

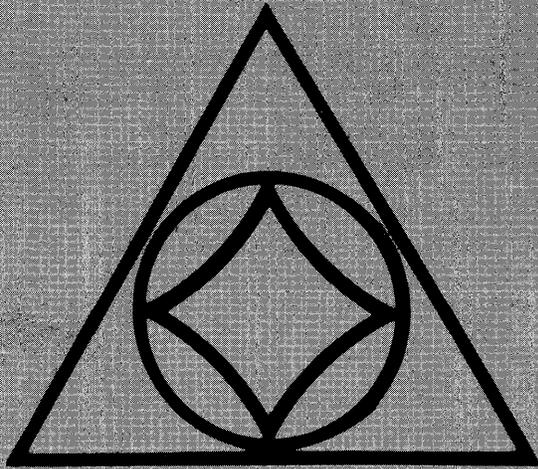
AFIPS

**CONFERENCE
PROCEEDINGS**

VOLUME 50

1981

**NATIONAL
COMPUTER
CONFERENCE**



ALEX ORDEN
Editor and Program Chairman

MARTHA EVENS
Co-Editor

ALBERT K. HAWKES
Conference Chairman

AFIPS PRESS
1815 NORTH LYNN STREET
ARLINGTON, VIRGINIA 22209

AFIPS

CONFERENCE PROCEEDINGS

1981

NATIONAL COMPUTER CONFERENCE

May 4-7, 1981

Chicago, Illinois

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1981 National Computer Conference or the American Federation of Information Processing Societies, Inc.

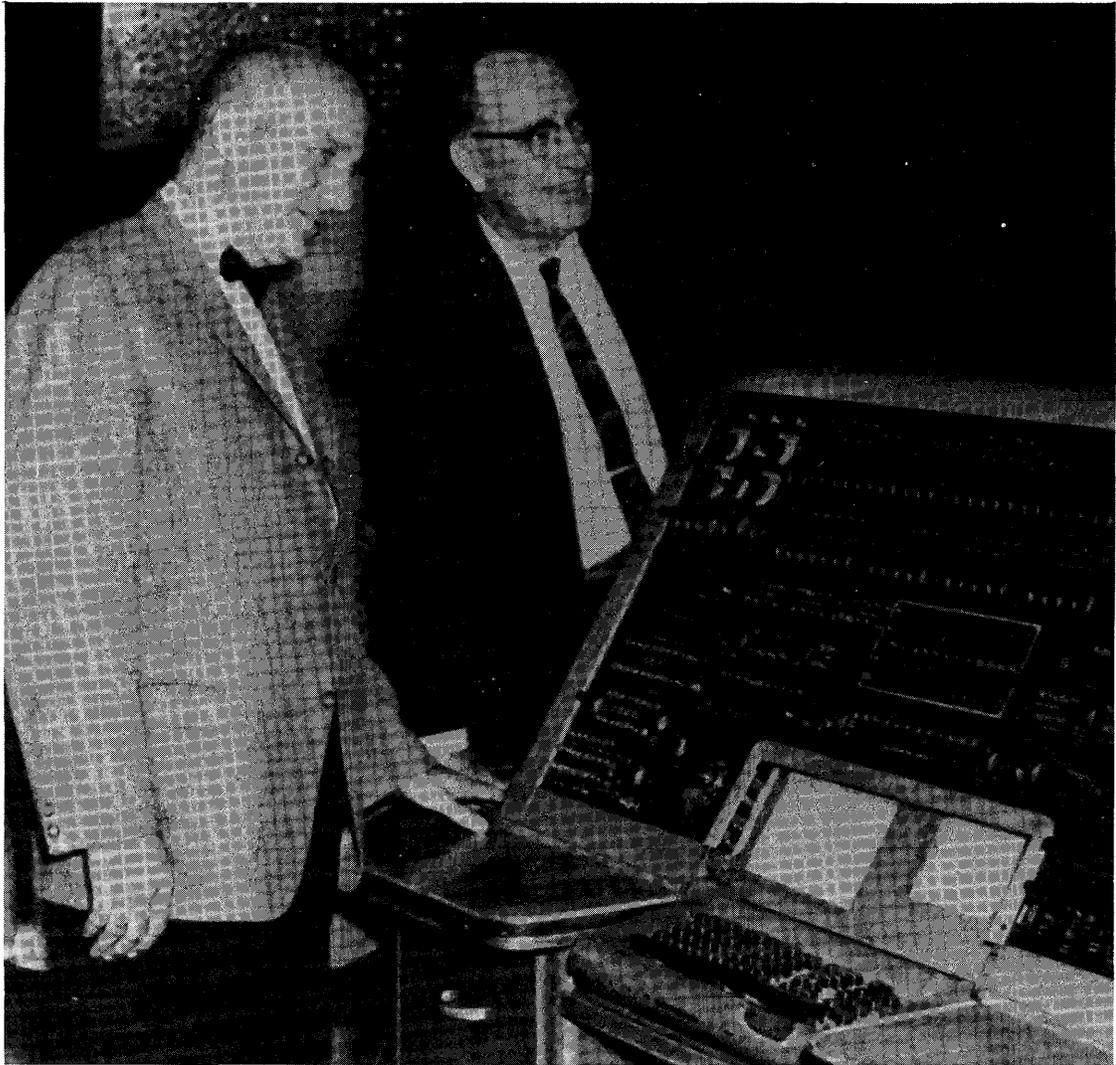
Library of Congress Catalog Card Number 81-65717

AFIPS PRESS

1815 North Lynn Street
Arlington, Virginia 22209

© 1981 by AFIPS Press. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) reference to the AFIPS Proceedings and notice of copyright are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to republish other excerpts should be obtained from AFIPS Press.

Printed in the United States of America



This edition of the *Proceedings* of the National Computer Conference is dedicated to J. Presper Eckert and to the late John W. Mauchly, whose pioneering efforts extended the frontiers of technology for the good of all mankind.

Preface

ALBERT K. HAWKES
1981 NCC Chairman

The National Computer Conference is an institution of the U.S. computer industry. Each year at the NCC, thousands representing companies, government agencies, and universities gather, and a forum is provided for hundreds of them. Much of what occurs is of course ephemeral, a great deal is of current and very practical value, and some lasts for many years. The *Proceedings* of the NCC is a principal element of that set of things from the conference with lasting value.

This 1981 Conference *Proceedings*, the fiftieth volume in the series, is an exhibit of the labors of Dr. Alex Orden, Program Chairman of the 1981 NCC. The myriad of details connected with organizing a program of this scope are treated in Professor Orden's introduction, and the reader will be wise to use that section as a road map before beginning to traverse the extensive volume. Authors of papers appearing here have contributed a great deal also, as have referees, session organizers, panelists, and other presenters of information at the conference whose contributions could not be made fully apparent in this volume. Another person who must be mentioned is the one who shepherded production of this *Proceedings*, Elizabeth G. Emanuel of AFIPS Press; her work was excellent and in keeping with the fine tradition at AFIPS established by Nelle W. Morgan many years ago.

It is an honor to prepare this preface to the lasting record of the 1981 NCC. In the same sense, it also is a rare privilege to be Conference Chairman. Having served on the NCC Committee starting nearly ten years ago, and serving as its chairman for two years covering the first and second NCCs, I did induce a number of friends and colleagues to serve as conference chairmen—Stephen S. Yau in 1974 and Carl Hammer in 1976. Thus it was a signal duty this year to try to put a National Computer Conference together for the Chicago area, which has been my home for the best part of my life.

One realizes early in planning an NCC that appointing a strong and able Conference Steering Committee is the only way to assure that this NCC institution can renew itself each year. These competent and willing volunteers who put this NCC together include Drs. Orden, Yau, and Hammer and Richard B. Wise (Director of Operations), Marvin W. Ehlers

(CSC Secretary and Special Projects Manager), M. Mildred Wyatt (Communications and Promotion), Sam Papa (Personal Computing, whose cochairman James Gerdes produced a companion *PC Digest*, which complements this volume for NCC '81), George Eggert (Professional Development Seminars), Fred Harris (Special Activities, including a very great step forward in NCCs by improving access to the conference for handicapped persons), Robert C. Spieker (NCC Liaison), Charles W. Schmidt (NCC '81 Fiscal Officer), Raymond Dash and Anthony S. Wojcik (the two Chicago area Vice-Chairmen of Professor Orden's Program Committee serving on the NCC '81 CSC), Joseph Leubitz (Registration), and Forest Mayberry (Conference Facilities). Serving ex-officio on the CSC over the two years that work was done on this conference and its planning were many AFIPS staff members. Principal thanks are due particularly to Paul J. Raisig, Executive Director; James H. Kroell, Director of Conferences, and his predecessor Gerard Chiffreller; Betty Lou Cooke, Conference Manager, and her predecessor Carol Sturgeon; Sam Lippman, Conference Operations Manager; and Christopher Hoelzel, AFIPS Press Manager. In addition, when trying to think of those who contributed a great deal to this conference and to establishing another mark in an excellent tradition, two of my mentors from the National Computer Conference Committee should be named. Over many years, Dr. Morton M. Astrahan and Jerry L. Koory stand out for their encouragement, support, and kindly advice given freely.

The idea of the NCC as an institution, which I mentioned previously, does remind one of what Benjamin Disraeli said over a century ago: "Individualities may form communities, but it is institutions alone that can create a nation." This great nation is known for its industry, and there are few more telling examples of American ingenuity producing something good and possibly great than what computer science and technology are today. The NCC, as an institution, has helped computing in the United States and in the world. May it go on, ever eclipsing those of us honored to be part of the tradition, so that NCCs always serve what is best in the U.S.A.

Introduction

ALEX ORDEN
1981 NCC Program Chairman

Since 1951 the National Computer Conferences have provided a forum for presentation of advances on all fronts of the computer field: research, development, and application. The *Proceedings* of the meetings give an unparalleled history of the field.

The *Proceedings* of 1976 include an interesting memoir by Herb Grosch on the history of the conferences, entitled "Conference Maketh a Ready Man, or Twenty-Five Years in the Better Joints." From 1954 to 1972, except in 1965, there were two conferences a year. From time to time the name has been changed—from 1951 to 1981 the series has consisted of

2 Joint Computer Conferences

1 Western Computer Conference

9 Eastern Joint Computer Conferences

8 Western Joint Computer Conferences

10 Spring Joint Computer Conferences

11 Fall Joint Computer Conferences

9 National Computer Conferences

—50 in all.

In recent years important developments in the history of digital computation have been commemorated by conference sessions and other special events on a day designated "Pioneer Day." This year's theme for that day is the completion 30 years ago of the first UNIVAC I system. Two historic events thereby coincided: (1) The emergence of computers from the laboratory machines of those days into the marketplace marked the birth of the computer industry. (2) Since the UNIVAC was the first machine designed for data processing rather than for mathematical computation, its development was the first major step toward the enormous development in computer-based data processing that has since occurred. It's all too easy in the 1980's to overlook the difficulties and doubts that had to be overcome back then. On that account, this volume is dedicated to J. Presper Eckert and the late John Mauchly, whose vision in the development of the UNIVAC made possible the penetration of computers into almost every field of human endeavor.

For 30 years, as the computer industry and the computer field have become ever larger and more complex, the National Computer Conferences have had a unique role in bringing together the computer R&D community, industry representatives, educators, application and facility systems analysts, DP managers, and end users to survey ongoing developments and examine new directions. Each year the formation of a new Program Committee sets the stage for a fresh assessment of trends in the field and an independent effort to assemble a well-balanced program. Drawing on several hundred paper submissions and session proposals—some invited by the members of the Committee, some independently submitted—the Committee sets up a program, which tends to consist of about one-third refereed papers, which appear in the *Proceedings*; one-third prepared talks on topics that do not lend themselves well to formal publication; and one-third panel discussions on trends and opinions.

For this year's NCC, in order to provide a framework in which all aspects of the computer field might be considered, the Program Committee was initially organized (about a year before the Conference) as five groups, each covering a wide area: (1) computer hardware and architecture, (2) software, (3) information processing management, (4) applications, and (5) social and economic implications. We knew of course that some important topics would straddle two or more of these areas. Indeed, as development of the program progressed, some members of the Committee freely ignored the area boundaries.

The identification of significant current trends emerged in part when we converted the initial five broad areas into a ten-track program. In the hardware/architecture area we found emphasis on microcomputer design, on developments in microprogramming, and on fault-tolerant computing; and from that general area we spun off five sessions as a separate track on network technology, with developments in local nets predominant. In the software area we found a strong emphasis on reliability, software validation techniques, and quality control. From software we extracted a six-session track on database systems, with emphasis on distributed databases. In information processing management we developed coordinated groups of sessions on DP project management, on management of transitions to new technology and methodology, on application systems audit and quality control, on the DP production process, and on personnel; and we formed a separate five-session track, Capacity and Performance Analysis. In the Applications area we separated out a track called Visuals, Natural Language Processing, and Artificial Intelligence, leaving—under the rubric Computers at Work—sessions in such still budding application areas as law, hospitals, energy, and simulation modeling. Since there seemed to be no clear dividing line between sessions on diverse issues in computer education and sessions on computer-related social issues, and since most social issues have much to do with the diffusion of knowledge, we formed a track called Education and Societal Issues. Finally, although there is now an annual AFIPS Conference on Office Automation, those rapid developments should nevertheless be included in the NCC. Therefore we provided the track entitled Automating the Office.

The published record of recent advances in the computer field, as seen at NCC '81, resides in the papers appearing in this volume. Since there was much more to the Conference, a condensed view of the entire program—in the "Conference at a Glance" form that appeared in the Program Brochure—is attached to this introduction.

It has been a challenging experience to coordinate this complex activity. It would not have been possible without the devotion and hard work of the Program Committee, the referees of the papers, and the AFIPS staff. Their names appear elsewhere in this volume. I particularly thank Liz Emanuel, who managed the editorial work on the *Proceedings* at AFIPS with great competence; and Martha Evens of the Illinois Institute of Technology, who joined me in dealing with the refereeing and selection of papers.

MONDAY, MAY 4 — NATIONAL COMPUTER CONFERENCE

PRINCIPAL TRACKS	1:30 to 3:00 PM	3:15 to 4:45 PM
HARDWARE & ARCHITECTURE	1.1 Design Tools for System Architectures George Kraft	1.2 Innovative Architecture & Commercial Computers Krishna M. Kavi
NETWORK TECHNOLOGY AND CAPACITY & PERFORMANCE ANALYSIS	2.1 Transport and Session Protocols in the Context of the ISO Reference Model Leslie Jill Miller	2.2 Packet Speech Danny Cohen
SOFTWARE	3.1 Programming Languages for Small Systems Leon Levy	3.2 Software Development Tools R. Stockton Gaines
INFORMATION PROCESSING MANAGEMENT	4.1 System Implementation Strategy Ken Zoline	4.2 Audit and Control in a Database Environment Steven Ross
EDUCATION & SOCIETAL ISSUES	5.1 Survey and Comparison of Model Curricula for Information Systems Education Thomas Ho	5.2 Joint Business-University Professional Development and Research Programs Robert A. Rouse
AUTOMATING THE OFFICE AND COMPUTERS AT WORK	6.1 Integrated Word- and Data-Processing Systems Robert Elliott	6.2 Office Automation Technology: Futures James Carlisle
DATA BASE SYSTEMS AND COMPUTERS AT WORK	7.1 Distributed Database Management Systems — Transaction Environment James Swager	7.2 Database Machines C. Robert Carlson
VISUALS, NATURAL LANGUAGE PROCESSING & ARTIFICIAL INTELLIGENCE	8.1 Image Analysis K. S. Fu	8.2 Pictorial Database Models & Query Languages K. S. Fu

PLENARY SESSION 10:00-11:00 AM

TUESDAY, MAY 5 — NATIONAL COMPUTER CONFERENCE		
PRINCIPAL TRACKS	8:30 to 10:00 AM	10:15 to 11:45 AM
HARDWARE & ARCHITECTURE	1.3 Microprocessor Architectures — What Next? K. Vairavan and Tadao Ichikawa	1.4 Perspectives on the History of Computing Paul Armer
NETWORK TECHNOLOGY AND CAPACITY & PERFORMANCE ANALYSIS	2.3 Local Networks and the ETHERNET in Particular (I) Gregory T. Hopkins	2.4 Local Networks and the ETHERNET in Particular (II) Gregory T. Hopkins
SOFTWARE	3.3 Functional Capabilities of Dictionary Systems Belkis Leong-Hong	3.4 Operating Systems Joseph Leung
INFORMATION PROCESSING MANAGEMENT	4.3 Technology Transfer: Management Issues Conrad Weisert	4.4 Planning for Technology Transfer Robert Scheer
EDUCATION & SOCIETAL ISSUES	5.3 Computers and the Future of Literacy Frederick Goodman	5.4 Issues Concerning National Computer Literacy in 1985 Robert Seidel
AUTOMATING THE OFFICE AND COMPUTERS AT WORK	6.3 Word Processing in Litigation & Information Retrieval Haley Fromholz	6.4 Computer Applications in Law Firm Management Haley Fromholz
DATA BASE SYSTEMS AND COMPUTERS AT WORK	7.3 Distributed Database Architecture Hal Uhrbach	7.4 Database Practicum Susan Rosenbaum
VISUALS, NATURAL LANGUAGE PROCESSING & ARTIFICIAL INTELLIGENCE	8.3 Intelligent Computer-Aided Instruction Mark Fox	8.4 Computer-Based Educational Aids Arthur Melmed

	1:30 to 3:00 PM	3:15 to 4:45 PM
PLENARY SESSION 12:00 NOON-1:00 PM	1.5 Fault-Tolerant Computing Systems Gerald Masson	1.6 Contemporary Fault-Tolerant Computer Designs William C. Carter
	2.5 Management of Capacity Planning Leonard Lipner	2.6 Network Capacity Planning Jeffrey A. Bloom
	3.5 Software Reliability in Real-Time Systems Bharat Bhargava and David Clapp	3.6 PASCAL: Standardization and Extension A. Winsor Brown
	4.5 Implementing Technology Transfer Denny O. Wallace	4.6 Systems Assurance: A Step Beyond EDP Audit James Krause
	5.5 Effects of Computers on Personal Life Abbe Mowshowitz	5.6 Where is the Story?: A Journalists Panel on Trends in Computing Brad Schultz
	6.5 Simulation of Natural Systems Roger M. Firestone	6.6 Future Office Systems Tom Sinopoli
	7.5 Research & Development in Distributed Database Systems Cory Devor	7.6 Database Systems Advances in Medical Systems Meera Blattner
	8.5 Communicating with Computers in Natural Languages — Current Capabilities Martha Evens	8.6 Communicating with Computers in Natural Language — Future Promises Norman K. Sondheimer

WEDNESDAY, MAY 6 — NATIONAL COMPUTER CONFERENCE

PRINCIPAL TRACKS	8:30 to 10:00 AM	10:15 to 11:45 AM
HARDWARE & ARCHITECTURE	1.7 Microprogramming—The Challenge of the 1980's (I) Samir S. Husson	1.8 Microprogramming—The Challenge of the 1980's (II) Samir S. Husson
NETWORK TECHNOLOGY AND CAPACITY & PERFORMANCE ANALYSIS	2.7 Capacity Planning in a Production Environment James Cooper	2.8 Simulation of Computer Systems: Software & Hardware Norman Schneidewind
SOFTWARE	3.7 Software Maintenance Stephen S. Yau	3.8 Quantitative Measures for the Quality of Systems and Programs Carma McClure
INFORMATION PROCESSING MANAGEMENT	4.7 Production Process in the Eighties Russ Melton	4.8 Business Communication; Security & Vulnerability John Donovan
EDUCATION & SOCIETAL ISSUES	5.7 Protection of Proprietary Interests in Software Susan Nycum	5.8 Planning Agenda for a National Health Information System Marion Ball
AUTOMATING THE OFFICE AND COMPUTERS AT WORK	6.7 Combining Office Automation and Data Processing—Its Technology and Usefulness Dan Zatyko	6.8 Form Processing in the Office Environment Mitch Zolliker
DATA BASE SYSTEMS AND COMPUTERS AT WORK	7.7 The Impact of Computing on the Handicapped in the Eighties Samuel C. Lee	7.8 Simulation: A Tool for Business Decision-Making Suresh K. Jain
VISUALS, NATURAL LANGUAGE PROCESSING & ARTIFICIAL INTELLIGENCE	8.7 Artificial Intelligence Applications to Electronic Circuit Design Tom Mitchell	8.8 Prospects for Artificial Intelligence Application in Industry N. S. Sridharan

	1:30 to 3:00 PM	3:15 to 5:15 PM
PLENARY SESSION 12:00 NOON-1:00 PM	1.9 Higher Level Microprogramming Languages and Optimization (I) Bruce Shriver	1.10 Higher Level Microprogramming Languages and Optimization (II) Bruce Shriver
	2.9 Special Session: Pioneer Day—Univac I Carl Hammer	2.10 Special Session: Pioneer Day—Univac I Nancy Stern and Henry Tropp
	3.9 Maintenance of Programs & Systems Ned Chapin	3.10 Software Development Facilities Louis Brocato
	4.9 Data Entry Productivity Lawrence Feidelman	4.10 Special Project Management Clifton Merry
	5.9 Private Sector Policy Issues on the Use of Computer Technology in the Healthcare Industry Karen Duncan	5.10 Alternative Data Processing Strategies for Hospital Information Systems David Mischelevich
	6.9 Definition & Measurement of Application Software Productivity Benn Konsynski	6.10 Electronic Mail: Current Developments Walter Ulrich
	7.9 Computer-Assisted Analysis in Energy/Economic Models Harvey Greenberg	7.10 Large Scale Database Applications Eugene Kozik
	8.9 Imaging & Computers Diana Merry	8.10 Educational Uses of Personal Computers Michael Tempel

THURSDAY, MAY 7 — NATIONAL COMPUTER CONFERENCE

PRINCIPAL TRACKS	8:30 to 10:00 AM	10:15 to 12:15 PM
HARDWARE & ARCHITECTURE	1.11 Adaptable Architectures Svetlana Kartashev and Steven I. Kartashev	1.12 Architecture of Specialized Hardware Systems William E. Farley
NETWORK TECHNOLOGY AND CAPACITY & PERFORMANCE ANALYSIS	2.11 Implementations of Experimental Local Networks William Lidinsky	2.12 Local Networks: The Fundamental Technology of Office Automation Harvey Freeman
SOFTWARE	3.11 Quality Assurance — An Emerging Technology Gene Altshuler	3.12 The User Interface Howard Lee Morgan
INFORMATION PROCESSING MANAGEMENT	4.11 Motivation of Computer Personnel J. Daniel Couger	4.12 Recruitment, Retention, & Certification of Data Processing Professionals Thomas A. Browdy
EDUCATION & SOCIETAL ISSUES	5.11 Computer Professional as an Expert Witness Alex Hoffman	5.12 Library & Business Computer Use: What's the Difference? Peter Lykos
AUTOMATING THE OFFICE AND COMPUTERS AT WORK	6.11 The Electronic Office: A Futuristic Forecast Richard Federico	6.12 Office Automation: The Federal Experience Ira W. Cotton
DATA BASE SYSTEMS AND COMPUTERS AT WORK	7.11 Computing & Energy Technology Assessment Ellen M. Leonard	7.12 Automated Testing for Increased Productivity Leonard Gardner and John Savage
VISUALS, NATURAL LANGUAGE PROCESSING & ARTIFICIAL INTELLIGENCE	8.11 Recent Computer Advances in Legislative Reapportionment Lee Papayanopoulos	8.12 Applications of Artificial Intelligence to Law L. Thorne McCarty

PRINCIPAL TRACKS	1:30 to 3:00 PM	3:15 to 4:45 PM
HARDWARE & ARCHITECTURE	1.13 Single Chip Computers — Where Are They Headed? K. S. Padda	1.14 The Application of Peripheral Array Processors Walter J. Karplus
NETWORK TECHNOLOGY AND CAPACITY & PERFORMANCE ANALYSIS	2.13 Use of Models in Capacity Planning Satish K. Tripathi	2.14
SOFTWARE	3.13 The Public Release of Smalltalk-80 Daniel H. Ingalls, Jr.	3.14 Computer-Based Tools for Software & Systems Engineering Gerald Estrin and Ray Houghton
INFORMATION PROCESSING MANAGEMENT	4.13 User Requirements Analysis Raymond Yeh	4.14 A Survey of Project Management Software Packages Linda Taylor
EDUCATION & SOCIETAL ISSUES	5.13 Developing Software Engineers in Industry Pei Hsia	5.14 Developing Software Engineers in the Universities Frederick E. Petry
AUTOMATING THE OFFICE AND COMPUTERS AT WORK	6.13 Choosing a Computer Language for a First Problem-Solving Course Robert J. McGlenn	6.14
DATA BASE SYSTEMS AND COMPUTERS AT WORK	7.13 Computing Applications in Magnetic Fusion Energy Research John T. Hogan	7.14 Computational Methods in Inertial Confinement Nuclear Fusion Keith A. Taggart
VISUALS, NATURAL LANGUAGE PROCESSING & ARTIFICIAL INTELLIGENCE	8.13 Expert Systems and Knowledge Engineering N. S. Sridharan	8.14

CONTENTS

Preface	v
Albert K. Hawkes	
Introduction	vi
Alex Orden	
COMPUTER HARDWARE AND ARCHITECTURE	
Software sympathetic chip set design	3
Richard F. Hobson	
A computer-aided VLSI layout system	11
W. A. Dees, K. M. Parmar, A. Goyal, R. Y. Tsui, B. D. Rathi, and R. J. Smith, II	
A multiprocessor description language	19
William T. Overman, Stephen D. Crocker, and Vittal Kini	
Fault tolerance by means of external monitoring of computer systems	27
Algirdas Avizienis	
The fault-tolerant 3B-20 Processor	41
L. E. Gallaher and W. N. Toy	
Firmware engineering: Methods and tools for firmware specification and design	49
Wolfgang K. Giloi, Reinhold Gueth, and Bruce D. Shriver	
New directions for micro- and system architectures in the 1980s	57
Harold W. Lawson, Jr.	
Microprogramming—The challenges of VLSI	63
Alice C. Parker and Wayne T. Wilner	
Vertical and outboard migration—A progress report	69
Andrew Heller and Andries van Dam	
Firmware testing and test data selection	75
Helmut K. Berg	
Specifying target resources in a machine independent higher level language	81
Scott Davidson and Bruce D. Shriver	
The design of a firmware engineering tool: the microcode compiler	87
Perng-Yi Ma	
Microcode compaction: Looking backward and looking forward	95
Joseph A. Fisher, David Landskov, and Bruce D. Shriver	
V-Compiler: A next-generation tool for microprogramming	103
Dave Patterson, Ross Goodell, Michael D. Poe, and Simon G. Steely, Jr.	
Adaptable pipeline system with dynamic architecture	111
Svetlana P. Kartashev and Steven I. Kartashev	
Modular crossbar switch for large-scale multiprocessor systems—Structure and implementation	125
Bernhard Quatember	
Some potential deadlocks in layered communications architectures	137
Joseph Hellerstein and Wesley W. Chu	
General-purpose integrated indexing circuits—A proposal	141
A. C. D. de Figueiredo	

The VALI (Variable Language Interpreter)	145
James D. Mooney	
The architecture of MANIP—A parallel computer system for solving NP- complete problems.	149
Benjamin W. Wah and Y. W. Ma	
Parallel sorting machines: Their speed and efficiency	163
Leon E. Winslow and Yuan-Chieh Chow	
NETWORK TECHNOLOGY	
Packet communication of online speech	169
Danny Cohen	
Highlights of a group effort in algorithmic development for packet-switched voice networks.	177
J. D. Markel	
A modular approach to packet voice terminal hardware design	183
G. C. O'Leary, P. E. Blankenship, J. Tierney, and J. A. Feldman	
Engineering computer network (ECN): A hardwired network of UNIX computer systems	191
Kai Hwang, Benjamin W. Wah, and Fayé A. Briggs	
A protocol for a new double-loop computer network and its implementation.	203
S. Leventis, G. Papadopoulos, S. Koubias, and J. Constantinides	
ILLINET—A 32 Mbits/sec. local-area network	209
W. Y. Cheng, S. Ray, R. Kolstad, J. Luhukay, R. Campbell, and J. W-S. Liu	
SOFTWARE	
A survey of currently implemented Pascal extensions	217
T. N. Turba and S. H. Costello	
A standard tool for information resource management	225
Michael E. Meyer	
SAGA: A system to automate the management of software production	231
R. H. Campbell and P. G. Richards	
The development facility approach to improved software development	235
David W. Johnson	
CARL—Experience of an application using clusters	241
E. Levinson, L. S. Levy, and J. B. Salisbury	
The software configuration management database	249
Edgar H. Sibley, P. Gerard Scallan, and Eric K. Clemons	
EUCLID—A language for compiling quality software	257
David B. Wortman, Richard C. Holt, James R. Cordy, David R. Crowe, and Ian H. Griggs	
The design and implementation of a new UNIX kernel	265
Charles Crowley	
A security policy for a profile-oriented operating system	273
Charles R. Young	
Distributed task force scheduling in multi-microcomputer networks	283
André M. van Tilborg and Larry D. Wittie	
The assignment of computational tasks among processors in a distributed system	291
Camille C. Price	
Software reliability in real-time systems	297
Bharat Bhargava	
A state- and time-dependent error occurrence-rate software reliability model with imperfect debugging	311
J. G. Shanthikumar	

On the complexity of measuring software complexity.....	317
G. Michael Schneider, Robert L. Sedlmeyer, and Joe Kearney	
Quantitative measures of MIS quality assurance during hardware conversion.....	323
John W. Center	
Taking the measure of program complexity.....	329
Jean Cochrane Zolnowski and Dick B. Simmons	
Salvaging your software asset (tools based maintenance).....	337
Michael J. Lyons	
Maintenance is a management problem and a programmer's opportunity.....	343
John Reutter, III	
Productivity in software maintenance.....	349
Ned Chapin	
Improving software testing in large data processing organizations.....	353
M. A. Holthouse and C. W. Lybrook	
Compiler validation—An assessment.....	361
George N. Baird and L. Arnold Johnson	
An approach to transfer verification and validation technology.....	367
Mark K. Smith, Leonard L. Tripp, Leon J. Osterweil, Richard N. Taylor, and William E. Howden	
Easy interactive access to batch image analysis software.....	375
Ronald L. Danielson	
A unified approach to online assistance.....	383
Nathan Relles, Norman K. Sondheimer, and Giorgio Ingargiola	
An experimental system to support a very high level user interface.....	389
William L. Batchelor and Lucian J. Endicott, Jr.	
Principles of good software specification and their implications for specification languages.....	393
Robert Balzer and Neil Goldman	
Modular documentation: A software development tool.....	401
Roy E. Anderson	
Specification technique for parallel processing: Process-data representation.....	407
Ken Hirose, Kiyoshi Segawa, Nobuo Saito, Norihisa Doi, Masahiro Hirata, Toshiharu Yamasaki, and Masayuki Takata	
A tiny portable language-independent macroprocessor and some applications.....	415
Robert C. Gammill	
 CAPACITY AND PERFORMANCE ANALYSIS	
Finite queueing approximation techniques for analysis of computer systems.....	423
Dimitris A. Protopapas	
Throughput-response measurements in a distributed CAD/CAM processing network.....	431
J. R. Rao and W. L. Hanna	
 DATABASE SYSTEMS	
Effective inference control mechanisms for securing statistical databases.....	443
Vangalur S. Alagar, Bernard Blanchard, and David Glaser	
Using partitioned databases for statistical data analysis.....	453
Ruven Brooks, Meera Blattner, Zdzislaw Pawlak, and Eamon Barrett	
Development of an automatic sleep EEG analysis and staging system.....	459
M. W. Vannier, E. Othmer, S. Othmer, and P. Fishman	

Embedding an information system within a generalized network environment	463
Darrell L. Ward	
The design of the Clinical and Research Information System for Psychiatry	469
Ruven Brooks	
A concurrency control algorithm in a distributed environment	473
Paul Decitre	
An alternative approach to distributed database updating	481
Richard J. Greene	
Multibase—integrating heterogeneous distributed database systems	487
John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong	
Architecture of a distributed database information resource	501
James R. Swager	
Optimization of the file access method in content-addressable database access machine (CADAM)	507
Sadayuki Hikita, Haruaki Yamazaki, Kiyoshi Hasegawa, and Yutaka Matsushita	
Parallel sort and join for high speed database machine operations	515
Mamoru Maekawa	
Highly parallel associative search and its application to cellular database machine design	521
Sakti Pramanik	
A generalized database access path model	529
Georges S. Nicolas	
Database programming with data abstractions	537
Burt Leavenworth	
Feature analysis of selected database recovery techniques	543
Bharat Bhargava and Leszek Lilien	
Data compression procedures utilizing the similarity of data	555
Yahiko Kambayashi, Narao Nakatsu, and Shuzo Yajima	
 INFORMATION PROCESSING MANAGEMENT	
Choosing application development tools and techniques	565
V. Kevin Whitney and Jane G. Morse	
A software requirements analysis and definition methodology for business data processing	571
Isao Miyamoto and Raymond T. Yeh	
A methodology for information system design	583
Colette Rolland	
 EDUCATION AND SOCIETAL ISSUES	
CSDP: A model for continuing education in data processing	593
Dennis M. Oliver, Robert A. Rouse, and Robert J. Benson	
People teaching people: A cooperative education venture	597
Edwin F. Kerr	
Computers and the future of literacy	601
Frederick L. Goodman	
 COMPUTERS AT WORK	
Keeping CAI humane in the humanities	605
Helen J. Schwartz	

The effects of computers on library staff and users: How can the administrator cope?	609
Richard W. Boss	
Libraries as local database producers	613
Robin Crickman	
Data files as library materials: Policies, procedures, and politics	617
Richard C. Roistacher	
Computerized weighted voting reapportionment	623
L. Papayanopoulos	
Hospital information systems tutorial: A guide for computer scientists and practitioners	631
David J. Mishelevich	
VISUALS, NATURAL LANGUAGE PROCESSING, AND ARTIFICIAL INTELLIGENCE	
Issues in the development of natural language front-ends	643
James Hendler, Thomas P. Kehler, Paul Roller Michaelis, Brian Phillips, Kenneth M. Ross, and Harry R. Tennant	
Text-critiquing with the EPISTLE system: An author's aid to better syntax	649
Lance A. Miller, George E. Heidorn, and Karen Jensen	
Shifting to a higher gear in a natural language system	657
Bozena Henisz Thompson and Frederick B. Thompson	
Computer speech for people with cerebral palsy	663
Jay Hewitt	
GRASS3, a language for interactive graphics	665
Nola Donato	
VISION II: A dynamic raster-scan display	671
Robert Rocchetti	
The development of the reactor safety film	677
Nancy N. Sheheen and Patrick J. Hodson	
The MODEL/IMAGES2 system: An application of computer graphics and three-dimensional geometric modeling to the jet impingement problem	681
W. R. Winfrey and S. R. Ricketts	
The applications of artificial intelligence to law: A survey of six current projects	689
Sandra Cook, Carole D. Hafner, L. Thorne McCarty, Jeffrey A. Meldman, Mark Peterson, N. S. Sridharan, James A. Sprowl, and D. A. Waterman	
An automated reasoning system	697
L. Wos, S. K. Winker, and E. L. Lusk	

**COMPUTER HARDWARE
AND ARCHITECTURE**

Software sympathetic chip set design

by RICHARD F. HOBSON

Simon Fraser University
Burnaby, British Columbia

ABSTRACT

The current status of special function unit (SFU) use in microcomputer systems is reviewed. Also outlined are areas where more sophisticated SFUs can be used to improve low- and high-level software environments in a microcomputer system. A structured machine model is presented to help containerize and control classes of software and hardware artifacts.

INTRODUCTION

With the 1980's comes an era wherein hardware modularity, specialization, and structure are closing in on the software territory of the 1970's. To simplify and speed up complex microcomputer software systems, more functionality and control are being packaged in hardware or firmware controlled specialized function units (SFUs). SFUs are appearing either as isolated performance enhancement devices or as part of an announced chip set. Ad hoc microcomputer hardware module expansion may lead to more of the architecture irregularities plaguing software engineers. To avoid this, we should look seriously at architectures based on a coherent structure of such modules.

In this article we review the current status of SFU use in microcomputer systems. We also outline areas where more sophisticated SFUs can be used to improve low- and high-level software environments in a microcomputer system. To this end, a model system is described. The model is partially motivated by an evaluation of the potential of a scientific arithmetic processor chip relative to a typical first-generation microcomputer (see Appendix I). While a performance factor increase of 3 or 4 was expected, the actual factor was closer to 10. More recently, the Intel 8087 has raised this factor to 100. Speed notwithstanding, we must strive to avoid strikingly awkward software sequences, whether to perform operations on data types that are not natural to a microprocessor (e.g., floating-point data) or to interface one chip with another. One cannot just glue LSI chips together and retain, at the VLSI level, the same software appeal that each chip may have enjoyed individually.

The status of language-oriented computer design is briefly reviewed in Appendix II.

CURRENT TRENDS

LSI specialized function units have become the dominant choice for arithmetic processing, floppy- and hard-disk control, CRT refresh and control, communication protocol, DMA control, and memory management.¹ Single-chip computers with on chip ROM and RWM also fall into this category. The latter are best suited for small dedicated tasks.² With such "support staff" potential, a central processor can have more time to execute higher-level control functions for an operating system or an application package. Code sections will also be shorter, more to the point, and more reliable.

The latest microprocessors have been enhanced in a variety of ways over their precursors.³⁻⁷ Prominent 16-bit contenders portray traditional register architecture philosophy more than HLL architecture philosophy (see Table I). Their new bus-sharing protocols are significant because multiple SFUs can be set up with a variety of processor interconnection schemes.

The main mode of communication between microprocessors and special devices is via I/O sequences of the type described in Appendix I (see also Wakerly⁸). The next most common mode would be a message buffer, with cooperating processors either sharing the same buses or separated by bus arbitration hardware.⁹ Ultimately we would like to be able to configure a microcomputer system with a collection of SFUs in such a way that the instruction set can be dynamically expanded to include new functions. The Intel 8086's ESCAPE mechanism is such a technique.

What we have seen so far is a packaging of clearly identifiable functions. What is not clearly identifiable as a function, notably high-level program and environment control, has been left over for the general-purpose microprocessor and the programmer. The next step is to identify hardware control, program control, data control, operating system control, and language interpretation as functional areas; hence potential candidates for LSI modules. Research is needed to iron out communication and high-level language/environment support problems before functional module unification will be achieved.

TABLE I—Some architecture features employed in prominent new LSI microprocessors

Features	Processors		
	I8086	M68000	Z8000
Base relative addressing	T	T	T
Memory management	F	New chip	New chip
Memory segmentation	Partial	Partial (New chip, New chip)	Partial
Multiprocessor bus	T	T	T
Dedicated coprocessors	T	F	F
State marking (on stack)	F	T	T
Unused opcode trap	F	T	T
Opcode expansion	F	Via trap	Via trap
Supervisor state	F	T	T
Floating-point instructions	F	To come	F
Floating-point processor	New chip	F	F
Channel processor	New chip	F	F
Instructions/data separate	Optional	Optional	Optional

TOWARD A PHILOSOPHY FOR CHIP SET SPECIALIZATION

Basic computer organization is often introduced in terms of logical functional units for input, output, memory, arithmetic and logic, and control. In general we do not think of these units as constituting a multiple-processor or distributed-processor machine. Nor, for the most part, do we care. Enter LSI technology. Now digital systems must be packaged as functional units for cost effectiveness and design simplicity.

How should the logical functions be distributed? What range of functions is relevant?

Since high-level-language (HLL) notation is the preferred way to describe an application, one technique is to identify separable activities that all or many HLLs require for run-time support. These requirements may then influence the resulting architecture,¹⁰ but only part of the architecture. Since languages are simply tools, we should not look exclusively at their requirements. Some applications have requirements that may not be adequately representable with known HLLs. Text editing or word processing, for example, have data structure and real-time interaction requirements that are difficult to describe effectively with conventional HLL techniques. Real-time operating system functions also form a special class. We have heard it said that a semantic gap often exists between machine architecture and programming language features.³⁹ It is also clear that such a gap exists for programming environments as well. It is the total programming environment that should be considered when designing a multiple-processor chip set—hence, computer architecture that is oriented to a high-level-programming environment rather than to an HLL.

Traditional logical partitioning remains essentially valid. It is the degree of specialization, the sophistication of instructions, and the physical separation of units that is outdated. To begin with, input and output belong to one (or more) separate processor modules, as do other specialized operations. A pro-

gramming environment is also destitute without sophisticated memory management. Finally, for control, the environment should be managed by a small real-time control processor. To illustrate, consider a specialized unit model (SUM.4) consisting of four units: environment control unit (ECU), program management unit (PMU), data management unit (DMU), and arithmetic/logic processor unit (ALPU). They represent an SFU system hierarchy, as depicted in Figure 1. The following subsections describe some of the functions required for these units to support an HLL-oriented programming environment. The main objectives are

1. To reduce the complexity of system software by providing a variety of high-level functions as software primitives or built-in tasks
2. To promote structured machine design
3. To provide better single user run-time support for a variety of high-level languages and real-time applications

ECU

The environment control unit provides operating system functions and high-level I/O interpretation (see Table II).

A small real-time operating system runs in the ECU, the application control system (ACS). ACS contains fixed tasks for all devices that may be attached to SUM.4, e.g., for network control, console graphics, or editing. Provision must be made for transient or user support tasks that may be required to augment the functional capabilities of ACS for different applications—in particular, high-level-language I/O interpretation tasks. Interface “pipes” or tasks for PMU and DMU communication are important parts of ACS.

ECU architecture must be oriented toward variable interval interrupt servicing and I/O translation needs. Many of the existing microprocessors have appropriate features for an ECU, but in terms of LSI rather than VLSI. A user microprogrammable or custom VLSI ECU could contain firmware for a complete operating system nucleus.

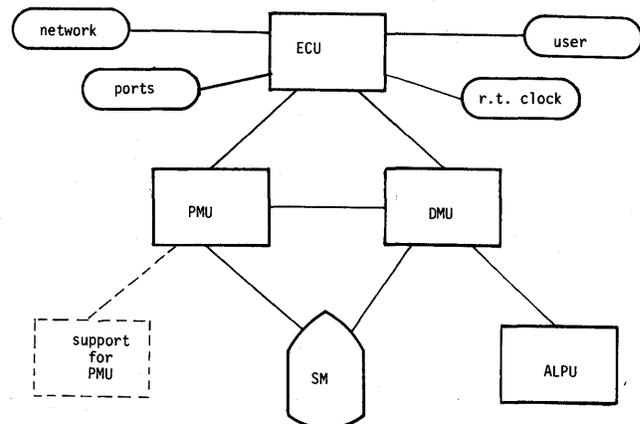


Figure 1—Block diagram showing the SUM.4 hierarchy

PMU

The program management unit is unique in terms of current microcomputer architecture in that it is the first of several possible processors that may contribute to the interpretation and execution of user code.

In an interpretive mode (e.g., for debugging or supporting a language such as APL), some internal form of the source code is fetched from segment memory (SM), checked for syntax errors, and translated into formatted, directly executed language (DEL) instructions for the DMU to execute. Operations modifying the execution environment, such as procedure call, are partially handled by the PMU.

The value of a separate PMU is not so clear for compilable languages. However, if we can get enough overlap between interpretation and execution of a compilable language, there is little reason to compile. Interpreters permit advantages, such as interactive debugging, execution environment protection, and run-time recovery, that cannot reasonably be matched by compilation. Microprocessors have been successfully used to interpret UCSD P-code machine instructions.³⁶ This experiment gives confidence that two or more "simple" processors with appropriate firmware can be very efficient at interpreting a suitably chosen intermediate form of Pascal, an easily compiled language. Greater benefits should result from more complicated languages. The key phrase here is "suitably chosen intermediate form." As long as run-time errors can be trapped and tied to the offending source statement expression through "reverse compilation," compilation per se is tolerable. Thus, even for compilable languages, the PMU can play an important role.

PMU activities are language-oriented. For multilingual support a user microprogrammable processor is required. This processor needs to manipulate code streams and communicate effectively with segmented memory. Additional special function support may be added to the PMU (see Figure 1).

DMU

A data management unit is mainly responsible for controlling operations on various data aggregates. Instructions or instruction bursts are normally fetched from the PMU interface. If the DMU is equipped to execute them directly, it does so; otherwise DMU action centers around providing operand data for the ALPU or ECU to manipulate. Before operation execution is permitted, operand validity must be checked. Such verification is facilitated by the use of data tags.

DMU and PMU design requirements are similar: Both are language dependent; both manipulate data aggregates (assuming that various forms of a program constitute data aggregates). The same processor architecture may thus be used for DMU and PMU units; only their microcode is different.

ALPU

Arithmetic and logical functions that the DMU cannot handle efficiently are provided by one or more specially designed units constituting the ALPU. This unit should operate only on

TABLE II—A possible partitioning of functions performed by various specialized units

ECU

- Task initiation (with PMU).
- Single-step control (with PMU).
- User command interpretation.
- Service special events (e.g., errors).
- HLL data formatting.
- Peripheral communication.
- Special device control (e.g. graphics).
- Text editing and command input.
- Real-time clock control.
- Execute I/O subtasks.
- Load microcode.
- Test other modules.

PMU

- Interpret HLL programs (or a suitable intermediate form).
- Execute procedure CALLs.
- Establish environments, manage tasks.
- Help maintain segmented memory.
- Service exceptional conditions.

DMU

- Execute DEL instructions from the PMU or ECU interface.
- Verify data operations (e.g., bounds checking).
- Manipulate operand data for the ECU and ALPU.
- Perform some functional operations (e.g., data rearrangements).
- Help maintain segmented memory.
- Report conditions to PMU.

ALPU

- The usual scientific calculator functions.
- Logical functions on strings.
- Adjust automatically to data size change (necessary for APL).

atomic or scalar items and need not have memory-accessing skills if the DMU interface is properly designed. The Intel 8087 is a good example of functional sophistication in this class, but it is too dependent on the Intel 8086.

SM

Segmented memory is not treated as a separate unit in this scenario. Memory modeling is, however, an important consideration in the quality of a programming environment. Variable-length containers for procedures and data aggregates greatly simplify run-time memory management firmware. Beyond that, segments can be associated with property lists for database content identification and protection. SM interfacing functions are divided among the DMU and PMU in SUM.4. Memory management is an integral part of interpretation, whereas a separate system is required to be interfaced with compiled code.

The unit interface

A number of multimicroprocessor communication schemes have been described in the literature.^{9, 12-15} For system modu-

```

program EX1 (output);
const  H = 34; D = .0625; S = 32.; L = 32;
       SEPARATOR = '-----';

procedure PLOT (var XS, YS: real;
               var XO, LIM: integer);
const  TWOPI = 6.28318;
var    X, Y: real; I, N: integer;
begin  for I = 0 to LIM do
       begin X = XS*I;
            Y = EXP (- X)*SIN (TWOPI*X);
            N = ROUND (YS*Y) + XO;
            repeat WRITE (' '); N = N - 1;
            until N = 0;
            Writeln ('');
       end;
end; {plot}

```

Figure 2—Pascal program example (see text)

larity and simplicity, I recommend using two or three port memories with an asynchronous wait when there is memory contention. Semaphores can be implemented by having a “hog” mode, permitting a processor to retain memory selection beyond one cycle. There are no bus contention problems with this model. Memory port multiplexing can be combined with refresh control and address translation in one cascable LSI module.

Messages are received indirectly by polling status word locations or directly through signal interrupts.

Example

To work with Pascal a user asks the ECU to load the Pascal firmware/software assist package (or it may be loaded implicitly). Figure 2 shows a simple Pascal program adapted from Jensen and Wirth.¹⁷ The program causes a damped sine wave to be plotted along a vertical axis according to supplied scaling parameters. A source operand recoverable, internal format is used to store the program. Variable name literals are stored in a master symbol table for run-time recovery.

The ECU conveys execution requests to the PMU, which then sets up an execution environment, allocates variables, initializes values, and begins sending code to the DMU. An implementer would probably use individual segments for data aggregates such as sets, arrays, and records, whereas scalars would be kept in the variable-length program segment. Environments for more complicated languages can easily be managed.

Built-in procedures such as WRITE and WRITELN are implemented through a task in the ECU. More interesting I/O procedures, such as a plot package, might be entirely implemented through the ECU. This facility permits a smooth interface between language, environment, and hardware.

There are many interesting implementation problems to be solved. For example, should I, the FOR loop variable in Figure 2, be incremented and tested by the PMU or the DMU? Control is simpler if the PMU performs decision-making operations and maintains iteration counters. This means that control variables, such as I, can only be modified by the

DMU with the PMU's permission. We see an opportunity here to improve program structure by distinguishing control from action.

To implement time-shared multitasking, timekeeping duties go to the ECU, which notifies the PMU to switch environments. In this case task maintenance is a PMU responsibility.

Implementation

Once an instruction set has been chosen, implementation details are irrelevant to most users. In a research environment, one favors microprogrammability because new primitives can easily be added to upgrade a unit. With custom chip fabrication nearing the grasp of lower-volume applications, we can visualize chip sets designed specifically to support a structured architecture.^{18, 19} As wafer-scale integration becomes economical, we will probably see the equivalent of these chip sets laid out as individual modules on a single wafer. Indeed, large hardware projects, like large software projects, must be divided into a number of coherent pieces with well-defined interfaces. In the VLSI era, multiple-processor systems will be essential for design simplicity as well as for greater throughput.

With modular hardware design, units can be developed by teams of specialists without relying heavily upon each other. Prototype modules can be implemented on a single PC board with the intention of gradually combining them into a single hybrid package and finally onto one chip.

CONCLUSION

The current proliferation of microcomputer hardware needs a focus. As a greater variety of SFUs are produced, we will be faced with organizational problems such as befell software in the past decade. The remedy? Structured architecture. We should think of hardware as a kind of petrified production software system. Hardware design therefore qualifies for all of our software engineering experience.

A structured model has been presented to help containerize and control classes of software and hardware artifacts. I am presently engaged in building a SUM.N ($N = 4-6$) prototype to study performance, function distribution, communication techniques, and language/environment support.

ACKNOWLEDGMENTS

Research support from NSERC of Canada and from a Simon Fraser University President's Research Grant is gratefully acknowledged.

APPENDIX I—THE CASE FOR SPECIALIZATION

Specialized LSI processor chips have been available for several years. Only recently, however, have microprocessor ven-

dors begun to realize their potential. This section compares the performance of an Intel 8080A microprocessor (MP) with that of Advanced Micro Devices' Arithmetic Processor Unit, the AM9511.^{20, 21} Floating-point addition is the benchmark operation. This exercise demonstrates the desirability of LSI special function units because they are much faster than general-purpose processors of the same technology and because, with a well-designed interface, software problems can be greatly reduced.

Benchmark data have a 24-bit normalized (sign magnitude form) binary mantissa and a 7-bit 2's complement exponent. FADD, our 8080A floating-point addition subroutine, contains 263 instructions (loops were avoided for speed) and assembles into 361 bytes of code. Operand pointers for the expression $Z = X + Y$ are passed in stream following the CALL. The simplest cost formula for FADD, assuming two positive numbers, can be represented as follows:

Entry overhead and setup	:	76 (8080A clock
Fetch X	:	233 cycles)
Overhead	:	32
Fetch Y	:	233
Comparison of exponents and overhead	:	113
Assuming $\text{exp } X - \text{exp } Y = N$, adjust Y N bits	:	$93N + 52$ (align mantissas)
Operation overhead	:	102
Mantissa addition (no carry)	:	125 (no renormalization)
Mantissa addition (carry)	:	225 (1 bit renormalization)
Store results	:	231
Exit overhead	:	41
typical add	:	1286 (carry, but no alignment shift)
"Fastest" add (without argument passing)	:	509 clock cycles

FADD is neither expressive nor conceptual, and it is certainly not software-sympathetic. Fetching the operands requires eight successive sequences of the form

```
LDAX B
STAX D
INX B
INX D,
```

consuming 24 clock cycles each. With a block move instruction the movement should only require 10 clock cycles per byte, i.e., five cycles each direction (the 8080A does a two-byte POP in 10 cycles and a two-byte PUSH in 11 cycles¹⁹). This leads to an improvement factor of 2.4 for operand handling. Another awkward operation for the 8080A is the calculation of 2's complement overflow, although most MPs do include an overflow flag.

For a contrast to the above, consider Advanced Micro Devices' Am9511 arithmetic processing unit (henceforth, the APU). This chip is made from similar N -channel silicon gate MOS technology and is rated at the same clock speed (2 MHz). The APU has an 8-by-16- or 4-by-32-level cascading

TABLE III—A summary of AM 9511 instructions

(Single, double, floating):	ADD, SUB, MUL, DIV.
(floating):	SQRT, SIN, COS, TAN, ASIN, ACOS, ATAN, LOG, LN, EXP, PWR.
(other):	NOP
	FIXS (convert top of stack [TOS] to single precision integer),
	FIXD (convert TOS to integer double), FLTS, FLTD,
	CHSS (change sign of integer single on TOS), CHSD, CHSF,
	PTOS (push integer single TOS to NOS etc.), PTOD, PTOF,
	POPS (pop integer single NOS to TOS etc.), POPD, POPF,
	XCHS (exchange integer single NOS with TOS), XCHD, XCHF,
	PUPI (push floating point constant pi onto stack).

arithmetic stack in reverse Polish notation calculator tradition. As can be seen from Table III, an impressive list of fixed or floating-point operations are available in comparison to an MP instruction set. Floating-point addition for the APU has a listed execution time range of 56–350 clock cycles (not counting argument passing). The lower figure is comparable to our above figure for the fastest add (not counting argument passing), 509 clock cycles. At the other extreme, the APU can do any addition within 350 clock cycles. Our benchmark requires up to 2748 clock cycles to add two positive numbers, including a 23-bit exponent equalization shift. If mixed signs are permitted, renormalization requires an additional 126 cycles per bit. Not counting operand fetch and store (for either unit), the APU is an order of magnitude faster than an 8080A at floating-point addition! The APU obtains this advantage through an optimized register level architecture and because irrelevant instruction fetches are avoided. Because more storage, stack operations, and instruction fetches are involved, other APU operations should also be considerably faster than equivalent 8080A operations. Operand passing is equally bad for both benchmarks because the 8080A was not equipped for block data transfer. Recently, the Intel 8087 arithmetic processor stretched this difference in performance by another order of magnitude!⁴⁰

Chip Communication

The software interface for these units should be considered as much a candidate for optimization as their operation logic. A simple interface between an MP and our APU example would be via 2 I/O ports, as depicted in Figure 3. Binary floating-point addition, using Zilog Z80 block move instructions, may be represented as follows:

```
LDI HL,X ;REG(HL) := ADDR(X)
MVI C,PUSH# ;REG(C) := push port #
MVI B,4 ;REG(B) := data precision
MOV D,B ;save precision, block move is
;destructive
```

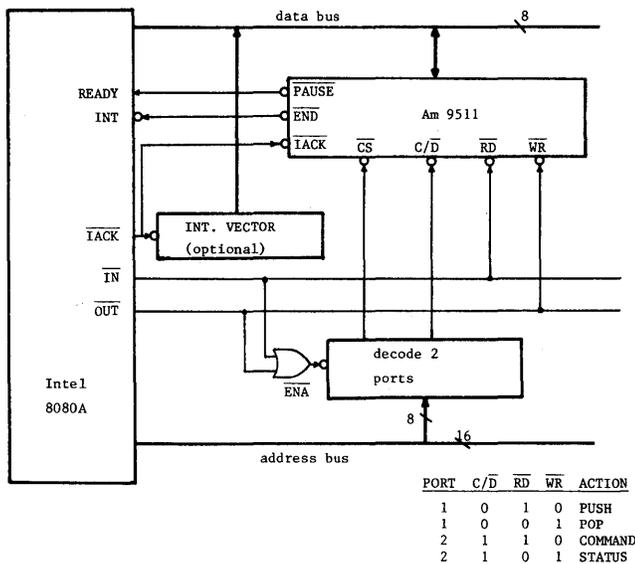


Figure 3—A simple I/O interface for the Intel 8080/A and the Am 9511

```

OUTIR      ;move value of X to APU stack
LDI  HL,Y
MOV  B,D
OUTIR      ;move value of Y to APU stack
MVI  A,PLUS ;REG(A) := APU operation code
MVI  C,CTL# ;REG(C) := control port#
OUT  (C),A ;perform operation
LDI  HL,Z
IN   A,(C) ;fetch status
....
MVI  C,POP# ;REG(C) := pop port #
MOV  B,D
INIR      ;move results of X + Y to Z.

```

Although the above code will obtain floating-point addition results faster than FADD, its appeal is lost because there is a large semantic gap.³⁹ It bears little resemblance to the fact that we are fetching X and Y for floating-point addition. Sympathetic software should provide a clean notational link between application and hardware. It is sometimes possible to fake hardware operations by using low-level software macros and subroutines, but these constructions occupy more space and take time to develop, maintain, and standardize.

Direct-memory access (DMA) chips are occasionally used for speed and software improvement, but even with DMA the above example requires a messy communication routine. Until very recently, support chips have not been designed to help improve instruction notation or expressiveness. There are few attempts to integrate their effects into the microprocessor's instruction set.

Symbolic code for the statement $Z = X + Y$ might cause a compiler or an interpreter to produce reverse Polish notation code:

```

PUSH X
PUSH Y

```

```

FADD
POP Z.

```

Or, for a three-address format, the statement is already directly executable. Leaving aside the virtues of direct HLL execution, these forms are software-sympathetic. We expect the MP to know who is responsible for executing such instructions. We also expect the MP to know how to communicate with the implied device. Intel's coprocessor technique is a partial solution to this problem.²¹

A coprocessor monitors the system bus looking for a special opcode called escape (ESC). Six bits within the two-byte ESC sequence may contain a coprocessor opcode. ESC causes the master processor to put an operand address on its address bus and perform a memory read. The data so read can be used immediately (e.g., PUSH) or may be ignored. Once selected, the coprocessor drops its TEST line for synchronization with the host. For continued coprocessor interaction, the main processor must see a WAIT instruction following ESC. Once activated, the coprocessor is free to access memory by putting the main processor on HOLD through the bus request/bus grant protocol.

Now consider the previous example using Intel 8086 assembler-type code with an arithmetic coprocessor (the Intel 8087):

```

....
ESC PUSH,X ;send X to coprocessor.
WAIT      ;synchronize.
ESC PUSH,Y
WAIT
ESC FADD,AL ;request floating-point operation.
WAIT
ESC POP,Z  ;deposit results.
....

```

The coprocessor technique is a much needed improvement over our previous example. But we do not see any advantage in having the coprocessor understand a complicated instruction stream. Nor does it seem necessary for a coprocessor to access memory itself. These irregularities can be removed with an improved interface and an appropriate instruction set.

It is clear that SFUs will play a vital role in future micro-computer design. Coprocessors, SFUs in general, are still more of an exception than a rule.

APPENDIX II—WHITHER LANGUAGE-DIRECTED COMPUTER DESIGN?

High-level language features have been influencing computer designers for some time.²³⁻³⁰ Where are the results? In the beginning, because there were no high-level languages, hardware technology dictated architectural features. In any case, technology was not capable enough. By the time an HLL executing processor was considered feasible, register architecture had a firm grip on both ends of the commercial market, namely IBM 360 and PDP 8. Already the enormous investment in software exerted great pressure on IBM to remain upward-compatible.

DEC had an opportunity to make revolutionary changes to minicomputer architecture when they designed the PDP 11. Indeed, software played a large role in that design, but not

HLL software. They were more interested in how easy it was to write assembler code, how a compiler would produce machine code, how a loader would work, how relocatable code would be, etc.³¹ The main uses of minis at that time were still in laboratories. A register-oriented architecture was the demonstrated choice for data acquisition and process control applications. So, while some improvements are visible, e.g., stack features and memory-to-memory operations, DEC soon found themselves in the same hammerlock condition as IBM and others. DEC's new VAX does exhibit several features that are convenient for arithmetic expression evaluation, environment control, and certain COBOL operations.

Microcomputers have repeated this history.³² If it were not for their simple architecture, the early microprocessors could not have been produced on a single chip. The jobs they were designed to fill were considered known a priori and suited a simple register/stack environment. When the potential of a microcomputer was finally realized, economic considerations again prevailed. New products were simple extensions of previous best sellers. For example, the Intel 8080 evolved from the Intel 8008. Interfacing remained the main application.

Stacks provide the most widespread connection between HLLs and machine design.³³⁻³⁶ Notable in this area are the Burroughs B5700, B6700, and B7700 series. More recent entries include the Hewlett-Packard 3000 and 300, the Microdata 32/S, and the WD9000 by Western Digital. Microprocessors in general are beginning to exploit stack techniques, although not on the same scale as the above. Stacks provide an efficient logical mechanism for run-time control of block structured languages. Stacks also facilitate execution of the Polish string representation of an arithmetic expression. However, recent work shows that stack architectures are often more convenient than optimal.^{39,41,42}

The SYMBOL computer is one of few HLL direct execution machines to have been built.³⁷ Seven autonomous SFUs perform program translation, Polish string execution, virtual memory management, I/O, etc. Programs tend to flow through the machine in a pipelined fashion for greater throughput. Results from this project firmly establish the feasibility of direct HLL execution. But the popularity of an HLL is a delicate marketing issue. Although old languages tend to fall into disfavor (although not disuse), new languages require several years to reach the hearts of programmers. Current trends in LSI technology indicate that a general-purpose machine with fewer SFUs and a simpler architecture than SYMBOL could be very successful.

Burrough's B1700 series was designed to interpret intermediate-level languages through microprogrammed interpreters.³⁸ Great effort was expended to abolish fixed-length word sizes and data formats at the hardware level. Instead, variable-length bit strings can be mapped into any desirable data structure. HLLs are supported through dedicated S-languages, which rely upon the use of stacks and special storage-to-storage instructions. The B1700 demonstrates that HLL-oriented computers are commercially feasible. It also demonstrates the flexibility of microprogramming for switching from one high-level environment to another. But as a latecomer to the industry, its popularity is hampered by well-established competition. Cost effectiveness is difficult to establish for machines such as SYMBOL and B1700 because work units are

different. Benchmarks are hard to agree upon. Again, current trends in LSI technology suggest that a combination of the B1700 and SYMBOL architectures should be very effective and practical.

Some microcomputers are dedicated to one HLL, for example MCM 900 and IBM 5100 APL machines. What matters when working on a dedicated machine—or any machine, for that matter—are implementation details affecting the programming environment and response time. We are at a stage now where specialized hardware units can be produced more cheaply than ever. It seems inevitable that they will be used to boost run-time efficiency in such machines.

What about new markets? If you consider ROM-controlled microcomputers such as the Apple, Pet, and TRS-80 to be HLL-oriented machines (many do), then HLL machines are exploding into an as yet unlimited market. A chip set version of SUM.4 would compete in this lower-cost but highly personalized computer market.

SUM.4's underlying philosophy recognizes the need for environments that support application programming in a variety of HLLs. Interpretation of an HLL is considered essential for debugging. During this phase speed is not a major factor because the system is generally I/O-bound. Hence direct execution should exist at least as a software option. Once a program is ready for production use, it should become as much an integral part of the environment as possible. This is difficult to achieve even in a unilingual environment. Structured digital system design can benefit from many historical architectural concepts, either globally or locally. What we need most are more working models for quantitative comparison.

REFERENCES

1. Posa, John G. "Peripheral Chips Shift Microprocessor Systems into High Gear." *Electronics*, 52 (1979), pp. 93-106.
2. Wakerly, John F. "Intel MCS-48 Microcomputer Family: A Critique." *IEEE Computer*, 12 (1979), pp. 22-31.
3. Morse, Stephen P., William B. Pohlman, and Bruce W. Ravenel. "The Intel 8086 Microprocessor: A 16-bit Evolution of the 8080." *IEEE Computer*, 11 (1978), pp. 18-27.
4. Stritter, Edward, and Tom Gunter. "A Microprocessor Architecture for a Changing World: The Motorola 68000." *IEEE Computer*, 12 (1979), pp. 43-52.
5. Peuto, Bernard L. "Architecture of a New Microprocessor." *IEEE Computer*, 12 (1979), pp. 10-21.
6. McKeivitt, James, and John Bayliss. "New Options from Big Chips." *IEEE Spectrum*, 16 (1979), pp. 28-34.
7. Stritter, Skip, and Nick Tredennick. "Microprogrammed Implementation of a Single Chip Microprocessor." *SIGMICRO Newsletter*, 9 (1979), pp. 8-16.
8. Wakerly, John F. "Microprocessor Input/Output Architecture." *IEEE Computer*, 10 (1977), pp. 26-33.
9. El-Ayat, K. A. "The Intel 8089: An Integrated I/O Processor." *IEEE Computer*, 12 (1979), pp. 67-78.
10. Allison, Dennis R. "A Design Philosophy for Microcomputer Architectures." *IEEE Computer*, 10 (1977), pp. 35-41.
11. Sites, Richard L. "How to Use 1000 Registers." *Proceedings of the Caltech Conference on VLSI*, January 1979, pp. 527-532.
12. Swan, R. J., S. H. Fuller, and D. P. Siewiorek. "Cm*—A Modular Multimicroprocessor." *AFIPS NCC Conf. Proc.*, 46 (1977), pp. 637-644.
13. Adams, George, and Thomas Rolander. "Design Motivations for Multiple Processor Microcomputer Systems." *Computer Design*, 17 (1978), pp. 81-89.
14. Gonzalez, Mario J., Jr. "Future Directions in Computer Architecture." *IEEE Computer*, 11 (1978), pp. 54-62.

15. Brinch Hansen, P. "Multiprocessor Architectures for Concurrent Programs." *ACM 78 Conf. Proc.*, Washington, D.C. December 1978, pp. 317-323.
16. Denning, Peter J. "Virtual Memory." *Computing Surveys*, 2 (1970), pp. 153-189.
17. Jensen, Kathleen, and Niklaus Wirth. *Pascal User Manual and Report*. New York: Springer-Verlag, 1974, p. 30.
18. Sutherland, Ivan E., and Carver A. Mead. "Microelectronics and Computer Science." *Scientific American*, 237 (1977), pp. 210-228.
19. Mead, Carver, and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.
20. Osborne, Adam. *An Introduction to Microcomputers*, Vol. II. Berkeley, California: Adam Osborne and Associates, 1976.
21. *Am 9511 Specification Sheet*, Advanced Micro Devices, 1977.
22. *MCS-86 User's Handbook*, Intel Corporation, October 1979.
23. McKeeman, W. M. "Language Directed Computer Design." *AFIPS FJCC Conf. Proc.*, 31 (1967), pp. 413-417.
24. Lawson, Harold W., Jr. "Programming Language-Oriented Instruction Streams." *IEEE Trans. Comput.*, C-17 (1968), pp. 476-485.
25. McFarland, Clay. "A Language-Oriented Computer Design." *AFIPS FJCC Conf. Proc.*, 37 (1970), pp. 629-640.
26. Chu, Yaohan, ed. *High-Level Language Computer Architecture*. New York: Academic Press, 1975.
27. Chu, Yaohan. "An LSI Modular Direct-Execution Computer Organization." *Computer*, July 1978, pp. 69-76.
28. Tanenbaum, Andrew S. "Implications of Structured Programming For Machine Architecture." *CACM*, 21 (1978), pp. 237-246.
29. Fadon, Emilio Luque, Lorenzo Moreno Ruiz, and Jose F. Tirado Fernandez. "High-Level Languages Processor Architecture." *Proc. ACM Annual Conf.*, Seattle, Washington, October 1977, pp. 479-483.
30. Battarel, G. J., and R. J. Chevance. "Design of a High-Level Language Machine." *AFIPS NCC Conf. Proc.* (1979), pp. 649-655.
31. Bell, C. Gordon, J. Craig Mudge, and John E. McNamara. *Computer Engineering*. Digital Press, 1978, p. 243.
32. Peuto, Bernard L., and Leonard J. Shustek. "Current Issues in the Architecture of Microprocessors." *IEEE Computer*, 10 (1977), pp. 20-25.
33. Bullman, David M. "Stack Computers: An Introduction." *IEEE Computer*, 10 (1977), pp. 18-28.
34. Blake, Russell P. "Exploring a Stack Architecture." *IEEE Computer* (1977), pp. 18-28.
35. The WD9000 Pascal MICROENGINE microcomputer chip set specification guide, Western Digital, 1978.
36. Bowles, Kenneth L. "UCSD Pascal: A (Nearly) Machine Independent Software System." *Byte*, May 1978, pp. 46, 170-173.
37. Laliotis, Theodore A. "Architecture of the SYMBOL Computer System." In *High-Level Language Computer Architecture*, Yaohan Chu, ed. New York: Academic Press, 1975, pp. 109-185.
38. Wilner, W. T. "Design of the Burroughs B1700." *AFIPS FJCC Conf. Proc.*, 41, pt. 1 (1972), pp. 489-497.
39. Meyers, Glenford J. *Advances in Computer Architecture*. John Wiley and Sons, 1978.
40. Palmer, John, Rafi Nave, Charles Wymore, Robert Koehler, and Charles McMinn. "Making Mainframe Mathematics Accessible to Microcomputers." *Electronics*, 53 (1980), pp. 114-121.
41. Hoevel, Lee W. "'Ideal' Directly Executed Languages: An Analytical Argument for Emulation." *IEEE Transactions on Computers*, C-23 (1974), pp. 759-767.
42. Flynn, Michael J. "Directions and Issues in Architecture and Language." *IEEE Computer*, 13 (1980), pp. 5-22.

A computer-aided VLSI layout system

by W.A. DEES, K.M. PARMAR, A. GOYAL, R.Y. TSUI, B.D. RATHI, and R.J. SMITH, II

University of Texas at Austin
Austin, Texas

ABSTRACT

The VLSI layout system is suggested as a practical approach for solving large and complex problems introduced by today's VLSI technology. Computer-based design aids are introduced which are utilized to effectively reduce design time and to increase product quality. A hierarchical description of VLSI circuits is utilized to partition the problem into manageable tasks. Each phase of the VLSI chip design cycle is discussed with special emphasis on layout techniques. The hierarchical VLSI layout system is applicable to the design of "semi-custom" or master-slice VLSI circuits. The placement and placement optimization portions of the proposed system have been implemented. Routing and routing optimization techniques are currently being developed.

INTRODUCTION

As VLSI chips become increasingly complex, reliability requirements, costs, schedules, and a host of other factors dictate that traditional chip design techniques cannot be expected to deal adequately with new requirements. It is proposed that future chips be designed using tools that promote the orderly management of design complexity, including computer-based design aids that are substantially more capable than those presently in use. This paper reports plans and specifications for a computer-aided layout facility applicable to the design of "semicustom" or master-slice VLSI circuits containing up to several hundred thousand gate equivalents.

We focus on layout-related aspects of the design problem, treating in only a peripheral manner other services and capabilities that would be required to reap maximum benefits from such a design system. The reported VLSI design system is in a preliminary stage. Software development has begun, and relationships between requirements, needs, techniques, and priorities are evolving rapidly. This plan defines what the authors believe to be an effective approach to the layout of VLSI chips. Though much remains to be resolved, the underlying approaches and design philosophies are likely to be retained in a system that evolves from this early work.

We begin with an introduction to hierarchical VLSI design methods, including a brief discussion of how these approaches

would be used prior to initial layout efforts. A model for VLSI layout is then presented, based on specific computer-aided layout capabilities. Preliminary specifications and designs for each subsystem used in the layout procedures are then developed.

PERSPECTIVES ON VLSI DESIGN

It is apparent that trends in IC fabrication technology are leading to the capability of manufacturing chips that become increasingly complex, at a rate that exceeds our ability to design. Clearly, these advances can be exploited fully only if substantial gains in design productivity can be realized. Indeed, motivation for the facilities described in this plan is derived from the need to rapidly decrease VLSI design costs.

One of the most effective methods for coping with a structurally complex situation is to decompose it in a hierarchical fashion into a sequence of more manageable subproblems. Application of this approach to VLSI chip design is one practical way to deal with the rapidly growing complexity of most chip design phases. VLSI design tasks can be partitioned into manageable subtasks if the interrelationships between subtasks can be managed efficiently.

Hierarchical approaches to chip design must begin during early phases of top-down planning and architecture-level design. This work results in a chip (system) that can be represented as a collection of functionally distinct subsystems along with appropriate interconnections between them. These descriptions may at first be incomplete and extremely tentative; but they represent preliminary partitions for each subsystem, as well as interrelationships between subsystems. If no standard design exists for a particular function, it can in turn be considered a design subproblem. Functional decomposition can be continued through as many design levels as are required to arrive at functional elements that are composed of and expressed completely in terms of basic cells that realize common standard functions. Note that the design level at which this occurs has a substantial impact on overall design costs: the use of relatively high-level, functionally complex standard cells reduces the number of specially designed custom subfunction elements that must be developed for a single project.

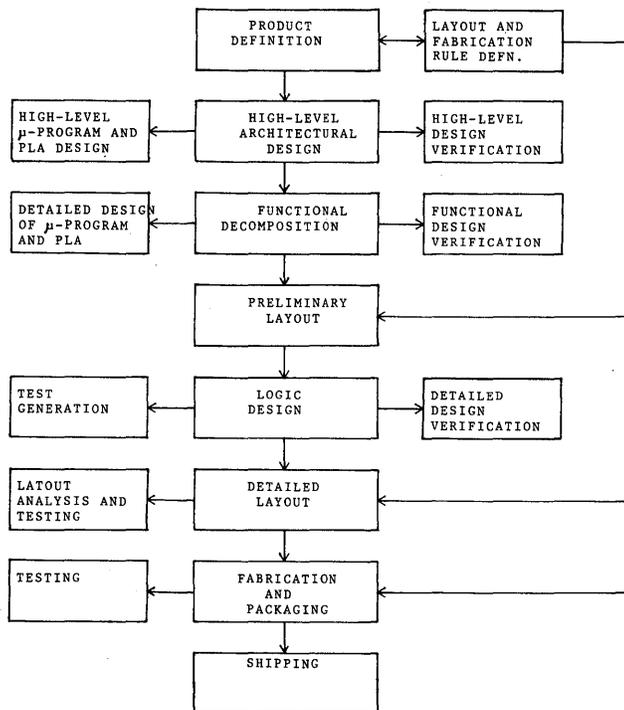


Figure 1—The VLSI design cycle

VLSI DESIGN CYCLE

Consider the sequence of stages through which a new VLSI chip design must pass. These stages are described as distinct, idealized steps, even though we recognize that in practice VLSI designs evolve through iterative repetition of closely coupled sets of stages (shown in Figure 1).

1. Product objectives are defined in terms of capabilities, marketing considerations, and processing technology.
2. Development of high-level structural organization. Preliminary instruction sets, registers, data and control paths, and other features that guide detailed design are defined here. Subsequently, high-level design verification is performed. Modern VLSI designs exploit micro-programmed control and PLA replacements for random logic. PLA and ROM contents are described here at an abstract level.
3. Decomposition of the high-level design into functionally distinct elements. Partitions are defined so that the amount of information transferred between functional elements is minimized, making each functional element a distinct design subproblem that can be individually verified for design correctness. Concurrently, the micro-programmed control and PLA designs are detailed.
4. Preliminary layout of each functional element involves allocation of areas and shapes on the chip, based on detailed layout and fabrication rules dependent on technology.
5. Transformation of each functional element into large collections of interconnected logic elements. Design verification at this stage may consist of development of a

physical simulator (hardware prototype) or extensive analysis using high-resolution simulators, or both; the objective here is to verify, at the greatest level of detail practical, that the design satisfies all applicable criteria. Generation of test patterns for fault detection is easily adapted to this stage of the VLSI design cycle, as is the verification of the microcode.

6. Placement and interconnection of functional elements. Detailed layout analysis, verification, and testing are performed to insure that the physical design (masks) accurately portrays the logic designs previously subject to careful scrutiny.

HIERARCHICAL DESIGN METHOD

Hierarchical decomposition may be applied to most large-chip design tasks. However, in this paper we are most concerned with providing layout-related design services, so let us focus on those aspects of the overall problem. The proposed computer-aided layout system is a collection of software tools that aid the IC designer in dealing with the chip in a hierarchical manner. The capabilities and services provided will allow future VLSI chip designs to be achieved with short turn-around time, in a cost-effective manner.

The hierarchical design method consists of top-down circuit partitioning followed by a bottom-up circuit layout.¹ This top-down design procedure insures that before a detailed layout of bottom-level element is started, a good estimate of the size and shape of the higher-level element is known. The hierarchical approach proposed here can greatly simplify placement and interconnection problems because of the relatively small number of elements and interconnections to be considered at each level of the hierarchy.

MODELS FOR VLSI LAYOUT

The implementation of the hierarchical structure in the design of an IC chip is accomplished with the use of a structure tree. The highest level of the tree is the chip, and the branches are its constituent cells. Different levels in the tree correspond to different levels of the hierarchy. At each level of the hierarchy, connecting constituent cells of a function requires the internal descriptive information of the functional cell, which is provided by an internal cell model. Placement and routing are done for the constituent cells of a function whose external cell model will be updated. The latter is required for interconnection with other constituent cells of a higher-level functional cell.

External Cell Model¹

The external cell model is used to define the external behavior of a functional or generic cell. This is in turn a detailed description of a set of placed and routed cells, viewed from outside the boundary of the set of cells. At the lowest level of the design hierarchy for an IC chip the generic cell or basic cell

from the cell library can be defined by its external model, which is all that need be known for use in the layout of higher levels. An external cell model for use in the hierarchical layout structure includes the following:

- **External cell ID/NAME.** Cell part number, generic cell type, dates of design and revision, designer's name, version number, technology and associated wiring rules, pointers to data structures for the functional cell, and boundary and I/O descriptions.
- **Functional cell description.** Functional behavior (such as a flip-flop, RAM, or ALU) is recorded to aid in selection of constituent cells for functional design during the design process.
- **Cell specifications and wiring rules.** Cell specifications parameterize the cell in terms of impedance, propagation delay, fanout, and total power. Wiring rules govern electrical and physical parameters by controlling the physical realization of cells.
- **Cell boundary description.** Defined in terms of area, size, shape (form factor), and number of layers.
- **Cell I/O description.** Defines and locates all nodes on the cell, including inputs, outputs, power, clocks, etc. Locations of these nodes may not be fixed in the generic cell description, thus permitting both interlayer and intralayer node float. Such float permits flexibility in routing and is finally reduced to zero by the router.

Internal Cell Model¹

An internal cell model is required for the layout of its constituent cells. During top-down design on an IC chip, the sizes of the cells at each level in the hierarchy are not exact, as their constituent cells are not precisely defined at the lower levels. As the design proceeds to lower levels, more accurate estimates of sizes of cells at higher design levels can be made. During bottom-up implementation, cell sizes at higher levels in the hierarchy are known, as their constituent cells have been completely designed. An internal cell model must include the following information:

- **Cell block ID/name.** A unique user-defined name or ID.
- **Constituent cell list.** All constituent cells at the next lower level of the hierarchy. Each constituent is identified by its unique ID/name. Note that a description of each cell can be obtained from its external cell model.
- **Net list.** All nets needed to interconnect constituent cells to form the new functional cell at the next higher level of hierarchy. For each net, there is a list of nodes belonging to that net.
- **Wiring rules.** Net routing is governed by a set of wiring rules, such as maximum total length of the net, maximum conductor width, and capacitance and parallelism limits.
- **External physical and electrical characteristics.** Describes the completed design. Includes external boundary imposed by the wiring rules, such as the separation between the constituent cell and the functional cell boundary and the geometries of the actual cell within the functional cell. External electrical information about the cell in-

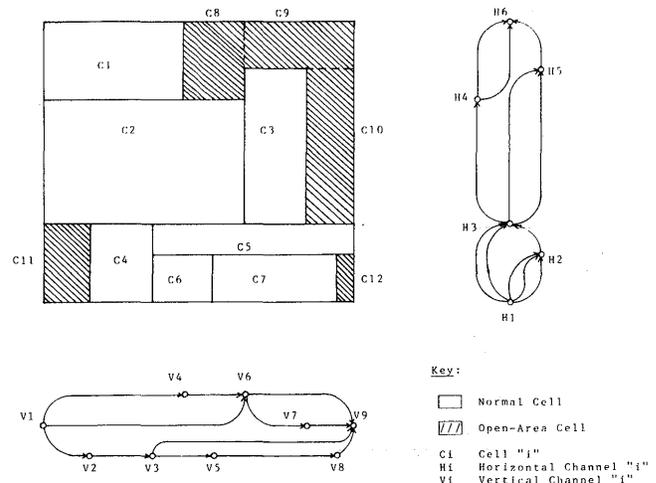


Figure 2—A typical placement of cells and the corresponding polar graphs

cludes total power dissipated within the cell, the strength of cell external nodes, etc.

Polar Graph Model^{1, 10}

To facilitate placement optimization and interconnection routing, a layout is represented by a pair of mutually dual graphs $G_x(V_x, E_x)$ and $G_y(V_y, E_y)$, where G_x and G_y are planar, acyclic directed graphs containing one source and one sink. Each pair of edges (e_x^i, e_y^i) represents a rectangle with X -dimension $l(e_x^i)$ and Y -dimension $l(e_y^i)$ where $l(e)$ denotes the length associated with edge e . Since a cell is modeled as a rectangular object and there exists a one-to-one correspondence between the edges of G_x and G_y , a pair of edges (e_x^i, e_y^i) represents cell i . Parallel edges are allowed in the dual graphs. Therefore, a vertex in the horizontal polar graph represents a vertical channel between cells that are represented by edges incident to and departing that vertex. Edges incident to a vertex on the horizontal polar graph represent cells which lie to the left of the vertical channel, while those edges departing a vertex on the same graph represent cells lying to the right of the vertical channel (see Figure 2).

SYSTEM ORGANIZATION

The support system nucleus and design file are two of the major components accessed by the user. To provide error checking and an orderly supervision and management of revision, these components are accessed only via database management utilities, as represented by the enclosing dashed lines. All major components of the VLSI design system are illustrated in Figure 3. Descriptions of each major component are subsequently provided.

The system centers on a common database, which is needed to tie all the elements of the design together. This is the nucleus of the system, which includes a cell reference library, design files of previous designs, and the support software necessary to access, update, and manage these files conveniently.

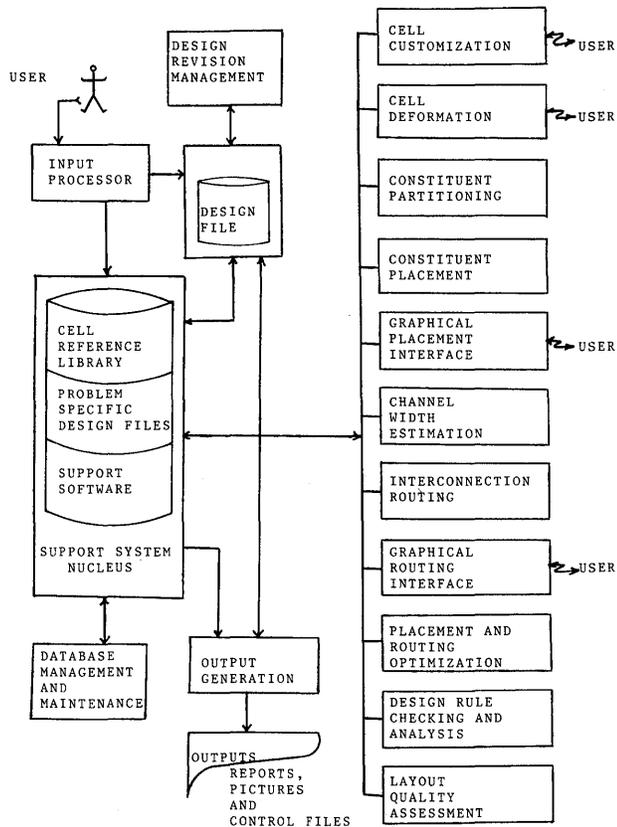


Figure 3—Computer-aided layout system structures

The support system nucleus enables all design-related information to be centralized in a common database that can be conveniently accessed, updated, etc., by other subsystems through the use of database access utilities.

Support System Nucleus¹

A collection of data, utility, and file access modules used as the foundation of the layout system provides the necessary CAD database management support. An input processor is defined to insure proper entry of data into the system database and the correct specification of a layout problem. An electrical and layout constraint file for each technology is provided in the database. The cell reference library contains detailed descriptions of the generic cells currently available, including standard supported cells, as well as those used by specific chip designs. Cells for a particular chip will be selected by their internal and external characteristics. If a generic cell does not exactly match the requirements, it is processed by the cell deformation and cell customizer modules and brought up to required standards. Of course, the designer has a great deal of control over this process. Certain functional elements that are frequently used in design with different shapes and characteristics are maintained as distinct cells in the cell library. As the number of varieties of a given functional cell increases, the system becomes more flexible and powerful.

All cells are given a generic name and the layout problem

description consists of the various constraints to be placed on the layout, the identification of the cells used, and their interconnection details. Should a cell in the design not be available in the cell library, its description is input to the system as a separate design subproblem and is appended to the cell library. Each layout problem description is maintained in a separate design problem file in the design library.

Input Processor

Verification of new cell descriptions being input to the cell library and cross-checking of the design problem files is performed by this module. Upon verification of an error-free design problem file, the required external and internal cell descriptions are obtained from the cell library.

Identification of the cells required in the design is performed during the top-down phase of design. As they become known, nets are identified and described in terms of interconnections to be performed, and estimates of the sizes of cells in higher levels of the hierarchy are refined.

Two modes of data entry are supported. In the interactive mode the layout being specified is verified when requested by the user. In batch mode, the user prepares a description in the form of a batch file, which is subsequently checked for validity and converted into a problem design file.

Design Decisions and Design Verification

Successful implementation of a complex VLSI circuit depends on more than correctly placing and interconnecting the cells making up the proposed circuit. The operational, functional, and electrical parameters specified by the designers must be satisfied by the final layout.

Many design decisions are related to the distribution of power, ground, and high-performance signals. Processing of the necessary data to optimally distribute these signals and accounting for the power dissipation is performed here. Clock and other high-frequency signals are checked, and necessary load and source impedances at critical nodes are calculated. Other design parameters verified here include the characteristics of the input and output circuits, voltages, and drive currents.

The final layout may also be processed here, and design constraint violations may be reported to the user.

Output Generation

The output generation includes extraction of information from database files and subsequent preparation of tabular reports, graphs, manufacturing files, and plots. Facilities for inspecting the layout at any stage of the design process are also provided here. Although emphasis is placed on automatic design, interactive design facilities to improve or suggest alternatives are nevertheless required. With these facilities, various elements in the database or design problem are displayed individually or in a specified combination.

Constituent Cell Placement

The objective of placement is to assign positions and orientations to cells and pads so that overall chip size is minimized. The program supports arbitrary-size rectangular cells and a mosaic layout, as opposed to the polycell approach, increasing flexibility and improving silicon use. Placement is divided into two parts, initial placement and placement optimization. Initial placement uses the mincut algorithm² to reduce cutline crossings and wire length as much as possible; placement optimization improves routability without changing relative positions of the placed cells.

This VLSI placement technique is based on an earlier paper by Lauther.¹⁰ However, substantial alterations have been made to achieve placements closer to manual ones for real-world VLSI placement problems. The following subsections briefly describe the improved placement strategies.

Initial Placement

Initial placement uses an envelope of specified shape whose area equals the sum of areas of all cells being placed and partitions it recursively until each partition contains exactly one cell. The partitions are based on the area of cells; that is, each final partition has an area equal to the area of the cell it contains, but it may not have the same dimensions. Thus, when partitions are converted into actual cells, dead area may be generated. Various techniques are used to reduce this dead area, as discussed.

A strategy administrator supervises initial placement, selecting positions and the directions for each cutline. A major advantage of this approach is the ability to match the procedure to the characteristics of the logic design. A partition imposed earlier in the sequence will have fewer signals crossing it than one imposed toward the end of the sequence. Clearly, then, the sequence of partitions greatly influences wiring densities. The quadrature placement with breadth first cuts² has currently been implemented; each cutline divides the current block into two partitions of approximately equal area. Options for performing slice cuts are available.^{4,10} However, they do not appear to be as effective, since peripheral cells may be placed on all of the four sides of the chip. The strategy administrator may optionally select cutlines manually, switch alternately between horizontal and vertical cutlines, or use an automatic procedure based on cell areas and shapes to select horizontal or vertical cutlines for a particular partition.

The mincut algorithm implemented attempts to put maximally connected cells and the cells belonging to the same affinity classes in the same partitions. An affinity class is represented as a set of cells belonging to a pseudonet with a large weight associated with it for crossing partition boundaries.

Constructive initial placement selects cells one at a time for placement in one of the two partitions created by a cutline. This strategy guarantees a partition satisfying area and peripheral conditions, locally minimizing the number of nets crossing the cutline.

Iterative improvement optimizes this placement by minimizing the cost on all nets cut by a cutline. Iterative improvement involves the seven steps shown:

1. Determine the gain (the reduction in number of connections cut by a cutline) for every single interchange of a pair of cells across the cutline.
2. The pair of cells that produce the maximum gain when interchanged are repositioned and marked non-interchangeable for subsequent passes of the iteration. If both area and peripheral conditions are satisfied after the interchange, a value 1 is generated for COND; otherwise it is 0.
3. Steps 1 and 2 are a pass. They are repeated until no further interchanges are possible. A sequence of incremental gains with COND is thus generated.
4. The total gain over the initial state is computed for the progression of passes.
5. The sequence of interchanges that produce the maximum total gain is determined by noting the pass where the maximum gain occurs with COND equal to 1.
6. All interchanges subsequent to the pass of maximum total gain are restored to their initial positions.
7. Steps 1 through 6 are iteration. They are repeated until no further gain occurs.

At the end of initial placement each partition contains exactly one cell. To introduce the actual cell dimensions, the arc lengths in the dual polar graphs are replaced by the cell dimensions, using a simple algorithm.¹⁰ Then for each cell the position of its lower left corner is calculated and recorded.

PLACEMENT OPTIMIZATION

The placement optimization subsystem can be used either through automated or interactive modes at various levels of the design hierarchy to reduce the silicon area of the problem. Overall chip size is reduced by both removing excess area introduced by the initial placement and reducing the estimated interconnection length of the problem. Interconnection length reductions result in saving silicon area and in reduced impedance and capacitance of the interconnected traces. A secondary goal of this subsystem is to modify the shape factor. Shape factor improvement is allowed only after area constraints have been met.

The techniques used for placement optimization are

1. Cell rotation
2. Channel squeezing
3. Abutment class dead-area use
4. Cell reflection
5. Cell deformation (or reshaping)

Cell rotation is an operation to reorient the cell with respect to the problem origin. The relative position of the cell to its neighboring cells remains the same. The operation is used both to reduce overall problem dimensions and to reduce interconnection length. Cells are classified into two sets, of which one contains all cells that are located on critical subgraphs and the other contains all cells excluded from the first set. Rotation candidates for area reduction are selected from the critical cell set; candidates for decreasing interconnection length are selected from the noncritical cell set. Cell rotation

is restricted to 90-degree counterclockwise increments, allowing only orthogonal movements.

Channel squeezing is a localized placement adjustment that modifies the incidence relationship for the modeled channels. Neighborhood relationships between cells are modified, but the general location of the cell within the problem envelope remains the same. Squeezing trials can be done on the IC at any hierarchical level for reducing problem dimensions. Only channels that lie on critical-subgraphs are candidates for squeezing optimization.

Cells which must be placed abutting one another are placed in an abutment class having a nonrectangular shape. An enclosing rectangle around the abutment class contains both the cells making up the abutment and dead silicon area. This area may be used if the abutment class neighboring relationship does not change. Cells incident to the abutment class boundary channel are candidates for area use. Selected cells are then placed in the interior of the abutment class, consuming the open area.

Cell reflection is a technique for reducing wire length that has no impact on previously placed cells. Features internal to the cell are reflected or mirrored either around their X -axis center line, Y -axis center line, or both center lines. Optimal reflection orientation is determined by evaluating changes in the minimum spanning tree length calculations over all nodes assigned to nets. The spanning tree calculation excludes nodes within the same net that belong to the cell being reflected.

Cell deformation allows the shape of a cell to be manipulated to suit the topology in the locality of the cell. Reshaping would allow the cell to be contracted or elongated along the X or Y axis. The resulting shape of the cell must be rectangular. The cell deformation technique is used both to reduce problem envelope dimensions and to reduce wire length if two adjacent cells have nodes connected to each other. Reshaping can be manual or automated, depending on problem constraints. Manual deformation allows the designer using interactive graphic tools to appropriately reshape cells. Automatic reshaping is the substitution of functional and electrical equivalent cells that have different shape factors.

The reduction of area is an iterative process in which its operations must have inverse functions. A minor placement optimization iteration is the selection of a trial operation, scoring the results, and either accepting or rejecting the trial. When a trial optimization operation fails to decrease area, then the inverse function is applied to restore the placement model to its previous state. Placement optimization is partitioned into two phases. The first phase attempts to reduce problem envelope dimensions; the second phase modifies placement, without increasing envelope dimensions, to reduce wire length of the problem.

Placement optimization is a major iteration within the design process. After the optimization system returns, the strategy administrator evaluates the modified polar graph. The new placement is scored against the best-known placement up to that point. If the new placement is better, then the cell locations and orientations are saved. The strategy administrator then has the option of reentering the placement optimization system with the previous polar graph model, hoping for more enhancements, or to reenter with the original placement, but with different processing option values.

INTERCONNECTION ROUTING

The routing subsystem is used at all design levels during the bottom-up phase. Its main task is to provide 100% interconnection, using the least amount of silicon area consistent with design and wiring rules.

Various methods for interconnection routing have been used in the past for chip layout. These may be conveniently subdivided into several classes, including serial approaches similar to those proposed by Lee¹¹ and Hightower.⁷ The major drawback of these approaches is the totally serial nature in which they attack the interconnection problem, causing larger numbers of routing failures as problem complexity increases. Hashimoto and Stevens⁶ first introduced the idea of channel routers to alleviate some of the drawbacks of a totally serial approach, thus providing much greater flexibility in the routing of large numbers of nets. These channel routing methods have recently been applied to VLSI layout, for example, in Hightower and Boyd.¹⁷

Channel routing is a two-phase routing strategy consisting of a channel Assignment phase and a track Assignment phase. During channel assignment, nets are assigned to a sequence of channels or to open wiring areas between constituent elements, but not to tracks within the channels. Track assignment assigns all nets in a channel to specific tracks to permit 100% wiring. Routing is completed when track assignment for all channels has been (successfully) completed.

Channel Definition and Assignment

During placement, cells are assigned to areas without regard to channels required for completing the required interconnections, thus abutting cells one to another, as illustrated in Figure 2. The boundary between two abutting cells represents a potential channel, represented by a vertex in the polar graphs. When such a placement is performed, a great deal of open silicon area is left unused. These areas are represented as special cells in the polar graph and are used for interconnection routing at the current level in the hierarchy or for placement and/or routing at higher levels.

The locations of channels are computed from the polar graphs, and their dimensions are estimated by the routing area estimator module prior to channel assignment. Nets at the current level in the hierarchy are decomposed into point-to-point interconnections, using a minimum spanning tree algorithm, and the from-tos created are assigned to a sequence of channels to complete the interconnections. Capacitance and power loss estimates are verified against the wiring rules imposed.

*Track Assignment**

Once channel assignment is complete, the track assignment phase begins, assigning all from-tos that use a channel to spe-

* This phase is currently being programmed by one of the authors.

cific tracks within the channel. There is, however, an additional problem that must be dealt with.

When a from-to enters a channel at its periphery, the exact location of the point of entry is unknown—or rather, the permissible region of entry is known. These regions (end-float regions) may overlap with end-floats of other from-tos that use the channel, further complicating the task.

A channel, with its associated from-tos, may be represented as a graph, with a node on the graph representing the end-float region and an edge representing a transition from the *from* node to the *to* node. It is obvious that in the case of single-layer routing, this graph must be planar. Nonplanarities in the graph are very easily handled in multilayer routing situations where the channel graph for each routing layer must be planar.

Any of the techniques currently used for routing may be used to perform track assignment. Careful evaluation has, however, indicated that the Lee type algorithm, with modifications to allow parallel contention resolution, is best suited for this task.

ROUTING OPTIMIZATION

Connectivity improvement is introduced, since the channel router might fail to route all the necessary point-to-point connections. This module is invoked after the channel router terminates, but before die dimensions are increased to support the necessary conditions. Connectivity improvement is a two-phase process, where the first phase is the application of a Lee type of interconnection algorithm. The second phase consists of rip-up, reroute, and shove-aside techniques, which rearrange the interconnection structure to enhance the routability of the unconnected nets. Both phases are invoked to increase completion rates without increasing overall die dimensions.

After connectivity improvement, interconnections can still be left unconnected as a result of conflicts with previously routed features or failure to meet electrical or timing constraints. Problem dimensions are then increased to support the unconnected nets. After die expansion all critical channels are evaluated for a final area reduction, using channel compression. The channel compression operation attempts to modify routed channel features to reduce as much unnecessary space as possible. If a channel is compressed, another critical channel candidate list is generated. From this list channels are selected for compression.

QUALITY ASSESSMENT

The quality of the placement and routing procedures is assessed by evaluating metrics dealing with software behavior and problem results. The evaluation of placement and routing is based on attributes reflecting the quality of the system. These attributes include execution time, completion rates, silicon area use, signal distribution, and channel use. The software behavior quality assessment report is useful when considering the system's strengths and weaknesses, where weaknesses are identified as areas for future enhancements.

The quality of chip design is assessed after placement and routing by noting the difference between the design goal and the actual results in such areas as physical, electrical, and logical characteristics. The initial problem description, modeled with a register transfer language, is simulated for logic errors. Errors detected and corrected early will reduce the number of automated design iterations. Detailed timing simulation is also necessary for evaluating signal propagation delays. Register transfer and behavior simulation are normally deficient in such detailed and accurate timing simulations. Physical characteristics such as total die size and shape factor are calculated to verify that physical constraints are not violated.

Electrical simulation and design checkers are used to guarantee that the problem results lie within the limits imposed by the designer. Electrical simulation estimates voltage, current, and power dissipation so that technological constraints are not violated. Design checkers are also used throughout the design process during and after human interaction. These checkers verify that wiring and technological rules are not violated during interactive sessions.

INTERACTIVE GRAPHICS

Until very recently the role of interactive graphics was limited to the display and manipulation of digitized manual layouts. As the complexity of new chips grows, this approach will no longer be practical: graphics must be used in a more cost-effective manner, and digitizing manual layouts will decline in popularity. There will always be a demand for manual layout of special-case designs. However, as design pressure increases, it is likely that a substantial number of new designs will be generated, using a combination of automated and graphical methods.

At any point during or after the placement or routing of a chip or constituent cell, graphical tools can be used to modify the evolving layout. With both automated and interactive tools available, it is practical to use each method under circumstances best suited to the situation at hand.

CONCLUSIONS

This paper has described a system for the automated layout of semicustom and gate array VLSI. The software required is substantial: more than 160,000 lines of programs, documentation, and related materials. Portions of the system outlined have already been programmed, and others are presently being developed. Until these and similar design tools come into productive daily use, the development of VLSI layouts will remain a time-consuming and costly task.

REFERENCES

1. "Computer-aided MOS VLSI Layout System." Electrical Engineering Department, The University of Texas at Austin, February 1980. (Prepared under the direction of R. J. Smith, II, by D. LaPlante, R. Tsui, W. Dees, W. Rogers, H. Bryce, B. D. Rathi, K. Parmar, T. Gunter, and C. Hobbs.)

2. Breuer, Melvin A. "A Class of Min-Cut Placement Algorithms." *Proc. 14th Design Automation Conference*, June 1977, pp. 284-290.
3. Lynn, Conway, and Carver Mead. *Introduction to VLSI Systems*. California: Addison-Wesley, 1979.
4. Corrigan, Lorretta I., "A Placement Capability Based on Partitioning." *Proc. 16th Design Automation Conference*, June 1979, pp. 406-413.
5. Deutsch, David N. "A 'Dogleg' Channel Router." *Proc. 13th Design Automation Conference*, June 1976, pp. 425-433.
6. Hashimoto, Akhiro, and James Stevens. "Wire Routing by Optimizing Channel Assignment Within Large Apertures." *Proc. 8th Design Automation Workshop*, June 1971, pp. 155-169.
7. Hightower, David W. "A Solution to Line-Routing Problems on the Continuous Plane." *Proc. 6th Design Automation Workshop*, June 1969, pp. 1-24.
8. Klomp, J. G. M. "CAD for LSI—Production's Interest Is in Its Economics." *ACM SIGDA Newsletter*, 6 (1976), pp. 11-15.
9. Koller, Konrad W., and Ulrich Lauther. "The Siemens-AVESTA-System for Computer-Aided Design of MOS-Standard Cell Circuits." *Proc. 14th Design Automation Conference*, June 1977, pp. 153-157.
10. Lauther, Ulrich. "A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation." *Proc. 16th Design Automation Conference*, June 1979, pp. 1-10.
11. Lee, C. Y. "An Algorithm for Path Connections and its Applications." *IRE Transactions on Electronic Computers*, September 1961, pp. 346-365.
12. Losleben, Paul, and Kathryn Thompson. "Topological Analysis for VLSI Circuits." *Proc. 16th Design Automation Conference*, June 1979, pp. 461-473.
13. Oakes, M. F. "The Complete VLSI Design System." *Proc. 16th Design Automation Conference*, June 1979, pp. 452-460.
14. Persky, G., Deutsch, D. N., and D. G. Schweikert. "LTX—A System for the Directed Automatic Design of VLSI Circuits." *Proc. 13th Design Automation Conference*, June 1976, pp. 399-407.
15. Preas, B. T., and C. W. Gwyn. "Methods for Hierarchical Automated Layout of Custom LSI Circuit Masks." *Proc. 15th Design Automation Conference*, June 1978, pp. 206-212.
16. Preas, B. T., and W. M. vanCleemput. "Placement Algorithms for Arbitrary Shaped Blocks." *Proc. 16th Design Automation Conference*, June 1979, pp. 474-480.
17. Hightower, D. W., and R. L. Boyd. "A Generalized Channel Router." *Proc. 17th Design Automation Conference*, June 1980, pp. 12-21.

A multiprocessor description language

by WILLIAM T. OVERMAN,
STEPHEN D. CROCKER, and
VITTAL KINI

USC/Information Sciences Institute
Marina del Rey, California

ABSTRACT

A language for describing multiprocessor systems is presented. The language, called MPDL, provides a flexible and unambiguous model of concurrency and allows for hierarchical construction of concurrent systems. MPDL encourages the user to encapsulate interprocess synchronization and communication in a special component called a connector. This encapsulation helps facilitate multilevel modeling and abstraction of communication protocols. A simulator for the language has been implemented and is running at ISI. This paper describes MPDL and evaluates the language in terms of two examples.

INTRODUCTION

The language and simulation system reported on here were developed as part of the Multimicroprocessor Emulation project at USC/Information Sciences Institute.* The goal of this project was to develop language and tools for exploring multiprocessor architectures, with specific emphasis on the support of closely coupled architectures and a wide variety of processor interconnection schemes.

As a part of this effort we have designed a language for describing the structure and behavior of networks of processors. Along with this we have implemented an Interlisp-based simulator with extensive debugging facilities to aid in the development of multiprocessor descriptions.

Our original intention was to use an existing hardware description language, such as ISPS¹ or SMITE², with extensions, to describe multiprocessor systems. However, we found that these languages were not adequate for expressing the necessary interconnections and interactions among processors.³ Therefore we developed a new language, called MPDL (Multiprocessor Description Language), which embodies a clean and flexible model of concurrency and provides a hierarchical interconnection language, but otherwise uses ISPS

* This work was sponsored in part by the Rome Air Development Center and in part by the Defense Mapping Agency, both under contract DAHC15-72-C-0308.

constructs for describing the sequential behavior of individual processors.

The next section introduces MPDL by walking through an example and then describing the major components of the language. The section following that briefly indicates the capabilities of the simulation system we have built for MPDL. The fourth section discusses two examples that have been described in MPDL and run on the simulator and evaluates the language with respect to these examples.

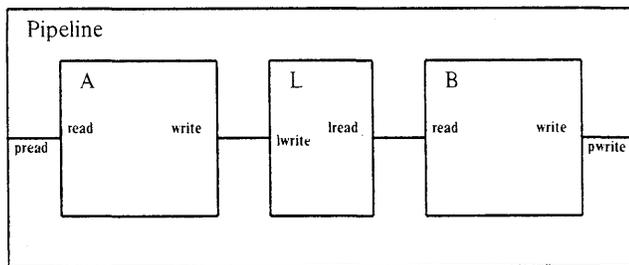
LANGUAGE DESIGN

The following are the key ideas in MPDL:

- Communication and synchronization among processors occur through shared variables. Wait and delay constructs are provided as abstractions.
- Shared variable access is restricted, and all shared variables and their accessing functions reside in special components called connectors that link processors together. This encapsulation of shared variables permits abstraction of communication protocols.
- The writer has control over the granularity of actions. Mutual exclusion and synchronization must be made explicit.
- Hierarchical description capabilities are provided. Modules may be combined to form new modules.

The example in Figure 1 illustrates these ideas. The following text describes the example.

Pipeline is a composite module composed of two processors, *A* and *B*, which communicate through the connector *L*. *Pipeline* is described as being a composite with two ports called *pread* and *pwrite*, the first returning an eight-bit value and the second returning no value. It contains a connector, which is to be called *L*, and its description is to be an instance of the connector *HandshakeLink*, which can be found in the library (*HandshakeLink* is described below). It also contains two instances of the processor *Handshake*, which are to be known as *A* and *B* within this composite. Processor *A* has two ports, one linked to *pread*, the port which was passed into



```

composite Pipeline (pread <7:0> , pwrite)
  connectors L = HandshakeLink;
  processors A = Handshake (pread, L.lwrite),
             B = Handshake (L.lread, pwrite)
endcomp;
processor Handshake (read <7:0> , write)
  repeat
    call write(read())           !read data and pass them on
  endrep                          !the processor never terminates
endprocessor;

connector HandshakeLink
  declare data <7:0> , sig <> init 0;           !buffer and handshake signal
  proc lread <7:0>
    wait sig eq 1 endwait;                     !read entry
    sig←0;                                     !wait until data are ready
    return data                                !signal "buffer empty"
  endproc;
  proc lwrite (d <7:0>)
    wait sig eq 0 endwait;                     !write entry
    data←d;                                    !wait until buffer is empty
    sig←1                                     !signal "data ready"
  endproc
endconn

```

Figure 1—A two-processor pipeline system using a handshake protocol

Pipeline, and the other to *lwrite*, an entry in the connector *L* (specified by the qualified name *L.lwrite*). Processor *B* is connected to the *lread* entry of *L* and to the *pwrite* port of *Pipeline*. The port connections are actually access paths through which procedures will be invoked (see below).

Processor *Handshake*, of which *A* and *B* are copies, is described following the composite. The processor has two ports named *read* and *write*. The action of the processor is simply to loop forever, calling the procedure bound to the *read* port and then calling the *write* port with the value returned by *read*. By calling the port, the processor is executing the procedure in the context of the connector. Variables can be shared among processors by having the processors linked to entries in the same connector.

The connector linking the processors is called *Hand-Link* in the library and is described in the connector statement. The connector has two variables declared in it, *data* and *sig*, which can be accessed by each of the procedures in the connector. The connector has no executable body of its own but has procedure definitions which are exported to be bound to the ports of processors. In this case we have the procedures (also called *entries* of the connector) *lread* and *lwrite*.

The behavior of the *Pipeline* module is to simply pass data from its *pread* port to its *pwrite* port with some buffering in the middle so that it can read and write data at differing rates. The nature of the language is that all processors in the system

run in parallel. If we think of the locus of control as being a token, then each processor starts out with one token, and the connectors have no tokens. When a processor calls a port, control passes to the entry in the connector to which that port has been bound. Since many different processors may be linked to a connector, it may turn out that a connector has more than one token in it. However, we require that at most one processor be connected to any one entry in the connector so executions within the connector should be at unique locations.

When processor *A* is started up, it calls the *pread* port of *Pipeline* (bound through *A*'s *read* port). We do not know here what *pread* is bound to, but assume it returns with a value. *A* then calls the *write* port, which is actually the *lwrite* entry in *L*. The action there is to wait for *sig* to be zero (it is initially zero, so execution continues). Then the shared variable *data* is set to the value passed to *lwrite* (the value returned from *read*); *sig*, which serves as a synchronization variable, is then set to one, indicating that there are data available to be read. The next time *A* enters *lwrite*, the wait statement will cause it to suspend and allow *B* to run. *B* is linked to the *lread* entry of *L*, so it checks that *sig* is one, resets *sig* indicating that it has read the data and *data* can be filled with a new value, and then returns *data*. Upon return back into processor *B*, the *pwrite* port of *Pipeline* is called, and we assume it is written out somewhere. Processor *B* then calls *read* (bound to *lread*) again and will be suspended when it encounters the wait statement.

All of the activity between wait statements is uninterruptible; so in this example, each processor has only one interaction point—at the wait statements in the connector entries.

Language Definition

The MPDL User's Manual⁴ gives a complete definition of the language. This section describes a very small number of statements, namely those representing the major structural components of a description (processors, connectors, and composite modules) and that illustrating the model of concurrency embodied in MPDL (the combination wait/delay statement).

Processor

Each processor in a system description is an independently running machine. As we have seen, the processors communicate with each other through shared variables housed in connectors. The processor statement looks very much like a normal function or procedure definition, having internal declarations and a statement list, but with one difference. The "formal argument" list in a processor statement identifies a set of ports that must be connected when the processor is actually used. The formal names are used within the processor as ordinary functions. However, when one of these ports is called, control passes out of the processor and into the connector to which the port is bound. Control returns to the processor in what looks like the normal subroutine return mechanism, and execution continues in the processor. (The

description of the connector, below, describes what happens within the connector). Ports are bound to connectors using the composite statement (see below).

Connector

Connectors are similar to devices found in software engineering such as Clusters⁵, Simula classes⁶, and Parnas modules⁷. They differ from Monitors⁸ in that they impose no synchronization. The connector encapsulates variables that are shared among processors, along with the accessing functions for those variables. One can think of portions of the connector code as actually being part of the processor that is bound to it, and can think that the connector simply provides a mechanism to isolate that portion of a processor's description responsible for communicating with other processors. This tends to help identify the interprocess communication protocol whose correct operation is essential to the operation of the system as a whole.

Connectors also make it possible to do multilevel modeling and abstraction of processor behavior and interprocess communication. We can change the level of detail within a processor and change its communication interface simply by modifying the connector so that it handles the change in protocol; we do not necessarily have to change other processors.

The connector statement specifies a set of declarations representing variables internal to the connector and a set of procedures whose entry names are to be exported to be bound to the ports of processors. These procedures have access to the variables within the connector, and thus the variables are shared among the processors.

Connectors are passive components and are activated only when a processor calls one of the entry points in the connector. Because multiple processors are connected to a connector, and different processors may each call entries within the connector at the same time, it may be the case that there will be many statements within the connector executing at the same time. Note, however, that if execution locations are thought of as tokens, then the number of tokens in the system is conserved, and there are always exactly as many tokens as there are processors in the system.

Composite

The composite statement allows one to assemble collections of processors, connectors, and other composite modules. Connector entries are bound to processor ports to create a complete system. The composite statement describes an entity called a composite module. This can be treated exactly as a processor in future composition steps, thus facilitating hierarchical description.

The composite statement lists the connectors to be included in a composite module and gives them local names, if necessary, to distinguish multiple copies of the same connector. Similarly, a set of processors (and/or composite modules) is included and possibly renamed. The composite entity being defined has a list of formal arguments identical to the formal arguments in a processor statement in that they represent

ports to be bound to the composite module. The processors and composite modules that are to compose the new composite are bound to connectors by specifying a connector entry to be bound to each port in the processor/composite. One exception is that the port of a processor/composite may be mapped to one of the formal arguments of the composite module being defined. These latter ports will be bound at a later time, when the newly defined composite module is composed with other modules.

WaitStmt

The writer uses the wait statement to specify the granularity of the actions in each processor. This statement can be thought of as a call to the scheduler that allows other processors to run, and all of the actions between wait statements are considered to be uninterruptible. The wait statement combines busy waiting on an expression, delay for a specified time, and waiting with an associated timeout. The following example illustrates the syntax and optional components of the statement.

```
wait
  a gt 1: a←0; b←0,
  b gt 1: b←1,
  c gt 1,
  delay d+e: d←0
endwait
```

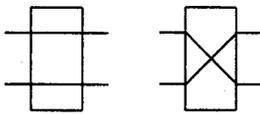
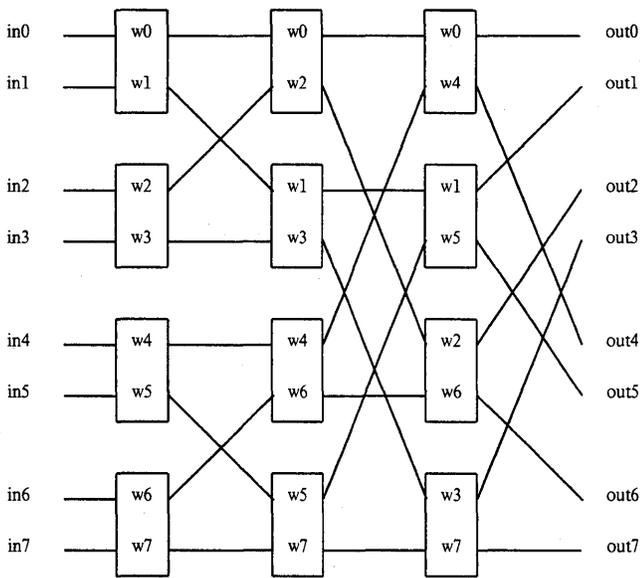
This statement waits until a , b , or c becomes greater than 1 and then performs the statements following the true condition. If c becomes greater than 1 first, then no statements are executed, but execution of the processor continues at the statement following the wait statement. If none of the conditions becomes true within $d+e$ time from the time the wait statement was encountered, then the timeout action list ($d←0$) is executed.

More generally, if one or more wait conditions are present, then the processor suspends until the expression in one of the clauses is true (if one is true immediately, then the processor does not suspend at all). If a delay clause is also present, then the processor waits until a clause is true or until the specified amount of time has elapsed. If this second condition occurs (timeout), then the action list after the time expression is executed. If no timeout action list is present (and timeout occurs), then the wait statement is just released and execution continues at the end of the wait statement.

If only a delay clause is present, then the processor merely suspends for the specified amount of time. A fine grain of granularity can be achieved by inserting delays of zero time where zero is actually interpreted as a small (epsilon) amount of time. A zero delay causes the processor to suspend and allow any waiting processors to evaluate their wait conditions.

MPDL SIMULATION SYSTEM

A simulator for MPDL has been implemented in Interlisp and now runs at ISI. The simulator allows the user to run system



straight exchange
Figure 2—Binary N-cube diagram

descriptions and provides extensive monitoring facilities, which we have found to be useful in the development of multiprocessor descriptions.

The user interface for the MPDL simulator consists of a set of commands that allows the user to do the following types of things:

- Parse an MPDL multiprocessor description, build an interconnection structure to be simulated, and start the simulation.
- Focus attention on any of the various control contexts (processors, procedures, and connectors) in the simulation, using tree traversal or search commands.
- Execute a list of MPDL actions within any context (scope) of the simulation, e.g., examine the values of simulation variables, assign new values to variables, etc.
- Display the declaration structure and current state of any context and its subordinate contexts.

The simulator also provides a powerful break and trace facility for use in debugging user simulations. The package consists of a set of commands that allow setting of breaks and traces on any construct in the description being simulated. The kinds of things that may be done are as follows:

- Breaking and/or tracing any construct in the MPDL language.
- Making these breaks and traces conditional. The conditions are expressed in the MPDL description language

itself and can be evaluated in the context of any process in the system.

- Specifying a set of actions to be executed once it is determined that the specified break or trace ought to occur and before it is entered. These are, again, specified in MPDL, and can be evaluated in any context.

The POPART system⁹ is used to generate a parser and a grammar-based editor for MPDL. Given the MPDL grammar, the POPART system produces the parser and an extensive set of editing, printing, and program transformation commands. Typical commands allow the user to search, delete, replace, print, and prettyprint portions of the description.

The appendix gives a summary of the commands provided by the MPDL development system.

```

composite cube8 (in0, in1, in2, in3, in4, in5, in6, in7,
                out0, out1, out2, out3, out4, out5, out6, out7)
  connectors wa0 = w, wa1 = w, wa2 = w, wa3 = w, wa4 = w, wa5 = w,
             wa6 = w, wa7 = w,
             wb0 = w, wb1 = w, wb2 = w, wb3 = w, wb4 = w, wb5 = w,
             wb6 = w, wb7 = w;

  processors
    stage1 = stage (in0, in1, in2, in3, in4, in5, in6, in7,
                  wa0.wr wa1.wr, wa2.wr, wa3.wr, wa4.wr,
                  wa5.wr wa6.wr, wa7.wr),
    stage2 = stage (wa0.rd, wa2.rd, wa1.rd, wa3.rd, wa4.rd, wa6.rd,
                  wa5.rd, wa7.rd, wb0.wr, wb2.wr, wb1.wr, wb3.wr,
                  wb4.wr, wb6.wr, wb5.wr, wb7.wr),
    stage3 = stage (wb0.rd, wb4.rd, wb1.rd, wb5.rd, wb2.rd, wb6.rd,
                  wb3.rd, wb7.rd, out0, out4, out1, out5, out2, out6,
                  out3, out7)

  endcomp;
  composite stage (in0, in1, in2, in3, in4, in5, in6, in7,
                 out0, out1, out2, out3, out4, out5, out6, out7)
    processors s1 = switch (in0, in1, out0, out1),
              s2 = switch (in2, in3, out2, out3),
              s3 = switch (in4, in5, out4, out5),
              s4 = switch (in6, in7, out6, out7)

  endcomp;
  processor switch (in0, in1, out0, out1)
    declare upper < 6:0 >, tag0 < 3:0 > := upper < 6:3 >, data0 < 2:0 > :
      = upper < 2:0 >, lower < 6:0 >, tag1 < 3:0 > :
      = lower < 6:3 >, data1 < 2:0 > := lower < 2:0 > ;

    repeat
      upper ← in0();
      lower ← in1();
      tag0 ← tag0 srr 1;
      tag1 ← tag1 srr 1;
      decode tag0 < 3 > @tag1 < 3 >,
        0: call LISP (ERROR),
        1: call out0 (upper); call out1 (lower), !straight
        2: call out0 (lower); call out1 (upper), !exchange
        3: call LISP (ERROR)
      endec;
      delay 23
    endrep
  endprocessor;
  connector w
    declare buf < 6:0 > ;
    proc wr (d < 6:0 > )
      buf ← d
    endproc;
    proc rd < 6:0 >
      return buf
    endproc
  endconn

```

Figure 3—Binary N-cube MPDL description

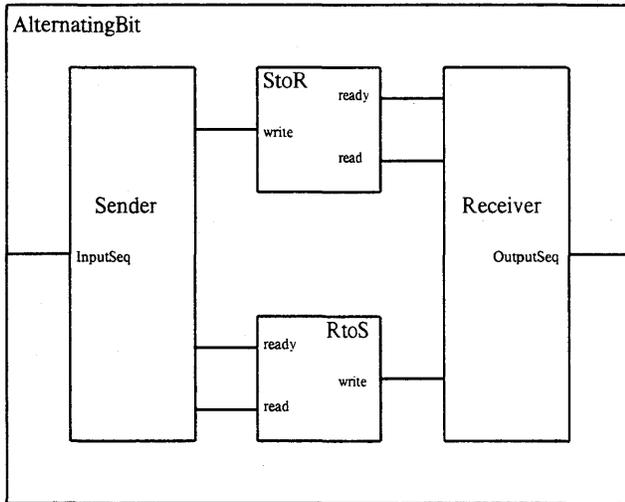


Figure 4—Alternating Bit Protocol diagram

EXAMPLES

We have performed two experiments that have pointed up strengths and weaknesses in MPDL. The first experiment is an interconnection network suitable for interconnection of an array of single-instruction, multiple-data (SIMD) machines. This example has been developed by researchers at Purdue University in conjunction with the development of efficient image-processing architectures¹⁰. The interconnection network consists of a series of stages of switching elements, where each switch can send its inputs through straight or exchanged. A diagram and the MPDL description of the interconnection network appear in Figures 2 and 3. In our example the switching elements are processors, and they communicate with each other through trivial connectors called wires. We were able to successfully describe and run a 48-processor system.

One apparent and one definite language weakness are illustrated by this example. MPDL is explicitly geared to description of asynchronous systems, and thus it appears ill-suited to describe synchronous systems. However, the delay construct that MPDL provides and the fact that the user has control over the granularity of the atomic actions makes it relatively easy to describe such systems.

There is a definite weakness in the interconnection facilities, however. In this example, there is a relatively succinct mathematical description of the individual switch/wire bindings. This mathematical description assigns wire bindings as a function of the bits in the binary representation of the switch number and stage number. However, our simple interconnection language does not provide any kind of parameterization facilities and thus the description became a long tedious list of individual bindings. The binding occurs in Figure 3 in the *cube8* composite description where *stage1*, *stage2* and *stage3* are defined. We would like to extend the interconnection language to include powerful features that would allow concise description of such systems.

The second example is a simple data transfer protocol called the alternating bit protocol¹¹ which has two processors

transferring a stream of data across an unreliable medium. A diagram and the MPDL description of the alternating bit protocol are shown in Figures 4 and 5. MPDL served very well for describing the protocol, and the wait statement with timeout condition was the perfect construct for this particular application. The simulation system allowed us to investigate this protocol and other (more complicated) versions very conveniently. The connector concept isolated the character-

```

composite AlternatingBit(InputSeq<7:0>,OutputSeq)
  connectors StoR = MsgBuf, RtoS = MsgBuf;
  processors Sender(InputSeq,StoR.write,RtoS.read,RtoS.read),
    Receiver(OutputSeq,RtoS.write,StoR.read,StoR.read)
endcomp;
processor Sender (InputSeq<7:0>,SendToMedium,
  MediumToSendReady<>,MediumToSend<>)
  declare SendSeqNo<> init 0,
    Message<7:0>,
    Timeout<7:0> init 2;
  repeat
    Message←InputSeq();
    label ResendLoop
    repeat
      call SendToMedium(SendSeqNo@Message);
      wait MediumToSendReady(): if MediumToSend() eq
        SendSeqNo
        then SendSeqNo←~ SendSeqNo;
        leave ResendLoop
      endif,
      delay Timeout
    endwhile
  endrep
endlab
endrep
endprocessor;
processor Receiver (OutputSeq,RecToMedium,
  MediumToRecReady<>,MediumToRec<8:0>)
  declare ExpectedSeqNo<> init 0,
    ReceivedSeqNo<>,
    Message<7:0>;
  repeat
    wait MediumToRecReady() endwhile;
    ReceivedSeqNo@Message←MediumToRec();
    call RecToMedium(ReceivedSeqNo);
    if ReceivedSeqNo eq ExpectedSeqNo
      then call OutputSeq(Message);
      ExpectedSeqNo←~ ExpectedSeqNo
    endif
  endrep
endprocessor;
connector MsgBuf
  declare buffer<8:0>,
    readyflag<> init 0;
  proc write(msg<8:0>)
    if LISP(RAND,0,1)
      then buffer←msg;
      readyflag←1
    endif
  endproc;
  proc ready<>
    return readyflag
  endproc;
  proc read<8:0>
    readyflag←0;
    return buffer
  endproc
endconn

```

Figure 5—Alternating Bit Protocol MPDL description

ization of the medium and allowed easy experimentation with different types of media, such as loss-free, and free-running (separate processor) media.

SUMMARY

We have designed a language, MPDL, which provides a flexible and unambiguous model of concurrency and allows for hierarchical construction of concurrent systems. Furthermore, we have introduced a construct called a connector which encourages a designer to encapsulate interprocess synchronization and communication in a single place.

Examples have been developed which have pointed out relative strengths and weaknesses in MPDL. The interconnection network that was modeled pointed out the significance of describing a synchronous system with an asynchronous language and demonstrated that there is a need for an interconnection meta-language which allows concise description of regular, repetitive structures. Data transfer protocols were conveniently modeled and relied on the abstraction and timing capabilities of the language.

ACKNOWLEDGMENTS

We would like to thank Sarma Sastry for his help with the design and implementation of the simulation system. Credit is also due Victor Lesser, Alice Parker, Mike Lyle, Sarma Sastry, Joel Goldberg, and Bill Landreth for the vision and refinement of concepts embodied in the MPDL language. We

would also like to thank Jim Kuehn and H.J. Siegel for their patience and understanding as the initial users of the MPDL simulator.

REFERENCES

1. Barbacci, M.R., G.E. Barnes, R.G. Cattell, and D.P. Siewiorek, "The ISPS Computer Description Language," Tech. report CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, August 1979.
2. TRW Defense and Space Systems Group, "SMITE Reference Manual," Tech. report RADC-TR-77-364, TRW, November 1977.
3. Parker, A.C., D.E. Thomas, S.D. Crocker, and R.G.G. Cattell, "ISPS: A Retrospective View," *Proceedings of the Fourth International Symposium on Computer Hardware Description Languages*, IEEE, Palo Alto, CA, October 1979, pp. 21-27.
4. Overman, W.T., V. Kini, and S. Sastry, "Multiprocessor Description Language (MPDL) User's Manual," Tech. report ISI/WP-15.3, USC/Information Sciences Institute, August 1980.
5. Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Comm. ACM*, Vol. 20, No. 8, August 1977.
6. Dahl, O.J., B. Myhrhaug, and K. Nygaard, "The SIMULA 67 common base language," Tech. report S-22, Norwegian Computing Center, 1970.
7. Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," NRL Report 8047, Naval Research Laboratory, June 1977.
8. Hoare, C.A.R., "Monitors: an operating system structuring concept," *Comm. ACM*, Vol. 17, No. 10, October 1974.
9. Wile, D.S., "POPART: Producer of Parsers and Related Tools: System Builders' Manual," Unpublished, USC/Information Sciences Institute
10. Siegel, H.J., et al., "Parallel Image Processing/Feature Extraction Algorithms and Architecture Emulation: Interim Report," Tech. report TR-EE 79-51, Purdue University, November 1979.
11. Bartlett, K.A., R.A. Scantlebury, and P.T. Wilkinson, "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links," *Comm. ACM*, Vol. 12, No. 5, May 1969.

APPENDIX—MPDL SIMULATION SYSTEM COMMAND SUMMARY

Initialization Commands

parse: < filename >	parse an MPDL Program from a file
build: < compositename > < parameterlist >	build a simulatable structure
start:	start the simulation
execute: < compositename > < parameterlist >	a combination build: and start: command
simulate: < compositename > < parameterlist > < filename >	a combination parse:, build: and start: command

Context Commands

cn:	display the root fully qualified name of the current context
pp: { ↑ } { < qualifiedname > }	display the full current context tree
pp: { nx }	display the context trees at the same level as the current context
p: { ↑ } { < qualifiedname > }	display one level of the context tree
p: { nx }	show the tail of the current context
ar: < arrayname > { < startindex > } { < endindex > }	display the contents of an array
setar: < arrayname > < filename > { < startindex > }	load an array with values from < filename >
↑:	go to the top context
dn: { < integer > }	go down a level in the context tree
out: { < nameorinteger > }	go up a level in the context tree
nx: { < nameorinteger > }	go to a sibling context (to the right)
bk: { < nameorinteger > }	go to a sibling context (to the left)
!nx:	go out: until a nx: is possible and then do the nx:
f: { ↑ } < qualifiedname > { nx }	find the partially qualified context, either below or to the right
setc:	set the current context to be the break context
bn:	print the name of the break context
ex: { ↑ } { < qualifiedname > } < mpdlactionlist > ..	execute the actionlist in the specified context

Tracing Processor Activations

trs: {↑}{<qualifiednamelist>}	trace the scheduling of Processors
utrs: {↑}{<qualifiednamelist>}	untrace the specified Processors
ptrs: {↑}	show which Processors are traced
pstat: {↑}{<qualifiednamelist>}	display the status of the named Processors

Editing Commands

pp*	pretty print the current expression
p*	print an abstraction of the current expression
pppt*	print the parse tree
↑*	go to the topmost expression in the current context
out*	pop out a level
f* <pattern> ..	find pattern. Pattern may contain metavariables.
ref*	refind the previously searched for pattern
nx*	go to the next element in an iterated object
bk*	go to the previous element in an iterated object
atf* {<nonterminal>}	go to the (possibly missing) nonterminal field (or print valid field names)
stype*	show the syntax type of the current expression
r* <editorexpression> ..	replace the current expression. (use \$\$ and metavariables)
rall* <pattern> = = > <editorexpression> ..	replace all occurrences of <pattern> with <editorexpression>
ib* <editorexpression> ..	insert before
ia* <editorexpression> ..	insert after
d*	delete the current expression
val* {<metavariablename>}	show current metavariables or their values
ice* <pattern> ..	instantiate current expression (like f*in setting metavariables)
sgv* <metavariablename>	make the metavariable global
rgv* {<metavariablename>}	remove the global variable assignment (or all of them)

Break Commands

br: {ANY} {<tag>}	break the current expression
br: ASK	enter full break dialogue
tr: {ANY} {<tag>}	trace the current expression
tr: ASK	enter full trace dialogue
pbr: {ALL}	print current break information
ptr: {ALL}	print current trace information
ubr: {<integer>}	unbreak the numbered break
utr: {<integer>}	untrace the numbered trace
warn: {off}	turn assertion warnings on or off

Fault tolerance by means of external monitoring of computer systems

by ALGIRDAS AVIŽIENIS

University of California at Los Angeles
Los Angeles, California

ABSTRACT

A frequently suggested solution to the problem of increasing the reliability of an already existing computer system (to be called the object machine [OM]) is to employ a functionally and physically separate *monitor* computer (to be called the monitor machine [MM]) that probes the operation of the OM in real time. The purpose of the monitoring is to assure that the functional performance of the OM does not deviate from the behavior specified by its design and by the programs being executed.

This paper systematically assesses the architectural and fault-tolerance issues that have to be resolved to effectively implement the monitoring process. The goal of the implementation is to create an integrated and uniformly fault-tolerant OM/MM complex, beginning with a given OM design.

Four principal problems are addressed in the subsequent sections: (1) implementation of the monitor machine; (2) implementation of the monitoring (OM/MM) interface; (3) specification of the monitoring function; and (4) the cost and effectiveness of monitoring.

The paper concludes with examples of model technical specifications for the architectural properties needed by the OM and the MM to attain a fault-tolerant implementation of the monitoring process.

INTRODUCTION: SCOPE OF THE PROBLEM

A frequently suggested solution to the problem of increasing the reliability of an already existing computer system (to be called the object machine [OM]) is to employ a functionally and physically separate monitor computer (to be called the monitor machine [MM]) that probes the operation of the OM in real time. The purpose of the monitoring is to assure that the functional performance of the OM does not deviate from the behavior specified by its design and by the programs being executed.

This paper presents a systematic assessment of the architectural and fault-tolerance issues that have to be resolved in order to accomplish an effective implementation of the monitoring process. The goal of the implementation is to create an integrated and uniformly fault-tolerant OM/MM complex, beginning with a given OM design.

Four principal problems are addressed in the subsequent sections:

1. *Monitor Machine implementation*: Its functional design, relative size, monitoring method, fault tolerance and/or avoidance, assistance in OM recovery, constraints to avoid interference with the OM, and other interfaces (operator, remote master machine, etc.)
2. *Implementation of the monitoring (OM/MM) interface*: The method by which the MM gains access to the OM, factors that limit the access, and the effect that this access has on OM design.
3. *Specification of monitoring*: Which functions of the OM operation and which stored OM data are to be monitored, and what are the symptoms of incorrect functional performance by the OM?
4. *Monitoring effectiveness and cost*: how to predict and to measure the effectiveness and the cost of a given approach to reliability monitoring.

The paper concludes with examples of model technical specifications for the architectural properties needed by the OM and the MM to attain a fault-tolerant implementation of the monitoring process.

PROPERTIES OF THE MONITOR MACHINE (MM)

The complexity and other properties of the monitoring operation depend very strongly on the nature of the MM itself. The main issues in the choice of the MM are

1. *Design*: Is it general purpose, a custom design, or a copy of the OM?
2. *Fault tolerance*: How is its own reliability assured?
3. *Recovery function*: Is it expected to participate in OM recovery?
4. *Constraints*: How is the integrity, security, and privacy of the OM assured in the case of a failure or a misuse of the MM?
5. *Other interfaces*: Is it internally program-controlled, operator-controlled, or connected to a remote MM master facility?

Design of the MM

Three fundamental choices are available in the selection of the MM:

- (a) The MM consists of m exact copies ($m \geq 1$) of the OM.
- (b) The MM is a general-purpose digital computer (usually much smaller than the OM).
- (c) The MM is a custom-designed (special-purpose) digital system.

All three approaches have been used in practice, depending on application constraints and cost considerations.

The m -copy approach

The m -copy ($m \geq 1$) approach is used in real-time applications in which reliability requirements are extremely high. Monitoring is simplified, since it can be a direct comparison of corresponding variables of the m copies. Recovery after failure is also facilitated.

With $m=1$ (the duplex case), the surviving copy is identified and becomes the OM, now without monitoring. The identification requires built-in self-test features in each copy. The use of duplexing for reliability is a very common solution. Successful examples of this approach are the central processors of the ESS system of Bell Telephone Laboratories.^{9, 10, 18} The success of the application of duplexing beyond pure monitoring, i.e., to attain recovery, differs widely as a function of the quality of the self-test procedures. Some notoriously unsuccessful cases have been reported.

With $m=2$, triple-modular redundancy (TMR) with majority voters is used to monitor correct operation and to implement instant correction in the variables being voted upon. A disagreement detector (DD) is used to determine when one input to a voter differs from the other two. The DD serves as an instant indicator of a potential failure and allows discarding or replacing the failed member. TMR has been successfully used in the U.S. space program—e.g., in the SATURN V launch vehicle computer²—and is currently being investigated for application in microcomputer systems²⁰ and in aerospace control computers.¹⁹

Values of $m > 2$ are occasionally used when recovery from two or more failures is needed. An example is the four-computer ($m=3$) space shuttle computer complex, in which each computer monitors output variables of the other three.²¹ Interface restrictions severely limit the observable variables in this design.

In general, the m -copy approach to monitoring has been justified by the criticality of application, and it usually follows fault detection by recovery actions that are implemented by the surviving copies. Because of the identical design, any one copy can serve as the OM. In TMR cases, the OM outputs are produced through majority voters, thus using the MMs to increase the probability of getting correct outputs.

The distinct general-purpose-MM (GP-MM) approach

This approach is found in some of the very recent large-scale computer systems. A small general-purpose machine

(minicomputer) or a peripheral processor is employed as a maintenance processor, which performs a certain set of monitoring operations in real time on the large system (OM). In addition, it also may act as a control console, peripheral processor, communication processor, diagnostic processor (not in real time), logout storage device, and interface unit to a remote central diagnostic facility.

The use of a general-purpose minicomputer or a peripheral processor as the MM for a large-scale OM offers a significant cost advantage. First, the MM is an off-the-shelf product, already operational and provided with software and maintenance support. Second, in addition to being the MM, it usually performs several other services for the OM. An evident disadvantage is the rather limited ability of the MM to perform real-time monitoring. This is due to the variety of tasks it is expected to carry out and also to the rather limited number of interface points that can be established between the MM and the OM. This problem is further discussed in the section "The Monitoring Interface."

State-of-the-art examples of the distinct GP-MM approach are

1. The Data General Nova 1200 minicomputer used as a console processor for the Amdahl 470 V/6 computer system.¹
2. The Data General Eclipse S-200 minicomputer used as a maintenance control unit for the CRAY-1 computer system.¹⁶
3. The PDP-11 minicomputer used as a front-end processor and diagnostic computer (with direct diagnostic bus linkage) for the KL-20 central processor of DECSYSTEM 20.¹⁷

The custom-MM approach

This approach also is found in some large general-purpose computer systems. Although custom-designed MMs also perform some other functions, such as diagnostics (non-real time), initial loading, etc., they are much more closely tailored to the monitoring function and cannot be readily saturated by other real-time tasks supporting the OM.

Because the custom design of the MM occurs along with the design of the OM, a more favorable monitoring interface can be created than for the GP-MM. A significant disadvantage of the custom-MM approach is the cost of designing, building, and developing the software and maintaining the MM. High-volume production cannot be expected to reduce the cost per unit because the custom-MM is of very limited applicability.

Examples of the custom-MM approach are

1. The maintenance control unit (MCU) of the CDC STAR-100 computer.¹⁴ The MCU has both an I/O channel connection and a special set of internally connected interfaces that allow it to monitor CPU status and gather event counter data.
2. The control and maintenance unit (CMU) of the Burroughs BSP array processor¹², which has access to most data paths and registers of BSP and itself runs under control of maintenance software of a B 7800 computer (the system manager for BSP).

3. The test-and-repair processor (TARP) of the JPL-STAR computer.³ The TARP is an MM embedded within the OM that carries out very extensive real-time monitoring of the execution of every instruction and initiates automatic recovery in case a fault is detected.
4. The maintenance and support processor of the IBM 4341 Processor¹⁵, which has a high-speed parallel link to the CPU. It can read out, analyze, and store machine status information upon detection of an error.

Relative size of the MM

The word *size* is used here to designate the complexity and cost of the MM when compared to the OM.

In the *m*-copy approach, the MM size is *m* times the size of the OM. In addition, there are interface elements that perform comparisons and usually also implement recovery action. The relatively high cost of this approach limits its use to reliability-critical real-time applications, such as aircraft control, spacecraft control, air traffic control, telephone exchange control, etc.

In order to reduce the cost of *m*-copy monitoring, the copies are sometimes assigned various background tasks while also monitoring the OM. Two modes of operation—monitored and simplex—are sometimes made available.²⁰ Such sharing of the MM, however, may easily lead to severe reduction in the effectiveness of the monitoring function. Latent faults in the OM or the MM may accumulate more readily when different tasks are carried out by the OM and MM for significant time intervals.

In the distinct MM approaches (both general-purpose and custom), the MM is usually much smaller than the OM, and only a single copy of the MM is used. The relatively small size of the MM is due to the usually very large size of the OM and to the less critical reliability requirements compared to those for which the *m*-copy approach was used. The emphasis here is not on uninterrupted operation but on fast fault location and recovery, usually by reconfiguration and manual repair.

An MM (GP or custom) larger than the OM is difficult to justify economically, since it has to be compared to the simple *m*-copy alternative. One potentially attractive exception can be postulated here: a single central master-MM could serve as the MM for a number of OMs, which could be located remotely or form a computer network. A step in this direction has been taken by some manufacturers who provide a remote central diagnostic system accessible via local MM's.^{1,17}

Fault Tolerance of the MM

It is evident that the MM may itself malfunction. As a consequence it may fail to execute properly its monitoring function. Even more dangerous is the possibility of the faulty MM actively interfering with the correct operation of the OM. This issue is discussed separately in the section "Interface Constraints on the MM."

The extent of fault tolerance provided in the MM depends on the criticality of the monitoring requirements. In the most critical cases, the *m*-copy approach with $m > 1$ provides very

complete fault tolerance at a relatively high cost. Standard fault-tolerance techniques, such as TMR,²⁰ duplexing,¹⁰ etc., are applicable to the protection of the MM.

The GP-MM and custom-MM approaches do not offer a clearcut solution to the MM fault tolerance problem. The existing GP-MMs use the standard fault detection and recovery techniques as provided by the manufacturers. The Custom-MMs that have been discussed do not show systematic fault tolerance. This is probably due to the fact that resource limitations did not allow designers of the Custom-MM time to incorporate fault tolerance. The relatively small size of the MM makes the probability of its failure much less than that of the OM and thus relegates the MM fault tolerance question to the background. An interesting and unique example of MM fault tolerance is the Custom-MM called TARP (test-and-repair processor) of the JPL-STAR computer,³ which uses hybrid redundancy for its own fault tolerance.

OM Recovery Assistance Functions in the MM

In addition to the OM monitoring, in most cases it is cost-effective to include OM recovery assistance functions in the MM. The functional and physical isolation of the MM assures a high probability that the MM will remain operational during an OM failure and will be able to execute the OM recovery procedures.

In the *m*-copy MM implementation the OM recovery is usually closely integrated with the monitoring function in order to provide extensive fault tolerance. These recovery procedures have been discussed in the section "The M-Copy Approach."

In both the GP-MM and custom-MM approaches the extent of OM recovery assistance remains at the discretion of the designer. The principal constraint that limits the introduction of recovery assistance is the usually quite restrictive monitoring interface (see the section "The Monitoring Interface"). An initial program load for the OM is the most common recovery assistance feature of the existing MM's.

Interface Constraints on the MM

The presence of effective reliability monitoring using an MM implies the possibility that the MM has access to internal points of the OM that are otherwise not accessible without meeting strict authorization requirements. Such access raises the possibility of two forms of interference with OM that can originate in the MM:

1. An MM hardware fault (physical fault) or a design error in the MM software (manmade fault) may cause the MM to interfere with OM operation.
2. The path to the OM provided via the MM may be used by individuals who have access to the MM as a means to gain unauthorized access to the OM, bypassing the OM security and privacy mechanisms.

Safeguards against both forms of interference need to be introduced into the MM design.

Interference due to faults in the MM

One form of possible MM interference with OM operation is the occurrence of MM faults that cause a false alarm about the OM. The extent of interference with OM operation caused by such a false alarm depends on how many automatic OM recovery functions are provided in the MM. The fundamental solution to the false alarm problem is to provide the MM with fault tolerance. If MM fault tolerance provisions do not exist, an independent verification of the alarm is needed when routine OM recovery does not succeed. This can be carried out by an operator initiating an MM test procedure either executed by the OM or as an MM self-test. An alternative is to involve the assistance of a remote diagnostic machine to test the MM.

A second form of MM interference with the OM can be physical interference through the MM/OM interface. Passive MM data acquisition interfaces can be adequately protected by physical isolators. A more difficult problem is presented by active MM input lines to the OM. For an extreme example, a "stuck-on-one" master reset signal from the MM would completely paralyze the OM. In addition to physical isolators, such active interface lines need self-checking, which can be implemented by duplexing or other fault tolerance techniques.

MM software errors (design faults) can also cause both false alarms and interference (in the form of incorrect commands) through the MM/OM active interface. Recognition of software fault conditions will be facilitated when the MM is provided with fault tolerance with respect to physical faults. This will reduce the probability of misinterpretation of design fault symptoms as being caused by physical faults.

The probability of detecting the occurrence of both design and physical faults in the MM can be improved by providing certain defensive measures in the OM software. Verification of commands received from the MM by means of an exchange of messages is one such measure. Another is the execution of an MM test program from the OM, either periodically or upon the occurrence of certain MM commands.

Unauthorized access via the MM

The introduction of an MM makes it necessary to review the security and privacy protection mechanisms that are implemented in the OM. The goal of this review is to insure that there are no previously unforeseen paths that could be used for unauthorized access, bypassing the existing safeguards of the OM.

The design of the hardware and software of the MM must also take into account the existence of security and/or privacy requirements for the entire system composed of the OM and MM.

A special condition is introduced by the provisions for remote diagnosis of the OM from a master diagnostic MM. Such provisions exist, for example, for the Amdahl 470V/6 and DECSYSTEM 20 computers. In this case, procedures for security must include the transfer of control of diagnosis of the OM from the local MM to a remotely located MM.

Interfaces of the MM

The OM/MM interface is discussed separately in the section immediately following. In addition to this interface, the MM needs the definition of the MM interface with the operator and linkages with other computers, especially a remote MM facility.

Another interface consideration for the MM is the recording, reduction, and presentation of reliability data collected during real-time monitoring. This function can readily be made a part of the functions of a general-purpose MM. An example is the Data General Nova 1200 minicomputer, which serves as the MM (Console Processor) for the Amdahl 470V/6 computer. In contrast to the IBM System/370, the machine check extended logout information is stored in the Nova 1200 memory under MM control.

The use of a programmable MM makes it necessary to provide an MM operator manual that defines the commands and instructions available to the operator. Furthermore, the manual needs to describe the procedures to be followed in case of reported OM failures and in case of suspected or indicated MM failures.

THE MONITORING INTERFACE

The guiding principle in design of this interface is that the OM and the MM are physically and functionally separate computer systems. A positive aspect of this constraint is that strong isolation is provided, which reduces the probability of related failures that simultaneously affect both the OM and the MM. Furthermore, the OM/MM interdependence with respect to design changes within each system is minimized. A limitation is imposed by the necessity to interface the different architectures, packaging, and physical layout of two separate machines.

The main issues in the choice of interface techniques are

1. *Methods of access*: Standard I/O provisions, dedicated I/O devices, custom links; OM vs. MM control.
2. *Access-limiting factors*: Architectural, physical.
3. *Modification of the OM* to accommodate the use of an MM: Limited retrofitting or redesign (hardware, architecture, firmware, software features).

Methods of Access

The simplest method of supplying data about OM operation to the MM is to employ the *standard output procedures* of the OM to deliver the specified OM status data to the MM. The MM itself may be a multipurpose peripheral ("front-end") processor that allocates a high priority to the OM status message. In response to the data received it may in turn interrupt the OM to gain more data or to initiate a recovery procedure. The use of this approach is found in many older general-purpose computer systems.

Two limitations of this approach are (1) the rather loose coupling between events in the OM and the MM and (2) the limited bandwidth for real-time monitoring that is provided by the shared use of one channel. The loose coupling occurs because the flow of monitoring data from OM to MM is controlled by OM programs. It will succeed only if the OM

software involved in monitoring is itself not affected by any faults. The occurrence of such faults will be detected by the MM only as failure of the data to arrive or as damage (e.g., parity errors) in the data being delivered.

The OM conditions that can be reported to the MM via the regular OM output route are limited to those that are available to the OM software. The rate at which this information is made available to the MM is limited by both the speed of the output channel and the maximum acceptable overhead (in software and in time) that the delivery of data to the MM imposes on the OM.

Some reduction in the OM overhead and improvement in OM/MM coupling can be attained by use of a *dedicated OM channel* that is reserved exclusively for the transmission of monitored data to the MM. Delay in transmission to the MM is reduced, since no other functions contend for the use of the channel. The continuous availability of the dedicated channel allows regular scheduling of data transmission to take place in parallel with other OM events. The software protocol needed for transmission is reduced to a minimum. One remaining limitation is that a subset of OM software must function correctly in order to communicate the data to the MM. A second limitation is that the data to be sent must be from registers that are accessible to OM software.

A much more fundamental approach to OM access by the MM is provided by the approach of *custom-designed links* that connect the OM and the MM for the purposes of monitoring and recovery (including diagnostics). It should be noted that a few special links are sometimes provided in systems that depend on the standard channel interface. These are signals that usually are available at the operator's console. For example, the CRAY-1, which is connected to its MM (Data General ECLIPSE S-200) by a channel pair, also has additional control signals for the following operations: (1) Master Clear, (2) I/O Master Clear, (3) Dead Dump, and (4) Clear Parity Error.

A full OM/MM *custom link* consists of a separate and channel-independent path between the OM central processor and the MM. It allows the readout of certain storage elements or registers in the OM. The readout may be of two types. Certain registers are read out to the MM continuously while the OM is proceeding with its operation. Other readouts take place statically, after the OM has been stopped and a command has been issued from the MM. The MM also has direct control signals that cause an immediate OM response (such as stop), select registers to be displayed continuously, preset conditions for stopping the OM, etc.

It is very probable that the major advantages offered by OM/MM custom links will lead to their general acceptance in the next generation of medium- and large-scale systems. State-of-the-art illustrations or dedicated OM/MM links are

1. The Amdahl 470V/6 system, which has a custom link with its console processor (Data General Nova 1200).
2. The CDC STAR-100 system, which has a custom link with its custom-designed Maintenance Control Unit (MCU).
3. The IBM 4341 processor, which uses a service bus link to its maintenance and support processor.

Access-Limiting Factors

For any given unmonitored OM system, the access that can be provided for an MM is limited by both architectural and physical factors.

From the architectural viewpoint, access is limited by the properties of the I/O system of the OM and by the accessibility of various internal OM registers to the OM software that will supply the MM with status information from the OM. The instruction set of the OM has a major role in determining the ease with which the monitoring data flow from the OM to the MM can be set up. The second architectural component of the OM that may provide access to the MM is the system operator's console, which provides the capability of sending direct commands to the OM CPU and of displaying certain selected OM status information.

From the physical viewpoint, very severe constraints to access by the MM are imposed by (1) the OM packaging and (2) the length of communication lines between the OM and the MM, which determines the potential delays, synchronization problems, noise pickup, etc. An exceptionally emphatic example of access difficulties is found in the Space Shuttle computer system.²¹ Four identical computers (IBM AP-101 CPU with an I/O processor) are used to provide fault tolerance by means of the *m-copy* approach to MM design, as described in the section "The *M-Copy* Approach." The use of off-the-shelf hardware allowed a single serial bus for each computer (the ICC bus). Instead of broadcasting (transmitting) all critical command outputs to the other three computers on this bus, only a checksum word is broadcast and used as the means of verifying correct operation.

Retrofitting Modification of the OM for MM Access

The difficulties of accessing the OM for monitoring may be alleviated by introducing certain modifications in the OM. For convenience of discussion, we consider separately the cases of (1) limited OM retrofitting and (2) major OM redesign (see the section immediately following).

By the term *limited retrofitting* we designate changes that can be introduced as engineering modifications into an existing production model of the OM system. Most readily accommodated are changes in system software. The addition of a dedicated channel (or a high-priority MM device in an existing one) is also relatively simple. A more basic form of retrofitting is the introduction of new microcode that facilitates the delivery of OM status data during OM operations or generates diagnostic data under MM control.

Another very promising form of retrofitting is the monitoring of the data that are being received at the operator's console by linking the console display registers to the MM. The console also offers the possibility of connecting the MM to share the command lines and the data input lines that have been provided for the operator's use on the console.

The most difficult form of retrofitting is the addition of new hardware links from the OM to the MM. However, it is also the most effective monitoring interface, as was discussed in the section "Methods of Access." Here we encounter the

problems of high packaging densities and compact topologies that are essential to insuring a high speed of OM operation. Access is most readily gained at connectors; however, the information passing these points is already rather readily available for output. Furthermore, it is also usually monitored by internal OM devices such as parity checkers. Direct hardware link access to the most critical system registers by retrofitting does not appear to be a practical solution in most of the currently existing systems.

Redesign of the OM for MM Access

The most fundamental approach to the introduction of a good OM/MM interface is to perform a redesign of the OM hardware so that a full custom link is provided to the MM.

The goal of the redesign is to provide a new model of the OM that is in all respects fully compatible with the previous OM model but contains a custom link to the MM. In addition to the link itself, there may be architectural additions to the OM that facilitate the monitoring process. Examples of such features are error detecting and correcting codes, instruction retry provisions, event counter hardware, and special operation codes for monitoring and diagnosis. These features will be discussed further in the section "Specification of Monitoring."

An example of a redesigned computer that has maintained full architectural compatibility with an earlier (unmonitored) system is the Amdahl 470V/6 computer. It has remained fully compatible with the IBM System/370 and added a sophisticated monitoring capability, using both dedicated logic in the OM and a custom link to its MM (Data General Nova 1200).

Effects of Monitoring on the OM-User Interface

In concluding the discussion of OM/MM interface, we note that the introduction of real-time reliability monitoring by an MM is very likely to become visible at the interface between the OM and its users. The usual manifestation of the presence of an MM will be the addition of some new constraints on the user.

Without attempting to identify all possible implications of adding the MM, we note the following two extremely visible effects:

1. The set of OM manuals and the detailed OM documentation will need to be revised to reflect the existence of the MM.
2. The presence of MM-related software in the OM may change the system software timing; this may be a limitation in tightly scheduled real-time OM systems.

Looking at item 2 above from the converse side, we note that a specific real-time application of the OM may become a limitation on the application of real-time monitoring techniques that impose timing constraints on the OM.

SPECIFICATION OF MONITORING

The goal of the monitoring specification is to assure the correct functional performance of the OM by means of real-time monitoring that is carried out by the separate MM, given that faults may occur in the OM.

The correct OM performance is assumed to be implicitly specified by the set of OM programs and the logic structure of the OM hardware. The present discussion assumes that there are no latent manmade (design) faults in OM software and hardware. This issue of manmade faults in OM and in MM will be discussed separately in the section "Identification of Manmade Faults."

The concept of real-time monitoring implies a comparison. The entities being compared are (1) subsets of operating states of the OM and (2) reference states being stored or generated in the MM.

To generate a monitoring specification, four questions need to be answered:

1. Exactly which operating states of a given OM are potentially accessible for the purpose of monitoring by a functionally and physically separate MM?
2. What are the symptoms of incorrect functional performance that can be identified by observation of these states?
3. Exactly which subset of those accessible states will be selected for monitoring?
4. How are the selected states going to be interpreted by the MM in order to decide whether the functional performance of the OM is correct?

A fifth major question, which is outside the scope of the present discussion, is

5. How is the MM going to respond to the conclusion that OM performance is not correct; i.e., what recovery action is to be taken?

The first four questions will be discussed in the following parts of this section.

Potentially Accessible OM States

The potentially observable states of the OM are either of static or dynamic nature. The *static* states consist of the digitally represented information that is at any moment held in the OM. This includes all stored data, stored instructions, and machine state variables that exist at any given time in the OM. The *dynamic* state information retains a record of the sequence of static states (from some previous time instant on) through which the OM has arrived at the present static state.

An important distinction that needs to be made in identifying the potentially accessible states is whether the state is being observed at a physical variable or at a logical variable level. At the physical variable level the quantities being observed are voltages, currents, positions of moving devices, temperatures, intensity of light, humidity, etc. At the logical variable level the physical variables are being interpreted di-

rectly as the True and False (and possible Indefinite) values of a two-state device, or as the n discrete states of an n -state device (for example, of a tristate bus).

In current practice, an MM frequently receives information at both the logical and the physical levels. For example, in addition to an extensive set of logic variables of the CDC STAR-100 system, the custom MM (designated MCU) receives the following physical variables: heat sink temperatures, freon pressures, room dewpoint, external power input, machine section power inputs, and presence of short circuits in machine sections.¹⁴

Although the set of potentially accessible OM states is very large, we must note that not all OM states are potentially accessible for monitoring. First, we have device constraints. The internal states of an LSI logic circuit or a magnetic bubble memory cannot be reached by an external probe. Second, we have packaging constraints. Some designs are so densely packaged that very few (if any) points other than external connectors can be practically accessed without a redesign of the OM. Third, we have architectural constraints. The system architecture may specifically exclude access to certain internal registers. The problem of choosing a subset of potentially accessible OM states for observation is discussed separately in the section "Choice of OM States for MM Monitoring."

Symptoms of Incorrect Performance

In order to perform monitoring, it is necessary to have the means to distinguish correct-operation states of the OM from faulty states. This is the single most important issue in devising methods of OM monitoring. Two distinct cases need to be discussed: (1) the MM is a *copy* (or $m > 1$ copies) of the OM, and (2) the MM is *distinct* from the OM (usually it is considerably smaller).

The M-copy MM case

When the MM consists of ($m \geq 1$) copies of the OM, the assumption is that during correct operation the corresponding states of the OM and the MMs will be identical. Presence of a faulty state is detected by a comparison. For $m > 1$, the faulty machine is identifiable because it disagrees with the other (two or more) machines. For $m = 1$, a disagreement only indicates that either one of the two machines (OM and MM) is faulty. The identification of the faulty machine requires either supplementary real-time fault detection in each individual machine or off-line testing—i.e., either externally applied diagnosis or self-tests for each of the two machines. At the cost of replication, the method of monitoring is reduced to its simplest form (comparison), and the only remaining limitations are (1) the number of points accessible to be compared and (2) the reliability or fault tolerance of the comparison device itself.

Failure to resolve uniquely which one of two or more identical machines is faulty may result for at least three reasons:

1. Latent faults or damaged information may accumulate in each one of the machines. By the time a disagreement

is noted, either a majority of machines is faulty or none is correct in the duplex case ($m = 1$). The probability that latent faults and damaged information will remain undetected for longer times and therefore accumulate in a majority of machines is higher when few comparison points are available. It is very high when the comparison device itself fails first (at least partially) in such a manner that it does not indicate some of the disagreements that occur.

2. There may be some faults that affect most or all $m + 1$ machines simultaneously. The most likely of such faults are those caused by external interference with system operation—for example, power transients, electromagnetic interference, sudden changes in environmental conditions. Such faults defeat the isolation provided between the individual machines and cause all machines to enter faulty (but probably not identical) states. The effect of such faults is similar to that caused by latent faults, as discussed in 1 above. Difficulties of the type discussed above can be minimized by the choice of a sufficient set of comparison points and by exercising stored data and programs by periodically moving them past comparison points.
3. In the case of $m = 1$, the self-test or external diagnosis may not be sufficiently complete or accurate. In this event, either there is no decision or (even worse) the good machine is wrongly identified as the faulty one.

The distinct MM case

Ordinary comparison does not suffice for monitoring when the MM differs from the OM. In cases of large general-purpose OM systems the MM is a much smaller system than the OM. This is dictated by cost considerations, which preclude the use of the much simpler m -copy MM approach. Both general-purpose minicomputers and custom-designed maintenance processors have been used to serve as MM systems.

Incorrect operation here cannot be identified by noting bit-by-bit disagreements, and it is necessary to identify other symptoms of incorrect functional performance (here called faulty operation) of the OM that can be recognized by the MM. The symptoms of faulty operation that can be looked for fall into two categories: static and dynamic.

Static symptoms of faults

The static symptoms are noted by observing (i.e., subjecting to a checking algorithm) arrays of binary information that reside in the OM. The array may be one byte, one word, or a block of words of arbitrary size. In order to distinguish correct arrays from faulty arrays, the array is encoded in an error-detecting code (EDC) or error-correcting code (ECC). Complete duplication of the array is a limiting case of EDC encoding; triplication is a limiting case of ECC encoding.

Besides such arrays, EDC or ECC encoding can also be applied to selected sets of individual logic variables that co-exist simultaneously during OM operation. A very simple

example is the 2-out-of-5 (generally, " k -out-of- m ") encoding of the consecutive states of a counter.

In most cases the checker, i.e., the hardware that performs the checking algorithm on the arrays or sets (parity checkers, Hamming code error correctors, etc.), is located within the OM itself, and only one output signal of the checker (indicating that detection or correction occurred) is available as an input to the MM. The advantage of such checker location is that there is no need to send the entire array to the MM for checking there, and OM/MM data communication requirements are kept small. The disadvantage is that checking is not done in a separate location (isolated from the OM) and we have to remain concerned about detecting OM faults that affect the checker hardware itself.

Dynamic symptoms of faults

The checking of static symptoms (especially parity checking, cyclic redundancy checks on block transfers, Hamming SEC/DED checking) is very widely used in current-generation computers. It provides highly useful information about the OM; however, it will not indicate when faulty operation alters the specified algorithm to some other algorithm. Examples of such events are as follows: an addition is done instead of a subtraction; a multiplication is terminated a few steps prematurely; the wrong location is accessed in a memory; the instruction counter is not incremented or is incremented twice. All such events may be caused by small transient faults affecting the sequencing logic for a given algorithm, or causing a faulty address decoding to take place within a memory.

The information that is contained in an instruction (operation code, address, tag bits) contains the information on the events that are expected to take place (information transfers, calculating algorithms, condition code settings, memory reads or writes, etc.) during its execution. This information allows the monitoring of the execution of an instruction for dynamic symptoms of a fault: incorrect sequences of events, failure to perform expected events, time-bounds violations (too long or too short), occurrence of unspecified events. Instead of duplication, only certain key events can be monitored, thus very significantly reducing the complexity of monitoring hardware. An example of such monitoring is found in the JPL-STAR (Self-Testing-And-Repairing) computer,³ in which the test-and-repair processor (TARP) serves as a custom-MM and the rest of the computer is the OM. The OM part consists of processor, memory, I/O channel, and read-only memory modules. This part performs spacecraft guidance and control computations that are being monitored in real time by the MM (TARP) for both the static symptoms (EDC encoding of machine words) and the dynamic symptoms (as described above).

In addition to the monitoring of the main events in the execution of an instruction, faulty operation can also be recognized because violations of software-controllable constraints on current program execution take place. Examples of such violations are memory bounds violations, incorrect resource requests, and unauthorized use of privileged instructions. These violations are referenced to the execution of an

entire program rather than a single instruction. They may be caused by programming errors (manmade faults) as well as by physical faults, and an identification of the cause is essential if recovery is to be attempted.

In conclusion, we note that dynamic symptoms may be observed at several levels: microinstruction, single-instruction, application program, external control (from MM). A judicious choice of the proper level is a key issue in the implementation of the OM/MM system.

Choice of OM States for MM Monitoring

The number of potentially observable OM states is very large. The OM states that are actually observed by the MM are usually a small subset of this set. Several practical factors limit the monitored set of OM states. They are

1. *The OM design*—Does it contain numerous built-in checking provisions? Examples are parity checkers, memory access monitors, and memory bound limit registers. All such built-in checks are natural choices for external monitoring. When built-in OM checks do not exist, the alternative is to do that checking in the MM.
2. *The interface constraints* imposed by the OM/MM monitoring interface—As were discussed in detail in the section "The Monitoring Interface."
3. *The cost* of conveying information across the OM/MM interface, especially where retrofitting or redesign of the OM is involved.
4. *The capability of the MM* to interpret the OM states—the MM may be too slow, or it may have an instruction repertoire insufficient to handle the OM states that can be conveyed via the OM/MM interface.

The goal of the selection procedure is to select a set of OM states that will give the best possible detection coverage (probability of detection given that incorrect operation has occurred in the OM) at an acceptable cost. A program that automatically generates various sets of OM states and evaluates the available coverage and the cost would be a very effective design tool.

In addition to the selection of OM states to be monitored, the questions of their interpretation in the MM also requires attention. Some states, such as parity error messages, are already interpreted by the checking hardware in the OM and need only a response to be programmed in the MM. Other states may need more interpretation in the MM before a decision can be made whether a symptom of faulty operation is present. For example, consider the case when two memory modules produce simultaneous outputs to a bus. If the two modules are duplexed for protective redundancy, the operation is correct; otherwise, one module has misinterpreted the command and is operating incorrectly. In current practice, the sets of OM states being monitored are usually very small. They include constraint violation signals at the program level and checking signals from the OM reporting parity errors and similar static (error-code) symptoms, as discussed in the section "Static Symptoms of Faults."

Identification of Manmade Faults

Manmade faults that lead to incorrect performance of the OM may be of two types: (1) design faults, including software imperfections and latent hardware design errors, and (2) interaction faults, caused by inappropriate operator action.⁸

The symptoms of both classes of manmade faults are frequently the same as those of physical faults. This is especially true for faults that are recognized at the program level. The symptoms are violations of program-level constraints (see the section "Dynamic Symptoms of Faults"). A fundamental method to handle manmade faults is software fault tolerance⁶. Software fault tolerance is still a topic of research, and it is not available in current OM systems.

An alternate method to distinguish whether a fault is physical or manmade is to employ retries and (usually MM-based) diagnostic procedures that identify physical faults. Remaining undiagnosed faults are then considered to be manmade. Finally, there is also the possibility of manmade faults in the MM. Especially critical are MM faults that can cause the OM to be unnecessarily interrupted or even to enter incorrect operation. Systematic protection against the effects of manmade faults in the MM is an important constraint in the design of the MM and of the OM/MM interface.

The entire manmade fault problem is currently handled by ad hoc procedures in both the OM and the MM. These procedures usually involve extensive participation and judgment exercised by a maintenance expert. Some of the recent systems (e.g., Amdahl 470V/6, DECSYSTEM 20) provide a remote diagnostic center, staffed by top maintenance experts, as support to local maintenance personnel. The problem of automatic, MM-based handling of manmade faults in the OM (and also in the MM itself) remains a high-priority topic for further research and experimental implementation.

EFFECTIVENESS AND COST OF MONITORING

At present, the evaluation of the effectiveness and the assessment of the cost of monitoring remain wide-open issues for research, systematic development, and experimentation. The goal of this section is to outline the various alternatives in approaching these two issues.

Monitoring effectiveness (ME) in this context means a quantitative estimate of the success of a given OM/MM system in detecting an incorrect functional performance (also called faulty operation) of the OM. The probability of successfully carrying out a recovery of the OM from the detected faulty operation is not included in ME.

Fault Identification and Effectiveness Measurement

The first fundamental requirement for the prediction or measurement of ME is to identify the classes of faults to be considered in the prediction or measurement of the ME of the OM/MM system.

We note that physical faults include the classes of transient and permanent faults, local and distributed faults, and deter-

minate and indeterminate faults.⁸ Some classes may be excluded as being too unlikely or unimportant in a given situation. The ME may be considered for individual classes of faults or for all fault classes (that were identified as significant) at once.

Manmade faults include the major categories of design faults (hardware and software) and interaction faults that are introduced by inappropriate operator actions at the person-machine interface. If an attempt is to be made to establish ME figures for manmade faults, great care is needed to describe the expected faults in terms of their symptoms. Such a description is a prerequisite for all ME prediction.

In general, once the classes of faults that are of interest are identified, the next step is to generate a description of their symptoms as they could appear in the OM. This procedure is very difficult, since it requires the superimposition of the fault onto the logic structure of a given part of the OM and the derivation of its symptoms as they appear at points within the OM at which the symptoms can be observed. The observation can be made either directly by the MM or by checking logic in the OM that reports its observations to the MM (e.g., a parity checker within the OM). An illustration of an analytic approach to symptom derivation (with respect to physical faults in arithmetic processors) is found in Avizienis, 1971.⁴ The generation of fault symptoms is greatly facilitated by the use of digital-logic simulation programs that can analyze the behavior of faulty circuits.²² A good example of such programs is the LAMP system¹³ developed at the Bell Labs.

It must be noted that very frequently the fault identification and symptom generation issues are entirely bypassed in the description of existing OM systems. The description provides only the list of error signals that are generated by the checking mechanisms within the OM. These signals implicitly define the fault classes that are considered as being of interest. An explicit description of these faults requires a reverse analysis, going from the symptoms to the causes, i.e., to physical or manmade faults.

Four approaches that have been used to derive quantitative predictions of system fault tolerance and that are applicable in the prediction and evaluation of the ME of a given OM/MM system are:

1. Analysis, using mathematical models of the system⁷
2. Simulation, using either functional-level or digital-logic-level system descriptions^{13, 22}
3. Experimentation, using a copy of the system that is instrumented with fault injection and data acquisition devices⁵
4. Field-data collection on the performance of systems after their delivery to users⁹

Cost Assessment

In this paper the MM is defined as a functionally and physically separate computer system. This separation of the MM and the OM significantly facilitates the cost assessment of monitoring. Major cost-contributing items of the real-time monitoring by an MM are

1. The procurement, programming, and maintenance of the MM
2. The setting up of the necessary OM/MM interface, including software modifications in the OM
3. Retrofitting or redesign costs associated with the OM
4. Introduction of monitoring-induced constraints at the OM-user interface: changes in machine manuals, programming manuals, user procedures, etc.
5. Specification, evaluation, and later refinement of the monitoring techniques

The cost items are readily identifiable, but the benefits derived from the existence of an MM are much more difficult to identify. Two principal benefits are

1. A reduction of the probability of an undetected OM system failure and of the consequently incurred losses to the system user
2. A reduction of the life-cycle operation cost of the OM system by (1) the reduction of the expenditures for manual maintenance and repair and (2) the reduction of down time, during which the OM is not available for use.

The accurate identification of benefits has remained an important issue in all applications of fault-tolerant computing except in the cases in which human lives are severely endangered by faulty operation. The initial cost is readily apparent as part of the procurement costs, whereas the operational costs over the lifetime of the system (where the benefits are accrued) are not readily evident. Even worse is the fact that the organizational unit responsible for procuring a system usually is not responsible for the life-cycle operational costs. In this case the major benefits of monitoring (or general fault tolerance) do not offer a direct incentive to the procurement group and are given a relatively low weight in the competitive selection process.

CONCLUSIONS

In retrospect, three conclusions are offered on the topic of real-time reliability monitoring of computer systems:

1. *In the current practice of computer system design and operation, real-time reliability monitoring has two distinct aspects of application:* (1) as a fault-detection method in fault-tolerant systems and (2) as a maintenance aid in general-purpose installations.

In fault-tolerant systems the usual choice is *m*-copy monitoring: duplex processors in ESS systems,⁹ triplex operation with voting in SIFT²³ and in the Symmetric FT Multiprocessor,¹⁹ four machines in the space shuttle computer.²¹ Exceptions are the test-and-repair-processor (TARP) in the JPL-STAR system³ and the CCU in its successor, the FTSC system,¹¹ which are custom-MMs embedded in their OMs. In addition, error-detecting and error-correcting codes are frequently used. The methods are usually considered to be too costly for general application.

In general-purpose systems, real-time monitoring is slowly entering system designs through the automation of maintenance procedures and through growing sophistication and

automation of operator consoles. Its evolution is accelerated by competition in the GP system market and is limited by cost considerations and by the inertia encountered in the design process. Limited-capacity general-purpose MM's are now found, among others, in the AMDAHL 470V/6,¹ DECSYSTEM-20,¹⁷ and CRAY-1¹⁶ computer systems. Custom-designed MM's are in the STAR-100,¹⁴ BSP,¹² and IBM 4300¹⁵ systems. The last is an especially sophisticated and advanced design.

2. *The exposition of principles of real-time reliability monitoring as presented in this paper had to be distilled from many diffuse sources.*

Fault tolerance literature deals with real-time monitoring as a part of the entire problem, i.e., as one of several fault-detection techniques. Major insights are contributed by the intensive use of real-time fault detection methods and by recent progress in analytic reliability modeling.

General-purpose systems treat real-time monitoring in the maintenance sections of system literature. Other functions assigned to the MM often obscure the extent of monitoring and downgrade its treatment. There is absolutely no attempt to present any quantitative measures of monitoring effectiveness. Implementation details are not revealed because of the competitiveness of the market. Important insights are contributed by the fact that relatively small MM's have to monitor large OMs. Even a sketchy description of the existing monitoring procedures inevitably offers a good look into machine architecture and its limitations.

3. *The time is here for significant advances in real-time reliability monitoring.* This paper presents model outlines for technical specifications of an OM and an MM, based on the discussion in the sections "Properties of the Monitor Machine," "The Monitoring Interface," "Specification of Monitoring," and "Effectiveness and Cost of Monitoring." Furthermore, several areas of research and development are identified that should contribute to rapid progress in the introduction of real-time reliability monitoring as a key attribute of computer systems of the future.

Model Technical Specifications

This section proposes the outlines of two technical specifications, one for the OM and one for the MM.

The OM specification is based on the assumption that the OM most likely is a large-scale, general-purpose computing system, although the specification does accommodate other types of computers. The current status of the OM is assumed to be one of two alternatives:

1. The OM is already in existence as a manufacturer-specified computing system.
2. The OM is to be designed as an evolutionary improvement of an existing computing system.

The outline of a model OM technical specification is shown in Table I. It covers only the monitoring-related information that needs to be specifically identified about a given OM. This information should serve two purposes:

TABLE I—Outline of a model technical specification: Monitoring-related attributes of the object machine OM*

A. Relevant Attributes of the OM

Attributes of the Object Machine OM	Detailed Aspects To Be Identified for Each Item	Examples of Typical Items
1. Built-In OM fault signals and their accessibility to the MM	1.1 Definition of signal 1.2 Classes of faults covered: 1.3 Form of external access: 1.4 Protection of external access:	Parity checkers, Comparators, Temperature monitors, etc.
2. Points accessible for monitoring by the MM	2.1 Definition of point: 2.2 Form of access: 2.3 Entities observed at point: 2.4 Symptoms of faults: 2.5 Classes of faults covered:	Buses, registers, microinstructions, physical variables, etc.
3. Points accessible for introduction of control signals by the MM	3.1 Definition of point: 3.2 Form of access: 3.3 OM condition during access: 3.4 Effect of control signal:	External interrupt, takeover of sequences control, forced readout stopped OM, etc.

B. Cost and Performance Aspects of Monitoring

(To be established for every attribute identified in A.1, A.2, A.3 above)

1. Cost of OM modifications (if any)
2. Cost of OM/MM interfacing
3. Cost of MM monitoring of this attribute
4. Effect on the OM/user interface
5. Proposed measure of effectiveness
6. Testing procedure of effectiveness

*Three versions of this specification can be generated:

1. For the OM as it exists at the present time.
2. For a retrofitted OM, as defined in "Retrofitting Modification of the OM for MM Access."
3. For a redesigned OM, as defined in "Redesign of the OM for MM Access."

1. Provide the technical data to establish whether the given system is a suitable candidate for real-time reliability monitoring, or to choose the most suitable OM system from among competing candidates.
2. Together with the general OM technical specifications, provide the definition of OM properties for the MM technical specification.

The MM specification is shown in Table II. It assumes that an OM specification is given and that an MM is to be acquired, either as an off-the-shelf item or as a custom design. The outline of the model specification identifies the MM properties that need to be selected. Detailed discussion of these properties has been presented previously in this paper.

The underlying assumption is that cost constraints will lead to the choice of an MM that is a smaller system than the OM; that is, either the GP-MM or custom-MM approach (see "Design of the MM") will be made. It must be stressed, however, that the specification is also applicable when the *m*-copy MM approach (see the section on this approach) is

chosen for the MM implementation because of very strict reliability requirements for the MM/OM system. In this case the specification becomes significantly simplified.

The goal of the specification is to describe an MM in terms of the following:

1. The monitoring tasks to be performed
2. The expected effectiveness of monitoring
3. The recovery assistance and other auxiliary tasks required from the MM
4. MM reliability goals and constraints on the interfacing with the OM
5. The maximum cost allowed for MM implementation

Objectives for Research and Development

Some goals for future R&D can be informally summarized as the finding of good answers to the following four questions:

1. How can we design or retrofit the OM to make it very well suited for real-time reliability monitoring?

TABLE II—Outline of a model technical specification: Required attributes of the monitor machine MM

This specification is referenced to a given OM specification.

Attributes of the MM	Detailed Aspects To Be Specified
1. Monitoring effectiveness (ME)	1.1 Goals for the probabilities of detection of faulty operation of the OM (Detailed for each class of Fault Signals and Fault Symptoms given in the OM Specification) 1.2 Required Analytic ME prediction methods 1.3 Acceptance criteria for ME verification by simulation 1.4 Specification of acceptance experiments of OM/MM System 1.5 Procedures for field verification of ME
2. UM recovery assistance (RA)	2.1 Specification of required RA procedures 2.2 Acceptance criteria for RA procedures
3. MM fault-tolerance	3.1 Reliability goal for the MM 3.2 Protection of the OM/MM interface 3.3 Fault-indication requirement in case of MM failure 3.4 Acceptance criteria for MM fault-tolerance
4. OM protection against interference by the MM	4.1 Isolation requirement against faults in the MM (physical and/or man-made) that can cause faulty OM operation or false alarms 4.2 Security and privacy protection requirements for the combined OM/MM system 4.3 Acceptance criteria for MM security provisions
5. MM operational requirements	5.1 MM/Operator interface 5.2 MM/Remote Master-MM interface 5.3 Monitoring data collection and processing 5.4 Other required MM functions (exclusive of 1-4 above)
6. Cost constraints	6.1 Absolute procurement cost limits 6.2 Cost vs. Monitoring Effectiveness (ME) tradeoff boundaries 6.3 Software support and custom program development cost limits 6.4 MM maintenance cost constraints
7. Physical constraints	7.1 Relative locations of OM and MM 7.2 MM operating environment 7.3 Communication links to the OM

2. How do we generate a complete, correct, and unambiguous specification of the monitoring operation (and possibly also recovery assistance) that an MM is required to carry out?
3. How do we implement the OM/MM monitoring interface and the MM itself to meet the goals of reliable MM operation, noninterference with the OM, secure OM operation, and additional application of the MM (for purposes other than monitoring)?
4. How do we predict and measure the effectiveness of the proposed or existing real-time reliability monitoring system (i.e., the OM/MM combination)?

A more specific discussion of the four R&D areas follows.

Problems of OM implementation

Four problems which are related to OM implementation require investigation:

1. Identification of fault types that are observed in contemporary OM systems and classification according to their relative criticality to OM operation
2. Study of the applicability of the various known fault detection methods to improve the coverage of significant faults expected to occur in the OM.
3. Study of the methods for cost-effective and fault-tolerant communication of fault signals and data to be monitored from the OM to the MM. In current systems the communication links are not fault-tolerant. An important part of this question is whether the fault signals should be generated in the OM or whether data should be independently monitored and fault signals generated by the MM itself.
4. Study of the methods of retrofitting dedicated OM/MM links into existing OM machines in which such links have not been provided by the designer.

Specification of the monitoring function

In its full generality, this specification is the cataloging of all the symptoms of faulty operation that should be recognized by the MM immediately upon their occurrence. In addition, some recovery assistance functions may also be specified.

The difficulty of generating the monitoring specification depends on how many fault classes are to be identified by the MM. The area in which research contributions are most immediately needed is the recognition of dynamic symptoms of faults (as discussed in "Dynamic Symptoms of Faults") and the recognition of software design and interaction faults. A second area of concern is the verification that a given monitoring specification is correct, complete, and unambiguous.

Problems of interface and MM implementation

The aspect of the monitoring interface that has been left relatively unexplored is the fault tolerance of OM/MM communication and the protection of OM security from interference via the monitoring link.

The second major problem in this area is the fault tolerance of the MM itself. Currently, the MM in large GP systems is either a minicomputer or a custom-MM, both without fault tolerance provisions. The lack of MM fault tolerance can be accepted as long as occasional OM failures are acceptable. If, however, the MM serves as the hard core of the OM/MM system, which itself has a high reliability requirement, MM fault tolerance becomes a critical issue, and cost-effective methods for MM fault tolerance need to be developed.

Prediction and measurement of monitoring effectiveness (ME)

This problem has remained the least explored of all the aspects of real-time reliability monitoring. Research and development are urgently needed here, because an orderly development of the entire field of reliability monitoring depends on the existence of objective measures of performance.

Work needs to be done in four areas: (1) analytic ME modeling, (2) ME prediction by simulation methods, (3) experimental ME evaluation, and (4) development of methods for ME assessment during field operation.

Research by experiment: An experimental OM/MM system

The most promising vehicle for the focusing of research efforts and the validation of new ideas on real-time reliability monitoring would be an experimental OM/MM system equipped with adequate provisions for continued evolution of the designs and with extensive instrumentation for experimental ME evaluation.

Past experience, specifically the JPL-STAR experimental computer,³ has shown that major benefits accrue from the building of such systems. First, the building of a system does

not allow any difficult issues to be quietly ignored or postponed, as frequently happens in paper studies of system architectures. Second, the real-life demonstration of successful performance removes many practical reservations by system designers and accelerates the acceptance of new ideas into general design practice. Third, a retrospective assessment of the design experience almost inevitably leads to new insights and a better second-generation design.

Because of the intense competitiveness of the computer industry, an experimental project of this nature would be most beneficial to the entire international user community as part of a university computer system research project. Such a location would allow a reasonable access to research results and would allow proposals for experimentation by interested researchers from industry, academic institutions, and government agencies throughout the world.

ACKNOWLEDGMENT

The research reported in this paper has been supported in part by a contract with the National Bureau of Standards, U.S. Department of Commerce.

REFERENCES

1. Ahmdahl Corporation, *470V/6 Machine Reference Manual*. MRM 1000-1, 1976.
2. Anderson, J. E., and F. J. Macri. "Multiple Redundancy Applications in a Computer." *Proc. 1967 Ann. Symposium on Reliability*. Washington, D.C., January 1967, pp. 553-562.
3. Avizienis, A., et al. "The STAR (Self-Testing-And-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design." *IEEE Transactions on Computers*, C-20 (November 1971), pp. 1312-1321.
4. Avizienis, A. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design." *IEEE Transactions on Computers*, C-20, (November 1971), pp. 1322-1331.
5. Avizienis, A., and D. A. Rennels. "Fault-Tolerance Experiments with the JPL STAR Computer." *Digest of COMPCON '72 (Sixth Annual IEEE Computer Society Int. Conf.)*, San Francisco, California, 1972, pp. 321-324.
6. Avizienis, A., and L. Chen. "On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution." *Proceedings 1977 Int. Computer Software and Applications Conference*, Chicago, Illinois, November 1977, pp. 149-155.
7. Avizienis, A., "Fault-Tolerant Computing—Progress, Problems, and Prospects." *Proc. IFIP Congress 1977*, Toronto, Canada, pp. 405-420.
8. Avizienis, A., "Fault-Tolerance: The Survival Attribute of Digital Systems." *Proc. IEEE*, 66, (1978), pp. 1109-1125.
9. *The Bell Systems Technical Journal*, 56 (1977) (special issue on the IA Processor), pp. 119-315.
10. Beuscher, H. J., et al. "Administration and Maintenance Plan of No. 2 ESS." *The Bell System Technical Journal*, 48 (1969), pp. 2765-2815.
11. Burchby, D. D., L. W. Kern, and W. A. Sturm. "Specification of the Fault-Tolerant Spaceborne Computer (FTSC)." *Proc. 1976 Int. Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, June 1976, pp. 129-133.
12. Burroughs Corp. *Introduction to Burroughs Scientific Processor*, 1977.
13. Chang, H. Y., G. W. Smith, Jr., and R. B. Walford. "LAMP: System Description." *The Bell System Technical Journal*, 53 (1974), pp. 1431-1449.
14. Control Data Corp. *Control Data STAR Computer System: Hardware Reference Manual*, 60256000-01, 1970.
15. Cordero, H., Jr. "4341's Infrastructure Is New from the Substrate Up." *Electronics* (November 8, 1979), pp. 110-115.

16. CRAY Research, Inc. *CRAY-1 Computer System: Reference Manual*, 2240004, Rev. B-02, July 1977.
17. Digital Equipment Corp., *DECSYSTEM 20 Technical Summary*, 1976.
18. Downing, R. W., J. S. Nowak, and L. S. Tuomenoksa. "No. 1-ESS Maintenance Plan," *The Bell System Technical Journal*, 43 (1964), pp. 1961-2019.
19. Hopkins, A. L., Jr., T. B. Smith, III, and J. H. Lala. "FTMP—A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft." *Proc. IEEE*, 66 (1978), pp. 1221-1239.
20. Siewiorek, D., M. Canepa, and S. Clark. "C.vmp: The Architecture of a Fault-Tolerant Multiprocessor." *Proc. 1977 Int. Symposium on Fault-Tolerant Computing*, Los Angeles, California, June 1977, pp. 37-43.
21. Sklaroff, J. R. "Redundancy Management Technique for Space Shuttle Computers." *IBM Journal of Research and Development*, 20 (1976), pp. 20-28.
22. Szczena, S. A., and E. W. Thompson. "Modeling and Digital Simulation for Design Verification and Diagnosis." *IEEE Transactions on Computers*, C-25, (1976), pp. 1242-1253.
23. Wensley, J. H., et al. "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control." *Proc. IEEE*, 66 (1978), pp. 1240-1255.

The fault-tolerant 3B-20 Processor

by L. E. GALLAHER and W. N. TOY

Bell Telephone Laboratories
Naperville, Illinois

ABSTRACT

The 3B-20 is the first Bell System general-purpose telecommunication processor designed to meet a broad range of applications. New features such as memory management are incorporated into its design to support a modern operating system. Hardware supports are provided to efficiently execute a high-level language. The 3B-20 Processor is designed to meet ESS reliability requirements and the proven technique of self-checking duplex operation forms its basic architecture structure.

1. INTRODUCTION

The 3B-20 Processor is the first member of a family of processors designed for a broad range of Bell System applications. Its development is a natural outgrowth of the continued need for high availability, real-time control of Electronic Switching Systems (ESSs),¹⁻³ including existing as well as new telecommunication applications. With the rapid growth of integrated circuit technology, the processor architecture is evolving to include as many features as possible to significantly reduce software development and maintenance costs.

The 3B-20 architecture takes advantage of LSI technology to expand its functionality and yet maintain a high reliability standard. Some of the design goals are to

- Achieve highest performance that is consistent with system cost (e.g., provide hardware facilities such as data cache, high speed interrupt stack, address translation cache, and microprogram critical functions which require too much time in software).
- Reduce software complexity (e.g., provide a modern real-time operating system to manage system resources, thereby creating a more useful and more reliable programming environment for the user).
- Reduce programming effort (e.g., provide both an efficient high-level language, such as C-language⁴ and a comprehensive set of software development tools).
- Improve reliability and fault-tolerance (e.g., provide built-in error detection and correction codes, recovery features, and fault diagnostics).

- Improve integrity and security (e.g., implement hardware features such as memory management protection and privilege instructions).

These goals are considered from the viewpoints of both hardware and software architecture in order to realize the most cost effective system for a wide spectrum of applications. Much of the development effort for the past four years has been directed to achieve these goals.

This paper gives an overview of the hardware structure of the 3B-20 Processor. The operating system, software development system, software test facilities, maintenance architecture and other related topics will be presented and published later.

2. GENERAL DESCRIPTION

As indicated earlier, high availability is one of the major objectives in the design of the 3B-20 Processor. The successful deployment and field operation of many ESS systems have demonstrated the simplicity and robustness of duplex configuration in meeting the ESS reliability requirements.⁵ Hence, duplex configuration forms the basic structure for both the hardware and software architecture. Experience gained in the design and field operation of the No. 3A Processor provided valuable inputs for the 3B-20 Processor design.⁶

The 3B-20 Processor falls into the category of a concurrent self-checking design. Extensive checking hardware was incorporated as an integral part of the processor. Faults occurring during normal operation are quickly discovered by detection hardware. This eliminates the need to both run the standby processor in the synchronous and match mode of operation and also the need to run the fault recognition program to identify the defective unit when a mismatch occurs. Self-checking implementation simplifies the maintenance program. Reconfiguration into a working system is immediate, without extensive diagnostic programs to determine which subsystem unit contains the fault. Improved software reliability has become an increasingly important factor in meeting the total system's reliability goal. Furthermore, the self-checking design will permit more straightforward expansion from simplex to duplex, or multiple processor arrangements.

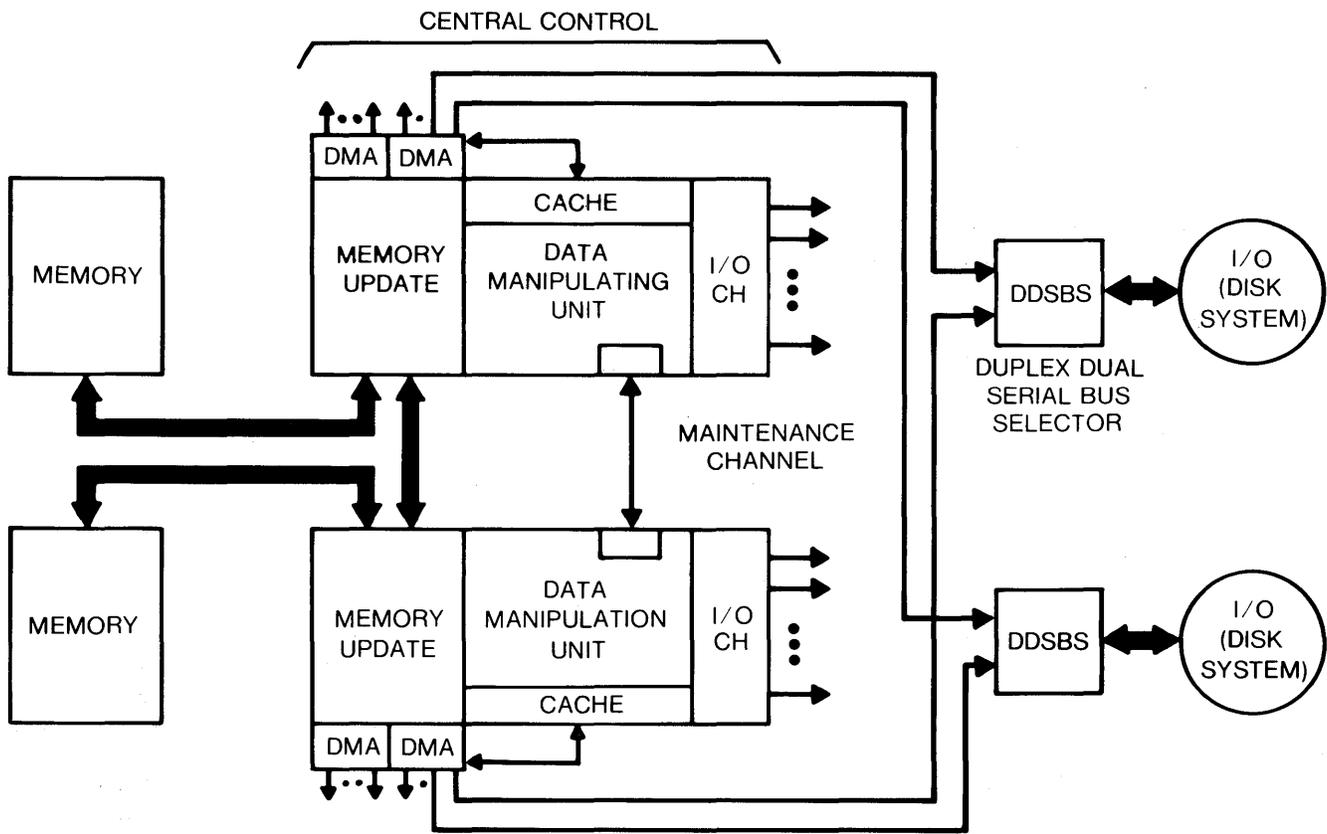


Figure 1—3B-20 Processor general block diagram

a. Duplex Configuration

Figure 1 shows the general block diagram of the 3B-20 Processor. The Central Control (CC), the memory and the I/O disk system are duplicated and grouped as a switchable entity although each CC can access each disk system. The quantity of equipment within the switchable block is small enough to meet the reliability requirement; therefore, the complexity of a recovery program to manage additional working states is avoided. Although each CC has direct access to both disk systems, this capability is mainly used to provide a valid data source for memory reload under trouble conditions. The processors are not run in the synchronous and match mode of operation as is done in early systems.¹⁻³ However, both stores (on-line and standby) are kept up-to-date by the memory update hardware concurrent with instruction execution. This is achieved by having the on-line memory update circuit write into both memories simultaneously when memory data are written by the CC. Under trouble conditions, when the control is switched to the standby processor, its memory will contain up-to-date information without performing a complete transfer from one processor to another. The DMAs interface directly with the memory update circuit in order to have access to both memories. A DMA write also updates the standby memory. In addition, it reads the cache to determine whether the written word is in cache. If it is, the cache word is invalidated to ensure consistent data throughout the cache and memories. Communication between the DMA and the

peripheral devices is accomplished by using a high speed dual serial channel (DSCH). The duplex dual serial bus selector (DDSBS) allows both processors to access a single I/O device. For maintenance purposes, the duplex 3B central controls are interconnected via the maintenance channel (MCH). This high speed serial path provides diagnostic access at the microcode level. It has the capability of transmitting a stream of microinstructions to exercise the processor from the on-line processor or from an external unit for diagnostic purposes.

b. Central Control Structure

The 3B-20 Processor is a 32-bit machine with a 24-bit byte address. Most of the data paths in the CC are 32 bits wide (plus 4-parity check bits). The processor's design is based on the register type of architecture with multiple buses to allow parallel data transfers within the CC. Separate I/O and store buses are provided to facilitate both the concurrent store and I/O operations. A detailed block diagram of the central control structure is shown in Figure 2. The major components and their associated functions are

(1) Microprogram Control (MC)

It provides nearly all of the complex control and sequencing operations required for implementing the instruction set. Mi-

crocode can support up to four different emulations in a 3B-20 Processor. Other complicated sequencing functions are also stored in the microstore (MIS). Its address bus (MSA) is 16 bits. Although this allows addressing of 64K, 64-bit words ($K=1024$), only 16K words can be equipped initially because of physical space allocation. The microcontrol (MC) unit sequences the microstore and interprets each of its words to generate the needed control signals specified by the microinstruction. To optimize the execution of microinstructions, execution time depends upon the complexity of the microinstruction. Each will be allocated only sufficient time required in multiple of 50 nanoseconds to implement the microinstruction. These times are 150, 200, 250 and 300 nanoseconds. The wide 64-bit word allows a sufficient number of independent fields within the microinstruction to perform a number of simultaneous operations. Some common and high runner instructions are implemented with a single microinstruction.

(2) Data Manipulation Unit (DMU)

The arithmetic and logic operations are carried out in the Rotate Mask Unit (RMU) and the Arithmetic/Logic Unit (ALU). These two units are connected in series to comprise the Data Manipulation Unit (DMU). The RMU provides the capability to rotate or shift any number of bit positions from 0 to 31 through a two-stage barrel switch network. In addition, a selection of AND/OR operations can be performed on bits, nibbles (4 bits), bytes, half-words, full words and miscellaneous predefined patterns. The RMU outputs feed directly

into the ALU. Any bit fields within a word can easily be manipulated and processed by the DMU. The ALU is implemented using AMD 2901 bipolar 4-bit processor slices.⁷ The chip contains two key elements: the two-port (A and B) 16-word random access memory (RAM) and the high speed ALU. Data in any of the 16 words addressed by the 4-bit A-address-field input can be used as an operand to the ALU. Likewise, data in any of the 16 words defined by the 4-bit B-address-field input can be simultaneously read and used as a second operand to the ALU. The result can be directed to the RAM word specified by the B-field. To take advantage of the above feature, the internal 16-word RAM is dedicated as general registers. This enables the arithmetic and logical operation involving general registers and/or the output of the RMU to be performed at the optimum speed.

(3) Special registers (SREG)

The 16 general registers reside inside the DMU and are available to the programmer. A number of special registers (SREG) associated with the operation of the CC are external to the DMU. Most of them are not explicitly specified by the instruction. They are characterized by their special dedicated functions with additional inputs from sources other than the internal data bus. Their outputs are used to control and direct the operation of the processor. Some of the special registers are grouped together as a functional block, i.e., error register, program status word, hardware status register, system status register, interrupt register, timers, etc. Others are separated and grouped along with their functional blocks. For example, the store output and address registers are dedicated to memory operation. The program counter is used with program sequencing. In addition, a 32-word RAM is provided within the SREG block which is essentially available only at the microcode level. It is used for scratch-pad space and pre-assigned registers to facilitate and enhance the power of microprogram sequences.

(4) Store interface control

The store interface circuit controls the data from the main memory to the CC for processing. As previously indicated, several special registers are associated with the store interface. Associated with the store address control (SAC) are the program address (PA), the program counter (PC), the store address register (SAR), and the store control register (SCR). Associated with the store data control (SDC) are the store data register, the store instruction register (SIR) and the instruction buffer (IB). The SAC and SDC together make up the store interface which handles the memory addressing, the updating of the program counter, the fetching and prefetching of instructions. The circuit ensures a continuous flow of instructions to be interpreted by the microcontrol unit.

(5) Store address translation (SAT)

Memory mapping is important in the implementation of a multiprogramming system. Address translation hardware is

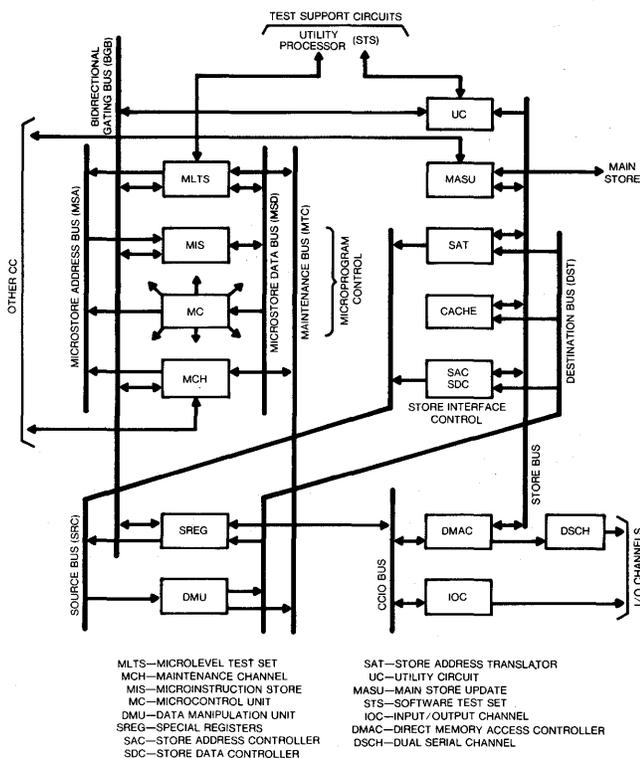
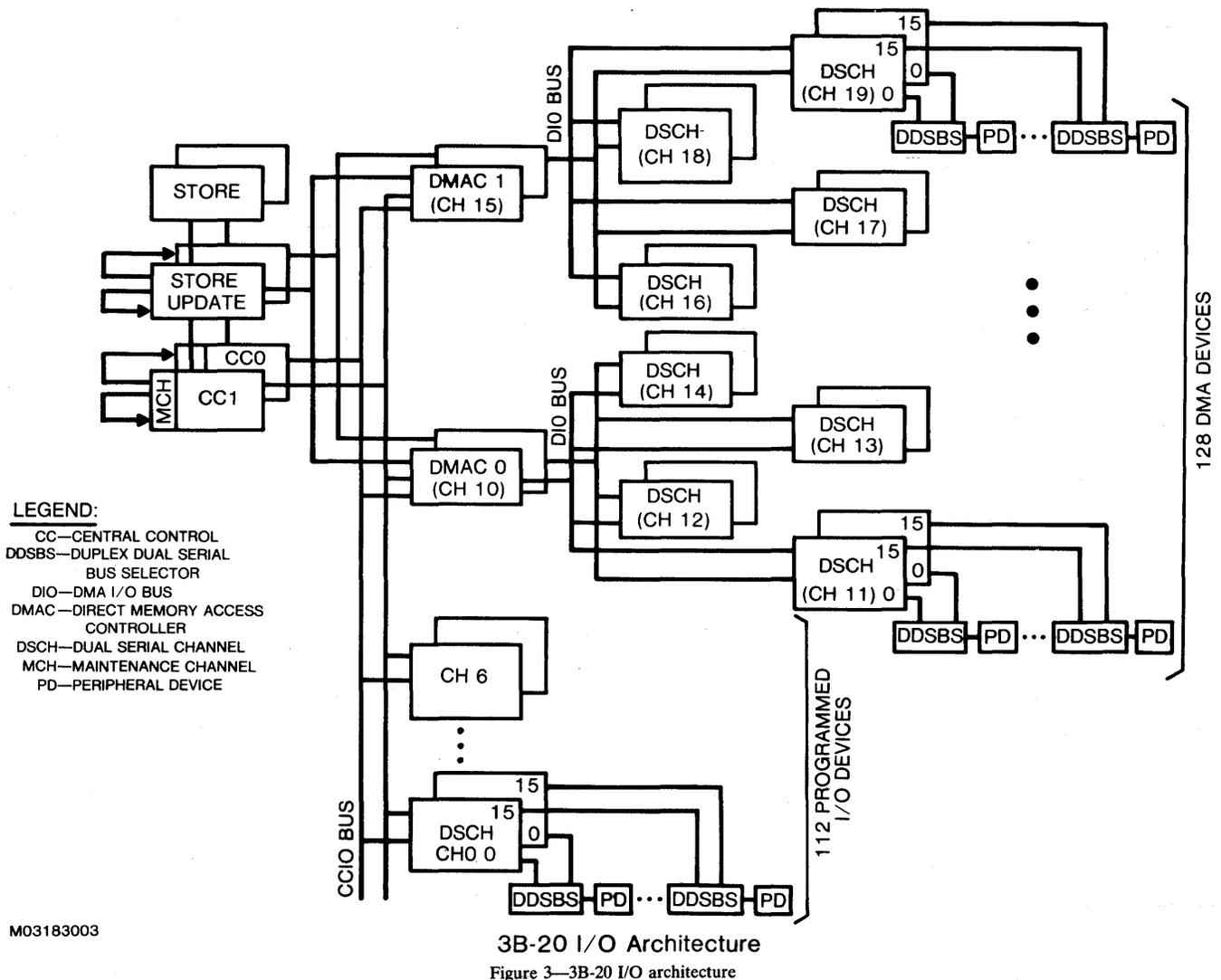


Figure 2—3B-20 Central control



3B-20 I/O Architecture
Figure 3—3B-20 I/O architecture

M03183003

provided to facilitate memory management in a more efficient manner. The store address space is divided into 128 segments, each having up to 64 pages, with a page containing 2K bytes. Both the virtual and the physical address spaces are 24 bits. The complete virtual to physical address translation tables are stored in the main memory. A significant amount of time is required by the CC in the repetitive task of dynamic address translation in using the main store tables. This translation time is reduced substantially by storing the likely-to-be-used physical address translations in a high speed cache-like address translation buffer (ATB). The store address translation facility containing the ATB is the mechanism which provides a mapping between a program-specific virtual address and its corresponding physical address.

(6) Main Store Update (MASU)

The memory update unit provides direct communications to the memories as both DMA and CC attempt to use the

memory. The cross coupling between the memory update units permits the on-line processor to access either memory or both for concurrent write operation.

(7) Input/output interface

The primary communication path between the CC and the I/O channels is through the CCIO bus. It is a local high-speed, direct coupled, parallel bus. Physical slots within the CC housing are allocated to a number of circuits called I/O channels (IOC). The IOCs may be the dual serial channel (DSCH) or the direct memory access controller (DMAC).

(8) Cache

The cache is an optional circuit equipped to improve the overall system performance by reducing the effective memory

access time. The cache is a 4-way set associative memory arrays, each containing 2K bytes, giving a total of 8K bytes.

(9) Maintenance channel

The maintenance channel (MCH), as described in the previous section, provides diagnostic access to the CC at the microinstruction level. The test support circuit is covered in Section 4c.

3. I/O FACILITIES

The I/O facilities are designed to meet a wide range of applications with different needs and capabilities. A modular and flexible I/O communication structure is provided by means of dedicated point-to-point serial channels. The loose coupling of the processor to the peripherals allows considerable freedom to grow and expand the system without physical constraints.

The I/O architecture is shown in Figure 3. Although the channel address ranges from 0 to 19, the processor can be equipped with as many as seven programmed I/O channels and as many as eight DMA channels. Each channel can control up to 16 devices. This means it is possible to equip the system with 128 DMA devices and 112 programmed I/O devices for the maximum configuration. Programmed I/O channels are directly controlled by microcode via the CCIO bus. The DMA facility provides autonomous control of data transfer between the main store and peripheral devices (PDs), thus alleviating the constant need for the CC to process I/O requests. A common DMAC controls up to four DSCHs with each corresponding to a DMA channel.

a. Dual Serial Channel (DSCH)

All standard peripheral devices use the DSCH as means of communication. It is a semiautonomous unit providing up to 16 private serial point-to-point data transmission paths, giving a unique link for each device. Each link consists of two bi-directional data leads, a transmit clock, a receive clock and a request lead. Each of the two data leads operate at 10 MHz for cable distances of up to 100 feet. The normal data transfer operation is a 32-bit word message. The dual serial channel allows concurrent transmission of 16 data bits each to form 32-bit words. In addition to the 16-bit data, the transmission includes 3 start code bits plus 2 parity (one for each byte). When the DSCH operates in a word-transfer mode, its transfer time is about 4.5 microseconds per 32-bit word. The DSCH can also operate in a block transfer mode of transferring 16 32-bit words at a rate of 3.0 microseconds per word as a single sequence. Although the addressing and loading of data is performed under program control, the actual transmission is done autonomously under control of the DSCH hardware.

The programmed I/O operation is controlled directly by the processor, with no need for a device to initiate action on its

own except on DMA operation. The service request functions are incorporated as part of the DSCH to allow a device to signal the processor via the interrupt mechanism. This is implemented over the request line of the DSCH link. Furthermore, the DSCH includes a feature which allows it to be used for processor-to-processor communication in a multiple processor configuration. The interconnection of DSCHs requires each circuit to look like a device to the other so that communication can be initiated by either end of the DSCH.

b. Direct Memory Access (DMA)

The DMA structure provides the facility for moving data between I/O devices and the main memory without having the processor involved in the handling of each transferred word. A DMA unit consists of a DMA controller (DMAC) and one to four DSCHs. Each DSCH provides up to 16 private serial links interfacing with just as many devices. This means a DMA can accommodate up to 64 devices and all may be active concurrently. The I/O structure can be optionally equipped with two DMACs capable of handling up to 128 concurrent DMA-controlled devices.

In virtual to physical address translation, the translation page tables are stored in the main memory. The translation process is the same as that used within the CC itself with the tables shared between I/O and executing programs. Each page is provided with protection bits defining DMA read and write access capability. The maximum transfer size of a single DMA transfer is one segment or 64 2K byte pages of memory. The physical addresses of the 64 pages are not necessarily contiguous. As part of the initialization process for a DMA function, the processor passes the page table pointer to the DMA. It in turn uses the page table pointer to obtain the physical address of the page. As the DMA to memory transfer crosses the page boundary, the DMA circuit automatically accesses the page table to obtain the next physical address for the new page. This is repeated until all pages specified by the DMA operation are completed.

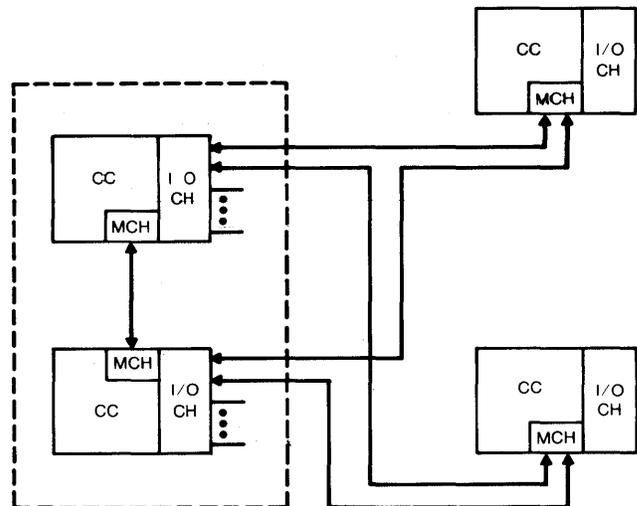


Figure 4—Maintenance channel interconnection

c. *Peripheral Devices*

A broad range of general-purpose peripheral devices is provided for the 3B-20 Processor system. High reliability and maintainability continues to be the design philosophy of the 3B-20 peripheral system. The critical components are duplicated and the software ensures that valid data sources are maintained. Duplex dual serial bus selectors (DDSBSs) permit controlled switching of a working standby device for a faulty on-line device when duplication of peripheral devices is needed. Some major peripheral devices developed for the 3B-20 system are as follows:

(1) **Moving head disk system**

The disk system provides a reliable and flexible mass storage medium for program and data. A backup copy of system programs and critical parameters can be reloaded quickly in the event of a duplex main store failure. The disk system is comprised of the disk file controller (DFC) and the moving head disk drive (MHD). The DFC interprets and executes commands from the processor to cause information transfer from and to the MHD. Each DFC occupies one of 128 channel slots and supports up to 16 MHD drives which are available in 80 and 300 megabyte sizes.

(2) **I/O processor (IOP)**

The IOP provides the control for a wide range of data link facilities and is the most flexible of the family of devices. An IOP supports up to 16 peripheral controllers (PCs) with each being a microprocessor base controller programmed to handle a specific terminal or device. For example, one type of PC is the line controller (LC); each LC can support up to four independent lines (data links or terminals).

(3) **Magnetic tape system**

The tape drive accepts the industry-standard (IBM compatible) 9-track tapes at a density of either 800 or 1600 bits per inch. The tape controller is derived from the basic PC and occupies one of the 16 slots of the IOP.

(4) **Scanner/signal distributor (SC/SD)**

This device is useful in monitoring and controlling power, equipment states, environment conditions, etc. The SC/SD circuit board provides 32 scan points and 32 signal-distribute points. It occupies one of the PC slots of the IOP. When an IOP is fully equipped with 16 SC/SD circuit packs, a total of 512 scan points and 512 signal-distributor points are provided. The cables of the SC/SD to the external circuits can be as long as 1000 feet.

4. SOFTWARE SUPPORT FEATURES

The excessive cost of designing, updating and maintaining software dominates the cost of producing computer systems. Considerable attention has been focused on providing various supports, i.e., high-level language, operating system and software test, in the development of the 3B-20 Processor. The combined software and hardware effort has yielded an integrated and cost-effective system.

a. *High-Level Language Support*

The most common approach to increasing software productivity and reducing software maintenance cost is the extensive use of a high-level language suitable for the applications. The design of the 3B-20 Processor instruction set was based on the fact that C-language programs would dominate the programming environment. Considerable studies were directed to measure and determine the characteristics of a large, diverse sample of C programs. Based on the result of these studies, the instruction set was optimized to be space and time efficient for compiled C programs. Some features provided for the instruction set are concerned with: (1) symmetrical resources; (2) addressing modes; (3) address manipulation; (4) flexible data structure; (5) stack instructions; and (5) procedural instructions.

From the compiler's viewpoint, the most important attribute of a processor instruction set is regularity. It is the key feature needed to abstract the various processor resources for uniform treatment by the compiler. The 3B-20 instruction set includes a wide range of address modes, i.e., indexing, direct, indirect, covering various data structures. The identical treatment of the addressing modes applied to all data types (bytes, half-words, full words, and instructions) without any exceptions, makes it possible to compile compact and efficient codes. The subroutine is one of the most important concepts in software. The principal idea in modular, structural programming is the partitioning of large programs into many small, understandable procedures or subroutines. Efficient instructions have been provided to handle subroutine entry and exit in addition to stack manipulation.

b. *Operating System Support*

The operating system provides a more useful and more reliable programming environment for the users. Higher productivity in application programming is made possible by the high level, simplified facilities provided by the operating system. The duplex multienvironment real-time (DMERT) operating system for the 3B-20 Processor is a general manager of processor, memory, input/output, and processes. Certain hardware has been incorporated into the design to reduce the overhead of an operating system.

As previously indicated, a high speed address translation cache memory called the address translation buffer is provided to reduce the overhead associated with the address translation function. If only one ATB is provided, the ATB must be "flushed" whenever a process switch occurs. This

would incur considerable overhead in the constantly flushing and reloading of the ATB. To improve this situation, the ATB is provided with eight sections, each composed of two sets of 128 entries; each entry can contain the virtual-to-physical translation for one page (2K bytes) of memory. Eight special registers in the CC (SBR0, SBR1, . . . , SBR7) are used to point to eight different segment tables in the main memory. These eight registers define eight address spaces in the 3B-20 Processor and are assigned by software.

Context switching is necessary upon interrupt. A memory stack is provided to facilitate the saving and restoring of the hardware context. In the 3B-20 Processor, a local high speed 8K-byte RAM is provided for this function. The addressing of the stack is part of the kernel virtual address space and has been assigned a fixed segment number and pages 0 to 3. Whenever the kernel virtual address falls into this range, the store operation is directed to the high speed RAM; otherwise, the virtual address is translated by the ATB and pointed to the main memory. The combination of fixed mapping by special circuit and dynamic address translation by ATB allows the high speed stack to be extended into the main memory when the use of the high speed RAM is exceeded.

c. Software Test Support

The software test facility is an option provided at both the microprogram level and the macroprogram level. As indicated in Figure 1, the microlevel test set (MLTS) is attached to the microcontrol section of the central control. It has the capability of direct access to a support computer system for assembling and loading the writable microstore through the MLTS. The primary purpose of the MLTS used in the development of the 3B-20 Processor is for initial debugging and troubleshooting the processor core hardware and subsequently, the microprogram sequences. Features incorporated into the MLTS allow the stepping, freezing, examining and tracing the execution of a microprogram sequence.

The utility circuit (UC), on the other hand, provides a similar set of facilities, except that at the macroprogram level for software debugging and troubleshooting. The UC and its associated software form an extensive Test Utility System (TUS) for software testing. A large number of matchers are incorporated into the UC for the tracing and monitoring of a variety of system conditions so that a programmer can observe and follow the execution of a program sequence. Much of the program debugging can take place in real time concurrent to program execution. The UC thus directly extracts and records information such as transfer trace from the internal data buses, thereby "capturing" the history of the machine while it is running at normal speed.

5. MAINTENANCE FEATURES

Increased support in this area is most appropriate to facilitate a more reliable and more maintainable system, thereby reducing the maintenance cost. It is the labor-intensive expense that is most vulnerable to inflation. For real-time applications, as in the electronic switching systems (ESS), high availability

and uninterrupted operation is essential. This requires the system to function correctly even when a fault is present and maintenance is being performed. The 3B-20 is designed to meet the ESS standard so that the expected amount of accumulated processor downtime does not exceed an average of two minutes per system per year.⁵ Software and hardware are designed to function jointly to insure the reliability objectives are met. Software features include such components as fault recovery programs, audits, and diagnostics. Hardware features include redundant processors, error detection circuits, maintenance access and controls, and diagnostic microcode. These components contribute to the effective maintenance design.

a. Self-Checking Features

Self-checking is implemented in the 3B-20 design for concurrent error detection. The maintenance philosophy is to provide a sufficient amount of hardware to enable detection of nearly all service-affecting single hardware faults. To minimize the potential sources of errors in the main memory, single-bit Hamming correction and double-bit error detection is employed. Most software faults such as memory protection violations, illegal instructions and out-of-range addresses are also detected. Some of the fault detection techniques used in the 3B-20 Processor are

- Parity per byte on data paths throughout the internal CC, memory buses, and peripherals
- Single-bit Hamming correction and double-bit error detection on the main store data
- Duplicated arithmetic and logic unit and other control logic
- Microprogram sequence check
- Parity per byte on data in ATB, cache, microstore, interrupt stack
- Clock timing check
- Decoder check
- Memory management hardware for detecting address error such as a protection violation
- Memory address consistency check (half word and full word)
- "Watchdog" timers for software faults

Faults detected by the processor check hardware are collected together into a single error register. The action taken for a particular fault depends on its impact on the system.

b. Hardware Fault Recovery Features

Fault detection is the first and most important step in realizing a highly reliable system. Almost of equal importance is the rapid recovery by the system. Recovery is achieved by a combination of hardware and software so that continuity is maintained. As soon as an error is detected, immediate action takes place to reconfigure the system into an error-free working system. The recovery process involves two steps: reconfiguration and initialization. To facilitate this rapid recovery

from system faults, the 3B-20 Processor provides the following:

- *A memory update unit*—It couples the on-line memory to the off-line memory. The off-line processor's memory is updated on each memory write operation to allow continuous agreement with the on-line memory.
- *A maintenance channel (MCH)*—It directs the off-line processor to initialize and recover when the on-line processor has detected critical error conditions
- *Initialization microcode*—The nondestructive microcode makes critical recovery decisions when error conditions are detected. This microcode is particularly important if total software sanity is lost.

c. Diagnostic Hardware

Hardware has been integrated into the design of this system to allow a systematic approach for identifying failures via software. This diagnostic software depends heavily on the maintenance channel and its associated circuitry. Its primary function is the diagnosis of one processor by the other. The MCH is an autonomous portion of the processor which, under control of the other processor, can provide information about the state of the machine. It thus exercises the machine at its most basic level by direct access to the microprogram control.

The MCH characteristics are like those of the dual serial channel (DSCH) and the access protocols are compatible with each other. A 64-bit (plus 8-parity bits) MCH command is formatted into two successive 32-bit messages in the standard DSCH manner. This allows a master-slave CC configuration. For example, in a multiple-processor configuration, a master duplex CC can be given maintenance control over simplex slave CCs by the DSCH-to-MCH connection, as illustrated in Figure 4. The dual-access provision of the MCH permits either CC of the duplex master to control the simplex CCs.

Summary and Status

The 3B-20 Processor is a general-purpose high availability machine supporting a broad spectrum of applications. A comprehensive set of software tools and facilities is provided to improve programming productivity and also to reduce the cost of software development and maintenance. The hardware architecture is designed to efficiently support high-level language, particularly C language.

The processor is a 32-bit machine with a 24-bit addressing. Hardware features have been provided to support a modern general-purpose operating system, e.g., virtual-to-physical address translation. Other features include microprogram im-

plementation, emulation capability, high speed data cache, high speed interrupt stack, self-checking circuits, extensive diagnostic access, craft interface for emergency manual control, and high availability duplex operation.

The standard I/O communication between the CC and the peripherals is by means of a dedicated point-to-point dual serial channel, capable of transmitting an effective rate of 20 megabits per second. It can operate in a word transfer mode or a block mode of 16 words per block. The loose coupling of the channels between the processor and the peripherals permits considerable freedom in expanding a system. A wide range of general-purpose peripherals have been provided with the 3B-20 Processor. Some of these are the moving head-disk system, magnetic tape system, high speed printer, scanner and signal distributor and data terminals.

An important provision in the 3B-20 Processor is a complete set of maintenance facilities, from error detection through fault recovery and diagnostics. Approximately 30 percent of the internal CC logic is devoted to self-checking. This allows concurrent error detection and immediate recovery. The combined hardware and software features give an integrated package of maintenance facilities to meet the high ESS reliability requirements.

The development of the 3B-20 Processor started in early 1977. The machine is currently in production at Western Electric Company. About a half-dozen projects requiring high availability, real-time telephone related applications are under development using the 3B-20 Processor. The first application, called NCP (Network Control Point), is expected to be cut into service in mid 1981.

7. ACKNOWLEDGMENTS

The design of the 3B-20 Processor was accomplished through the combined efforts of many designers in Bell Laboratories and Western Electric Company. The authors wish to acknowledge the contributions of all the team members. Their work is summarized herein.

REFERENCES

1. "No. 1 ESS Description," *BSTJ*, September, 1968.
2. "No. 2 ESS Description," *BSTJ*, October, 1969.
3. No. 1A Processor Description," *BSTJ*, February, 1977.
4. Kernighan, B. W., and D. M. Ritchie. *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
5. Toy, W. N. "Fault Tolerant Design of ESS Processors," *Proceedings IEEE*, October 1978.
6. Storey, T.F., "Design of a Microprogram Control for a Processor in an Electronic-Switching System." *BSTJ*, February 1976.
7. AMD. *The Am2900 Family Data Book*. 1979.

Firmware engineering: Methods and tools for firmware specification and design

by WOLFGANG K. GILOI and REINHOLD GUETH

Technical University of Berlin
Berlin, Germany

and

BRUCE D. SHRIVER

University of Southwestern Louisiana
Lafayette, Louisiana

INTRODUCTION

Microprogramming has become the means of implementing the machine language instructions of a conventional computer. In the future, the vertical migration of functions from the software levels of a system to the microprogramming level may become equally important. The vertical migration of functions of a computer is undertaken to realize architectures having improved performance, functionality, reliability, or data security. The increased volume of microcode brought about by vertical migration tends to increase the complexity of the firmware development process and calls for a firmware engineering discipline that provides tools for the design and specification, implementation, validation, and maintenance of firmware. We present a rationale for the specification and procedural design of firmware based on the use of an appropriately defined specification language. The features of such a language and the supporting software system are outlined and demonstrated by the example of an existing APL-based firmware development system.

Firmware Engineering versus Software Engineering

Modern software engineering methodology is characterized by features such as: structured programming, hierarchical decomposition based on the use of abstractions for data, operations and control, execution-independent definition of program semantics, and verification by correctness proof or by testing. The core of any software development system is a formal specification method. Only formal specifications can be tested for consistency, completeness and ambiguities. A formal specification of the program modules of a software system is the starting point for the process of *hierarchical decomposition* in which a complex software system is decomposed into a hierarchy of less complex modules. This results in a more manageable design and validation process that can be carried out at each stage of a stepwise refinement process.

The system decomposition is supported by the use of program and data abstraction, allowing decisions concerning irrelevant implementational details to be deferred until the appropriate stage of the design process. The result obtained after each refinement step must be validated with respect to the given specifications. There exist three major approaches to software specification: (1) the operational method, (2) the denotational method, and (3) the axiomatic method.

Only the denotational and the axiomatic specification methods allow for execution-independent semantic definitions. The operational method requires the introduction of representations that are already at the highest level of abstraction. It thus has the potential danger of "overspecification" i.e., the introduction of implementational details not pertinent at the given level of abstraction. Any specification system must be sufficiently *abstract* and *precise*. By "sufficiently abstract" we mean that no more information is contained in a specification than absolutely necessary. By "sufficiently precise" we mean that all abstractions used are completely and unambiguously defined. In general, it should be possible to define an abstraction without specifying its implementation representation, thus avoiding premature binding.

Specification systems should meet some additional requirements. A specification system should: (a) allow for specifications to be given at all levels of abstraction in the design process, (b) allow mappings to be defined between the objects and functions at various levels of abstraction, (c) lend itself to the application of automated checking of the completeness and consistency of the specifications, and (d) lend itself to the application of computer assisted verification of the system. However, since verification by correctness proof requires an automated theorem prover, validation by testing will still be predominant.¹

When selecting a firmware specification system, an important question to pose is whether software and firmware are similar enough in nature so that existing software development systems (including their specification subsystems, in particular) can be used in the firmware development process

as well. This would be very desirable because the vertical migration of primitives to the microprogramming level suggests a view of firmware design as an extension of software design, involving only additional levels of decomposition. As shown in the following section, however, the firmware differs in some decisive aspects from software. Consequently, what is needed is an *adaptation* of software development methods to the task of firmware development, rather than a simple *adoption*.

Characteristics of Firmware

There are some decisive differences between firmware and software that affect the level of abstraction and the nature of the design steps to be taken in the course of firmware development. Firmware represents a "target machine" by implementing the data objects, operations, and control constructs of the microprogramming language of a given "host machine".² All abstractions of a firmware design must eventually be mapped onto real hardware resources. Both the hardware and logical architecture of real machines are often based on marketing considerations such as cost, hardware performance, backward compatibility (and also on tradition). Therefore, the firmware interface provided by the hardware usually is strongly determined by the underlying hardware architecture rather than by the constructs found at the higher levels of interpretation.

The limited reservoir of hardware resources, especially those available for the representation of data objects, is a serious constraint imposed on the design of firmware. Consequently, a firmware specification system must provide means for introducing bindings of data objects and operations to specific real hardware resources. In contrast, in software design it is usually sufficient to introduce bindings to virtual resources and transformation and leave the mappings of the virtual resources to the real ones to the compiler and operating system. The elementary data objects of firmware design are the contents of certain hardware resources for which the generic name *carrier* is used. The basic data type of the data objects of the firmware is the *bit vector*. More complex data types can be defined to represent arbitrary information items of the machine based on the bit vector.

A common feature of firmware data types, which is also found in high-level microprogramming languages³ is that they define *data representations* and not the semantics of the represented information. Contrasted with software data types these data types have no *predefined* semantics; their interpretation thus is context-dependent. Therefore, firmware specification systems must include models for the information interpretation (the definition of semantics).

The primary goal of software engineering is to produce a reliable and maintainable software system at minimal cost. This is often a very elaborate and costly goal and, if the penalty for failure is high, other important system characteristics (such as execution time) may become of secondary importance. The performance of firmware, on the other hand, determines directly the performance of a computer and, therefore, execution time of the microprograms is of primary concern.

A firmware design process, including its specification system, should offer provisions for stating performance requirements. For example, the firmware design system should allow its user to describe parallel activities of the hardware and to verify execution speed requirements. In the firmware design process, the timing characteristics of microprogram execution by the host machine may have to be considered, and the microprograms must be validated with respect to freedom of timing conflicts and resource contention. This distinguishes firmware from software where the notion of run time generally does not exist.

An operational specification readily allows for the introduction of models for the host machine semantics, and it lends itself in a natural way to the definition of parallel activities and timing characteristics.

The operational specification method initially requires the definition of representations. This is undesirable in software development but is no serious disadvantage in firmware design because the representations of data objects and operations are restricted by the host machine architecture. The operational specification method can be combined with a procedural design process to form a design system having the advantage of allowing for design validations through testing.

OPERATIONAL FIRMWARE SPECIFICATION

For the reasons listed above, the operational specification method is the preferred starting point in the firmware development process.

Operation Abstraction

A microprogram can be interpreted as a state transforming function, $f_{i,j}$ of an abstract machine M_i . The operational specification method requires the existence of a programming language L with semantically well understood constructs. These constructs are used to define the semantics of the operations of M_i . The data objects of M are defined by algorithms formulated in L . To validate the construction, a program written in L is executed, that is, validation is carried out by testing. Therefore, an essential prerequisite of the operational specification method is the existence of an interpreter for L , to perform the state transformations by which the semantics of the operations and constructs of a microprogram are defined.

An operation $f_{i,j}$ of an abstract machine M_i is specified by representing $f_{i,j}$ by a program, $P_{i-1,j}$, to be interpreted by an abstract machine M_{i-1} . $f_{i,j}$ may be a transformation performed on data objects $d_{i,u}$. $P_{i-1,j}$ may be a transformation performed on data objects $d_{i-1,v}$. The program $P_{i-1,j}$ can be validated with respect to the specification of $f_{i,j}$ if the values are defined in terms of the data objects $d_{i-1,v}$. Therefore, mappings $m_{i,t}$ must be given for all types of data objects of M_i so that $d_{i,u} = m_{i,t}(d_{i-1,v})$.⁴ These mappings, together with the specifications for the operations and control constructs of the microprogram, should be formulated in L in order to render them executable by the interpreter for L .

A program P performs transformations in the state space S_{i-1} given by the values of the data objects $d_{i-1,v}$. By virtue of

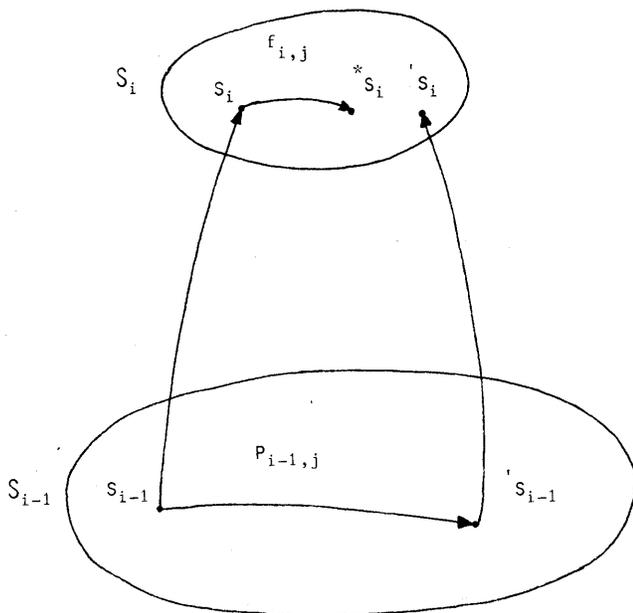


Figure 1—Validation of a program implementing a higher-level function

mappings $m_{i,b}$, S can be mapped onto S_b , i.e., the initial state s_{i-1} and the final state $'s_{i-1}$ of a transformation in S_{i-1} can be related to s_i and $'s_i$, respectively. Conversely, if the execution of operation $f_{i,j}$ by machine M_i leads to the final state $*s_i$, then validation of the program $P_{i-1,j}$ implementing $f_{i,j}$ is to demonstrate that $*s_i = 's_i$, as illustrated in Figure 1.

Data Abstraction

Data abstraction of both user defined and predefined data objects is achieved by using abstract data types.⁵ The definition of an internal representation, in the cases where there is not a predefined representation, is based on the elementary data types of the host machine. To this end, one may start, for example, with a state machine input-output specification of the operations of a user defined abstract data type.⁶ Subsequently, by introducing representations, the axiomatic specification can be substituted by an appropriate operational specification, and validation by execution becomes feasible. This approach introduces internal representations even in cases that are not yet predefined, resulting in an “over-specification” at this level. For the reasons given above, such an approach that may be questionable in software specification seems quite acceptable in firmware specification.

Control Abstraction

Control constructs establish an ordering of execution of operations. A control construct of an abstract machine M_i can be implemented by a program that transforms specific data objects of the machine given at the next lower level. Examples of specific objects are the instruction counter or semaphores.

In contrast, the ordering of operations can be defined in software in a more abstract manner, e.g., through iterators of path expressions. This difference results from the early resource binding of firmware data objects. For example, to migrate the calling sequence of a high-level programming language into firmware, certain hardware resources must be provided in the host architecture from the beginning, such as instruction counter, stack pointer register, and local base register. The extent to which control constructs may occur as part of the firmware design task can be included in an operational specification.

Anatomy of the Specification Language

A language to be employed in the operational specification and procedural design of firmware should primarily satisfy the requirements listed earlier. The language must allow for stating resource bindings, for defining the semantics of all possible interpretations of the elementary data types, and for stating timing specifications. These requirements preclude the unmodified adaption of existing operational software specification systems. Conventional hardware description languages (CHDLs), which may satisfy some of these requirements, exhibit deficiencies that make them unsuitable for the operational firmware specification. CHDLs generally lack the abstraction and expressive power needed for a specification language; in particular they lack the capability to provide precise and abstract specifications of software oriented data objects. Furthermore, they do not allow for the necessary mappings between different levels of abstraction. An important attribute of a realistic firmware specification and design system is for it to allow for a stepwise refinement of the specified firmware product. The similarity between the abstractions and semantic models introduced at the higher levels and the functionality and structure of the host machine should be increased with each refinement step. If the designer finds out at a lower level that certain specifications provided at a higher level hinder the reaching of the desired proximity, the higher level specifications should be modified. This makes firmware design an iterative process in which phases of specification, refinement, and validation alternate. Such an approach raises two stipulations concerning L : (1) the language L should be able to be used interactively and (2) it must allow for definitions of abstractions and models at all possible levels of refinement.

Meeting the second condition allows the intermediate validation steps to be performed on specifications in which different portions of the system are defined at different levels. The representations of all the data objects, functions, and program constructs at the next lower level need not be defined in one step. The user can thus control the complexity of a refinement step. The requirements for the data types, control constructs, and operations that a suitable language L should satisfy are discussed in the following section.

One-dimensional and two-dimensional arrays suffice as elementary data structures in L , because the only data objects to be represented are: memory arrays and register files, represented by boolean matrices; single register or memory cell

TABLE I—Operations on arrays

Clause	Operation
Indexing	Selection of vectors of a matrix or of single array elements
Mask	Selection of subvectors (fields in a vector)
Shift	Shift and fill operation
Rotate	Cyclic shift
Catenate	Concatenation of vectors

contents, represented by boolean vectors; fields or single bits in bit vectors, obtained by mask or indexing functions.

Table I lists the functions of the basic type *array*. Data objects of this type are specified together with their dimension. Explicit reference data types such as "pointer" are not needed; referencing is restricted to the naming of objects and their substructures or elements.

Data dependencies and procedural dependencies establish a partial ordering of the activities of a program. Two activities that are not in a predecessor-successor relationship to each other may be executed simultaneously, provided there are independent hardware resources available for the parallel execution. In microprogramming, where a number of independent hardware resources exist, such parallel execution is exploited for performance optimization. The operations of a program may be grouped into several cooperating processes to indicate asynchronous sequences of activities occurring in independent hardware resources. Therefore, the language should contain constructs for process declaration as well as inter-process communication and synchronization. Such an expressive power of *L* is unusual, but is essential to validate performance requirements as well as the requirement that hardware bindings are conflict free. To illustrate this point, Table II lists the constructs as can be found in the FIT system, a firmware development system that was designed and implemented at the Technical University of Berlin.⁷

A language *L* should facilitate the specification of the host machine. Resource binding of microoperations to functional hardware and control word organization should be supported in a machine independent way. The specification language should allow constraints, such as restrictions on parallel execution of operations, to be expressed.

The interpreter of *L* should include virtual time facilities, so that the time consumed by a specific operation may be specified and its effect on the specified time behavior and performance requirements may be validated. Virtual time specifications also allow the recognition of possible resource contentions among parallel activities and provide a basis for the formulation of synchronization conditions. The time raster used must be fine enough to allow subcycles to be considered. The specification language *L* should be as syntactically simple as possible, containing no more than the minimal number of types, operators, and constructs needed to provide it with the required expressive power. Except for standard operations, such as arithmetical, relational, and logical operations, functions and statements should be named in a mnemonic and self-explanatory fashion. In other words, a specification should be easily readable and intelligible.

THE LEVELS OF REFINEMENT

General

The syntactic framework of the language *L* outlined above allows the user to formulate an operational specification in a firmware design process at different levels of abstraction. Figure 2 lists the levels of such a specification and design process, together with the design decisions to be made and the refinements obtained at each level. The use of specification at the problem-oriented level is primarily motivated as a vehicle for the designer to develop a correct and complete understanding of the design task and demonstrate this to other people involved.

In the data refinement portion of the firmware design process, two fundamental refinements may be distinguished: (1) definition of binary representations of given data objects and (2) mapping the formats of binary data objects to formats available in the real machine. The resulting data objects can be bound to carriers of the host machine.

There are two basic design steps in operation refinement: (1) introduction of operations on binary data objects and (2) refinement to operations similar to microoperations. The resulting operations can be bound to operations of the host machine and thereby to functional units of the hardware.

The Machine-Independent, Hardware-Oriented Level

At the *machine-independent* level, microoperations are specified in terms of carrier-to-carrier transfer statements. A state change in a carrier is brought about by transferring a new value to it, and a data transformation is performed by letting

TABLE II—Control Constructs in *L*

Clause	Purpose
→FUNC	Procedure head
→ENDFUNC	Procedure end
→CALL	Procedure call
→RET	Procedure return
→GOTO	Unconditioned jump
→IF ...→DOTO ... (→ELSE)	Conditioned branching
...→FI	Alternative branching
→CASE ...→ESAC	Iteration clause
→REPEAT ...→UNTIL	Repetition clause
→FOR ... FROM ... TO ...→ROF	Opens declarative part of process
→DECLARE	Opens procedural part of process
→PROCESS	Reiterates process execution
→FOREVER	Terminates process execution
→END	Parallel execution of operations
→PAR	Switches signals and state indicators on or off respectively
→ON	Wait for signal to become true
→OFF	Critical section lock
→WAIT	Critical section unlock
→LOCK	Initiation of a process on behalf of another process by "no-wait send"
→FREE	
→INIT ...→EVENT	

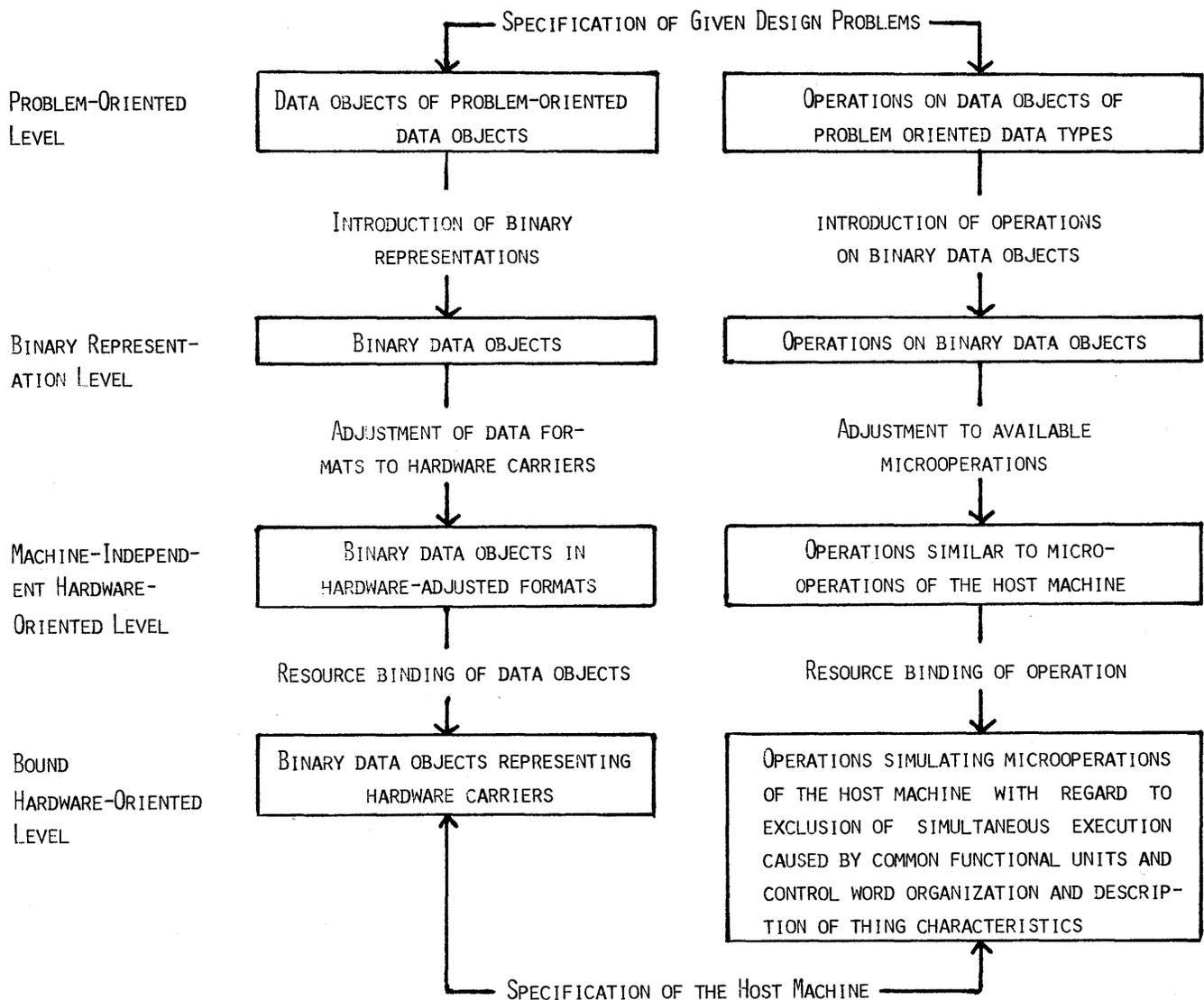


Figure 2—Specification and fundamental design decisions in data refinement and operational refinement

a value on a register transfer “pass” through an abstract functional unit. If the identity operation is included into the set of operations that may be carried out on a register transfer, then *any register transfer defines an operation and vice versa*. The register transfer thus becomes the basic microoperation of a machine. Within a microoperation, there may be a number of operations performed by appropriate functional units. At the machine-independent level, however, abstract functions rather than actual functional units are specified. Table III presents a sample of possible operations of a computer hardware. Besides such *data-transforming operations* there also are operations, to be executed within a microoperation cycle, to make possible the register transfers, i.e., establish the necessary connections between carriers and functional units via switching facilities such as multiplexors and gates. At the machine-independent level, these *path-controlling operations* are not explicitly specified but are implied by the register-

transfer statements. There usually is no semantic connection between the operations within one microinstruction. Only the microoperations cause state changes of the machine, and it is the control constructs in a microprogram which determines the order of register transfers that must take place in the execution of the underlying algorithm. It is a specialty of microprogramming that several microoperations may be under execution during a given microinstruction cycle and, therefore, microoperations may be executed concurrently, partly overlapping or completely in parallel. The control constructs listed in Table II in connection with the timing specifications allow the designer to deal with such concurrencies. Timing specifications allow the designer to deal with such concurrencies. Timing specifications designate the execution time of operations as a basis for synchronization and performance validation. Performance validation is carried out by executing programs written in *L* which execute the same func-

TABLE III—A sample of abstract hardware functions

Operation	Notation	Comment (in APL notation)
Addition or Subtraction	A <i>ADD2</i> (B) <i>CARRY</i> C	A, B are 2's-complement numbers, C is the carry-in
Multiplication	A <i>MULT2</i> B	$A \times B$, 2's-complement operands
Division	A <i>DIV2</i> B	$A \div B$, 2's-complement operands
Greater than	A <i>GT</i> B	$A > B$
Greater equal	A <i>GE</i> B	$A \geq B$
Equal	A <i>EQ</i> B	$A = B$
Less equal	A <i>LE</i> B	$A \leq B$
Less than	A <i>LT</i> B	$A < B$
Not equal	A <i>NE</i> B	$A \neq B$
Logical and	A <i>AND</i> B	$A \wedge B$
Logical or	A <i>OR</i> B	$A \vee B$
Logical not	<i>NOT</i> A	$\sim A$
Logical nand	A <i>NAND</i> B	$A \wedge B$
Logical nor	A <i>NOR</i> B	$A \vee B$
Rotation	K <i>ROT</i> B	Rotates argument B k steps for $k > 0$ to the left for $k < 0$ to the right
Shift and fill	K <i>SHIFT</i> (B) <i>FILL</i> C	k = number of shifts, B = argument, C = fill element
Concatenation	A <i>LINK</i> B	A, b ($p(A, B) = (pA) + (pB)$)
Portion of a vector	A <i>MASK</i> i1 i2	$A[i1 + i(i2 - i)]$ (0 - origin!)
Incrementation	<i>INCR</i> A	$a \leftarrow A + 1$
Decrementation	<i>DECR</i> A	$A \leftarrow A - 1$
Binary-to-decimal conversion	<i>DEC</i> B	B = binary number in 2's-complement, result = decimal equivalent
Decimal-to-binary conversion	K <i>BIN</i> D	D = decimal integer, result = 2's-complement, k-digit binary number representation
Index generator	<i>INDX</i> i1 i2	$i1 + i(i2 - i1)$ (0 - origin!)
Register initialization	<i>SET</i> "list of register names"	all positions in the named registers are filled with 1's
Register initialization	<i>RESET</i> "list of register names"	all positions in the named registers are filled with 0's

tion (the same machine operation) at different levels of abstraction. The time consumed by the execution of a program, $P_{i-1,j}$, is measured against the time specification of the function $f_{i,j}$ to which it is related by the mapping $m_{i,t}$. In the same manner it can be tested whether a certain operation is executed within a given subcycle.

The Bound Hardware-Oriented Level

At the bound level an appropriate model of the real machine must be given as a prerequisite for the firmware design. The documentation of the real machine behavior, as given by the hardware designer, must be sufficiently precise, complete, and understandable in order to provide the required model without major deficiencies. Deficiencies in the model may not be recognized until microprograms are executed on the real machine. An essential assumption for any formal specification method is that there exist a sufficiently well-defined model of the operations, data objects, and control constructs. At the hardware-oriented levels of a firmware design process, the primary question is not whether the design should be verified by correctness proof or validated by testing, but whether the underlying hardware behavior is sufficiently well-defined. In the design process depicted in Figure 2, this problem is mitigated by the fact that the method allows for arbitrary mod-

ifications and refinements of the model of the real machine and the consequent corrections of design decisions made at the higher levels. Programs constructed at the higher levels of firmware design have the inherent potential of parallel execution, limited only by the existing data dependencies among the operations. At the lower levels of the design process the possibility of parallel operations are further reduced by the constraints given by the microinstruction format and the need to avoid resource contention. The mappings introduced by the bindings of the higher level data objects to carriers of the real machine consist in addressing functions or identities. The operational specification can introduce the highest possible degree of parallelism and subsequently restrict it according to resource bindings and other constraints. In connection with time specifications and mappings, this allows the designer to carry out performance estimates for alternative resource bindings, providing data to optimize the system. The real machine performs state changes by the clocking of registers. Therefore, in a microprogram specification at the bound level, the register-transfer statements of the machine-independent level are substituted for by clock statements. The data objects are bound to real carriers, and the abstract operations are bound to real functional units of the host machine, such as ALUs, shift units, multiplexors, gates, and decoders. Since most of these functional units have a control input in addition to the data inputs, the microprogram specification must comprise

specifications pertaining to the generation of the control signals. With each clock statement, the time elapsed since the occurrence of the last clock statement must be specified. These timing specifications, in connection with the clock statements which mark cycle points of state transitions, indicate the beginning of cycles and subcycles. An example of a machine-independent level specification of a microprogram and its refinement to a bound level specification can be found in Giloi et al.⁷

CONCLUSION

The main goal of software engineering is to implement a reliable and maintainable software system that provides a required functionality. The performance aspect may be of secondary importance. However, performance is the major goal of firmware design. Emulators are, in general, less complex than software systems and cannot be designed in a straightforward top-down fashion, since there usually are rigid constraints to meet as given by the underlying hardware. In our opinion, the primary problem in firmware design is not so much "decomposition of complexity" that is important in software design but rather the problem of resource binding. Differences between firmware and software must be considered when one adapts methods and techniques of software engineering to firmware engineering.

In particular, these differences affect the decision whether to take an axiomatic or an operational approach to firmware specification and design. Unquestionably, the axiomatic approach offers the highest possible degree of abstraction. However, such a high degree of abstraction can be utilized only at the higher levels of firmware design, e.g., in vertical migration problems. At the lower levels of firmware implementation there usually exist many predefined constraints to be taken into account concerning data representations, resource bindings, and timing problems. Models for the definition of resource binding, hardware-related semantics and timing characteristics, however, do not exist in the axiomatic specification method in a practically usable form.

It has been predicted that eventually whole application programs rather than just individual functions will be migrated into firmware. In this case, it would be highly desirable to have a uniform software/firmware development system. Such a uniform approach, in which the firmware design is to be-

come an additional refinement step in the software design process, would not support the use of an operational firmware specification. However, we do not expect the large-scale migration of entire application programs into firmware to materialize. The vertical migration of application programs offers a performance advantage only if the control store is considerably faster than the main memory. In future LSI based architectures, this condition will not hold true. The firmware development system based on an operational specification and procedural design method presented in this paper has been implemented at the Technical University of Berlin. The core of the interactive system, named FIT (Firmware Implementation Tool),⁷ is a mnemonic, self-descriptive, and extensible specification language. A *simulator* interprets the microprogram specifications given in FIT at various levels of abstraction and executes them. Simulation runs include time bookkeeping and allow for performance measurements. A report generator provides the user with a numerical and/or graphical presentation of the results obtained. The realization of a microcode generator for a particular host machine, which will automatically translate the hardware-level specification of a microprogram into microcode, seems feasible, but remains yet to be done. The system is implemented in VSAPL and thus portable to computers supporting that language.

REFERENCES

1. Miller, E.F. (ed.). Special issue on "Program Testing," *COMPUTER* 11,4 (April 1978).
2. Davidson, S., and B.D. Shriver. "An Updated Overview on Firmware Engineering." In W.K. Giloi (ed.), *Firmware Engineering*. Berlin-Heidelberg-New York: Springer-Verlag, 1980.
3. Davidson, S., "Design and Construction of a Virtual Machine Resource Binding Language." PhD Dissertation, Computer Science Department, University of Southwestern Louisiana, Lafayette, Louisiana, August, 1980.
4. Robinson, L., and K.N. Levitt. "Proof Techniques for Hierarchically Structured Programs." *CACM* 20,4 (April 1977), 271-283.
5. Liskov, B., and S. Zilles. "Specification Techniques for Data Abstraction," *IEEE TRANS. SOFTWARE ENGINEERING* 1,3 (March 1975), 7-19.
6. Liskov, B., and V. Berziens. "An Appraisal of Program Specifications." In P. Wegner (ed.), *RESEARCH DIRECTIONS IN SOFTWARE TECHNOLOGY*, MIT Press, Cambridge, Mass. 1979.
7. Giloi, W.K., P. Behr, and R. Gueth. "FIT—A System for Firmware Specification, Implementation, and Validation." In G. Chroust (ed.), *PROC. OF THE IFIP WORKING CONF. ON FIRMWARE, MICROPROGRAMMING, AND RESTRUCTURABLE HARDWARE*, North-Holland, Amsterdam 1980.

New directions for micro- and system architectures in the 1980s

by HAROLD W. LAWSON, JR.

Linköpings University
Linköping, Sweden

ABSTRACT

After approximately 30 years experience with microprogrammed control concepts, we find ourselves at an interesting turning point leading into the 1980s. Experience with various microprogrammed control techniques has been obtained, some experience has been obtained in recent years in the redistribution of functions between various software and microarchitectural levels and we have a better idea of methods of timing and synchronization. This knowledge together with the possibilities brought on by VLSI will meet in the 1980s and have several interesting effects upon system architecture at microarchitecture and higher architectural levels.

We shall consider some of the opportunities and limitations of VLSI and their potential effects upon microarchitecture followed by a view of the future utilization of "programmed logic." More specifically, we shall be considering the organization of logic, synchronization, microarchitecture characteristics, target machine properties and special purpose microprogrammed machines. Further, we shall touch upon relationships to function distribution, computer aided design and the possibilities for architectural synthesis.

INTRODUCTION

The forecasting of what will happen in the future, particularly in this dynamically changing field, can be dangerous. However, experiences with microprogrammable computers and possible directions as a result of ongoing VLSI technology can lead, at least in this author's point of view, to an indication of some possible new directions in the area of microarchitectures and system architectures.

Let us first clarify some of the important terminology and concepts used in this paper. When we refer to *architecture*, we mean the activities of design related to specifying the eventual assembly of available or new "building blocks;" thus, constructing a *system* or *subsystem*. In the computer environment, we can observe many levels of systems and subsystems, as indicated in Figure 1. Traditionally, we divide the computer system architecture into software and hardware parts. The architect's designing of a total computer system is the composite of many sub-architectural activities. Observe that we can, and shall in this paper, also refer to architectural design

activities as the *organizing* and/or *structuring* of systems and their parts.

In order to improve our view of global computer architectural activities, we shall abstract a common denominator of the activities. We shall view the building blocks used at all levels as *processes* and the architecture that is evolved at each level of the system and the system as a whole as a *system of cooperating processes*. For example, gates are processes, which when combined into combinational and sequential circuits form a system of cooperating processes. Likewise, collections of microsubroutines, subroutines (procedures, functions, etc.) of higher levels are processes that are organized into systems of cooperating processes.

In observing Figure 1, we see that the cooperating processes of higher levels are dependent upon the cooperating processes of a lower level. This relationship has been referred to as an *interpretive hierarchy*.¹ Going a bit deeper into this subject, we can state that "*One level's system of cooperating processes forms a processor for the next higher level*".

Consequently, we can say that architects design *processors*. This point of view is valuable in forming an understanding of computer systems² as well as in considering the new concepts presented later in this paper. The question, what is *micro* to what in a system architecture, is not always clear, and in order to achieve a uniformity in structure between various architectural levels, perhaps we should use the denotation *programmed logic* as a general concept that can be applied at all levels. Further, we can use the term *base logic* to refer to the lowest level of processes (nonreducible hardware actions in this case). This point of view is quite useful, since the realization of programmed logic can indeed vary among architectural levels (e.g., *Programmable Logic Arrays*, microprograms, conventional programs written in lower or higher level languages). We shall also return to this more general notion of programmed logic in later sections of this paper.

The majority of the comments made in this paper are related to microarchitectures (see Figure 1) that are designed to emulate conventional target system instruction repertoires and interpret HLL (higher level language) oriented instruction sets for uni-processor systems. However, some brief consideration will be given to the role of microarchitectures in special purpose systems. In a later section, we shall use the process-processor concepts introduced above to consider some possible new directions in the VLSI context. With these

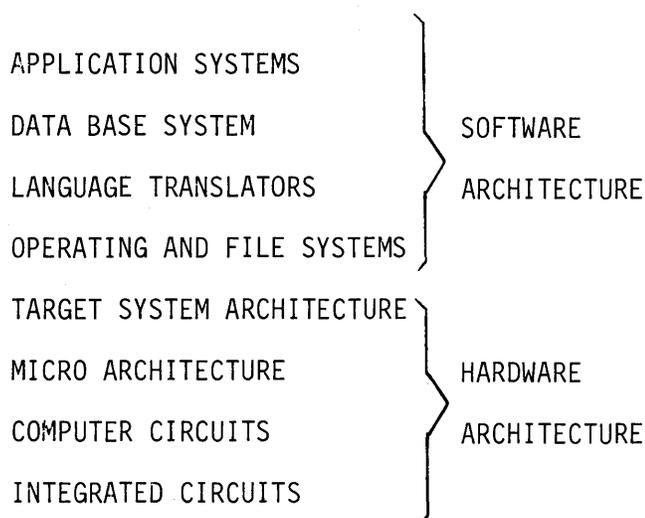


Figure 1—System levels

points in mind, let us first consider the existing views of microarchitectures and their possibilities for obtaining parallelism in microprogram execution.

MICROARCHITECTURE STRATEGIES

Many of those active in the field divide microarchitecture strategies into *horizontal* (minimally encoded) and *vertical* (highly encoded) strategies.^{3, 4, 5} Further structuring, such as in the Nanodata products,⁶ use a combination of microprograms and nanoprograms with corresponding architectural levels. Within these strategies, the introduction of parallelism is made by the designer(s) of the micro- and/or nanoarchitecture. Activation and termination of parallel base logic is accomplished in relationship to synchronous (single or multi-phase) clock cycles.

Let us briefly consider the principles of these traditional microarchitectural approaches, which will be essential to our understanding and later comparison to an alternative approach. In the horizontal strategy, the control of hardware processes is determined by eliciting the activation of base logic processes from a micro control word as exemplified in Figure 2. Each field of the micro control word can cause the activation of a process and, in fact, several processes can be activated simultaneously (parallel processes) during a single clock cycle or clock cycle phase. The designers of the processes must assure that the execution of their hardware base logic processes (circuits) can be completed within the limits of the clock requirements. This factor determines performance and frequently causes problems for designers when their logic does not fit into the timing requirements. "Glitches" are frequently introduced at this point to solve "the problem". In this au-

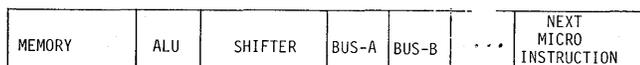


Figure 2—The horizontal microinstruction approach

thor's opinion, experience has shown that this has led to complicated microarchitectures that are difficult to understand and consequently difficult to microprogram, test, maintain, etc.

The vertical strategy does not provide any inherent mechanism for achieving parallelism in the microarchitecture. Some architectures try to achieve the parallelism by dividing the activities into processing and control parts. Where appropriate non-conflicting combinations of consecutive microinstructions appear, they are executed in parallel (see Figure 3); the next two microinstructions in this case activate different base logic and thus can be executed in parallel. This approach was used in the MLP-900.⁴

Simple one-stage pipelining of microinstructions is another approach to improving performance, using the vertical strategy and is commonly found in bit slice microarchitectures.⁷ The use of these commonly known microarchitecture strategies leads to what this author refers to as *unnatural parallelism*, that is, man-made parallelism. The possibilities for parallelism are designed into the architecture and must be thoroughly understood by the micro/nano programmer to be exploited; often, with significant difficulties due to the magnitude and/or complexity of their structure.

It is possible to introduce *natural parallelism* at the microarchitecture level. That is, parallelism that is achieved automatically by the nature of the implemented instruction interpretation mechanism in relationship to base logic processes that have not been designed with the type of design timing constraints mentioned earlier. The microprogrammer, while possibly being aware of potential parallelism, is not faced with the complexities of understanding and exploiting man-made parallelism that has been determined by stringent clock time-requirements and possible "glitches." This natural form of parallelism was used successfully in the Datasab FCPU.^{8,9,10} Since this latter form of parallelism at the microarchitecture level has not received widespread description in the technical literature and since it has an important extrapolation to potential new directions of VLSI-based microarchitectures, we shall devote the next section to considering the general principles behind this strategy. For those readers not familiar with the FCPU, the collected published papers¹¹ are available from the author upon request.

THE ASYNCHRONOUS APPROACH TO MICROARCHITECTURE

The use of asynchronous control as an architectural technique is well known and is well developed, particularly for input/output systems. The advantage is that asynchronous control provides a convenient mechanism for coupling processes that are executed by their processors at different rates of speed. It is possible and, as we shall see, useful to use this architectural strategy at the microarchitecture level. This is precisely what was proved by the implementation of the FCPU. In this section, we shall not present the exact FCPU methodology; instead, we shall consider a general model of the approach.

The model of this approach to microarchitecture is given in Figure 4. We divide the conventional components of a microarchitecture into processes. The processes cooperate by pass-

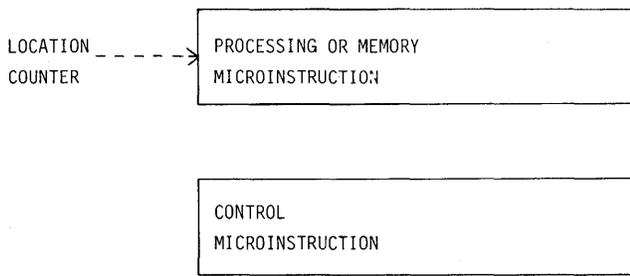


Figure 3—A vertical microinstruction approach

ing “data” and status information to each other in producer-consumer relationships over synchronization registers. Microinstructions are distributed to the execution process by way of a microinstruction pipeline mechanism that prefetches microinstructions from the control memory. Let us consider an example of the activation of the processes by considering three microinstructions that fetch the contents of a memory location and add these contents to the contents of a register in the ALU and test for overflow.

```

READ FROM<memory address>TO<synchronization register-1>
ADD <synchronization register-1> TO <ALU register> STA-
TUS <synchronization register-2>
TEST OVERFLOW <synchronization register-2> TRUE
<overflow label>
    
```

This example illustrates the point that producer-consumer relationships exist between the base logic processes and that the designers of the Memory, ALU, and Control processes are not inhibited in their design by having to meet a stringent clock cycle requirement. Their base logic process executions take as long a time as (naturally) required. The asynchronous execution permits prefetching and parallel microinstruction execution to the extent that progress is not directly inhibited due to waiting for previous results (including status).

The approach indicated here involves a centralized microinstruction control and distributed base logic processes execution. Other models with less central control of microinstruction supply and truly autonomous distributed control amongst the processes may be possible; however, the full implications of this distribution in the context of conventional and higher level language oriented instruction-set interpretation remains a problem for further research. The use of producer-consumer relationships in the operating system milieu is well known and is also, in that environment, a means to synchronize process execution where processes execute at different rates of speed. An additional area for research concerning asynchronous control for microarchitectures involves the use of monitor concepts as, for example, described by Hoare¹² for handling more than the one-level producer-consumer relationships (as described above) and the possible introduction of several processes of equal capability (e.g., several ALU processes). This asynchronous form of microarchitecture control has its cost in synchronization logic. However, even at the time the FCPU was designed and constructed (1971-73) when custom LSI was not widely available, the partitioning of the design, implementation, and testing provided significant project related advantages that have been docu-

mented by Lawson and Magnhagen.¹³ With VLSI, such mechanisms become trivial and, in fact, desirable and probably necessary, as we shall show in the following section. The units of an asynchronous microarchitecture like the FCPU are indeed *self-timed circuits*, and corresponding advantages for the design, implementation and testing for VLSI circuits have been described by Seitz.¹⁴ These are the techniques that allow for what has been referred to as natural parallelism.

NATURAL PARALLELISM IN THE VLSI ENVIRONMENT

The availability of VLSI and especially CAD tools for VLSI will lead to the possibility of realizing new approaches to the programming of base logic. Synchronization mechanisms, for example, implemented by newly defined logic cells will relieve the problem of costly implementation by standard MSI and SSI components.

A major research area for the future will involve methods for evaluating alternative “processor” architectures to support the processes to be implemented. In many situations, particularly when the process is not complicated, it may be useful to construct a processor per process. In other cases; where the process is more complicated and/or a family of related processes are to be supported, more sophisticated programmed logic processors may be justified. This programmed logic can then be shared (multiplexed) amongst the higher level processes to be executed by the processor. One trend that may well evolve from these advances is that complicated general purpose processors of the type we have built earlier may well be replaced by a “network” of special purpose processors. Note that this trend is already underway with the expansion of computer network technology. What is being

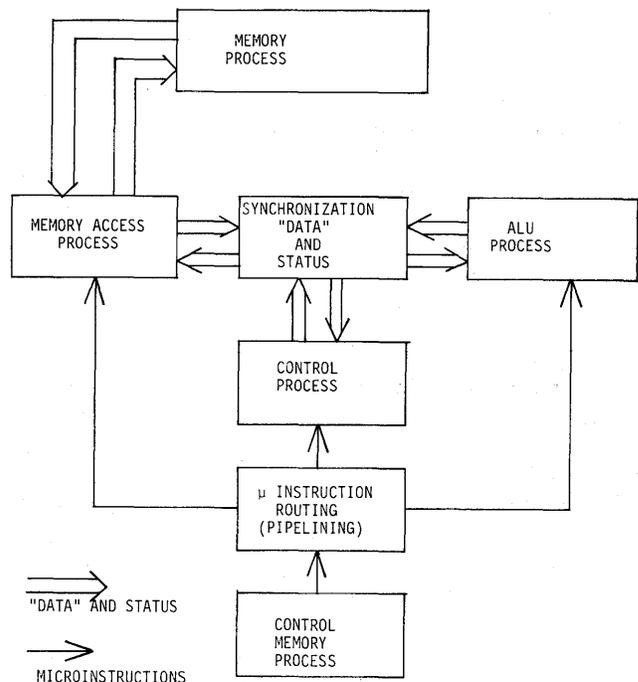


Figure 4—The asynchronous approach to microarchitecture

pointed to here is that this trend can well be extended down into chip architectures. The PLA as one keystone of VLSI programmed logic is already well established owing to the regularity of physical structure. It is clear that the use of this programming form for base logic will continue to be an important factor for low level processor realization and in many cases will provide an alternative to more conventional micro-architecture control strategies.

How about the organization of several base logic processes, particularly when the processing of less regular "data" (including instructions) is required? Shall we resort to the use of unnatural parallelism with all its possibilities for creating complicated structures that are difficult to microprogram, test, maintain, etc.? In this author's opinion, that approach asks for trouble. With VLSI, one will attempt more comprehensive architectures, and to be able to design, implement, and test these architectures, one needs to reduce, at all costs, the complexity of the structuring of processes and their programmed base logic processors.

It seems quite clear that VLSI design rules such as the IBM LSSD (Level Sense Scan Design)¹⁵ will be essential for understandability and testability. With this strategy, feedback loops in circuit logic are not permitted. Transformations take place from register to register in well-defined steps. Logic "tricks" are not possible. The step from these design rules to achieving true possibilities for natural parallelism is not a long one. This basic type of asynchronous chip organization has already been proposed.¹⁶

A general view of a chip organization for realizing natural parallelism (cooperating process execution) is shown in Figure 5. Here we see that logic close to the PADS (external chip connections) is used for PAD IN/OUT control. In this environment, signals are not delivered directly to and from the processors in their execution of processes but are delivered to and from synchronization registers. Consequently, the use of external connections to other components is multiplexed among the on-chip implemented processes. External connections to other components of similar skeleton structure would permit processes to cooperate in a logical system environment across physical component boundaries by a system bus. Two internal busses (for "data" and control signals), the V (vertical) and H (horizontal) busses, are proposed in this chip architecture, which can be used for connecting processors and thus the processes they execute to the synchronizing memory. Placement algorithms in the CAD system can be used to attach the processors to the most convenient bus. Processes dedicated to monitoring and testing must be included in this highly structured environment so that internal "data" and status can be delivered in and out of the chip for testing and failure analysis.

While we have, in the previous section, been discussing the use of asynchronous control for microarchitectures, which could be implemented using this VLSI strategy, the reader will observe that the structure proposed here can well apply to process realization in general. It is quite possible that with appropriate CAD tools and appropriate descriptive languages for expressing base logic (or convenient synthesis to base logic), newer forms of systems architectures may well, in many situations, avoid a microarchitectural level of the form we have been considering, be it horizontal or vertical (syn-

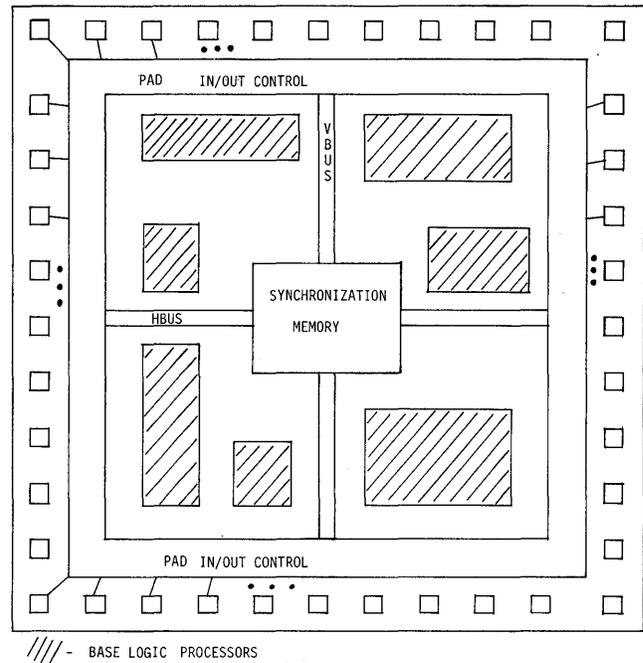


Figure 5—Chip organization for natural parallelism

chronous or asynchronous). *Programmed logic* for processor realization will only be introduced when deemed necessary for supporting families of similar processes and/or when advantages for design, testing, maintenance, etc. can be obtained.

Diverging for a moment from the subject of micro-architectures for emulation and HLL instruction interpretation, we can observe that interesting applications of VLSI technology involving systolic algorithms¹⁷ have been proposed where there is inherent natural parallelism in the application that permits the design of regular arrays of logic, normally with homogeneous "data" pipelined synchronous control of processing. Unfortunately, emulation and HLL instruction interpretation do not fit into this class of "application" parallelism. Process execution in this environment is extremely heterogeneous. However, with asynchronous approaches as described above, we can hope to put some order into these complex processes.

To give a better idea of the new concepts introduced in this section, let us consider some processes of varying complexity with an idea of the type of processors required.

<i>Process</i>	<i>Processor</i>
Real time clock	Sequential circuit with registers and random logic.
Bus arbitration	Programmable Logic Array
Input/Output control	Programmable Logic via a static "control memory"
High level language machine interpretation	Programmable logic via a dynamic "control memory"

The key point to be made here is that the processor architecture complexity need only be as complex as required for the processes that it supports (executes). We can observe that the

VLSI architecture concepts presented here provide for a form of distributed control. We have discussed cooperating process execution within a chip and among two or more physically near chips. The principles however, can well apply, in general, to remotely located network process execution. Remember, asynchronous control is the natural mechanism for permitting process execution at their own rates of speed, regardless of their physical location. Of course, this is already known and done in the world of packet-switched communication networks.¹⁸

IMPLICATIONS OF IMPROVED ARCHITECTURES

The possibilities of redistribution of functions between previous levels of hardware, firmware and software architectures can be expanded with the introduction of a clearer notion of programmed logic and resulting process and processor organization. The view that all system architectural levels are composed of subsystems of cooperating processes, with clean intra- as well as interlevel interfaces, allows us to begin to understand redistribution possibilities (perhaps in some cases, as mentioned above, eliminating what we know as microarchitectures today).

Global design rules can, one hopes, be evolved to allow a uniform view of activities at all levels and thus permit us to encapsulate redistribution possibilities into the CAD (in a broad sense) systems of tomorrow. In this environment, assuming the availability of powerful simulation engines, the necessary evaluation and insight can be gained. The synthesis of the programmed logic of processors of all levels (hopefully fewer levels than today) can be accomplished in a natural symbiosis between human and machine.

Tomorrow's integrated CAD system will hopefully provide the basis for not only attacking structural problems, designer-manufacturer problems, testing problems, etc., but will also provide the basis for more understandable architectures. One hopes that, in addition to other byproducts of the CAD system, the system will produce its own educational training aids in the form of animations of the static and dynamic structure. Such animations, developed from simulation, in this author's opinion will be possible in the future. A pedagogically designed system will definitely provide improved economical results by reducing complexity, and consequently costs, for users and maintainers. Work on tomorrow's CAD systems, as well as the design of VLSI based applications, is a major research activity of the Computer Architecture Laboratory at Linköping University.

SPECIAL PURPOSE SYSTEM ARCHITECTURES

The use of microarchitectures in various types of special purpose machines, that is, the use of microprogram control for applications other than emulation and HLL instruction interpretation, has become popular in the last several years. Several of these projects have been possible due to the availability of bit-slice ALUs and related microsequencers. In order to round out our view of activities in system architecture with respect to the use of microprogrammed control we shall briefly review a few application areas in this regard.

Microprogrammed systems involving bit-slice logic have been introduced in a wide scale in signal processing. The requirement, as always here, is high processing capacity by use of relatively simple algorithms, perhaps applied over several signal sources in parallel. Due to the homogeneity of the algorithms involved and the possibilities to develop even greater processing capacity, signal processing applications are good candidates to be realized in VLSI without the use of microprogrammed base logic. These developments are already well under way.¹⁹

Another interesting area is that of database machines, where the "data" of the application is normally highly heterogeneous even though many attempts have been made to regularize data in order to apply associative searching.²⁰ Moving more conventional database approaches into database machines will result in a definite need for highly structured programmed logic. The ideas provided in previous sections of this paper definitely apply to this environment.

Let us now consider the area of array processing systems of the type used in image processing and analysis. Convenient and efficient implementation of, for example, picture operators in this environment has required the use of microprogram control. In this case, the microinstructions are routed to equal parallel base logic processes in a synchronous manner. Each base logic process interprets the same microinstruction with its "local data." The main problem here is handling conditional transfers of control. Consequently, the architecture of this type of machine is usually constructed to execute short sequences over several microcycles without branching. Due to the regularity in the data structures, the use of the systolic algorithm VLSI approach mentioned earlier¹⁷ seems quite appropriate in this area.

During the 1970s, research attention has been drawn to Data Flow Architectures and Dennis²¹ and Patil²² have made basic contributions leading to this research activity. The very nature of processing in this environment is asynchronous and the ideas presented in previous sections are definitely appropriate. Processing demands for a Data Flow Architecture to replace conventional machines are, of course, heterogeneous. Certain control aspects in this environment can utilize more regular forms of logic representing the realization of Petri-nets.²³ Ongoing experiments in this area use bit-slice microarchitectures²⁴ as well as planning for the use of VLSI base logic.²⁵ It will be interesting to see if practical systems using the data flow strategy become accepted during the 1980s.

SUMMARY AND CONCLUSIONS

The approaches used in realizing microarchitectures have been presented and compared in respect to parallelism and complexity. It has been argued that the use of asynchronous control, even in the microarchitectural environment, has many advantages for emulation and HLL instruction interpretation. The structuring possibilities provided by the asynchronous strategy lead very conveniently into structuring ideas in the VLSI area. Possibilities for using systolic algorithms, where homogeneous data structures are to be processed, have been mentioned and contrasted to the hetero-

geneous world of emulation and HLL instruction interpretation. Some implications of simplifying and unifying notions of programmed logic for redistribution and CAD have been presented. Finally, some areas where microarchitectures have been used in producing special purpose systems architecture have been summarized, and some trends for developments in these areas have been considered.

We can conclude that the 1980s will probably be an era of change for system architectures (including microarchitectures) due to the possibilities made available by VLSI and CAD systems. We must move in the direction of producing highly structured systems that will be easier to design, implement, test, maintain and utilize. Further, we can conclude that the systems of tomorrow will be easier to understand and will provide for widespread education for new categories of "users" and laymen. The generalization of the process-oriented approach advocated in this paper has been used successfully in presenting a well structured overview of computer systems concepts and terminology for people (laymen as well as professionals-to-be) from a variety of walks of life.² Hopefully, our architecture activities and educational activities will become closer to each other in the future.

REFERENCES

1. Lawson, H.W., "Function Distribution in Computer System Architectures," *Proceedings of the Third Annual Symposium on Computer Architecture*, January 1976.
2. Lawson, H.W., *Understanding Computer Systems*, Lawson Publishing Company, Linköping, Sweden, 1979.
3. Rosin, R.F., Frieder, G. and Eckhouse, R., Jr., "An Environment for Research in Microprogramming and Emulation" *Communications of the ACM* 15, no. 8, August 1972.
4. Lawson, H.W. and Smith, B.K., "Functional Characteristics of a Multilingual Processor," *IEEE Transactions on Computers* C-20 no. 7, July 1971.
5. Bell, C.G. and Newell, A., *Computer Structures: Readings and Examples*, New York, McGraw Hill, 1971.
6. Salisbury, A.B., *Microprogrammable Computer Architectures*, New York, American Elsevier, 1976.
7. Alexandridis, N.A., "Bit-Sliced Microprocessor Architecture," *Computer*, June 1978.
8. Lawson, H.W. and Malm, B., "A Flexible Asynchronous Microprocessor," *BIT* Volume 13, Number 2, June 1973.
9. Lawson, H.W. and Malm, B., "The Datasaab Flexible Central Processing Unit (FCPU)," *Infotek State of the Art Series*, Report 17 on Computer Design, 1974.
10. Lawson, H.W. and Blomberg, L., "The Datasaab FCPU Microprogramming Language," *Proceedings of the SIGPLAN/SIGMICRO Interface Meeting*, May 1973.
11. Lawson, H.W., "The Datasaab Flexible Central Processing Unit: Collected Published Articles (1973-75)," Linköping University, Report LITH-ISY-1-0330.
12. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM* 17, no. 10, October 1974.
13. Lawson, H.W. and Magnhagen, B., "Advantages of Structured Hardware," *Proceedings of the Second Annual Symposium on Computer Architecture*, January 1975.
14. Mead, C. and Conway, L., *Introduction to VLSI Systems*, Reading, Mass., Addison-Wesley, 1980.
15. Eichelberger, E.B. and Williams, T.B., "A Logic Design Structure for LSI Testability," *Proceedings of the Design Automation Conference no. 14*, 1977.
16. Alves Marques, J.M.C., "A Multiprocessor Architecture Adapted to VLSI Custom Design," *Microprocessor Systems: Software, Firmware and Hardware*, *Proceedings of EUROMICRO 80*, September 1980.
17. Kung, H.T. and Leiserson, C.E., "Systolic Arrays (for VLSI)," *Proceedings of the Society of Industrial and Applied Mathematics on Sparse Matrices*, 1978.
18. McNamara, J.E., *Technical Aspects of Data Communication*, Digital Press, Maynard, Mass., 1977.
19. Wanhammar, L., "The Implementation of Wave Digital Filters Using Vector-Multipliers," *Proceedings of EUSIPCO, Signal Processing: Theories and Applications*, 1980.
20. Batcher, K.E., "STARAN Parallel Processor System Hardware," *National Computer Conference, AFIPS Conference Proceedings*, 1974.
21. Dennis, J.B., "Modular Asynchronous Control Structures for a High Performance Processor," *Record of Project MAC Conference Concurrent Systems and Parallel Computation*, ACM, N.Y., 1970.
22. Patil, S.S., "Coordination of Asynchronous Events," MAC TR-72, *Project MAC*, MIT, June 1970.
23. Peterson, J., "Petri Nets," *Computing Surveys*, Vol. 9, no. 3, September 1977.
24. Dennis, J.B., Misunas, D.P. and Leung, C.K.C., "A Highly Parallel Processor Using a Data Flow Machine Language," *Computation Structures Group Memo 134*, Laboratory for Computer Science, MIT, Cambridge, Mass., January 1977.
25. Arvind, Gostelow K. and Ploute W., "An Asynchronous Programming Language and Computing Machine," TR 114a, University of California, Irvine, December 1978 (Minor Revisions June 1980).

Microprogramming—The challenges of VLSI

by ALICE C. PARKER
University of Southern California
Los Angeles, California
and
WAYNE T. WILNER
Xerox Corporation
Palo Alto, California

ABSTRACT

Digital system design has been affected dramatically by very-large-scale integration (VLSI). Microprogramming will be affected most greatly by the VLSI design problem. Hardware performing specific functions will be replaced by regular arrays of logic and memory. Design time for VLSI systems will depend on sophisticated design aids for hardware and microcode, and concurrent systems will be common. Microcode will be used in virtually all highly integrated systems.

INTRODUCTION

“So what everyone is expecting from VLSI systems is nothing less than a complete revolution in computer science. A revolution on the Copernican scale, in which what we have been calling the ‘central processing unit’ is seen to be not central at all, but just one of many similar units. And what we have been calling ‘control programs’ will not really be in control but will negotiate with other control programs.”¹

Progress in integrated circuit technology is affecting microprogramming and microprogrammable control structures. The reverse is perhaps even more important: *Progress in very-large scale integration of digital designs will be enhanced by the use of microprogramming.* As chip densities increase, innovative control structures will be introduced and the nature of the microprogramming task itself will change.

VLSI designers are constrained by objectives, the most important of which are

- To minimize cost by minimizing silicon area and pin count.
- To maximize performance in terms of either operations/second or bits of data/second.
- To increase chip functionality by adding newer, more powerful instructions.
- To increase chip fault-tolerance.
- To enhance the user interface.

- To achieve reasonable design turn-around time and minimize design costs.
- To minimize power consumption.

These objectives are conflicting. Usually silicon area is traded off against performance, functionality, fault tolerance, ease of use, and quick design time. If silicon area (and pin count) are held constant, the remaining objectives (except power consumption) compete and only one can be optimized for a given design; the rest will have been traded off, since they all require increased silicon area.

A number of techniques will be used by VLSI designers in order to achieve the above goals, including

- The use of regular structures to replace random logic.
- The exploitation of concurrency by the partitioning of processing into separate functional units as predicted by Wilkes.²
- The minimization of detail which must be dealt with.
- The addition of extra hardware or ROM in order to provide fault tolerance.

Regular structures provide many advantages to the VLSI designer. They can be tightly compacted on the layout and tend to minimize silicon area. The MIT SCHEME chip³ makes extensive use of such structures. The compact arrangement of regular structures can minimize the length of interconnections, a significant use of area now and a more significant source of delay in the future. Properly designed, regular logic structures can be used to execute parallel algorithms, enhancing performance (e.g., Bentley and Kung's tree machine⁴). If ROMs or RAMs are used to provide regularity, microcode can be written to support extensive, powerful instruction sets (as in the VAX 11/780), diagnostic programs, and interpreters of high-level languages (Western Digital's PASCAL machine, for example). In fact, it is universally agreed that future single-chip processors will be microcoded; random logic becomes too complex a design problem to be feasible. Much of this complexity comes from the relative difficulty of design changes with random logic. Finally, design

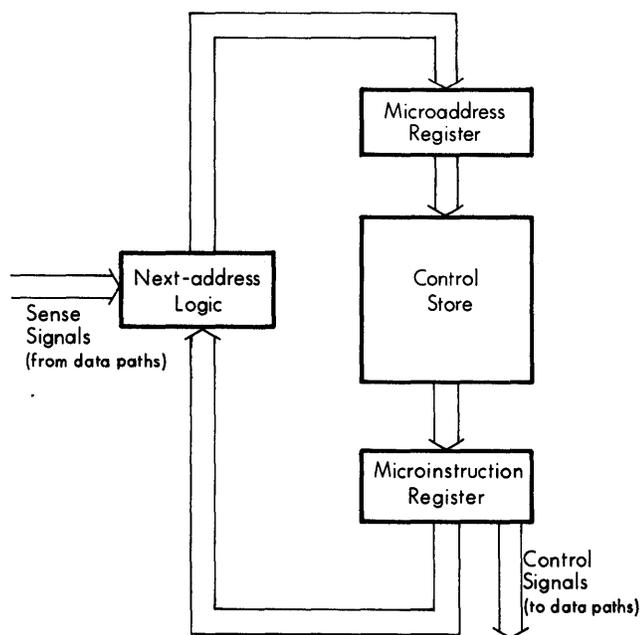


Figure 1a—A Basic microprogrammed controller

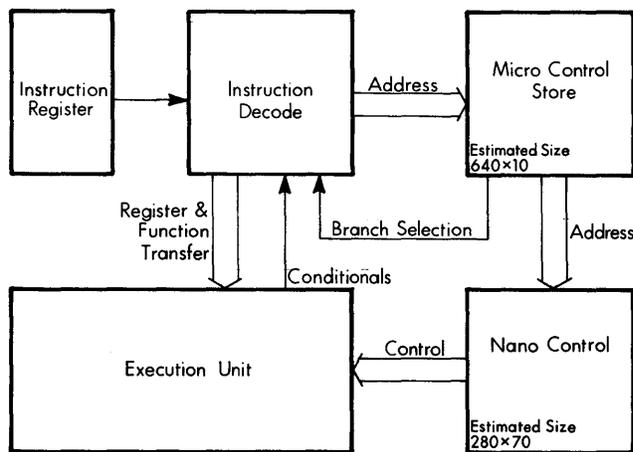
time and design costs are related directly to the irregularity of the structures used to implement a given design. Massive use of logic arrays and memory will be required to slow the exponential growth in design times for VLSI.⁵

Concurrent operation can be achieved by functional partitioning at any of a number of levels. At a primitive level, the fetch-execute cycle of the microcontroller is already being overlapped, or pipelined, as in the PDP-11/40. At the machine-instruction level, a similar pipeline can be constructed. At the data-processing level, activities can be functionally partitioned and in some cases can be subdivided to be executed by an interconnected network of primitive processing elements, which may in turn provide a regular structure to the design. Concurrent operation, at any level, implies separate hardware, and as a consequence it may require distributed control.

As VLSI systems become more complex, there is a competing goal to make them easy to design. High-level or machine-independent descriptions of microprogrammable machinery will reduce the amount of detail VLSI designers must supply. Automated tools are already necessary to cope with the complexity of contemporary designs. In order to handle the additional complexity due to increased circuit density and concurrency, many design steps and optimizing procedures will be relegated to tools. Some such design systems are being reported.^{6,7,8}

DIRECTIONS IN MICROPROGRAMMING

Because the design goals described earlier are competing, microprogramming and microprogrammed controllers will change along many different directions, depending on the constraints of each design problem. In general, however, microprogrammable hardware will become increasingly complex



MC68000 Control Structure
Figure 1b—MC68000 control structure

with significantly larger control stores, and will support communication with concurrent devices, as well as concurrency in its own operation. The microcode itself will become more sophisticated. Unfortunately, design time and cost will increase. Microcode production (now informally estimated at one line of optimized microcode per person per day) may reach even lower levels. After exploring the pathways microprogramming is going to take, we will propose solutions to the problems of design time and design costs.

The Shape of Future Microcontrollers

Obviously, complex microarchitectures are not recent design innovations (the QM-1⁹ and B1700¹⁰ are striking exam-

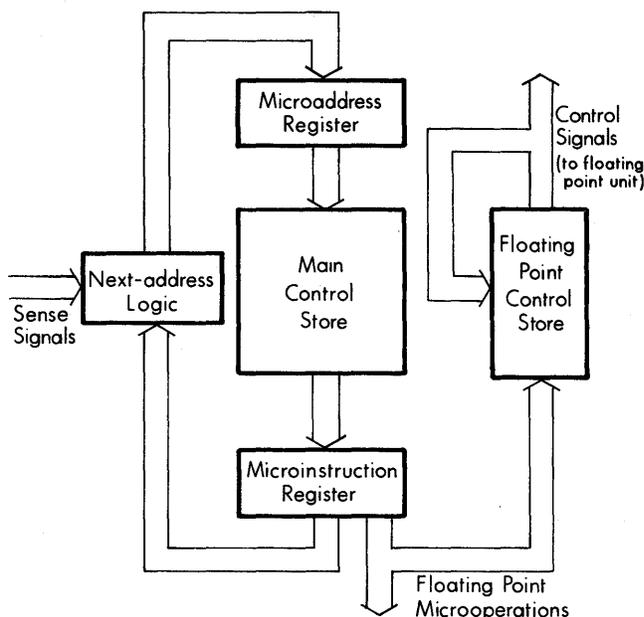


Figure 1c—A microprogrammed controller with separate floating-point control store

ples). However, the use of sophisticated microcontrollers to achieve VLSI design goals is on the increase.

Of course, complex digital devices can be controlled with basic microcoded structures like that shown in Figure 1a. The next-address logic is primarily combinational, forming or multiplexing the next micro-address. The advantage of this structure is that the control store itself, regular and compact, dominates the controller chip area. This structure therefore optimizes design time. If silicon area or performance takes precedence over design time, however, more complex structures may be required. Stritter and Tredennick have shown that the Motorola 68000 two-level control store¹¹ is smaller than a single-level implementation would have been. (See Figure 1b). Similar results are reported in Holloway et al.³

Another frequently used technique is the partitioning of machine instruction decoding or next address generation into separate PLAs or ROMs, as done in the PDP-11/60.¹² (See Figure 1d). This technique allows concurrent operation of the fetch, decode and execute cycles at the machine-language level but makes the job of code optimization more difficult.

When specialized sequences of microinstructions are partitioned into separate PLAs or microstores, a similar performance enhancement or a savings in silicon area can occur. An example of this is shown in Figure 1c, where long floating-point sequences are isolated in their own microstore. Execution might be faster because the smaller microstore may be more parallel, or may sequence at a faster rate. In the extreme, the entire floating-point timing may be different, or even self-timed, as proposed by Seitz.¹³ Further performance enhancements are possible when these control stores and PLAs execute concurrently.

If silicon area is the hardest constraint, such partitioning can result in narrower microinstructions, perhaps as a result of more vertical microcode in the second control store. The savings in silicon area is not a simple relationship, however, since the overhead logic involved in control of the second microstore or PLA may very well consume more area than is saved by the partition.

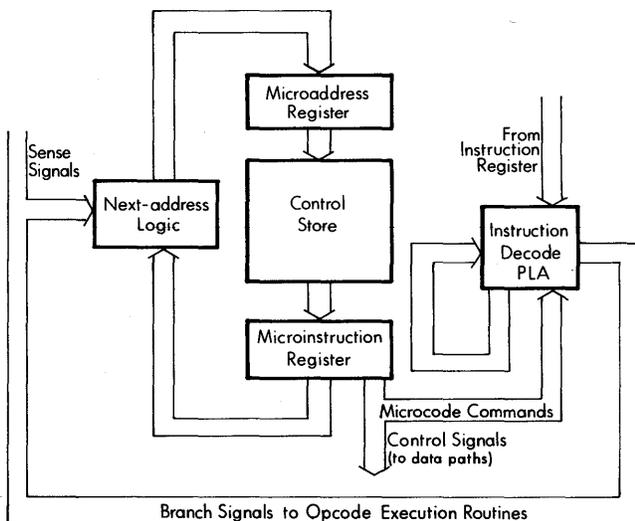


Figure 1d—Separation of instruction decoding in a microprogrammed controller

CONTROLLING CONCURRENT PROCESSING

Complications in traditional microarchitectures occur when these microarchitectures must support real-time response to external devices and communication between concurrently executing functional units. The introduction of interrupt-based concurrency and communication affects the microarchitecture differently than an alternative approach, message-based communication.

Synchronization between functional units within and across VLSI chips is likely to occur at a higher level than the microprogram level. Ideally, each functional unit will possess its own clock and communication between functional units will be asynchronous. Furthermore, a message-based communication scheme allows communication to occur concurrently with processing, and the microcode deals with the communication mechanism asynchronously or even indirectly via machine instruction execution. The main difference in the microarchitecture will be the addition of many more control and sense signals extending beyond the processing unit and controlling the communication mechanism. Direct control over communication will belong in the microcode when cost and design time are important constraints. When performance is to be optimized, separate interface logic will generate the necessary signals under microcode commands. Such a situation is shown in Figure 2. We have a lot to learn about how to control many functional units concurrently but it will become very important when multiple functional units are placed on a single chip. Fortunately, there are many familiar systems to model, such as freeways, markets, and companies.

Distribution of clock signals to multiple functional units, even on the same chip, will be replaced with self-timed circuits because problems with clock skew and signal degradation will make design times and costs unreasonably great. Only with high-performance, real-time processing requirements will it be necessary to provide central control with a single clock.

Furthermore, microprogrammed controllers are naturally sequential and synchronous. Therefore, the control over asynchronous communications will likely be PLA-based, instead, with microprogrammed hardware controlling each communicating unit's internal operation. In some cases the distinction between the control and data paths used for such asynchronous operation may become blurry. In these situations, arrays of gating and storage (SLAs¹⁴, for example)

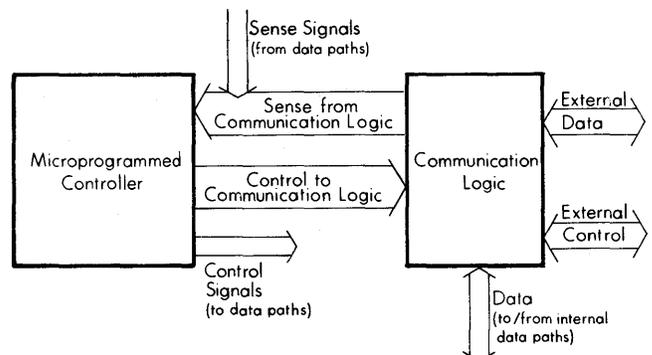


Figure 2—Separation of processing control from I/O control

provide regular structures containing both control logic and data-path hardware so that design time is reasonable.

Interrupt-based concurrency will most frequently be used when real-time response is important. In these cases, long sequences of microcode executing a single complex machine instruction will themselves need to be interrupted, and micro-interrupt hardware will be introduced. (See for example, IBM System/370.¹⁵)

Changing Tradeoffs

Traditional implementation tradeoffs are still valid for VLSI designs. The choices of PLA versus ROM and nature of microinstruction format still remain. However, the goals of minimal silicon area and reasonable design times complicate the implementation decision process, producing substantially different price-performance characteristics from previous digital systems.

Performance will be optimized primarily, as it has been traditionally, by evoking operations in parallel, which precludes significant encoding of microwords. The role of PLAs to replace random logic in high-performance systems is growing as PLAs become faster. Local control of specific functions is likely to be implemented with PLAs since their operation is inherently parallel and can be asynchronous with respect to the central control. In the extreme performance cases, relatively simple functional units controlled by small PLAs will be appropriate for executing inherently parallel problems, such as image enhancement. (See Figure 2). In most environments, however, the justification for a central control is that it reflects the "problem" being executed, where the "problem" is commonly instruction-set interpretation and execution. Furthermore, large PLAs are not as likely to be used for centralized control as microstores are. This is partly due to the difficulty the designer will face in thinking of larger, complex systems in a nonprocedural manner, and partly because of the sequential nature of the control task itself.

In competitive microprocessor designs, where design costs are absorbed in volume business, cost optimization means area minimization. Because control stores are regular structures, there will be a tendency to replace random logic with microcode. An example of this is the collapse of complex I/O device controllers into the microcode. Performance in these cases may be degraded since the opportunity for concurrency is lost, and providing more horizontal microinstructions increases silicon area again. Breaking a central microstore into local PLAs and control stores can supply the necessary parallelism and regularity of structure, while reducing silicon area. The ratio of microstore width to data path width should be close to one in order to cleanly route control signals to the data paths. Encoding of the microstore can shift the width ratio significantly. Also, if the control signals are run in polysilicon instead of metal, time considerations in routing the control signals to their destinations may become significant if the connections are not straightforward.

Exotic formats and encoding techniques are not likely to be employed. An example of an exotic format is the following: imagine a definition field in the microword which determines

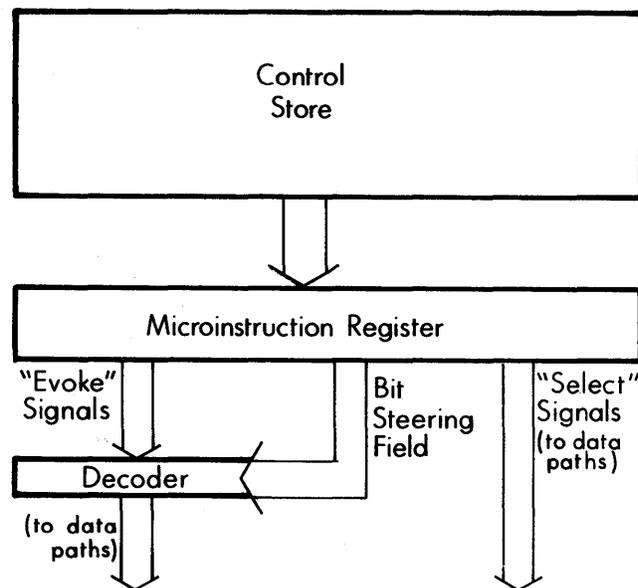


Figure 3—A novel encoding technique

the field sizes and boundaries to be decoded by a bit-steering field, also specified in the definition field. These techniques may consume more silicon in decoding than they save in encoding, or may degrade performance if simple decoding structures are used. Design time is increased since the microcode is harder to write, design changes in the microcode may require redesign of the entire controller, and the decoder's random logic itself takes time to design.

Straightforward encoding techniques, particularly along functional boundaries, will always be useful when conserving silicon. An encoded destination field, for example, may contain mutually exclusive destinations and therefore parallelism is not sacrificed. Design changes only affect this field and its decoder. Furthermore, migration of decoders out into the data paths saves silicon area, since the routing of $\log n$, instead of n , signals is done. This can only be done, however, if the encoded signals are related topologically.

Some variations in encoding techniques are possible. Nagle¹⁶ has proposed a scheme for overlaying mutually exclusive control signals in the same fields. A bit steering field demultiplexes the signals to the correct destinations. (See Figure 3). The innovation in this technique is that only fields which contain control signals causing state changes (evoke signals) need be multiplexed. Select signals, which select multiplexer inputs and ALU functions, can merely be overlaid in the same microinstruction field, provided the partitioning and overlaying is carefully done. In addition, these select signals, which reach the data paths first, are not delayed by decoders. Evoke signals are delayed by the demultiplexer but are not used until the end of each cycle anyway. The degree to which signals are overlaid in this scheme controls the amount of parallelism in the resulting microprogram.

Encoding in general can conserve silicon area if the microstore contain n words, the encoded fields contain f bits each and n is greater than f . The simplest decoding of an f -bit field requires $f(2^f)$ PLA cells. Additional space for unencoded sig-

nals would require $(2^f - f)n$ ROM cells. Thus, for example, encoding fields less than 8 bits wide will save considerable silicon area when the number of words in the microstore is greater than 10. Larger encoded fields require longer microprograms in order to be cost-effective, of course. (In this example, random logic replacing the PLA would consume more area due to the irregular structure, use of logic gates, and increased number of transistors per gate, but may be faster). As data paths support higher degrees of parallelism, however, microcode will become more parallel in most cases, and significant encoding will not be considered. (Colleagues have suggested a transmission gate decoder may be more silicon-efficient than the techniques described above.)

MICROPROGRAM DESIGN AIDS

Design times must be decreased in order to produce more complex integrated circuits. While regular structures alleviate this problem in some cases, the problems of correct microcode and optimal microcode for a given task remain. VLSI hardware and microcode design problems are large and technology-dependent; a human alone cannot handle the detail and explore enough alternatives involved in correct, optimal design. Design aids which explore alternatives and keep track of necessary details will be an integral part of future microprogramming projects. Research involving these design aids falls into three classes:

- microcode verification.
- microcode generation from a high-level language.
- synthesis of microcode and microcontroller hardware from a higher-level specification.

Although a survey of these research areas is not possible here, we will briefly present example projects.

An example of microcode verification is the work by van Mierop, Marcus and Crocker at ISI.¹⁷ Formal descriptions of both the host and target machines for a fault-tolerant computer were written in ISPS¹⁸ and the proof system verified that microcode running on the host machine correctly implemented the target instruction set.

Compilation of microcode from a high-level language is another approach to the production of correct microcode. The STRUM system¹⁹ was an early research project that produced verification conditions as it compiled microcode from a structured high-level language.

Machine independence is a highly desirable feature of microcode compilers.²⁰ While research in universal microprogramming languages is not new²¹, the problem of machine-independent compilation is, in general, unsolved. Some recent results reported on by Marwedel and Zimmerman⁶ are optimistic. They describe a system for the generation of microcode from a high-level language. The system is machine-independent because it is driven by a tabular description of the hardware. Although their proposed optimization algorithm is not yet implemented, hand-compiled examples indicate good results with word compaction and hardware allocation.

Optimal speed or cost can only be obtained in a digital system when crucial design decisions are made simultaneously. For example, once the number of registers in the data paths has been fixed, limiting the parallelism, completely horizontal microcode may be useless, since the data path hardware limits the design. On the other hand, adequate data-path resources are wasted if the microcode can only control a subsection of the hardware in each cycle. Therefore, microcode optimization *a posteriori* may produce less-optimal designs than if the entire design process occurs concurrently. The MIMOLA system⁸ allows a designer to interactively restrict data-path hardware and the system generates microcode for the available resources. The designer iterates, changing the hardware restrictions, until the desired price and performance goals are met. The resulting microinstructions contain precisely the amount of parallelism allowed by the data paths. Recent research²² shows that optimal designs can be achieved only if ordering of microoperations and design of data paths occurs simultaneously.

A more rigorous attempt at microcode optimization has been undertaken by Nagle et al.⁷ In this research, the microcontroller hardware and microcode are generated automatically for a fixed set of data paths. A convex cost-speed curve can be obtained for microcode controlling a given set of data paths, as the parallelism is changed. In this case, also, the microcode supports the degree of parallelism provided by the data paths. The designer restricts the microinstruction width and the optimization algorithm, then produces microcode ranging from horizontal to vertical, depending on the word-width restrictions and on each design problem. If the performance bounds are not met, the designer can loosen the width restrictions and iterate.

Some research issues must still be resolved in order to apply the design automation research described here to general VLSI design problems. Furthermore, much of this research does not include complicated branching schemes, pipelined microarchitectures, and other VLSI system features. However, these techniques provide a glimpse into the future of microprogramming.

CONCLUSION

Microprogramming will be affected most highly by the VLSI design problem. Hardware performing specific functions will be replaced by regular PLAs and control stores. Many special purpose integrated circuits will appear on the market, customized via PLA's or microcode. Bit-slice microprocessors will increase in sophistication, replacing random logic. The underlying geometry of VLSI systems is growing in importance, and should be studied closely.

The role of PLA's in the future is becoming clearer. While they will not replace large microstores, they certainly have a role in support of a large control store and are ideal for controlling local events and asynchronous communications.

Design aids for microprogrammers and hardware designers will become widely used, allowing generation of correct, near optimal microcontrollers. Virtually all digital systems which will be highly integrated will be microprogrammed.

REFERENCES

1. Wilner, W. T., "Microprogramming in Silicon," Keynote address, *11th Annual Microprogramming Workshop*, Nov. 1978.
2. Wilkes, M., "Computers Then and Now," *Journal of the ACM*, 15 (1968), 1, pp. 1-7.
3. Holloway, J., et al., "The SCHEME-79 Chip," AI Memo No. 559, M.I.T., January 1980.
4. Bentley, J. L., and Kung, H. T., "A Tree Machine for Searching Problems," *Proceedings of the 1979 International Conference on Parallel Processing*, Wayne State University and IEEE Computer Society, August 1979.
5. Moore, G., "VLSI: Some Fundamental Challenges," In Rex Rice (Ed.), *The Coming Revolution in Applications and Design*, New York: IEEE, 1980.
6. Marwedel, P., and Zimmerman, G., "Target-Machine-Independent Microcode Generation System for a High-Level Language," unpublished manuscript, Institute für Informatik und Praktische Mathematik der Universität Kiel, West Germany.
7. Nagle, A., "Automatic Design of Digital-System Control Sequencers from Register-Transfer Specifications," dissertation, Carnegie-Mellon University, Dec. 1980.
8. Zimmerman, G., "The MIMOLA Design System: A Computer-Aided Digital Processor Design Method," in *Proceedings of the 16th Design Automation Conference*, ACM and IEEE, June 1979, pp. 53-58.
9. Nanodata Corporation, *QM-1 Hardware Level User's Manual*, 1974.
10. Wilner, W. T., "Design of the Burroughs B1700," in *AFIPS Proceedings of the Fall Joint Computer Conference*, 41 (1972), pp. 489-497.
11. Stritter, S., and Tredennick, N., "Microprogrammed Implementation of a Single Chip Microprocessor," In *Proceedings of the 11th Annual Microprogramming Workshop*, ACM SIGMICRO and IEEE Tech. Comm. on Microprogramming, November 1978, pp. 8-16.
12. Bell, G., Mudge, C., and McNamara, J., *Computer Engineering*, Maynard, Massachusetts: Digital Press, Digital Equipment Corporation, 1978.
13. Mead, C., and Conway, L., *Introduction to VLSI Systems*. Reading, Massachusetts: Addison-Wesley, 1979.
14. Patil, S. and Welch, T., "A Programmable Logic Approach for VLSI," *IEEE Transactions on Computers*, 28 (1979), 9, pp. 594-601.
15. Davis, C. et al., "IBM System/370 Bipolar Gate Array Microprocessor Chip," In *Proceedings of the IEEE International Conference on Circuits and Computers*, October 1980, pp. 669-673.
16. Nagle, A., "Automatic Design of Micro-controllers," in *Proceedings of the 11th Annual Microprogramming Workshop*, ACM SIGMICRO and IEEE Tech. Comm. on Microprogramming, November 1978, pp. 112-117.
17. van Mierop, D., Crocker, S., and Marcus, L., "Verification of the FTSC Microprogram," In *Proceedings of the 11th Annual Microprogramming Workshop*, ACM SIGMICRO and IEEE Tech. Comm. on Microprogramming, November 1978, pp. 118.
18. Barbacci, M., Barnes, G., Cattell, R., and Siewiorek, D., "The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language." Technical Report, Dept. of Computer Science, Carnegie-Mellon University, August 1979.
19. Patterson, D., "STRUM: Structured Microprogram Development System for Correct Firmware," *IEEE Transactions on Computers*, 25 (1976), 10, pp. 974-985.
20. Mallet, P., "Methods of Compacting Microprograms," dissertation, University of Southwestern Louisiana, Dec. 1978.
21. Eckhouse, R., "A High-Level Microprogramming Language," in *AFIPS Proceedings of the Spring Joint Computer Conference*, 40 (1971), pp. 169-177.
22. Hafer, L., and Parker, A., "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic," In *Proceedings of the 18th Annual Design Automation Conference*, ACM and IEEE, June 1980 (in press).

Vertical and outboard migration—a progress report*

by ANDREW HELLER
IBM Corporation
Santa Theresa, California

and

ANDRIES VAN DAM
Brown University
Providence, Rhode Island

ABSTRACT

The primary method for gaining performance improvement on a fixed-hardware architecture is to tailor the soft components, i.e. the application program, the operating system, or the firmware, to the performance requirements. This paper deals with two specific forms of performance tuning called *vertical* and *outboard migration*. These terms refer respectively to migrating (pieces of) functions from higher levels to lower levels in a software/firmware/hardware hierarchy and to migrating such functionality to auxiliary processors such as I/O processors which can run in parallel with the CPU to offload it. The performance gains in vertical migration result from the elimination of CPU overhead, while those in outboard migration result from the ability to offload the CPU and have separate (special-purpose) processors execute the migrated code asynchronously and in parallel with CPU execution.

INTRODUCTION

Motivation

Performance is a continuing concern in the design of computer systems, despite the great advances that have been made in hardware technology which provide ever-greater computing and storage capacity at decreasing price. Rising expectations simply absorb increased capability to provide, for example,

- faster response,
- more user-friendly interaction,
- handling of more variables and exceptional conditions,
- more robustness, security and protection,
- greater “RAS”: Reliability, Accessibility/Availability and Serviceability
- an increase in the number of simultaneous users of a

timesharing system with a decrease in the actual multi-programming level (number of concurrently executing tasks and context switches), and

- capacity for new, larger problems which could not be handled previously.

The primary method for gaining performance improvement on a fixed-hardware architecture is to tailor the soft components, i.e. the application program, the operating system, or the firmware, to the performance requirements. This paper does not deal with such tuning in general, but with two specific forms of tuning called *vertical* and *outboard migration*. These terms refer respectively to migrating (pieces of) functions from higher levels to lower levels in a software/firmware/hardware hierarchy and to migrating such functionality to auxiliary processors such as I/O processors which can run in parallel with the CPU to offload it. A third form of migration, called *horizontal migration*, is the migration of tasks between more or less equal peer processors, typically CPUs connected in a distributed network, each of which may have its own peripheral processors. Horizontal migration is in effect a synonym for cooperative distributed processing, and is the subject of intensive research in its own right, most of which is not related to microprogramming and is therefore not directly relevant here.

The primary purpose of vertical and outboard migration is to achieve a better division of labor and therefore a price/performance improvement by migrating tasks to the real or virtual processors best equipped to handle them. In addition to such functional specialization, migration may be used to provide greater isolation and protection of critical modules by making them inaccessible to direct or indirect user modification, effectively “burying them in the silicon/hardware.” The performance gains in vertical migration result from the elimination of CPU overhead, while those in outboard migration result from the ability to offload the CPU and have separate (special-purpose) processors execute the migrated code asynchronously and in parallel with CPU execution.

* Portions of this paper have been adapted from an informal paper by Heller.¹

(This is traditionally done, for example, with I/O channels and arithmetic processors attached as peripherals.)

Vertical migration is effected in architectures which can be largely structured as hierarchies, starting with the hardware, then up to the firmware level and then building up with the operating-system kernel level, higher levels of operating system, and finally levels of application program. The software part of such a hierarchy is effectively induced by using subroutine/procedure modularization, whereby each call produces another micro-level in the hierarchy; it is customary, however, to group functionally related procedures and micro-levels into a few, relatively large (macro) levels, each with potentially differing entry/exit/invocation conventions. At higher levels in the hierarchy, these conventions tend to be increasingly more general and powerful, but they also require increasingly more processor cycles and memory space for status-saving and context-switching.

Primitives at any level of a hierarchy have two types of components: *mapping actions* and *execution actions*.² Mapping actions are those actions performed to map flow-of-control and data parameters from the invoking level to the level of the invoked primitive and back. Execution actions are those actions which perform the semantic operations for the invoked primitive.

A function which is migrated downwards (towards the hardware) in the hierarchy can reduce overhead and improve performance in two distinct ways: first, its execution actions are implemented on a faster and more efficient but more primitive level, and second, a more efficient, less general-purpose mapping action may be used to invoke it. Thus individual functions are speeded up by as much as an order of magnitude while the speed of the application program, as a whole, may improve by 25-50%.³ As an example of the first type of savings, implementing the execution actions of an algorithm in microcode is much faster than having the equivalent software instructions perform the same execution actions because, minimally, target-instruction fetch and decode are replaced by microinstruction fetch and decode, even if target operand fetch cannot be altered.

The second type of savings can occur, for example, when an entire target subroutine invoked by a procedure call is replaced by a microcode subroutine invoked by a single target instruction (e.g., a matrix multiply or an array reference), or when the mapping action overhead of setting up task control blocks is replaced by a simpler, cheaper, lower-level invocation mechanism that does not allow parallelism. This can be done if a function's ability to run "in parallel" with other tasks is not needed for a given application, i.e., if its previous (higher-level) implementation underutilizes the more general functionality (provided by the higher-level mapping action) available to the function at the higher level.

As described in the section "Operating System Migration," vertical migration of software to software or software to firmware can improve performance not only by eliminating overhead incurred by unused functionality, but also (for software to firmware migration) by introducing savings as a result of basing the migration on a more efficient algorithm (and its data structures) to take advantage of data and control flow possibilities at the more primitive levels.

Another application of vertical migration is to migrate

down to firmware (with hardware assistance) supervisory functions such as software for monitoring, checking, fault diagnosis, etc., to allow them to run (nearly) continuously without impacting CPU performance.^{4,5,6} At a critical time (e.g., after hardware failure), this logic thus has access to data and control paths not accessible at the target level (e.g., scan rings for reading and resetting low (chip) level status). This type of vertical migration leads to machines with higher reliability, availability and serviceability.

Selection criteria

What are the problems faced in selecting highly-used candidate functions to migrate into firmware or hardware and in selecting the target implementation media? In selecting new, highly-used software functions as candidates for migration, several criteria must be employed. The *frequency of use* is key in determining the potential performance benefits which might accrue from various alternatives. The *locality of reference* to the data used by a function is critical; that is, if the function references a large number of storage locations spread across the processor memory and is limited by storage accessing rather than by the processor, simply placing the function in microcode or hardware will do little to improve its performance. On the other hand, if a function uses small self-contained tables as its primary source of data (high locality of reference), or if the data can be restructured or can be represented by compact representations (lookasides and tables), or if it accesses memory serially, it may benefit significantly from migration into microcode. In fact, these data may even be candidates for placement in special, high-speed read/write storage areas accessible only to microcode or hardware. In such a case, functions having high locality of reference may benefit greatly from microcode or hardware implementation. On the other hand, functions that appear to be CPU-bound and have locality of reference may in fact be memory bus-bound, so that moving them to a lower level won't help significantly. Another important consideration in evaluating the desirability of migrating a function centers around the *interlocks* which the function places on other functions or processes that may be competing for use of the same data. If this intersection with other functions is large, the complexity of isolating the function is high and therefore the interface should probably be reconsidered. This leads to a more difficult point, *interface selection*, an area which requires more planning, thought and good taste (system thinking) than any other area in system design today. It is important also to consider the *stability* of the function being considered for migration into either microcode or hardware. If the function is still undergoing frequent changes in either its interfaces or its algorithms, it is not ripe for "casting into silicon."

PROGRESS TO DATE

Operating system migrations

Vertical migration has been used frequently in modern operating systems. In IBM's MVS, for example, it was ob-

served that the locking function was executed frequently and that the data referenced by this function was highly localized. Consequently, the LOCK and UNLOCK microcode assists were added to the MVS system extension feature for certain system locks. The SVC ASSIST facility enhances the performance of the entry into a supervisor call routine by providing most of the housekeeping and lookups necessary in the first level of the SVC handler; this processing occurs as part of the execution of each supervisor call. Such facilities as register saving, acquisition of necessary MVS locks, and setup of the proper environment for execution of the service routine are all accomplished by the microcode. In addition, facilities in the MVS assist such as SVC ASSIST, PAGE FIX, Invalidate Page Table Entry (IPTE), etc., are of value not only because of the high frequency of use of these functions but also because the data referenced by these functions can be organized to a large extent into tables that provide for high locality of reference (a small number of cache fetches results in a large number of useful data references).

Often in migrations of system functions, rethinking of actual desired use by the software of the architecture can cause substantial performance gains in the migration of functions. Functions like the recently announced cross-memory feature in the MVS System Extensions significantly reduce the complexity and expense that the multiple memory operating system supervisor (MVS) has in providing multiple memory services in a machine that provides simultaneously only single memory addressing. The cross memory feature is a significant step in the direction of hardware/firmware recognition of the unanticipated use the operating system has placed on the original architecture. It results in greatly enhanced security and protection, reduction in use of globally shared address space. As well, it provides significant performance improvements in transfer of control for applications residing in one private memory using systems/subsystems services residing in another.

Similar performance enhancements were provided for another of IBM's operating systems, VM/370. Here again many functions were migrated into microcode in various assists. These include the Virtual Machine ASSIST, the Shadow Table Bypass ASSIST, the Control Program ASSIST, the Extended Virtual Machine ASSIST, the Virtual Interval Timer ASSIST, and others. Other manufacturers' operating systems provide assists/accelerators for frequently used functions such as memory management, dynamic storage allocation, and FORTRAN or COBOL operations such as range checking, DO loops, array and record/structure accessing, etc.

Performance improvements on the order of a factor of 5 to 10 have been observed in migrating individual software primitives to firmware, while application programs using the primitives have gained 25% or more. Furthermore, similar results have been obtained for software-to-software migration, as described in papers by Stockenberg³ and Stankovic,⁷ done by using a methodology and tools which treat firmware and software migrations in an identical manner.

More is not necessarily better in vertical migration, however. When all of VM's Control Program (CP) was migrated to microcode in an experiment, only an 8%-10% gain in throughput was realized, whereas VM ASSIST microcode, including approximately 5% of the critical code, produced a

20%-50% increase in throughput on the 370/148. The reason for this seemingly baffling result lies in the nature of the architecture of each layer of the hierarchy; primitives, data paths, data and storage structures, and control paths at each level may be reflected to or hidden from successively higher levels. A host (micro) machine typically has access to many special-purpose resources and data and control flow possibilities (register, writeable control stores, busses, multiway branches and condition testing, etc.) while the target machine has fewer but more general purpose possibilities. Thus the target architecture in conjunction with hardware assistance may have access to control information which is not as easily accessed by the host. Migrating a software algorithm *as is* to firmware thus may provide no improvement; it may even create a performance loss. What is required is to redo the algorithm (and, if necessary, its associated data structures) to take advantage of the host architecture. The non-migrated software in turn may have to be adapted to the newly migrated primitives, and the interfaces between migrated and non-migrated functions then must be changed for optimal results.

In the case of the 370/148 and the totally migrated CP, for example, the micromachine went through the software logic, took an interrupt on a privileged instruction detection just like the software, went through initial status save and determined at that point in a micro subroutine, whether or not the operation would be allowed. If so, it re-issued the instruction after resetting state information. By evaluating the actual requirements of the function it was determined that many of the actions taken by the algorithm were unnecessary in many cases. In VM ASSIST, tables accessible to the firmware ASSIST program allow the firmware prior to the execution of an interrupt to determine whether the specified privileged operation is allowed and, if it is, to dispatch it, essentially without overhead. In this way the target machine was restructured—hardware/firmware was made “smarter” to allow software at higher levels to be smarter. Thus designing an interface to a function chosen for migration may require extensive planning and re-specification in order to ensure that the migration criteria of this section are met.

Indirect vertical migration and hardware/firmware/software redesign

Often the function that is migrated into microcode or hardware has no real or specifically identifiable software counterpart, but rather is created in response to the recognition that the software usage of the hardware *implies* a function that can be effectively realized only in hardware or microcode. Here one speaks of *indirect* function migrations. For example, in the MVS operating system, shared segments and all pages in them (common storage area, pageable link pack area, nucleus area, etc.) are always assigned to the same virtual addresses in all address spaces. Advantage was taken of this fact by providing an interface between the software and the hardware which allows the operating system to indicate to the hardware which segments at any instant are shared in this manner. This information permits the hardware to use only a single entry in its translation lookaside buffer to map the same segment or page in each virtual memory (rather than using a lookaside entry

for each occurrence of a shared segment or page in each virtual memory). It makes it possible to eliminate alias checking (which would otherwise be required by the architecture) in the hardware lookaside buffer for all segments or pages shared in this manner. Another way to view this is that by eliminating the alias checking for shared pages and segments, we cause the number of "hits" to the lookaside to increase and the number of changes to the lookaside to decrease, and hence the performance to improve.

This change required a pair of symbiotic changes; first, the software had to be changed to indicate which pages had the property described, and second, the hardware had to be changed to take advantage of this information. This was done as part of the MVS System Extension feature (the "common bit"). In this case, then, system functions were not directly migrated; rather their use of the machine architecture implied a corresponding hardware function.

Sometimes machines are designed at the architecture level without a thorough understanding of how specific hardware features really will be used by the system, and this can result in a mismatch of function among hardware, firmware and software. These mismatches will continue to surface as the software system continues to evolve, and machine design/architecture changes that recognize the actual uses of the architecture can have significant system performance impact. Another example of this kind of evolution in system usage occurs because of a change in software usage of hardware functions, coupled with a change in hardware economics. This is especially relevant for high-speed storage, i.e. in CPUs' cache memories and main store. Because of the increasing disparity between processor speed and main storage speed (at the high end, commercial CPU speeds have increased by nearly 10 times in the last 12 years, while memory speeds for main storage have only doubled or tripled) the importance of locating data in lookaside buffers has increased. As a result of this, it has become more important to increase the efficiency of lookaside and lookaside functions such as translation directory lookasides and cache memories. Segment table origin stacks (STO stacks) were introduced in 370 hardware so that address space changes did not cause the loss of all existing lookaside entries, and in the MVS System Extension feature the ability to purge specific lookaside entries selectively was added to the microcode and hardware (IPTE, Invalidate Page Table Entry). This type of function (designed to improve lookaside ratios) is also an indirect rather than direct function migration implemented to complement the software structure and its use of the architected interface.

OUTBOARD MIGRATION

Many of the more interesting candidates for function migration do not fall into the category of simple assists or implied special lookasides, but are rather functions removed from the central processor altogether. Many system functions that now execute in a serial fashion can be migrated into microcode or parallel hardware with great benefits realized by exploiting natural parallelism and natural breakpoints. Usually, though not always, significant restructuring is necessary to obtain the desired interfaces and to yield the potential per-

formance improvements. Many key system functions need not be performed in a precise, serialized or interlocked manner. Typical of such functions are the heuristics frequently employed in operating systems to improve performance. Specifically, aging and sorting of the real page frames in a virtual storage system need not be done in a serialized or interlocked process as it is in any case only an approximation.

Another class of candidates for such migration are functions which must be precise but whose operation need not be synchronized or coupled with the operation of the main processor. Examples of this type of processing are many. The processing of I/O support functions, access methods, automatic data backup or journaling, and reallocation of storage are but a few of the possibilities. Since these functions need not be executed serially with the main-line processing, they can be implemented in totally parallel microcode and hardware.

In addition to the obvious performance benefit of the instructions which are now not executed in the central processor other benefits are also accrued. The hardware interfaces to the "page management" process, for example, can be optimized to acquire large blocks of status information about many pages with a single reference to the storage control element or hardware memory management facility, or could be built directly into the storage controller. The output resulting from the execution of a request to the storage controller could result in a significant reduction of the total number of references to the key array by not requiring that the keys be accessed individually. The output resulting from the execution of this function could be placed in addressable queues that the in-line page fault function could utilize. This type of migration results not only in the offload but potentially in a significant improvement in the total performance of this function: it would be possible, for example, to provide such a function with the ability to retrieve several keys (and their associated reference bits) in a single hardware storage cycle, thus improving overall performance of the function.

Specialized, often inexpensive, hardware can be used for many of the parallel function candidates due to the simplicity of the functions executed (i.e., no floating-point or complex-variable operations are required).

The above are representative of the type of function that is likely to migrate, but it is important to understand the criteria used in selecting specific functions and interfaces. In looking at functions that could be asynchronous to determine their potential for offloading, and in designing the interfaces to the offloaded function, it is important to exercise great care in finding "natural" breakpoints. Natural breakpoints are logical boundaries in the process that (a) have only small amounts of required shared state information with the invoking process, and (b) occur at or after a detected and required task or major state change so that, for example, the cache disruption in a mainframe CPU and the overhead of the communication is not added to the disruptive effects in the system.

An example is the sequence of functions involved in processing data requests from an application program or a mainframe CPU to a database stored on an external disk. Here, there are many obvious breakpoints at which one could envision offloading all subsequent processing to a parallel process

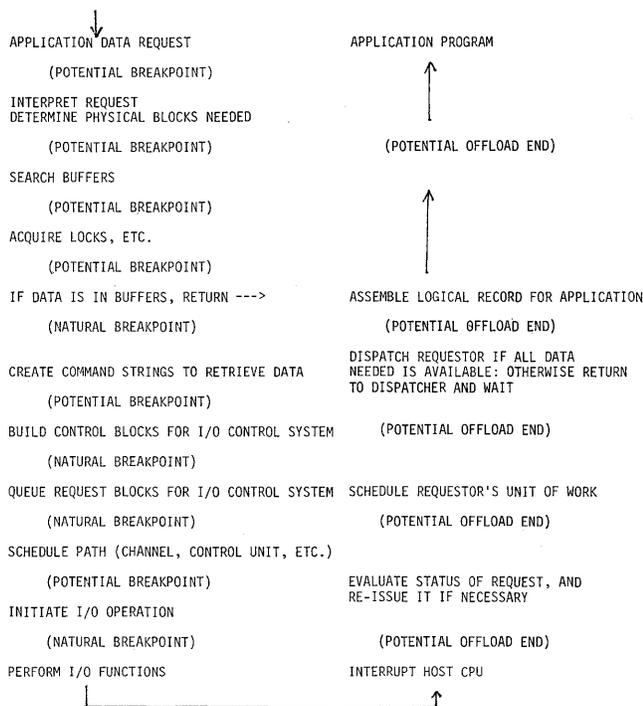


Figure 1—Possible breakdowns

or parallel hardware. But after examining these candidates carefully, there are several which have significant negative impact upon the performance of the system and therefore are not "natural breakpoints." Figure 1 shows some of the possible breakpoints that occur during the processing of a typical data management request on a System/370 under the MVS operating system.

In evaluating each of the possible breakpoints where parallel function might commence, it is important to look at the potential impact upon the system as well as the potential benefits. The first and most obvious breakpoint is the application interface. This interface, often referred to as the "data base/machine" interface, would at first glance provide the easiest and cleanest possible breakpoint, but on closer examination, it is found that in today's large-scale processors this is not really the case. Large CPUs with instruction lookahead, pipelining, storage caches, etc., operate best when executing instructions in a sequential fashion. Breaking the pipe, dispatching new units of work that would clear the cache, overlying the translation lookahead buffers and resetting the status of the machine are expensive in execution performance. In most data base systems, large in-storage buffers are used to store significant portions of the actively used data and indexes. As a result of this (and the clustering or packing algorithms employed in many systems to allow placing related items together in physical storage), many of the data requests find the data already in the buffers and require no asynchronous execution. Often more than 98% of the total data accesses to the data base are resolved without additional I/O processing. Because of all the hardware penalties mentioned above, it would be far better not to take a breakpoint at the request interface if the data is already in storage, but rather to continue resolving the reference and return in an in-line processing se-

quence. In smaller CPUs without much pipelining this trade-off becomes more difficult to evaluate, since the impact of a transition from one unit of work to another is relatively less costly to system performance than it is on larger machines where the impacts are obvious.

Creating a breakpoint at the physical record selection interface will have the same drawbacks as doing so at the application program data/request interface. After physical record selection has occurred, a local buffer search is performed to determine if the data required currently resides in high-speed storage. It is only at the point where buffer miss is detected (data not in storage, or page fault on the data if the machine is built on a "one-level store" virtual memory model) that this process has reached an interface in this sequence where a disruption is assured. This interface crossing indicates then that at some point asynchronous I/O processing will occur. The interface corresponds to the need to obtain data from an external device, and the system is consequently forced to make the transition to other work at some time as a result since the external devices are very slow relative to internal processor speeds. Creating a breakpoint here incurs no disruptions beyond those occurring in any case due to this external I/O processing. This is to say that a change to another unit of work, status saving, pipeline breaking, and cache disruption will occur in any case as a result of not finding the data in storage. Thus, this is a natural breakpoint unless the data resides on a device whose access time and transfer rate are such that the data could be made available in less time than it would take to perform the transition to another unit of work. (If such a large, fast store were available, no asynchronism would be implied anywhere in the entire process, and this device could be viewed as part of the memory buffer.)

This interface probably provides the cleanest breakpoint considering the machine architectures, software systems, and implementations in use today. The drawbacks in choosing this interface as a breakpoint are related to the diversity of underlying access method functions, stored data formats and sequences that would have to be programmed in the parallel engine(s). Although this interface is attractive technically, pragmatically it is improbable that it will be implemented quickly as a new breakpoint in the I/O process because of implementation complexity; we suspect that it will rather begin to appear in an evolutionary fashion. Additional strategies for asynchronous processing/outboard migration, including detailed I/O breakpoint strategies are examined in a paper by Heller.¹

PROBLEMS TO BE SOLVED

Like microcoding in general, vertical and outboard migration applied in industry and university projects have often been done without adequate support in methodologies and tools, based on ad hoc techniques and much special casing. Some typical problems include:

1. Identifying bottlenecks on a particular system, a very difficult measurement problem, requiring intimate knowledge of workload characteristics as well as of hardware and software behavior. Seldom does simply identi-

ifying CPU-intensive processes suffice, especially in today's multiprogramming and multiprocessor target and host configurations with much shared use of busses, memories, buffers, lookasides and peripherals.

2. Predicting and verifying performance improvement due to migrations, especially when migrations interact.^{3,7} That is, migrating one primitive typically affects the potential improvement to be expected from migrating a related primitive in ways that are not immediately obvious.
3. Doing software-to-firmware migration in ways other than recoding in a lower-level language—the ideal of a high-level language compiler which compiles both to optimized target code and to vertical (let alone horizontal) microcode is far from generalized due to both current host machine idiosyncracies and the language level at which most systems code is currently written. (Compiling design automation directives for actual hardware construction is even more unrealized today).
4. Finding systematic techniques for migrating functions and their data structures which minimize shared state and potential interlocks and do not impact the interfaces to related nonmigrated functions—or, alternatively, migrating and improving the migrated and non-migrated functions and their interfaces simultaneously.
5. Verifying the correctness of a migration (assuming that the original code was either formally or experimentally verified to be correct). Establishing modularization techniques and interface design which create more opportunities for migrating functions is a related problem.
6. Creating technology transfer—since user-friendly tools and methodologies can give an enormous boost to the speed with which migrations can be done, they tend to remain company-proprietary, rather than being put in the public domain via official publications.

FUTURE DEVELOPMENTS

We can expect that all the techniques which have been applied to tune main-frames will now come into use in the design of today's sophisticated microprocessor chips; these chips may in fact afford even more opportunity for vertical migration in that some have lower levels of nano- and even picoprogramming. Designers of small-scale systems may have to deal with design rules or constraints that may not apply to larger systems, however. For instance, the disparity between processor and memory speeds in NMOS or CMOS systems is not as great as in large-scale systems; this will affect interface boundaries and placement of functions. Furthermore, the temptation to migrate large numbers of functions directly into logic is strong for microprocessor designers, yet the impact on overall system performance may be quite different from that

of placing those same functions in microcode, because more complex circuitry and longer data paths are required for random logic than for more regular memory.

As chips become more complex, increasingly more micro-coded operating system support will be seen, for instance for object, process, and capability management.^{8,9} Also, control stores will become large enough to allow direct compilation to microcode rather than interpretation of a compact intermediate language code. Many higher-level facilities will be migrated after a period of experimentation at the software level. For this reason chip manufacturers have left uncommitted (PLA) areas on their chips or have separate control store chips, as on the Motorola 68000. A third trend will be increasingly parallel architectures using outboard migration as a way of seeking architectural remedies to technological limitations. It is hoped that increasing integration of all three migration techniques will be seen, using as much as possible the same tools and methodologies.

As a final note, it is expected that unfortunately there will be no equivalent in the migration field of the Mead/Conway phenomenon¹⁰ of do-it-yourself (amateur) VLSI chip specification (at least for simple chips). A deep knowledge of hardware-firmware-software architecture will continue to be needed to understand where migration can be useful and how it can be accomplished with minimum restructuring. The lack of broadly educated computer professionals who can deal with this complex problem area will continue to be a major bottleneck to progress in the field.

REFERENCES

1. R.A. Heller, "Experiences and opportunities in vertical migration of computing functions", in *Proceedings of a Conference on Information Processing*, Kiel, March 1980.
2. S.H. Fuller, V.R. Lesser, C.G. Bell and C.H. Kaman, "The effects of emerging technology and emulation requirements on microprogramming", *IEEE Trans. Computers*, vol. 25, no. 10, October 1976.
3. John Stockenberg and Andries van Dam, "Vertical migration for performance enhancement in layered hardware/firmware/software systems", *IEEE Computer*-vol. 11, pp. 35-50, May 1978.
4. Amdahl Corporation, Amdahl 5860 product announcement, November 1980.
5. G. Chroust, A. Kreuzer and K. Stadler, "A microprogrammed page fault monitor", Kepler University, Linz, Informatik-Berichte: SYSPRO 1980.
6. E. Feilmair and K. Stadler, "A CSECT monitor to measure program flow", Institute for Informatics, J. Kepler University, Linz, 1980.
7. J.A. Stankovic, "Structured systems and their performance improvement through vertical migration", Ph.D. thesis and Technical Report CS-41, Dept. of Computer Science, Brown University, May 1979.
8. G.J. Myers and B.R.S. Buckingham, "A hardware implementation of capability-based addressing", *Computer Architecture News*, vol. 8, no. 6, pp. 12-24, October 1980.
9. J. Rattner and G. Cox, "Object-based computer architecture", *Computer Architecture News*, vol. 8, no. 6, pp. 4-11, October 1980.
10. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1979.

Firmware testing and test data selection

by HELMUT K. BERG

Honeywell, Inc.
Bloomington, Minnesota

INTRODUCTION

Loosely stated, program verification is the task of systematically demonstrating that a program achieves its intended purpose, i.e., the task of proving the absence of errors from a program. This task may be considered as a backward mapping from a given program to a statement of the requirements for that program. There exist two fundamental approaches to establishing such mappings. A program may be exercised for a specific set of input values; the successful completion of program execution constitutes a necessary condition for the correctness of that program. A more rigorous approach is to provide an argument that a program satisfies its requirements for all legitimate input values, thus constituting a necessary and sufficient condition for the correctness of that program. In this paper, we are concerned with the first of these two fundamental approaches.

Verification techniques are dependent on the notations and concepts used to express requirements and programs. Central to verification is the availability of a specification, i.e., an unambiguous representation of the requirements definition. It embodies the system requirements in the sense that any system which fits this description will satisfy the requirements. With respect to verification, specifications constitute redundant information which is necessary for the construction of a backward mapping from a program or program system to a statement of the system requirements, i.e., to carry out the verification.

Verification methods are related to the approach taken to specify the semantics of program systems, programs, and programming constructs. Three main approaches to semantic definitions of programming constructs (i.e., programming languages) have developed. These approaches are: the operational approach, the denotational approach¹ and the axiomatic approach.^{2,3} All three approaches have been applied to firmware verification.⁴ In this paper, we are concerned with the operational approach.

In the *operational approach*, the semantics of the programming constructs of a programming language are defined by virtue of a more primitive programming language. The semantic definition specifies, for each programming construct, a *translation* into the constructs of the primitive language. With respect to verification, the operational approach requires the execution of (a trace through) programs written

in the primitive programming language. The effect of program execution may then be determined by observing how the state is changed by the individual transformations. Obviously, this method of determining the semantics of programs is applicable only to specific sets of input values. Hence, a program is verified by demonstrating the equivalence between results of program executions for specific sets of input values and corresponding execution-independent specifications of the expected results. The concept underlying this verification method, which maps between individual program states, is the commonly known concept of *program testing*. The specification of the particular sets of input states and corresponding output states constitutes the definition of *test data*.

We offer the following definitions of correctness proof and testing,⁵ noting that both verification methods must be supported by techniques and tools for debugging.

- *Correctness proof* is the attempt to show the absence of errors from a program.
- *Testing* is the attempt to show the presence of errors in a program.
- *Debugging* is the attempt to locate and correct known errors in a program.

Theoretically, correctness proof is a stronger argument about the correctness of a program than testing. Correctness proofs demonstrate correctness for the domain of all legitimate initial states, whereas a test can demonstrate the correct behavior of a program only for a particular initial state. Consequently, testing is inherently limited, because the number of test cases required to completely exercise even a small program may be prohibitively large or even infinite. Nevertheless, testing is by far the most widely used verification technique. However, neither proofs nor tests can, in practice, provide complete assurance that programs will not fail.⁶ Tests actually provide information about a program's behavior in its actual environment, whereas proofs are limited to conclusions about behavior in a postulated environment.

DIFFERENTIATING METHODS OF TESTING

Testing techniques in general, and firmware testing techniques, in particular, may be differentiated by:

- the types of errors,
- the level at which tests are carried out,
- the approach used to carry out tests,
- the environment in which tests are carried out,
- the test strategy,
- the test data selection.

To elaborate on these criteria, we provide the following definitions. We call the deviation of a system's behavior from its specification a *failure*. An *error* is caused by a *fault* and leads to an *erroneous state* which, in turn, indicates a failure. By *error detection* we mean the recognition of the effect of an error, i.e., of an erroneous state; by *error location* we mean the recognition of the cause of an error, i.e., of a fault.

Errors can be classified as performance errors and logic errors.⁶ *Performance errors* lead to results which are not within specified time and space limitations. *Logic errors* lead to erroneous results and may be classified further into:

- *requirement errors*, which lead to failure with respect to real requirements as given by the purpose of a system;
- *specification/design errors*, which lead to failure with respect to understood requirements as given by the requirements definition; and
- *construction errors*, which lead to failure with respect to the specification or the design.

At higher levels of abstraction software testing does not deviate essentially from firmware testing. Therefore, we concentrate on microprogram tests, which are carried out at the following three levels:

- Tests at the *microprogram level* consider complete microprograms by either analyzing their code or investigating the machine states resulting from their execution.
- Tests at the *microinstruction level* consider single microinstructions by either analyzing the assignment of microoperations to them or investigating the machine states resulting from their execution.
- Tests at the *microoperation level* consider individual microoperations by monitoring their execution.

Two major approaches to testing have developed.⁷

- *Static analysis* is concerned with investigating the logical structure of a program.
- *Dynamic testing* is concerned with investigating the execution of a program.

The following two test environments may be distinguished for firmware tests.

- The test environment for *on-line tests* is the physical processor upon which the microprograms under test are executed.
- The test environment for *off-line tests* is independent of the physical processor upon which the microprograms under test are executed.

There are three general testing strategies.⁷

- *Bottom-up testing* starts with low level test modules that are gradually integrated into higher level test modules until the integration of the entire system is tested.
- *Top-down testing* starts with tests at the system level, using dummy lower level modules, and gradually embodies the lower level modules until all test modules at the lowest level are tested.
- *Mixed Testing* is predominantly top-down, but bottom-up testing is used on certain test modules.

Two fundamental properties of test data selection criteria have been identified.⁶

- *Reliable test data criteria* ensure selection of test data which are consistent in their ability to reveal errors as opposed to necessarily being able to detect all errors.
- *Valid test data criteria* ensure that for every error there exists a complete set of test data capable of revealing the error as opposed to necessarily being able to select such a set of test data.

A FUNDAMENTAL THEOREM OF TESTING

Before we proceed to descriptions of program correctness and the concepts of test reliability and validity, we introduce the notions of abstract machines and abstract programs.

Abstract Machines, Abstract Programs, States, and State Spaces

An *abstract machine*, A , is defined by a pair, $A = (d, F)$, where d is the state of A , and F is a set of transformations for effecting state changes. The transformations $f_i \in F$ act upon a set of data objects (O_1, \dots, O_n) ; the state d is given by the states of the data objects, O_i .

A *data object*, O , is defined by a triple, $O = (n, v, t)$, where n is its *name*, v is its *value*, and t is its *type*. The name of a data object may be used to reference the object in a transformation description. The value of a data object defines the *state* of the object. The type of a data object defines the form of its value (e.g., integer, real) and the operations which may legitimately be performed upon it. The state of an abstract machine depends upon the execution of an abstract program on that abstract machine, as the state is given by the collection of the values of the data objects referenced by that abstract program.

It is convenient for the treatment of verification, to associate a state with an *abstract program*, rather than with the appropriate abstract machine. Then, the *state* defines the effect of executing a program for a specific set of input values up to a certain point. Additionally, a *state space*, D , of a program may be defined as the cartesian product, $D = D_1 \times \dots \times D_n$, of the sets, D_i , of the legitimate states (value ranges) of all the data objects, O_i , referenced by that program. Legitimate sets of input and output values of a program, which are of the form (v_1, \dots, v_n) , may then be identified by defining subspaces of the state space, D .

Tests, Test Cases and Program Correctness

A (micro)program, M , consists of a set of (micro)instructions, m_i , i.e., $M = m_1; \dots; m_n$. The execution of a program, M , is a function, E , whose domain is the set of legitimate initial states and whose range is the set of legitimate final states of M .⁸ We call the function, E , the execution function, and denote the set of legitimate initial states and the set of legitimate final states of a program, M , by ϕ and ψ , respectively. Let the initial state as defined by a particular set of input values be denoted by d_0 . Then, the final state, d_f , resulting from the execution of a program, M , for an initial state, d_0 , is defined, $d_f = E(M, d_0)$. The execution $E(M, d_0)$, of a program, M , is defined if $d_0 \in \phi$ and M terminates; the execution, $E(M, d_0)$, of a program, M , is undefined, if $d_0 \notin \phi$ or M does not terminate.

A test case is a triple, $(d_j, m_p; \dots; m_r, d_k)$, which specifies that the execution of a code segment, $m_p; \dots; m_r$, of a program, $M = m_1; \dots; m_n$, is to terminate with state, d_k , when executed starting with state, d_j . A test is the attempt to verify a test case, $(d_j, m_p; \dots; m_r, d_k)$, i.e., the attempt to establish the equivalence, $E(m_p; \dots; m_r, d_j) = d_k$, by execution of $m_p; \dots; m_r$ for d_j . A program test attempts to verify test cases of the form, (d_0, M, d_f) .

We are now in the position to define the correctness of a program. A program, M , is correct, if $(\forall d_0 \in \phi) (E(M, d_0) = d_f)$. We recognize that program correctness cannot readily be established by testing, as the set ϕ may be infinite, and consequently, an infinite set of test cases would need to be specified and verified. Hence, testing can validate programs only by verifying an appropriate set of test cases. Therefore, we are interested in finding a finite sample of the set of legitimate initial states which can be used to establish program correctness.

An ideal test of a program M is defined by a subset $T \subseteq \phi$, if

$$(\forall d_0 \in T) (E(M, d_0) = d_f) \rightarrow (\forall d_0 \in \phi) (E(M, d_0) = d_f),$$

for ϕ , d_0 , and d_f as defined above.

That is, if from successful execution of M for all initial states in T , we can conclude the program correctness, then T constitutes an ideal test.

The Goodenough-Gerhart Theorem

The primary problem in testing reduces to the question as to which test data set T constitutes an ideal test for a given program M , or how thoroughly does T exercise M . Usually, a thorough test is equated with an exhaustive test, i.e., with $T = \phi$.^{9,10} However, as seen above, exhaustive tests generally are not feasible, hence this definition of a thorough test does not provide insight into problems of test data selection. Goodenough and Gerhart developed a formal basis of testing⁶ which is aimed at the definition of criteria for test data selection that ensure thorough tests. This theory is based on the following concepts.

Let T be a set of test data and let C be the criterion for

selecting T , then the completeness of a test data set is defined by the predicate COMPLETE(T, C), which ensures that the successful execution of all test cases derived from T implies no errors in the tested program. That is, C defines what properties of a program must be exercised to constitute a thorough test. A test is successful, if the execution of a program M for all test cases (d_0, M, d_f) derived from a set of test data conforms with the specification of the program M . This property is denoted by the predicate, SUCCESSFUL(T) = $(\forall d_0 \in T) (E(M, d_0) = d_f)$.

To ensure that a successful test with a complete set of test data implies the correctness of an ideal test, the test data selection criterion C must be reliable and valid. A test data selection criterion C is reliable, if and only if for every set of test data T with COMPLETE(T, C), we have SUCCESSFUL(T) or not SUCCESSFUL(T). Reliability of a test data selection criterion is defined by the predicate, RELIABLE(C) = $(\forall T_1, T_2 \subseteq \phi) ((\text{COMPLETE}(T_1, C) \text{ and } \text{COMPLETE}(T_2, C)) \rightarrow (\text{SUCCESSFUL}(T_1) \leftrightarrow \text{SUCCESSFUL}(T_2)))$. A test data selection criterion C is valid, if and only if for every error in a program there exists a set of test data T with COMPLETE(T, C) such that we have not SUCCESSFUL(T). Validity of a test data selection criterion is defined by the predicate VALID(C) = $(\forall d_0 \in \phi) (E(M, d_0) = d_f) \rightarrow (\exists T \subseteq \phi) (\text{COMPLETE}(T, C) \text{ and not } \text{SUCCESSFUL}(T))$.

We are now in the position to state the fundamental theorem of testing⁶:

$$(\exists T \subseteq \phi) ((\exists C) (\text{COMPLETE}(T, C) \text{ and } \text{RELIABLE}(C) \text{ and } \text{VALID}(C) \text{ and } \text{SUCCESSFUL}(T)) \rightarrow (\forall d_0 \in \phi) (E(M, d_0) = d_f))$$

The theorem demonstrates that tests with complete test data sets that have been derived by reliable and valid test data selection criteria are thorough in the sense of ideal tests. Although the theorem may alleviate the need for exhaustive testing, it leaves us with the problem of demonstrating the reliability and validity of the test data selection criteria. In effect, the theorem states that in some cases a test is a proof of correctness.⁶

LEVELS OF MICROPROGRAM TESTING

The application of the Goodenough-Gerhart theorem to firmware may lead to formal techniques of firmware testing and test data selection. Formal methods necessitate an abstract view of program semantics in which semantic definitions need to be restricted to program execution on some abstract machine that models the behavior of a real machine. In software verification, high level programming languages are generally considered as the abstract programming language of the abstract machine by which the behavior of the real machine is modelled. For firmware verification, hardware-independence cannot be achieved. Therefore, firmware verification requires less abstract models of machine behavior that capture all effects in the hardware that affect the semantics of microinstructions. In particular, the parallel execution of microoperations and the synchronization of asynchronous oper-

ations require considerable elaboration in firmware semantic definitions.

Testing of microprograms, microinstructions and microoperations

Following the discussion in the section "Tests, Test Cases, and Program Correctness" a *test at the microprogram level* attempts to verify test cases of the form (d_0, M, d_f) , i.e., it attempts to establish equivalences, $E(M, d_0) = d_f$. A *test at the microinstruction level* then attempts to verify tests cases of the form (d_{j-1}, m_j, d_j) where d_{j-1} and d_j are the states reached before and after the execution of the microinstruction, m_j , respectively.

A *microinstruction*, m_j , consists of a set of *microoperations*, μ_i , i.e., $m_j = (\mu_1, \dots, \mu_m)$. The execution of a single microinstruction may be divided into several *subcycles* in which individual microoperations are executed. Hence, during the execution of a microinstruction, *substates* may be reached that result from the execution of microoperations in a particular subcycle. We define a *subcycle* by a pair, (cp_s, cp_e) , where cp_s and cp_e denote the cycle points at which the subcycle starts and ends, respectively. A *cycle point* is a time instant within a microinstruction execution cycle at which the input to a functional hardware unit or register must be present, or after which the output of a functional hardware unit or register is available for use.¹¹ In general, cycle points are defined by clocks. We identify the *individual microoperations*, μ_i , in a *microinstruction*, m_j , by the notation, $E(\mu_i, d_{j-1}, cp_k) = d_j(cp_k)$, $k \in [1:S]$, where S is the number of subcycles in the execution of m_j . That is, $E(m_j, d_{j-1}, cp_k)$ denotes the substate reached after the execution of the k -th subcycle in the execution of the microinstruction, m_j . The substate, $d_j(cp_k)$, is effected by only those microoperations, $\mu_i(cp_{i_1}, cp_{i_2})$, with $cp_{i_2} \leq cp_k$, i.e., whose execution terminated before the cycle point, cp_k . Obviously, we have, $E(m_j, d_{j-1}, cp_s) = E(m_j, d_{j-1}) = d_j$, i.e., the substate reached after the last subcycle is the state reached after microinstruction execution.

The *effect of executing a single microoperation*, $\mu_i(cp_{i_1}, cp_{i_2})$, in a microinstruction, m_j , for a given state $d_j(cp_{i_1})$, is denoted $E(m_j, d_j(cp_{i_1}), cp_{i_2}) = d_j(\mu_i(cp_{i_1}, cp_{i_2}))$. Using this notation, we obtain the following definition. A *test at the microoperation level* attempts to verify test cases for the form, $(d_j(cp_{i_1}), m_j, d(\mu_i(cp_{i_1}, cp_{i_2})))$, i.e., it attempts to establish equivalences $E(m_j, d_j(cp_{i_1}), cp_{i_2}) = d_j(\mu_i(cp_{i_1}, cp_{i_2}))$.

The *effect of asynchronous microoperations* may be observed as follows. For an asynchronous microoperation, μ_i , the cycle point, cp_{i_2} , at which it terminates is not known a priori. Therefore, we denote an asynchronous microoperation by $\mu_i(cp_{i_1}, ?)$. Nevertheless, the termination of an asynchronous microoperation needs to be synchronized with an internal clock. Hence, if the execution of an asynchronous microoperation terminates, there exists a cycle point, cp_a , at which the effect can be recognized. That is, *tests of asynchronous microoperations* involve test cases of the form $(d_j(cp_{i_1}), m_j, d_k(\mu_i(cp_{i_1}, ?)))$, whose verification requires the determination of the cycle point, cp_a , to establish $(E(m_j, d_j(cp_{i_1}), cp_a) = d_k(\mu_i(cp_{i_1}, ?)))$, $k > j$.

The Goodenough-Gerhart Theorem and Microprogram Testing

The fundamental theorem of testing described above requires that the predicates COMPLETE(T, C), RELIABLE(C), VALID(C), and SUCCESSFUL(T), be established for sets of test data, T , which have been selected according to an appropriate test data selection criterion, C . The major concern of this activity is the proof of the reliability of the test data selection criterion, C . Studying this problem, it has been found⁶ that reliable tests need to be designed not so much to exercise program paths as to exercise paths under circumstances such that an error is detectable if one exists. Tests based solely on the internal structure of a program are likely to be unreliable. Rather, in order to find test data which reliably reveal errors, all conditions relevant to the correct operation of a program must be known. The existing theories of testings are not sufficiently powerful to identify, for any program, criteria for test data selection which can hope to yield reliable tests.

For microprograms, the characteristics of their implementation, including special conditions relevant to data representation and the internal structure of the program representations, need to be used to ensure that all combinations of conditions relevant to their operation can be identified. Particularly, the following characteristics need to be considered: grouping of microoperations in microinstructions, the microinstruction sequencing mechanism, the timing of microoperation executions, the resource binding of microoperations, etc. Consequently, microprogram tests need to consider microprogram implementations at all three levels of abstraction described above.

The following relationships between microprogram correctness, error detection, and error location can be established. An error which is detected at the microprogram level may be caused by any number of combination of faulty microinstructions or microoperations in the microprogram. A testing technique must support the location of such errors. An error is located at the microprogram level, if the set of faulty microinstructions or microoperations can be identified. To identify faulty microinstructions or microoperations in an erroneous microprogram, tests at the microinstruction level may become necessary. Such tests allow the tester to trace through the computation. Thus, tests at the microinstruction level provide insight into microprogram control flow and states reached after the individual steps of microprogram execution. This information may facilitate error location. An error detected at the microinstruction level is located, if the set of faulty microoperations in m_j can be identified.

For firmware tests at these levels, test data selection methods as the one defined by Goodenough and Gerhart⁶ may be applied. Their method distinguishes between test data and test predicates. *Test data* are the actual values for which the tests are performed. *Test predicates* describe conditions and combinations of conditions which are relevant to the program's correct operations. That is, the test predicates define the aspects which are to be tested, whereas the test data cause the aspects to be tested.

An evaluation of this method shows that the primary source of test predicates are the program specifications. A test pred-

icate analysis must be carried out so as to prove their reliability. This analysis forces analysis towards the abstract properties of a program and its specifications. It has been recognized that knowledge of the internal program structure by itself is insufficient to yield adequately reliable tests. In particular, it might become necessary to add further conditions and test predicates to those obtained from program specifications and the program's internal structure, when the program implementation is considered. On the other hand, test predicates may be eliminated without impairing test reliability, when knowledge of the actual implementation is used.

The impact of implementation details on the abstract properties of microprograms is crucial to firmware testing and test data selection. These implementation details refer primarily to the execution of the microoperations contained in a microinstruction. Let us assume that faulty microinstructions have been detected. Analysis of faulty microinstructions may be successful in locating the faulty microoperations that cause the error. However, in certain cases, analysis of faulty microinstructions cannot provide sufficient insight into the semantics of microinstructions. For example, errors may be caused by violations of time-dependent relationships between individual microoperations, yet these violations may not be recognizable from the microcode. The location of such errors requires insight into the *transiency of microoperation execution*. As a result, test cases must be defined that specify the expected functional behavior as well as the transient behavior of a machine during microinstruction execution. This information may be furnished by defining test cases in terms of substates that are reached due to microoperation execution in subcycles of the microinstruction execution cycle.

Errors that are detected at the microoperation level may either be firmware errors or hardware errors. The test cases used incorporate semantic definitions of microoperations as well as specifications of the associated transiency of the hardware. Consequently, the inability to verify a test case may be due to a faulty microoperation or to the fact that the behavior of the controlled hardware resources is not consistent with the specifications given in the test case. In order to be able to distinguish between firmware and hardware errors, tests at the microoperation level must lend themselves to error location. An *error detected at the microoperation level* is located, if the logical fault in the specification of a microoperation or a failure of the controlled hardware can be identified.

Note that the identification of a hardware failure does not correspond to the location of the underlying error. However, in firmware testing, we are not concerned with the location of hardware errors but are content with locating firmware errors. Theoretically, tests at the microoperation level are sufficient to locate firmware errors, as they allow us to observe the effects of executing microinstructions in the framework of control signals sent to individual hardware resources and subcycles that define the execution times of the initiated elementary operations.

Obviously, systematic test data selection methods do not readily apply to tests at the microoperation level. Questions such as how to incorporate the transiency of microoperation execution have not yet been considered in testing theory. Furthermore, predicates such as $COMPLETE(T,C)$, $RELIABLE(C)$, $VALID(C)$, and $SUCCESSFUL(T)$ can hard-

ly be established at the level of microoperations. For example, to establish the fundamental predicate $SUCCESSFUL(T)$, all the substates reached at the different cycle points during the execution of each microinstruction would have to be verified by appropriate test cases. No formal approach to the solution of this problem has been proposed to date. Rather, in practice, the ability to detect and locate errors at the microoperation level is heavily dependent on the tester's skill in defining test cases and the facilities available for observing the execution of microoperation. As an overview of firmware testing techniques¹² demonstrates, the typical strategy is that of mixed testing, which incorporates static analysis and dynamic testing. Usually, top-down testing is used, until unlocatable errors are encountered and then traced down using bottom-up testing at the microoperation level.

Although the mixed test strategy reduces the number of test cases to be verified, it is unsatisfactory from the theoretical view of establishing microprogram correctness. However, as is outlined in the concluding remarks, the knowledge available in high-level microprogramming language design, testing theory, and firmware correctness proofs may be exploited to develop a theory of firmware testing. Work towards this aim will be reported in a forthcoming paper.

CONCLUDING REMARKS

Although testing is the predominant firmware verification method, many software test practices⁷ have not yet been adapted to firmware testing. Resulting deficiencies of firmware test methods concern in particular the development of systematic test strategies, the ability to carry out firmware tests at the level of high-level microprogramming languages, and the ability to combine firmware tests with execution time measurements. Further work on theoretical foundations of firmware testing is needed in order to establish a firm basis for the development of solutions to these problems. Theoretical approaches to software testing may be an appropriate starting ground for such an endeavor. However, it must be realized that to date we know less about the theory of testing than about the theory of correctness proofs.

Despite the fact that approaches to firmware testing are in their infancies, trends in firmware engineering look promising. The development of firmware engineering disciplines concentrates on rules for the systematic execution of working procedures in the specification, design, construction, verification, documentation, and maintenance of firmware.¹² Progress in this area will have a decisive impact on firmware testing, as testing is in fact performed in one way or another at every stage of the firmware development process. This need for testing in all stages of the development process requires the integration of techniques for specification, design construction and certification into comprehensive strategies.¹³

Most important in this respect is the development of high-level microprogramming languages that make the idiosyncrasies of the microprogram running environment transparent to the microprogrammer. To be successful with respect to firmware testing, high-level microprogramming languages must allow for the application of machine-independent software verification techniques. However, at the same time ac-

cess to target machine data and functional resources must be provided to facilitate resource binding, microcode optimization must be supported, and machine-specificities of the microprogram running environment need to be incorporated into the language concepts. Testing and test data selection at a machine-independent level must ultimately be related to the specific microprogram running environment.

Approaches to solutions to these problems include axiomatization of the running environment,^{14,15} explicit descriptions of the microprogram environment,¹⁶ correctness proofs of microprogram source code,^{17,18,19} and machine virtualization.^{20,21} It will be interesting to see to what extent these efforts will alleviate some of the problems identified in this paper and how these approaches could be integrated into a comprehensive theoretical basis for firmware testing.

ACKNOWLEDGMENT

The author wishes to express his gratitude to Professor T.G. Lewis of Oregon State University for his help in the planning phase of this paper and valuable discussions on the subject matter. These discussions led to the start of joint investigations of theoretical foundations of firmware testing.

REFERENCES

1. Stoy, J.E., "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", The MIT Press, Cambridge, MA, 1977.
2. Floyd, R.W., "Assigning Meaning to Programs", *Proc. of Symposia in Applied Mathematics*, American Mathematical Society, Vol. 19, 1967, pp. 19-32.
3. Hoare, C.A.R., "An Axiomatic Approach to Computer Programming," *Comm. of the ACM*, Vol. 12, No. 10, 1969, pp. 576-583.
4. Berg, H.K., "Correctness of Firmware-An Overview", *Firmware Engineering, Informatik Fachberichte*, Vol. 31, Springer-Verlag, 1980, pp. 173-224.
5. Dijkstra, E.W., "Notes on Structured Programming," Technical University Eindhoven, Tech. Report EWD 149, April 1970.
6. Goodenough, J.B.; Gerhart, S.L., "Toward a Theory of Test Data Selection", *IEEE Trans on Software Engineering*, Vol. 1, No. 2, 1975, pp. 20-37.
7. Fairley, R.E., "Tutorial: Static Analysis and Dynamic Testing", *Computer*, Vol. 11, No. 4, 1978, pp. 14-23.
8. Berg, H.K.; Boebert, W.E.; Franta, W.R.; Moher, T.G., "A Survey of Formal Methods of Program Verification and Specification", Digital Systems Program, University of Minnesota, DSP-79-02.
9. Poole, P.C., "Debugging and Testing" *Advanced Course on Software Engineering*, Springer-Verlag, 1973, pp. 278-318.
10. Stucki, L.G.; Svegel, N.P., "Software Automated Verification System Study", McDonnell Douglas Astronautics Corp., Rep. AD-784086, 1974.
11. Berg, H.K., "A Model of Timing Characteristics in Computer Control", *EUROMICRO Journal*, Vol. 5, No. 4, 1979, pp. 2067-218.
12. Davidson, S.; Shriver, B.D., "Firmware Engineering: An Extensive Update", *Firmware, Microprogramming, and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 1-40.
13. Lewis, T.G.; Malik, K.; Ma, P.-Y., "Firmware Engineering Using a High-Level Microprogramming System to Implement Virtual Instruction Set Processors", *Firmware, Microprogramming and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 65-87.
14. Dasgupta, S., "Some Implications of Programming Methodology for Microprogramming Language Design", *Firmware, Microprogramming, and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 243-252.
15. Berg, H.K.; Franta, W.R., "Firmware Engineering: Critical Remarks and a Proposed Strategy", *Firmware, Microprogramming and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 41-64.
16. Richter, L., "High-Level Language Extensions for Micro-Code Generation and Verification", *Firmware, Microprogramming and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 233-242.
17. Crocker, S.D.; Marcus, L.; van-Mierop, D., "The ISI Microcode Verification System", *Firmware, Microprogramming and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 89-103.
18. Patterson, D.A., "STRUM: Structured Microprogramming System for Correct Firmware", *IEEE Trans. on Computers*, Vol. C-25, No. 10, 1976, pp. 974-986.
19. Carter, W.C.; Joyner, W.H.; Brand, D., "Microprogram Verification Considered Necessary," *Proc NCC 1978*, AFIPS Conference Proceedings, Vol. 47, 1978, pp. 657-664.
20. Malik, K.; Lewis, T.G., "Design Objectives for High Level Microprogramming Languages", *Proc MICRO II*, IEEE Cat. No. 78CH1411-8, 1978, pp. 154-160.
21. Davidson, S.; Shriver, B.D., "MARBLE: A High Level Machine Independent Language for Microprogramming", *Firmware, Microprogramming, and Restructurable Hardware*, North-Holland Publ. Co., 1980, pp. 253-266.

Specifying target resources in a machine independent higher level language

by SCOTT DAVIDSON

Western Electric Engineering Research Center
Princeton, New Jersey

and

BRUCE D. SHRIVER

University of Southwestern Louisiana
Lafayette, Louisiana

INTRODUCTION

Each Higher Level Language (HLL) defines a virtual machine. A compiler for a HLL translates a program written for this virtual machine into a program for a lower level target virtual machine, which could be at the operating system, machine language, microcode, or other level. The space and time needed to run the compiled program are influenced by the difference between the functionality and semantics of the source and target virtual machines (semantic gap). As part of the translation process, a compiler *binds* source language constructs to target language constructs. For example, a variable in a program will be bound to a memory location, and an operator in a program (such as Plus) will be bound to a target machine functional unit (such as an Adder).

This binding is under compiler control in a machine independent HLL. This has two disadvantages. The first is that current HLLs generate code that makes use of only a subset of the available target virtual machine resources.¹ This means that the compiler for a HLL will often not be able to generate code that makes use of the target resources that would most efficiently implement the source program. Consider a FORTRAN program to do matrix multiplication. The semantics of FORTRAN requires that code must be written to do this operation component by component. Now consider a target machine having matrix multiplication hardware available. The object code produced for this program will take many instructions to perform a task for which one specialized instruction would be adequate. The resulting program is inefficient when compared to a program produced by a programmer writing in a language that allows use of the matrix multiplication hardware. No amount of optimization would eliminate this inefficiency.

One way of solving this problem would be to construct a compiler that would recognize that a segment of the source program was doing matrix multiplication and compile this segment into a matrix-multiply instruction. As will be argued below, this procedure is generally infeasible. Another solu-

tion would be to incorporate a matrix-multiply operator into the source language. This does solve the problem, but the resulting language is no longer machine independent. This is not a general solution to the problem, because a language based on this principle would have to include every possible target resource of every possible target machine.

A second difficulty with contemporary machine independent HLLs is that at times the language will not allow the construction of a program that requires the use of certain target machine resources. For instance, a Pascal program does not have access to the interrupt vector of a minicomputer. For this reason, operating systems written using machine-independent HLLs have had to include some routines written in a machine-dependent language, such as assembly language. These routines have access to the full instruction set of the target.

The problem of how to recognize the similarities between source and target constructs and how to bind the source constructs to the appropriate target constructs is called the *resource binding problem*.

This paper describes a machine independent solution to the resource binding problem. The Virtual Machine Resource Binding Language (MARBLE) has been designed and implemented to allow experiments to be performed to test the usefulness of this solution. This paper describes MARBLE, experiments conducted with the MARBLE compiler and the results of these experiments.

A SOLUTION TO THE RESOURCE BINDING PROBLEM

Placing Resource Binding

We begin our attempt at finding a solution to the resource binding problem by examining where binding is done in current translation systems. Figure 1 shows the phases of a typical HLL compiler.

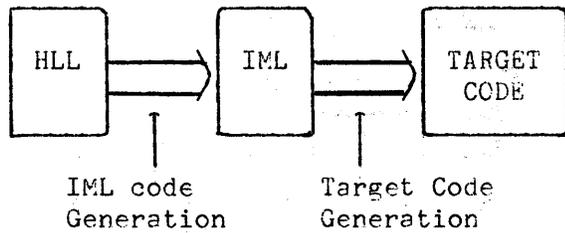


Figure 1—Phases of a typical HLL compiler

When the HLL is machine independent, a program in the HLL is usually translated into an Intermediate Language (IML) form. The IML, which is usually also machine independent, serves to simplify the compiler writing process by reducing the semantic gap between source and target, and serves to improve the portability of the compiler by making only the construction of a new IML to target translation phase necessary in order to move the compiler to a new machine.

For the case of a machine independent HLL, binding takes place in the IML to target translation phase. IML instructions are mapped into sequences of target instructions, and IML data resources are bound to target resources. It is also possible that a sequence of IML instructions can be mapped into a single target instruction.

Pattern matching techniques such as those used by Leverett et al.² can be used to recognize that some simple sequences of IML code represent specific target resources. However, more complex resources (such as a matrix-multiply functional unit) can be described in a number of ways in the source, and are therefore impractical to recognize.

For machine dependent HLLs, binding is done in the language itself, by building the resources into the language. Programs written in such languages are not portable except with great difficulty.

Where should binding be done in a resource binding language? Binding language constructs to target constructs at language definition time is not acceptable in a machine independent language. Waiting until machine independent IML code is generated from the source program is not acceptable because too much information about the intended use of target resources has been lost. A combination of both approaches must be used.

In order for the language to be machine independent, and thus allow programs written in the language to be portable, programs written in the language must be composed of machine independent constructs. For the reasons given above, the compiler should include an IML, and therefore a path from the HLL through the IML to the target code. Suppose a method was found for doing resource binding before the source code was translated into IML code. Bound code (code using data or functional resources to be bound to a specific target resource) cannot be translated into IML code, or else the information on binding will be lost. Therefore, a method must be devised by which the information on target constructs identified as being bound in the source program can be transmitted to the target code generation phase.

A solution to this problem lies in the definition of a Transmittal Phase (TP) around the IML phase. Those portions of

the source program representing target constructs are translated directly into these target constructs by the TP. Figure 2 shows the TP added to the compiler system. Note that a High Level Intermediate Language (HLIML) and a Low Level Intermediate Language (LLIML) have been added. The path through the compiler containing the IML is called the unbound path, and the path including the TP is called the bound path.

As mentioned above, machine dependent constructs cannot be built into the HLL without violating the goal of machine independence. However, it is possible to build models of machine dependent resources using only machine independent components. These models can be associated with the appropriate target constructs, and source code using these models can be efficiently translated into target code using the target resources. The responsibility for determining which target resource is to be used in a given situation is thus placed with the user, not with the translation system. However, the user should not be required to construct models of all parts of the target machine. A library of MARBLE representations of target resources could be available for this purpose. A novice user should be able to use target resources, but the more experienced the user, the more efficient will be the resulting code. A user not wishing to make explicit use of target resources need not use the models at all. Since the user can turn binding on and off, the user can direct the compiler to use specific resources, thus solving the second part of the resource binding problem.

Modeling Target Resources

There are three types of target virtual machine resources that can be modeled: data resources, functional resources, and control resources. In this section we briefly describe how these resources can be modeled in MARBLE. Some MARBLE features will be introduced for this purpose. A complete description of the syntax and semantics of MARBLE is given by Davidson.³

A natural way to model a data resource is with a data type. A type defines the size of the resource, which operators and functions can use the resource, and the structure of the resource. This mechanism keeps the use of a data resource in an algorithm distinct from the structure of that resource defined by the type.

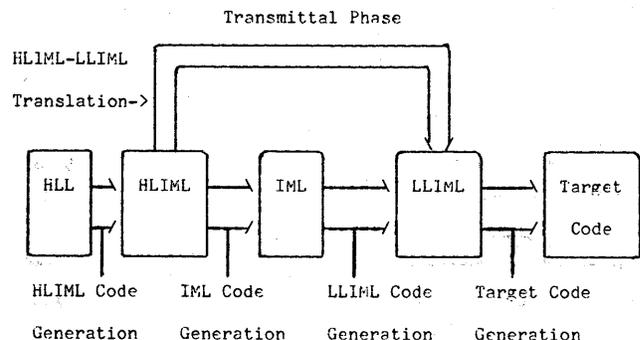


Figure 2—Transmittal phase added to compiler system

As a simple example consider a 16-bit target machine register, named `reg_a`. Using a type specification with the MARBLE base type `bit`, this register should be modeled as

```
type reg_a = bit 16;
```

The type-definition facility of MARBLE is similar to that of Pascal.⁴ This definition serves as a template, stating that any variable declared to be of type `reg_a` will be allocated 16 bits of storage. The declaration

```
var counter: reg_a;
```

makes this connection.

We bind a type to a target data resource through the use of the MARBLE construct named `bound` or “%” placed before the type name. If `reg_a` is to be bound, the type definition above would be written

```
type bound reg_a = bit 16;
```

and the variable declaration would be written

```
var counter: bound reg_a;
```

The effect of declaring a variable to be of bound type is that the variable is allocated to the target data resource represented by the type. All values assigned to the variable are stored in the target resource. In this way bound variables are distinguished from unbound variables, which are allocated space in the main store of the target machine.

Target functional resources can be modeled by MARBLE procedures and functions. The set of target functional resources is composed of the instruction set of the target virtual machine, excluding target control operations. The input and output resources required by a target functional resource are specified by the parameters of the MARBLE procedure or function that models this resource. The header of a multiply function that multiplies the numbers in the 16-bit registers `reg_a` and `reg_b`, and that places the results in the 32-bit register `reg_c` can be written

```
function bound multiply (in mier: bound reg_a;
                        in mcand: bound reg_b)
  returns bound reg_c;
```

The keyword `bound` before the function name means that if binding is turned on, this function is associated with a target functional resource, and parameters of bound type are associated with the proper input and output target data resources.

The semantics of a bound function are implemented by the target machine. When binding is turned on, the body of the function is ignored. A call to the bound function will compile into the appropriate target instruction, which will be inserted in-line; therefore there is no procedure call overhead. If binding is turned off, the body of the function will be compiled, and a call to the function will compile into a target machine subroutine call. If the function is never to be unbound (perhaps because the program is never to be ported) the body of the function can be replaced by the keyword `bound`. This

makes unnecessary the sometimes difficult job of writing code to model the semantics of the target-functional resource if that code will never be executed.

Target control resources, unlike data and functional resources, can be neither modeled nor bound in MARBLE. There are two reasons for this policy. The first is that the effect of a target control resource may not be limited to a well defined area of the program. For instance, the destination of a PDP-11 JMP instruction can be taken from a register, and therefore may be an unconstrained run-time value. This makes control flow analysis of the program impossible, and therefore makes the analysis necessary for register allocation and optimization difficult or impossible.

Second, those control constructs usually identified with “structured programming” can be easily translated into control constructs implemented by mechanisms that are fairly uniform over a wide range of target architectures (conditional and unconditional branches). The IML contains these constructs, which allows the binding of source control constructs to target constructs to be done efficiently at IML code generation time.

In summary, target data resources are modeled by MARBLE types, target functional resources are modeled by MARBLE procedures and functions, and target control resources are available only through high level control constructs. The models defined in a MARBLE program are models of the constructs of a specific virtual machine. These models are mapped into the appropriate target resources when they are bound by use of the keyword `bound`. Since all programs are built from machine independent constructs, the resulting program is guaranteed to be a valid MARBLE program.

Previous Work on Resource Binding

One field in which a solution to the resource binding problem is particularly necessary is that of microprogramming. At present, no widely accepted machine independent higher level microprogramming language exists.⁵ The diversity of microprogrammable architectures, and the requirement for the generation of efficient microcode, have contributed to this lack. In order to make efficient use of low level machine resources, the resource binding problem must be solved. Previous work in this area has been done by DeWitt,⁶ Dasgupta,⁷ Richter,⁸ and Giloi.⁹ Davidson³ reviews this work in detail.

AN IMPLEMENTATION OF MARBLE

This section describes an implementation of MARBLE as a microprogramming language. As noted in the previous section, the lack of an acceptable high level language for microprogramming has motivated research into resource binding. Thus, the implementation of a successful microprogramming language would be an indication of the validity of a solution to the resource binding problem. Here successful means both machine-independent and efficient. Though this implementation of MARBLE is as a microprogramming language,

it could be used for other types of target machines; thus MARBLE is not inherently a microprogramming language.

The MARBLE Language

The goal of the design of MARBLE was to produce a language that solved the resource binding problem and also kept the advantages of current HLLs. MARBLE is based on the language Pascal.⁴ A complete description of MARBLE is given by Davidson.³ The choice of Pascal to serve as a basis for the design of MARBLE was due to the usefulness of Pascal in the support of systematic programming. Other languages can be adopted to serve as resource binding languages, however; the major requirement is that the language have the facility to model target resources.

The type definition facilities of Pascal have been extended in MARBLE. The bit type, introduced above, allows a unsigned bit string to be declared. Arithmetic and logical operations are defined on variables of bit type. The view type allows several views of a data structure to be defined, with perhaps overlapping fields. Unlike most microprogramming languages, MARBLE has no bit extraction operator. Fields of a variable to be extracted must be defined in a view definition in the definition of the variable's type. This allows the fine structure of a target data resource to be modeled by a MARBLE program.

Shift operators have been added to MARBLE, and logical operators have been extended to work on variables of bit type. The goto has been replaced by the structured statement exit facility. An assert statement and case otherwise clause have been added to MARBLE.

The MARBLE compiler was implemented in four passes, and resembles the compiler of Figure 2. The HLIML was called the Intermediate Language 1 (IL1), and consisted of a control flow graph of the source MARBLE program. The IML consisted of the instruction set for a simple abstract microprogrammable machine. The LLIML closely resembled the code for the target machine, the Microdata 3200.¹⁰ The code produced by the compiler was both unoptimized and uncompact. Davidson³ gives further information about the MARBLE compiler.

EXPERIMENTS WITH THE MARBLE COMPILER

This section describes several experiments performed with the MARBLE compiler and the results of these experiments. Conclusions derived from these experiments will be presented in the next section.

Experiments

The purpose of the experiments with the MARBLE compiler described in this section was to determine whether the solution to the resource binding problem implemented by the compiler was satisfactory, both in allowing user-specifiable

TABLE I—Results of experiments with MARBLE compiler

Microprogram:	Prime	Microdata	
		Unbound	Bound
MARBLE statements	128	133	133
IL1 nodes	304	347	347
IML statements	684	709	113
Microdata instructions	856	885	264
IML registers used	7	8	2

resource binding and in being machine independent. The machine independence of the solution could be determined by writing a program using the target resources of one microprogrammable machine, and porting it to another microprogrammable machine. The success in allowing user-specifiable resource binding can be determined by examining the code produced by the compiler for a program using bound resources.

To meet these objectives, two MARBLE programs were designed and coded. The first was an implementation of a microprogram for the Prime 300 given as an example by Agrawala and Rauscher.¹¹ This program was to be compiled, unbound, into Microdata 3200 microcode. The second program constructed was a microprogram for the Microdata 3200, also taken from Agrawala and Rauscher.¹¹ The experiment consisted of compiling this Microdata 3200 program with binding turned both on and off. When compiled bound, the resulting Microdata 3200 program should make use of the target data and functional resources specified in the MARBLE program. The code resulting from the bound program can be compared with the code from the unbound program to determine whether resource binding resulted in an improvement in the efficiency of the generated microcode. The code can also be examined to confirm that binding is being done, and that the compiler is producing code that makes proper use of the target resources.

Results

The results of the experiments performed with the MARBLE compiler are given in Table I. These results are preliminary, in that only these two programs were compiled, and in that changes to the compiler pointed to by the results of the experiments were not made.

An examination of the IML and Microdata code produced for the Prime 300 microprogram revealed many places where optimization would be valuable. Although the code produced for the Prime 300 microprogram was not especially efficient, an examination of the resulting code (produced in the CAP-32 assembly language of the Microdata 3200) indicated that the semantics of the Prime 300 resources were accurately represented by the Microdata microprogram.

An examination of the code produced by the compiler for the Microdata microprograms bound and unbound indicates where the savings originate. Fifty-four percent of the savings

in IML code arises from discarding the bodies of bound functions and procedures when the program was run bound. Seventy-nine percent of the savings in Microdata microcode arose from this source. Eighty-one IML microinstructions were generated in order to call these procedures and functions when the program was run unbound; ninety-six Microdata microinstructions were required for this purpose. When this overhead is included, 62% of the savings in IML code and 94% of the savings in Microdata microcode results from the elimination of function and procedure calls. This is due to the simple types used by this program, which reduced the savings due to the use of bound data resources.

Discussion of Results

It is clear from Table I that the code produced by the MARBLE compiler for the Prime 300 program was not very good, in the sense that it was excessively long. Optimization and compaction would improve the code considerably, but the microprogram would still not be close in size to the assembly level microprogram, which was 22 Prime 300 horizontal microinstructions long. The greatest cause of this size is the procedure bodies and calls for the Prime 300 microoperations implemented as unbound functions. This expense was expected. The most significant result of the first experiment was that machine-dependent code for one microprogrammable computer was automatically compiled into code for another microprogrammable computer without alteration.

The code produced from the unbound version of the Microdata 3200 microprogram was also long and inefficient. This was for much the same reason as for the Prime 300 code. An examination of Table I indicates this quite clearly. Once the modeling overhead (procedure call overhead, procedures to implement bound functions, and code to extract components of data resource models) is removed, 264 Microdata microinstructions remain.

The assembly level microprogram to implement the Microdata 3200 microprogram used as data for the second experiment required 38 microinstructions, only 14% of the microcode generated by the compiler. There are several sources of this inefficiency. The most significant was that ALU registers were needlessly saved and restored. Optimization of the object code would reduce this problem. We estimate that this improvement, along with other optimization and compaction techniques, would produce code that would be approximately 100 microinstructions in length, three times the size of the assembly level microprogram.

There were two major results of these experiments. First, it was found to be possible to port a microprogram for one machine to another machine, with no changes to the microprogram required. Second, binding produces a microprogram 30% of the size of the microprogram when unbound. Most of this improvement is from the use of bound functions. The code generated when the program is compiled bound, unoptimized, and uncompact is 7 times the size of the equivalent assembly level microprogram. We believe that this gap could be reduced through the use of microcode compaction and optimization.

CONCLUSIONS AND FURTHER RESEARCH

The experiments described in the last section indicate that MARBLE provides one machine independent solution to the resource binding problem. The machine independence of the solution stems from the fact that all MARBLE programs are written using only machine independent constructs. An identifiable sequence of these constructs, representing a model of a target data or functional resource, can be bound to that resource. In this case, the code for the resource is discarded, and the code to use the appropriate target resource is inserted in the object code instead.

This mechanism allows a user access to specific target resources, as long as those resources can be modeled in MARBLE. Access to control resources is not allowed. All other resources of the target machine may be used.

The construction of the MARBLE compiler and the experiments performed with the compiler indicated several problems. The first is the difficulty of restricting access to variables of bound type to what is allowed by the target architecture. Though some such restrictions can be expressed in MARBLE, further mechanisms are required. The second problem is conflict in the use of resources by code generated by the bound and unbound paths. Restricting resources used by the unbound path to those that could be saved and restored would ease this problem; however this is not always possible.

We have presented a solution to a long-standing problem in the design of programming languages that must make efficient use of the resources of the target virtual machine. Much more work need be done in assessing the practicability of this approach.

REFERENCES

1. Kuck, D. J., *The Structure of Computers and Computations: Volume One*, John Wiley and Sons, New York, 1978.
2. Leverett, B. W. et al., "An Overview of the Production-Quality Compiler-Compiler Project," *Computer*, Vol. 13 No. 8, pp. 38-49, August 1980.
3. Davidson, S., "Design and Construction of a Virtual Machine Resource Binding Language," Ph.D. Dissertation, Computer Science Department, University of Southwestern Louisiana, August 1980.
4. Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Second ed., Springer Verlag, New York, 1975.
5. Davidson, S. and B. D. Shriver, "Firmware Engineering: An Extensive Update," *IFIP TC-10 Conference on Microprogramming, Firmware, and Restructurable Hardware*, North-Holland, Amsterdam, 1980.
6. DeWitt, D. J., "A Machine Independent Approach to the Production of Optimal Horizontal Microcode," Ph.D. Dissertation, The University of Michigan, 1976.
7. Dasgupta, S., "Towards a Microprogramming Language Schema," *Proc. 11th Annu. Microprogramming Workshop*, pp. 144-153.
8. Richter, L., "High-Level Language Extensions for Micro-code Generation and Verification," *IFIP TC-10 Conference on Microprogramming, Firmware, and Restructurable Hardware*, North-Holland, Amsterdam, 1980.
9. Giloi, W. K., P. Behr, and R. Gueth, "FIT—A System for Firmware Specification, Implementation, and Validation," *IFIP TC-10 Conference on Microprogramming, Firmware, and Restructurable Hardware*, North-Holland, Amsterdam, 1980.
10. Microdata Corporation, *3200 Microprogramming Reference Manual*, Irvine, 1976.
11. Agrawala, A. K. and T. G. Rauscher, *Foundations of Microprogramming*, Academic Press, New York, 1976.

The design of a firmware engineering tool: the microcode compiler

by PERNG-YI MA

TRW Systems Group
Redondo Beach, California

INTRODUCTION

The explosive rate of progress of hardware technology has made the microprogrammable processor characterized by horizontal microcoding extremely attractive for many high speed and real time applications, such as signal processors. However, the human microprogrammer currently has little but assembly language and text editor to help with code developments.¹ The lack of software tools to support microcode generation results in high costs and poor reliability especially when the volume of microcode increases. The application of software tools to microprogramming will be termed firmware engineering in the remainder of this paper.

The most obvious solution is to increase firmware tool capability by encoding the intended application program in a high level language (HLL), and developing a translation system for conversion of the HLL into horizontal microcode.² However, the general problems of firmware engineering using a high level language compiler are found in several studies.^{3,4,5,6}

1. Machine variety and complexity^{7,8,9}— The object code of the microprogramming system is the hardware-oriented and timing critical horizontal microcode. The gap from HLL to the microcode is extremely large.
2. Concurrency utilization—One benefit of microprogramming is that the horizontal microinstruction formats offer added speed of machine operation only if concurrent microoperations can be combined into a single microinstruction.^{10,11,12,13,14,15} The concurrency detection rules reported in the literature are usually machine dependent,³ and the optimization algorithms which determine optimal compaction of a sequence of microoperations is an NP hard problem.¹⁶ Devising a practical compaction algorithm is still an open research question.¹

To study these problems, we designed and implemented a compiler to produce microcode. This compiler, denoted *microcompiler*, is shown in Figure 1. The application program is encoded in a HLL. A machine-independent partial compiler is used to perform the lexical, syntax, and data flow analyses of the HLL and produces a stream of machine-independent intermediate language (IML) statements. The IML version of the intended application program and the underlying machine information are input to the machine dependent translation

sub-system. The underlying machine is described by a set of microoperations ($M_i, i = 1..n$). Each microoperation is a machine primitive operation and represented by (OP, I, O, F, P) , where OP = function, I = input data set, O = output data set, F = microinstruction field, and P = clock phase. This is called the *field description model* and is the collection of microoperations along with a methodology for checking the concurrency among microoperations.

The translation system requires three passes over the IML representation of the source code to produce compact microcode. In pass 1, a macro table is used to translate the IML into a set of machine dependent statements (MDIL). For each MDIL statement, some operands are bound to machine units; other operands still hold the symbolic variables from IML.

Pass 2 allocates the remaining symbolic operands of MDIL to one of the target machine general purpose registers. In general, the number of symbolic variable operands in a given program is greater than the number of machine registers. Thus, each register must be shared by more than one symbolic operand. A register allocation and deallocation scheme swaps operands between the machine's main memory and its general purpose registers. After all operands of the MDIL statement have been allocated registers, the field value and timing phase are assigned to each statement. The resulting codes from pass 2 are called microoperations.

Pass 3 is used to increase the system throughput by compacting the sequence of microoperations into the least number of horizontal microinstructions. However, complete optimization of microoperations is known to be an NP-complete problem.¹⁶ Thus, a linear order algorithm is developed which may not produce optimum compaction, but produces a best possible compaction given linear time to scan the stream.

Finally, the microcompiler was tested and studied using a HLL called VMPL and the PDP 11/40E as the target machine. VMPL (Virtual Machine Programming Language) is a machine independent structure programming language used to implement emulators. However, the design of a microcompiler discussed here is not restricted to a particular HLL or a specific microprogrammable machine. Any IML can be designed for a class of problems and freed from a specific hardware operation set. In fact, it is desirable to alter the IML instruction set to better reflect the HLL being translated. This field description model is intended to be machine independent and separate from the compiler. Only the contents of the model and the macro table are redefined in order to imple-

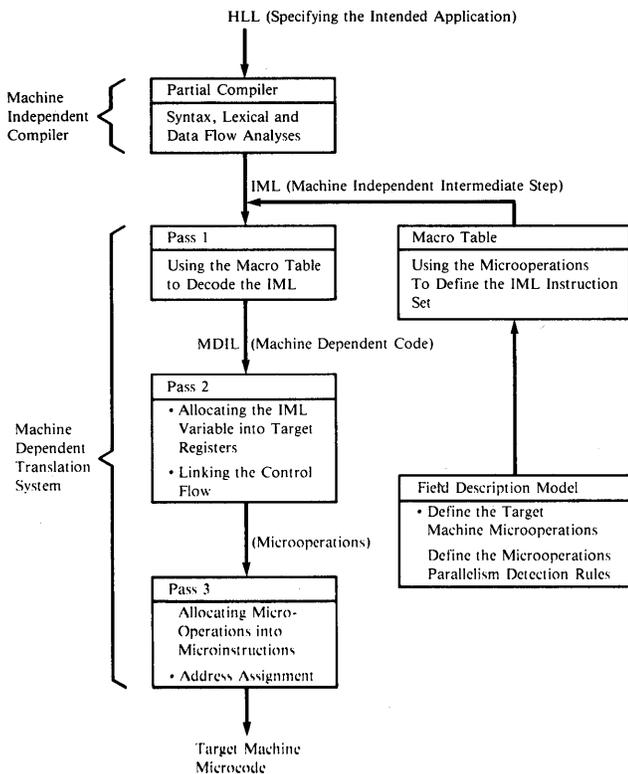


Figure 1—General structure of microcode compiler

ment the microcompiler for another microprogrammable machine.

PARTIAL COMPILER

The partial compiler translates the HLL source program into intermediate form by lexical scanning and syntax analysis just as any other compiler does, however, the unusual demands placed on the performance of a microprogram means we must use additional information to produce the best possible object microcode. Thus, code optimization is aided by the partial compiler in three general areas.

1. Attaching register allocation “clues” to symbolic variables in order to guide the microcode generator into a “least amount of swapping” pattern.
2. Attaching flow-of-control tags to the segments of straight-line microcode so that a program flow analysis can be done. This assists both variable-to-register binding and microcode compaction algorithms.
3. Producing an IML format which provides opportunities for subsequent code optimization.

The first area is satisfied by the partial compiler using a 3-character clue attached to each entry in the symbol table:

1—Scope

- Local variable
- Global variable
- Subprocedure name

2—Activity

- Temporary variable
- Permanent variable

3—Type

- Simple variable
- Memory
- Stack
- Field
- Flag (condition code)
- Parameter (actual)
- Parameter (formal)
- Procedure
- Concatenated variable
- Constant

The second area is accommodated by the partial compiler using additional tags or modifiers which aid the code compaction algorithms:

Tags

- Block code start/stop
- Label of IF-THEN-ELSE branch
- Label of a GO TO
- Label of a FOR loop
- Label of an EXIT (from a procedure)
- Label of a CASE selector

Finally, the design of an IML (machine independent intermediate language) is guided by the application, class of underlying machines, and overall firmware engineering goals. Malik^{17,18} studied several candidates for a suitable IML, they are: 3-operand format, 2-operand format, 1-operand format (stack), Polish Notation, and Program Tree. He studied these five formats with respect to the number of instructions, instruction “size,” stack and register requirements, complexity of interpretation, and Halstead’s information theoretical measure of “level.”¹⁹

The 3-operand format (quadruples) yielded the least execution time estimates, the highest level in Halstead’s measure,¹⁹ and provided the greatest opportunity for subsequent economization of microcode.

The results of the partial compiler are machine independent intermediate code (IML). The IML overall structure consists of a program portion and delayed information portion. The program portion has a declaration part and main body. The main body is in single entry/multiple exit block structure and used to represent the application program. The declaration part declares all variables used in the main body. The delayed information portion is created to contain information specified in the HLL program but cannot be directly supported in the IML main body due to infeasibility or highly machine dependent features, e.g., machine implicit I/O request.

The partial compiler passes this form of the translated HLL program to a 3-pass code generator that binds symbolic variables and instructions to the real machine, compacts the resulting bound instructions into the machine microinstructions. This 3-pass process is the subject of the following sections of this paper.

FIELD DESCRIPTION MODEL TO DESCRIBE THE TARGET MACHINE

The Field Description Model (FDM) is a compact representation of the machine hardware capability. From the functional

behavior viewpoint, a microprogrammable machine consists of a set of microoperations encoded and stored in a control memory. We adopt the notation and schema first proposed by Dasgupta^{11,20} to describe this model. An FDM is simply the collection of these microoperations.

$$FDM = \{M_i / i = 1..n\}$$

Each M_i , identified by a unique index i , is in turn defined by a set of five tuples,

$$M_i = \{OP, I, O, F, P\}_i$$

and each tuple is expanded by specifying its domain. Each domain enumerates all the legal values which the component can assume. The tuple components are:

- OP*: Designates the primitive operation to be performed.
- I*: Denotes the resource used as the input to the OP.
- O*: Denotes the resource used as the output to the OP.
- F*: Denotes the set of fields which are occupied in the microinstruction format when $\langle OP, I, O \rangle$ is executing.
- P*: Denotes the set of timing phases at which the $\langle OP, I, O \rangle$ can execute.

This model also provides concurrency detection rules between microoperations. Some definitions are explained first.

Microoperations M_i and M_j , are said to be data-independent, if $I_i \cap O_j = I_j \cap O_i = O_i \cap O_j = \text{empty set}$. Otherwise, there is an *I/O conflict* between M_i and M_j . M_i is said to precede M_j in sequential order if they are in separate control store (CS) cycles and M_i is executed prior to executing M_j . A field conflict between microoperations occurs if the same field is used by these microoperations in the same CS cycle. But there is a special kind of field tuple which can be shared by more than one microoperation in the same CS cycle as long as the values assigned to the field of each microoperation are the same. For example, a literal field can be shared by microoperation in the same CS cycle if the field values are the same. Obviously, if the literal field value of one microoperation is different from the others, it will cause a field conflict. Microoperations M_i and M_j are in *parallel*, denoted $M_i // M_j$, if they can be executed in the same control store cycle and produce the same output as if executed sequentially in separate control store cycles. Two microoperations, M_i and M_{i+1} , are said to be *invertible*, denoted by $M_i > < M_{i+1}$, if the execution of M_i and M_{i+1} yields the same result as the execution of M_{i+1} and M_i .

Machine constraints on the microoperation may be different for each machine. Therefore we seek general rules that work for a class of horizontal machines. This is done by generalizing the structure of an arbitrary machine by describing the machine as a set of microoperations in 5-tuple format.

General Rules

We assume every microinstruction is completed within a control store cycle. This cycle is divided into several minor phases and each microoperation is assigned to the corresponding phases. Given two microoperations, M_i and M_j , and M_i precedes M_j in sequential order, the timing phases used to

execute M_i and M_j are denoted by P_i and P_j , respectively. The general rules are:

Begin

CASE "THE RELATIONSHIP BETWEEN P_i AND P_j "
OF:

WITHIN THE SAME CONTROL STORE CYCLE, P_i
IS PRIOR TO P_j ;

IF THERE IS NO FIELD CONFLICT THEN $M_i // M_j$
WITHIN THE SAME CONTROL STORE CYCLE, P_i
IS NOT PRIOR TO P_j ;

IF THERE IS NO FIELD CONFLICT AND DATA
ARE INDEPENDENT FROM EACH OTHER,
THEN $M_i // M_j$.

ENDCASE

IF M_i AND M_{i+1} ARE DATA INDEPENDENT, THEN
 $M_i > < M_{i+1}$.

End.

Example. We use the PDP11/40E as the target machine and the following cases to illustrate these general rules. Pulses P_2 and P_3 are the first phase and second phase within the control store cycle CL3.^{21,22} In the following cases, one reference³ shows that there is no field conflict between M_5 and M_6 , neither in M_7 and M_8 .

Case 1: $M_1 : R_2 \rightarrow D$, P_2 : copy R_2 to register D in phase P_2 .
 $M_2 : D \rightarrow R_3$, P_3 : copy register D to R_3 in phase P_3 .
 $M_3 : R_3 + B \rightarrow D$, P_2 : add R_3 and register B to register D in P_2 .

$M_4 : D \rightarrow D_4$, P_3 : copy register D to R_4 in P_3 .

M_2 and M_3 are examined to detect parallelism. P_3 is not prior to P_2 , M_2 is not data independent from M_3 . This implies M_2 not $// M_3$. (If M_2 and M_3 are executed in one CS cycle, and M_3 is executed prior to M_2 , it will give the wrong result.)

Case 2: $M_5 : PS \rightarrow \text{stack}$, P_3 : copy the contents of the "PS register" to stack in P_3 .

$M_6 : R_3 \rightarrow D$, P_2 : copy R_3 to register D in P_2 .

$(F_5 \cap F_6 = 0)$ and $(M_5$ is data independent from $M_6)$ imply $M_5 // M_6$ which is independent of timing sequence.

Case 3: $M_7 : R_3 + B \rightarrow D$, P_2 : add R_3 and B to register D in P_2 .

$M_8 : D \rightarrow R_3$, P_3 : copy register D to R_3 in P_3 .

The pulses used by M_7 and M_8 and P_2 and P_3 , respectively.

$F_7 \cap F_8 = 0$ implies $M_7 // M_8$, which is independent of I/O conflict.

End example.

Machine Constraints

Some examples from PDP11/40E are now used to illustrate the effect of machine dependency on the parallelism detection rule.

Example. In the FDM of the PDP11/40E, the microoperation FLAG is used to set the machine flags for the previous ALU operation. Microoperation FLAG must be the

next one after the ALU operation, and it cannot be moved even if invertibility is possible.

Microoperation NOOP, which is used in an N-way branch operation on the PDP11/40E machine, has its own fixed position. It cannot be moved and/or be made parallel with other microoperations even if the general rule indicates parallelism.

End example.

The microoperations used for these special purposes lead to restrictions on the parallelism detection rules. Therefore, the users of this model must provide these kinds of machine-dependent rules in addition to the general rules.

PASS 1

Pass 1 maps the IML version of the application program into a machine dependent intermediate code (MDIL). The MDIL instruction is defined from the field description Model with the format $\langle OP, I, O \rangle$ which excludes the field and timing tuples from the 5-tuple $\langle OP, I, O, F, P \rangle$ representation.

This microoperations defined in the machine field description model are used to decode the instruction set of the IML. The delayed information portion of IML, which contains the items specified in the HLL but cannot be directly supported by the IML, are emulated by the machine microoperations. All the mappings from IML facilities to the machine microoperations are stored in the macro table. Pass 1 allocates the variables in the IML declaration part into data memory and uses the macro table to decode the IML main body and the delayed information.

The output of Pass 1 is a collection of machine-dependent code (MDIL) consisting of a set of blocks. The operands in MDIL are either the machine units defined by the FDM or the symbolic variables which will be bound to the machine general purpose registers in Pass 2.

PASS 2

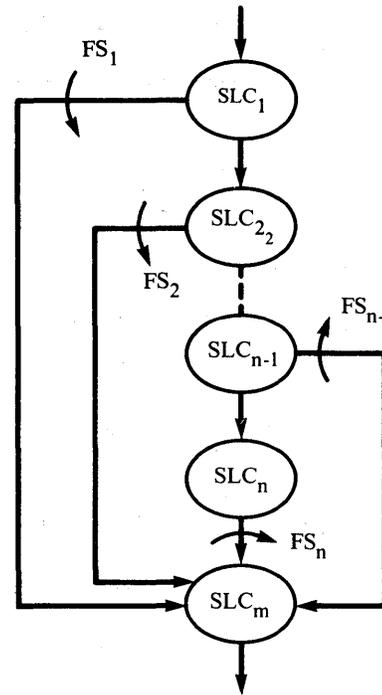
This pass accepts a set of blocks of MDIL from Pass 1 and allocates target machine registers to the symbolic variables remaining in the microcode. Furthermore, Pass 2 produces binary microoperations ready to be compacted by Pass 3.

This pass would be trivial if the target machine guaranteed an unlimited supply of registers and there were no branches within the object code. However, each block of MDIL is likely to contain more variables than the machine registers, and there are always branches generated from the partial compiler.

The general idea of this register allocation scheme³ is to keep the variables in their assigned registers as long as possible. When no register is available for a newly encountered variable, the variable replacement priority is applied to determine the least likely used variable which will be moved out of the register. The newly encountered variable is then assigned the free register.

The replacement priority is determined by the status (active or passive) and the kind (local or global) of each variable. If the contents of a variable held in a register is different from

a. n Final States $FS_1, FS_2, \dots, \text{ and } FS_n$ are to determine IS_m .



b. Final State FS_q is Determined from IS_p .

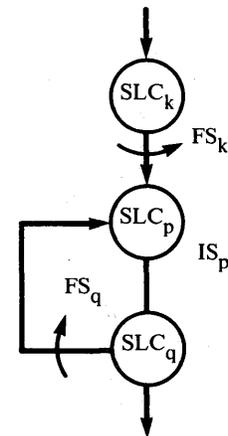


Figure 2—Forward and backward branching

the contents of its main memory location, then it is said to be active. Otherwise, it is called passive. When an active status variable is to be deallocated, a memory write is needed to swap it back to memory. However, memory write is not necessary for a passive status variable. In order to reduce redundant swappings, a passive status variable is assigned a higher priority to be replaced. A local variable is assigned a higher priority to be replaced than the global variable, since local variables are available only in the current block.

In particular, the register allocation algorithm handles forward and backward branches as shown by the program flow graphs of Figure 2. In minimizing the "thrashing" of register allocation and deallocation, the blocks of MDIL codes can be

analyzed for the flow of control governed by branch statements and labeled statements. These two statement types divide the blocks into a set of straight line codes (SLCs) which are sets of single-entry single-exit segments.

We define the state of an SLC as the assignment of operands to registers for the given SLC. Upon entry to the SLC we must define an initial state IS_i for SLC_i , and the final state FS_i as the state of SLC_i when register allocation is completed.

The initial state of SLC_m , denoted by IS_m , is defined as the assignment of symbolic variables to registers immediately before entering this SLC_m . Every SLC segment uses the previous segment's final state as its initial state, except in the cases shown in Figure 2. Referring to Figure 2a, the initial state of SLC_m is actually determined from the final states of $SLC_{i=1 \text{ to } n}$, and used as the basis to perform the register allocation/deallocation scheme on the current SLC_m .

The final state of an SLC is translated from the initial state by employing the least number of memory references for each SLC segment in the microoperation sequence. However, the final state determination of the SLC with backward branch is different.

Refer to Figure 2b, showing the SLC_q backward branch to SLC_p . The state immediately before the branch statement must be the same as the starting state of the SLC_p . It implies the final state, FS_q , is also dependent on initial state of SLC_p .

Finally, Pass 2 produces an output as a collection of 5-tuple format microoperations consisting of a set of SLCs. This 5-tuple format provides a convenient way to allocate the microoperations to microinstructions, which are then used in Pass 3.

PASS 3

This pass uses a set of rules to detect the concurrency of microoperations and combine sequences of microoperations into shorter concurrent microinstruction, or what we abbreviate as MI s.

The MI sequence is optimized if it is impossible to rearrange the sequence of microinstructions in a manner that will produce fewer microinstructions. DeWitt¹⁶ proved that this kind of absolute minimal reduction problem is an NP-complete problem. Here, by seeking a near-optimal solution rather than the absolute, we get a fast algorithm of complexity proportional to mn where m is a pragmatically determined constant less than n .

Linear Order Compaction Algorithm

Given an $SLC = \{M_1, M_2, \dots, M_k, \dots, M_n\}$, and MI_i refers to the i th microinstruction. (Note: M_i is a single microoperation, while MI_i is a microinstruction with several M_i s). As M_k is allocated, the possible relationships between M_k and MI_i are

- Case 1: $M_k \text{ not } > < MI_i$, and $M_k \text{ not } // MI_i$
- Case 2: $M_k \text{ not } > < MI_i$, and $M_k // MI_i$
- Case 3: $M_k > < MI_i$, and $M_k \text{ not } // MI_i$
- Case 4: $M_k > < MI_i$, and $M_k // MI_i$

If M_k is invertible with MI_i (Case 3 or 4), it may be moved past MI_i and the same test applied to MI_{i-1} . On the other hand, if M_k is not invertible with MI_i (Case 1 or 2), it is blocked by this microinstruction.¹¹

Early indication³ showed that *invertibility* caused the problem to be NP-complete. However, the data dependency among microoperations is obvious and limits the invertibility considerably. In this case, it is hard for a microoperation to cross too many microoperations ahead of it. A limitation of the times of comparing a microoperation with other microoperations is necessary.

In order to get a practical and efficient compaction, we impose the following restrictions:

1. The position of microoperation M_k is computed by searching backward over the previous microinstructions leading up to microoperation M_k .
2. The following restrictions are made:

Case 1: $\{M_k\} \rightarrow MI_{i+1}$

Case 2: $\{M_k\} \rightarrow MI_i$

In the next two cases, we compare M_k m times with the previous microoperations. In other words, M_k can compare with h MI s from MI_i to MI_{i-h+1} where h is a number of MI s and $\sum_{j=0}^{h-1} |MI_{i-j}|$ is nearest to m . ($|MI_k|$ means the number of microoperations in MI_k .)

Case 3: If M_i is invertible with all MI s and not parallel, then $\{M_k\} \rightarrow MI_{i+1}$.

Case 4: Compare M_k with MI_{i-j} , $0 \leq j \leq h-1$, until we find the MI nearest to MI_i that can accept M_k .

The compaction algorithm can be implemented by the use of invertibility and parallelism between microoperations and MI s.³ Now we consider the computational complexity to allocate n microoperation, using the number of comparisons between pairs of microoperations as a measure of this complexity. In the above restriction, M_k is limited to make m comparisons with the preceding microoperations. If $K \leq m$, at most k comparisons are necessary to optimally place M_k . If $k > m$, M_k requires a total of m comparisons before (sub-optimal) allocation. Indeed, if this occurs for each microoperation: $M_1 \dots M_n$, the total number of comparisons is

$$\begin{aligned} T(n) &= \sum_{j=1}^m j + \sum_{j=m+1}^N m = \frac{m(m+1)}{2} + m(n-m) \\ &= nm + \frac{1}{2}m - \frac{1}{2}m^2 \end{aligned}$$

Therefore, the algorithm complexity is proportional to n . Now we will pragmatically determine the value of m .

Determination of m in the Linear Algorithm

The compaction algorithm³ was applied to the abstract Husson machine²³ and the PDP11/40E to determine the best width of the MI and the best value of m in the linear algorithm. The

width of the *MI* means the number of microoperations allowed in the *MI*.

Early results indicate that four microoperations is the limiting width of a *MI* for a microprogrammable machine.³ Beyond this number, data dependency among microoperations limits the compaction of microoperations into *MI*s. This constraint leads to the following conjecture in the determination of the value of *m*. While we have no way of proving this conjecture, it is in line with independent work by Mallett.²⁴

Conjecture— Given a horizontal microprogrammable microinstruction width *W*, and a linear compaction algorithm that locally compacts the straight line code segments of length *n*, using peephole size *m* of time complexity *mn*, then

$$m = 2W$$

produces compact code within 10% of optimal.

For example, applying our conjecture to Mallett's results,²⁴ $W = 4$, so $m = 8$. Thus, $8n$ comparisons are required for a code segment of length *n*. In the 10 tests reported by Mallett,²⁴ an average of $5.6n$ comparisons were performed to compact code to within 3% of optimal. Thus, the conjectured value of $m = 8$ appears to be safely conservative when used to explain Mallett's results.²⁴

CONCLUSION

The techniques described in this paper have been successfully applied to the design and implementation of a HLL compiler for microprogramming.^{3,17} The main goal of efficient compaction of parallel microcode in a horizontal microprogrammable machine has been demonstrated.⁶ The system runs as a cross-compiler on a Cyber machine which downloads to a PDP-11/40E. The system is not portable, but produces transportable code. The HLL syntax can be changed by specifying new syntax rules. The object machine is changed by specifying a new FDM and the set of macros. The system includes a simulator for testing IML code before Passes 1, 2, and 3 are completed.

Clearly the work reported here is experimental and tentative. Many questions remain unanswered, for example:

1. HLL determination—How to determine a microprogramming HLL as to its capabilities such as (a) to describe the intended application algorithm; (b) to be compiled to microcode efficiently?
2. Hardware mapping—How to sufficiently use the hardware features to decode the machine independent intermediate language (IML)?
3. Target machine selection—How to select a machine which can produce the minimum object code for the application algorithm coded in HLL?
4. A machine description language is needed to describe the target machine in high level terms which produces the FDM automatically.

The early indications are that register allocation schemes and microcode compaction as described in this paper and elsewhere are solved problems.^{10,11,12,14,16,24,25} However, preliminary experience gained from using this system suggests that allocation and compaction are *minor* sources of inefficiency when compared with the problems listed above. A casual examination of several tested program running on a target machine, PDP11/40E, indicates that the number of microoperations generated from the hardware mapping is often 2 to 3 times larger (counting the instruction words) than the number of IML codes. However, it is unlikely that compaction algorithms will be able to improve code by more than 30% unless target machines are designed to encourage greater levels of concurrency. Machine selection is a particularly important factor to minimize the object code. More experimental studies on a variety of machines are needed to make conclusive statements about coding efficiency.

REFERENCES

1. Davidson, S., and B.D. Shriver, "An Overview of Firmware Engineering," *Computer*, Vol. 11, No. 5, May 1978, pp. 21-33.
2. Mallett, P.W., and T.G. Lewis, "Considerations for Implementing a High Level Microprogramming Language Translation System," *Computer*, Vol. 8, No. 8, Aug. 1975, pp. 40-52.
3. Ma, P.Y., "Optimizing Microcode Produced From a High Level Language," Ph.D. Dissertation, Oregon State Univ., Electrical and Computer Engineering Department.
4. Ma, P., and T.G. Lewis, "Design of a Machine Independent, Optimizing System for Emulator Development," *ACM TOPLAS*, Vol. 2, No. 2, April 1980.
5. Lewis, T.G., K. Malik, and P. Ma, "Firmware Engineering Using a High Level Microprogramming System to Implement Virtual Instruction Set Processors," *IFIPS Workshop*, April 1980.
6. Lewis, T.G., P. Ma, K. Malik, and C. Liu, "On the Problem of Portable Microprogramming," Technical Report, TN79-3, Oregon State Univ., Computer Sci. Dept.
7. Agrawala, A.K., and T.G. Rauscher, "Foundations of Microprogramming Architecture, Software, and Applications," Academic Press, Inc., 1976.
8. Katzan, Jr., H., "Microprogramming Primer," McGraw-Hill Book Company, 1977.
9. Husson, S.S., "Microprogramming: Principles and Practice," Prentice Hall, Englewood Cliffs, New Jersey, 1970.
10. Agerwala, T., "Microprogram Optimization: A Survey," *IEEE Trans. Comput.*, Vol. C-25, Oct. 1976, pp. 962-973.
11. Dasgupta, S., and J. Tartar, "The Identification of Maximal Parallelism in Straight Line Microprograms," *IEEE Trans. Comput.*, Vol. C-25, Oct. 1976, pp. 986-991.
12. Tabendeh, M., and C.V. Ramamoorthy, "Execution Time (and Memory) Optimization in Microprograms," Preprints Supplement, 7th Annu. Workshop on Microprogramming, pp. 119-127.
13. Tsuchiya, M., and C.V. Ramamoorthy, "A High Level Language for Horizontal Microprogramming," *IEEE Trans. Comput.*, Vol. C-23, Aug. 1974, pp. 791-802.
14. Tsuchiya, M., and M.J. Gonzalez, "An Approach to Optimization of Horizontal Microprograms," Proceedings of the Seventh Workshop on Microprogramming, Palo Alto, California, Sept. 1974.
15. Yau, S.S., A.C. Schowe, and M. Tsuchiya, "On Storage Optimization of Horizontal Microprograms," Preprints, 7th Annu. Workshop on Microprogramming, pp. 98-106.
16. DeWitt, D.J., "A Machine Independent Approach to the Production of Optimal Horizontal Microcode," Ph.D. Dissertation, The University of Michigan, 1976.
17. Malik, K., "Optimizing the Design of a High Level Language for Microprogramming," Ph.D. Dissertation, Oregon State University.
18. Malik, K., and T.G. Lewis, "High Level Microprogramming Language," *COMPCON*, 1978, pp. 88-91.

-
19. Halstead, M.H., "Elements of Software Science," Elsevier North-Holland, 1977.
 20. Dasgupta, S., "Parallelism in Microprogramming System," Ph.D. Thesis, University of Alberta, Aug. 1976. Tech. Rept., Dept. of Computing Science.
 21. Fuller, S.H., et al., "PDP11/40E Microprogramming Reference Manual," Dept. of Computer Science, Carnegie-Mellon Univ., Jan. 1976.
 22. Fuller, S.H., et al., "The PDP11/40E Maintenance Manual," Dept. of Computer Science, Carnegie-Mellon Univ., June, 1977.
 23. Ma, P., and T.G. Lewis, "On the Design of a Machine Description Model and a Compaction Algorithm for Microcode Generation," to appear on the Proceedings of Euro Micro 80 Symposium, London British, Sept. 1980.
 24. Mallett, P.W., "Methods of Compacting Microprograms," Ph.D. Dissertation, University of Southwestern Louisiana, Computer Science Department, Lafayette, La., 75401, Dec. 1978.
 25. DeWitt, D.J., "A Control Word Model for Detecting Conflicts Between Microprograms," Proc. 8th Annual Workshop on Microprogramming, pp. 6-13.

Microcode compaction: looking backward and looking forward

by JOSEPH A. FISHER

Yale University
New Haven, Connecticut

and

DAVID LANDSKOV and BRUCE D. SHRIVER

University of Southwestern Louisiana
Lafayette, Louisiana

INTRODUCTION

The past decade has seen significant advances in the state of the art in microcode compaction. Microprograms are compacted by placing several microoperations into each microinstruction, subject to the constraints of data dependency in the program and legal resource usage in the machine on which the microcode will execute.¹ The compaction process attempts to make the code run as fast as possible. In this paper we will not only survey the most recent past in microcode compaction but will also speculate about the near future.

The *classical microcode compaction problem* considered basic blocks of code for machines in which: (a) all operations have fixed cycle time, (b) data precedence may be described by a partial order on the operations, and (c) the choice of representations for an operation may be fixed before compaction begins.

A *basic block* of code can be entered only at its beginning and is jump-free, except possibly at its end. Compaction restricted to a basic block is called *local compaction*. Basic blocks are also known as straight-line microcode segments, or SLMs. This classical compaction problem was posed a decade ago.² The word "compaction" is now preferred to the word "optimization."³

The classical compaction problem is analogous to deterministic processor scheduling.⁴ As such, it is NP-complete, and thus it is believed that any algorithm that produces optimal results must be of at least exponential complexity. Despite this belief, there are techniques that apparently perform so well that for all practical purposes the classical problem may be regarded as solved.³

Research in microcode compaction continues in two directions. The first relaxes the classical problem's restriction to basic blocks, while the second relaxes the restriction to a simplified machine architecture.

Compaction that is not restricted by block boundaries is called *global* microcode compaction. Two relatively complete methods have been suggested for global compaction. Both

approaches apply techniques used in the solution of the local microcode compaction problem, but differ markedly in their methods of choosing operations to move past the block boundaries. The method of Tokoro et al.⁵ separately compacts individual blocks and then considers the motion of operations from their current blocks to adjacent blocks, as allowed by a "menu" of legal moves. However, Fisher⁶ argues that if each block is compacted separately, too many arbitrary decisions are made during a block's compaction that adversely affect the ability to move operations from block to block. His method starts by using dynamic information to pick a "main trace" of several blocks that might be executed one after the other for some choice of data. The trace is then compacted as if it were one large basic block. Traces are discussed in more detail later.

To the authors' knowledge, there have been no published implementations of global compaction; however, simulation results on small code samples have been encouraging.^{5,6} An important short-term research priority is the coding of global compaction methods in a system on which reasonable measurements may be made.

Many aspects of the compaction problem have not been well analyzed. For example, although speeding up a loop that is executed many times may significantly speed up a program, loops have not been extensively investigated in the literature. In conventional compiler optimization, faster loop execution can sometimes be achieved by "unrolling" the loop, a process that constructs more than one copy of the loop body. We will present a highly speculative method for speeding up microcode loops by unrolling them and rerolling after compaction has been done.

Data Precedence

Given a basic block of microoperations, hereafter called a block, the compaction process places every microoperation (MO) into a microinstruction (MI). The final sequence of MIs

must be semantically equivalent to the original block in its effect on data resources. A data resource is a register or, viewed on another level of abstraction, a variable. When an MO writes a value to a data resource, that resource must not be changed until that value has been read by all MOs that would read that value in the original microprogram. This motivates the following definition.

Definition. Two MOs, A and B, have a *data interaction* if they satisfy any of the following conditions:

1. An output resource of A is also an input resource of B.
2. An input resource of A is also an output resource of B. (This prevents B from interfering with A's read.)
3. An output resource of A is also an output resource of B.

A section of microcode will generally have fewer data interactions if its reads and writes are stated in terms of program variables rather than registers. This occurs because several variables may have to share the same register, potentially introducing data interactions that are not required by the original program. Since these extra interactions can artificially restrict the compaction process, register allocation should be delayed at least until compaction time, as discussed in a later section.

The data interaction concept can be used to define a partial order over the MOs of a basic block:

Definition. Given two MOs, A and B, where A precedes B in the original basic block, A *directly data precedes* B (written $A \text{ ddp } B$) if the two MOs have a data interaction and if there is no sequence of MOs, $C_1, C_2, \dots, C_n, n \geq 1$, such that $A \text{ ddp } C_1, C_1 \text{ ddp } C_2, \dots, C_n \text{ ddp } B$.

The relation $A \text{ ddp } B$ may also be worded, "B is *directly data dependent* on A." The transitive closure of the *ddp* relation is the data-precedence relation: $A \text{ dp } B$ if $A \text{ ddp } B$ or if there exists an MO C such that $A \text{ ddp } C$ and $C \text{ dp } B$. If a *dp* relation does not exist between two MOs, they are said to be *data independent*.

As long as data precedence is not violated, a compacted microprogram will preserve data integrity. A few integrity-preserving compactations that do violate precedence can sometimes be obtained by moving each write MO and its associated reads as a group, but this is widely regarded as an excessively complicated technique offering little gain.

The representation of the direct data-precedence relation as a graph forms a directed acyclic graph (*DAG*). Each node on a *DAG*, e.g. node i , corresponds to a unique MO in the basic block, M_i . If there is a link from M_i to M_j on a *DAG*, then $M_i \text{ ddp } M_j$.

THE CLASSICAL COMPACTION PROBLEM AND ITS SOLUTION

The classical compaction problem restricts itself to simplified models of machine behavior and to essentially jump-free microcode.

The Model Constraints

In the simplified machine model, each MI takes one machine cycle to execute. The number of fields in an MI and the

valid contents for each field are bound by the MI format. The classical problem restricts itself to one predetermined format per machine. Each MO requires one or more fields to execute. In addition, there are machine resources (ALUs, BUSs, etc.) that an MO can use. Fields are normally considered as resources. For compaction purposes, a legal MI is regarded as a set of MOs that do not conflict in their resource usage. Most of these assumptions can be straightforwardly extended to more general cases.

The Classical Problem

The classical microcode compaction problem may be formally stated as the following scheduling problem:

Given

1. a set $BB = \{M_1, \dots, M_n\}$ of microoperations,
2. a partial order $<<$ on BB (i.e. data precedence),
3. that each microoperation requires one time unit, and
4. a conflict function, $c: BB^* \rightarrow \{\text{false}, \text{true}\}$, where an element of BB^* is a set of MOs from BB ,

then

minimize $tmax$, the total number of time units required, under the constraints that if $M_r << M_s$, then M_s does not execute until at least one time unit after M_r , and that if S is a set of microoperations executed in the same time unit, then $c(S) = \text{false}$.

The conflict function must test for MI format compatibility, as well as for other conflicts in resource usage. A typical way of doing this is to associate each MO with a resource vector, with the k th component being the proportion of resource k used by the MO. For most microprogrammable machines, the resource vector suffices to test all possible conflicts. It is not difficult to prove that the classical compaction problem is NP-complete.³ In spite of the exponential complexity of NP-complete problems, extensive experiments in microcode compaction have demonstrated that heuristic algorithms find acceptable compactations in a reasonable amount of time.^{7,4,8}

List Scheduling

Four approaches to compacting a block of microcode have been identified (see Landskov et al.³ for a survey). Of these the most important are the following:

- (1) A branch and bound (BAB) algorithm. This well-known scheduling algorithm is the only approach that can search exhaustively to find a guaranteed-optimal solution. BAB is usually run with heuristics, since exhaustive searches are prohibitively expensive in execution time. BAB is also the most difficult to program of these three approaches.
- (2) A list scheduling algorithm. This is another algorithm from scheduling theory, in which MOs are assigned priority values before scheduling begins. Each MI is formed by considering in order the unscheduled MOs whose parents have all been scheduled (the data-ready MOs) and adding them to the MI if no conflict exists. List scheduling can be considered as a particular heuristic version of BAB.

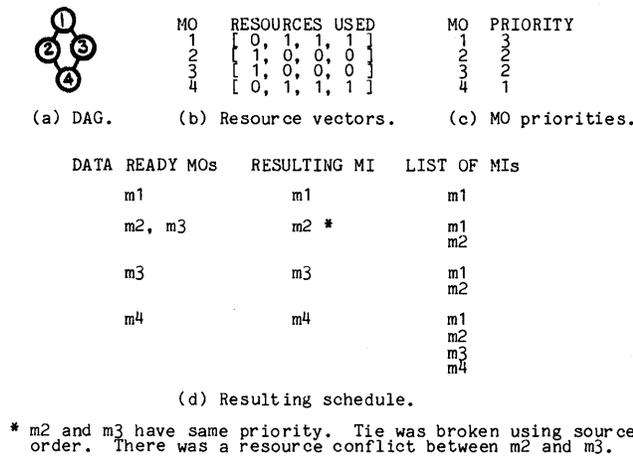


Figure 1—List scheduling applied to a simple block

- (3) A first-come, first-served (FCFS) algorithm that considers the MOs in source order and adds them one at a time to an initially empty list of MIs. Each MO is added as high as possible in the MI list. This method is used in Dasgupta⁹ and in Tokoro et al.⁵ With a minor modification, the FCFS algorithm is semantically equivalent to list scheduling with source order priority.

The effectiveness of source-order analysis is somewhat controversial. It is easy to construct trivial examples for which FCFS produces more MIs than necessary, although comparison tests on actual microprograms have always found FCFS results comparable to those of the other heuristic approaches.^{8,4} Various heuristics have been studied for BAB and list scheduling.^{7,4} A good heuristic for list scheduling is assigning the level of an MO in the DAG as the MO's priority. In this scheme, the priority of an MO is 1 more than the largest priority of any of its children in the graph. The higher priority MOs (i.e. those at higher levels in the DAG) are considered first in building MIs.

Figure 1 shows the effect of applying list scheduling to a simple block. In this example, two MOs conflict if they both have a 1 in the same element of their resource vectors. A list scheduling algorithm written in a PASCAL-like language may be seen in Figure 2. In this figure, except for DAG, cycle, and CurrMO, all of the variables are sets, with the implementation of the set type remaining unspecified. This figure is intended to outline the steps of the algorithm and should not be taken as an exact implementation.

GLOBAL COMPACTION

The typical compacted block is full of unused microinstruction fields (holes) that can potentially hold additional microoperations. Much better parallelism can be achieved if the partitioning can combine MOs from different blocks into the same MI, rather than being artificially restricted by block boundaries. Several global compaction techniques have been developed that allow the "motion" of MOs between blocks.

The Rules for Moving Microoperations

A set of rules will be developed that define the conditions under which an MO can be moved from one block to another. But first some additional terms must be defined.

The flow of control between basic blocks can be analyzed using *flow graphs*.¹¹ A flow graph is a directed graph with the basic blocks of a program as nodes. An edge is drawn from block B_m to B_n if upon exit from B_m the block next executed may be B_n. A cycle in the graph is called a *loop*. A data resource is *live* at the entrance to a block in a flow graph if the value stored in it may be read in that block or in some successor block without having been overwritten. A data resource which is not live at some point is *dead* at that point.

An MO is said to be *free at the top* of its block if no MO data precedes it, i.e. it is a root in the DAG. An MO is said to be *free at the bottom* if it does not data precede any MO, i.e. it is a leaf in the DAG.

Figure 3 shows the rules for moving an MO from one block to an adjacent block on the flow graph. Rule R1 states, for example, that an MO free at the top of a block can be removed from that block if a copy of the MO is added to the end of each parent block.

Block-Oriented Methods

Perhaps the simplest approach to global compaction is that of Dasgupta.⁹ He defines a *symmetric pair* of basic blocks in a loop-free microprogram as two blocks with the property that the second block is executed whenever the first one is, and vice versa. Blocks that are between a symmetric pair of blocks in the flow graph are called *internal blocks*. MOs which are free at the top of the second block and which are not data-preceded by any MO in an internal block are added where possible to an existing MI in the first block. This method is

```

begin
  prioritiesForMOs := CalculatePriorities (DAG)
  cycle := 0
  DRS := FormInitialDRS (DAG) (* Data Ready Set:=roots on DAG *)
  remainingMOs := SetSubtract (DRS, (*from*) AllMOs)
  while not empty (DRS) do
    cycle := cycle + 1
    MI[cycle] := EmptySet ()
    CandidateMOs := OrderMOsByPriority (DRS, prioritiesForMOs)
    while not empty (CandidateMOs) and not MIsFull (MI[cycle]) do
      CurrMO := PluckNextMO (CandidateMOs)
      if ResourceCompatible (CurrMO, MI[cycle]) then
        AddMO (CurrMO, MI[cycle])
        RemoveMO (CurrMO, DRS)
      end
    end
    newReadyMOs := AllParentsUsed (remainingMOs, DAG, MI[cycle])
    DRS := SetUnion (DRS, newReadyMOs)
    remainingMOs := SetSubtract (newReadyMOs, (*from*) remainingMOs)
  end
end.

```

Figure 2—A PASCAL-like specification of list scheduling

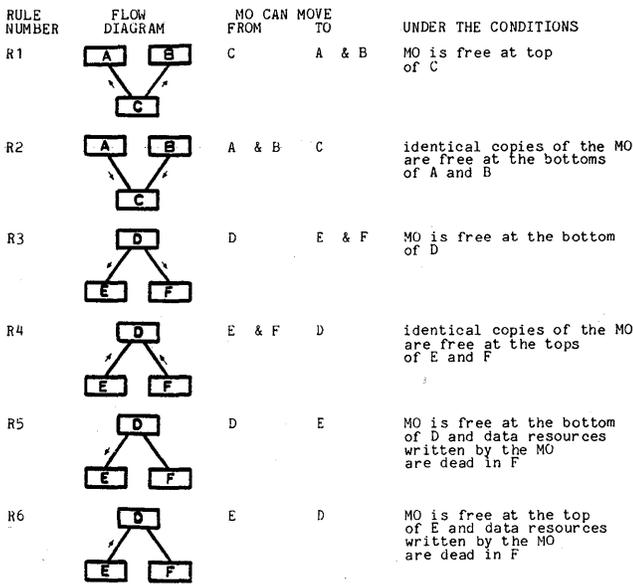


Figure 3—Rules for the motion of MOs across block boundaries

equivalent to applying rules R1 and R4 of Figure 3 as needed, starting with the second block of the symmetric pair and ending with the first.

The method of Tokoro et al.⁵ is based on a more general application of the rules for MO motion. In this method the blocks are compacted one by one in flow graph order. Legal motions between the current block and previously compacted blocks are considered. MO motions are made that appear “worthwhile” and that do not increase the number of MIs in a block.

For global methods that compact blocks individually, each MO motion between blocks requires recompacting the blocks involved, a time-consuming process. To alleviate this problem, Tokoro’s method takes advantage of the first-come, first-served nature of its FCFS local compaction technique. Adding an MO at the bottom of a compacted block does not require reanalyzing the relationships between the already compacted MOs, since the new MO is the “last to come.” This considerably simplifies the recompaction. A similar savings of time can be found for adding an MO to the top of an already compacted block. It should be remembered, however, that the effectiveness of the FCFS algorithm is still disputed and is little-tested in this context.

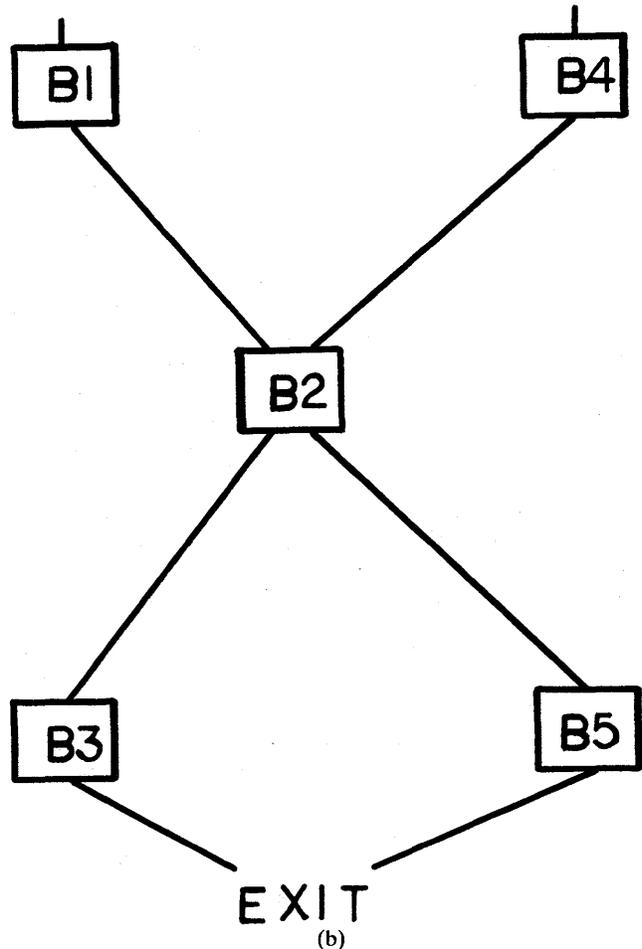
Another global method has been developed by Wood.¹² His method does not directly apply the rules of MO motion. Instead, he uses the concept of a *composite* microoperation, which is essentially a compacted block. Wood breaks up a microprogram into nested blocks, where a composite MO is treated as a node in a data precedence graph. The nested decomposition is always possible because his programs are written in a high level language with nested control structure. In Wood’s approach, MOs are able to be moved above and below a compacted loop as long as they do not have a data precedence relation with an MO in the loop. This approach does not detect many of the motions which are possible according to the MO motion rules of Figure 3.

A Trace-Oriented Method

This global compaction method⁶ operates on traces instead of basic blocks. A trace is a path through a microprogram’s flow graph, representing a sequence of microinstructions that

MO NUMBER	BLOCK NAME	READ REGISTERS	WRITTEN REGISTERS	TARGET BLOCK (S) IF JUMP MO	RESOURCE VECTOR
1	B1:	R1, R2	R3		[0, 1, 1, 0]
2		R3, R4	R5		[1, 0, 0, 0]
3		R5	R6		[0, 1, 0, 0]
4		R5, R6	R7		[0, 0, 1, 0]
5		R7, R3	R8		[1, 0, 0, 0]
6	B2:	R9	R10	B3, B5	[1, 0, 1, 0]
7		R9	R11		[0, 1, 1, 0]
8		R10			[0, 0, 1, 0]
9	B3:	R12	R13	EXIT	[0, 0, 0, 1]
10		R10	R15		[0, 0, 0, 0]
11		R13, R14	R16		[0, 0, 0, 1]
12		R16, R11	R17		[0, 0, 0, 1]
13		R17, R5	R18		[0, 0, 0, 1]
14		R18	R19		[0, 0, 0, 1]
15	B4:	R3, R4	R5	B2	[0, 0, 0, 1]
16		R7, R3	R8		[0, 0, 0, 1]
17	B5:	R13	R12		[1, 0, 0, 0]
EXIT:		the following registers dead at this point of the code: R1-7, R9-11, R13, R18			

(a)



(b)

BLOCK NUMBER	MOS FREE AT TOP	MOS FREE AT BOTTOM	REGISTERS LIVE AT TOP
B1	m1	m5	R1-2, R4, R5, R12-17
B2	m6, m7	m7, m8	R8, R9, R12-17
B3	m9, m10	m10, m14	R5, R8, R10-12, R14
B4	m15, m16	m15, m16	R3, R4, R7, R9, R12-14
B5	m17	m17	R8, R13-17

(c)

Figure 4—Example microprogram (a) The microoperations (b) The flow graph (c) Flow Graph Information

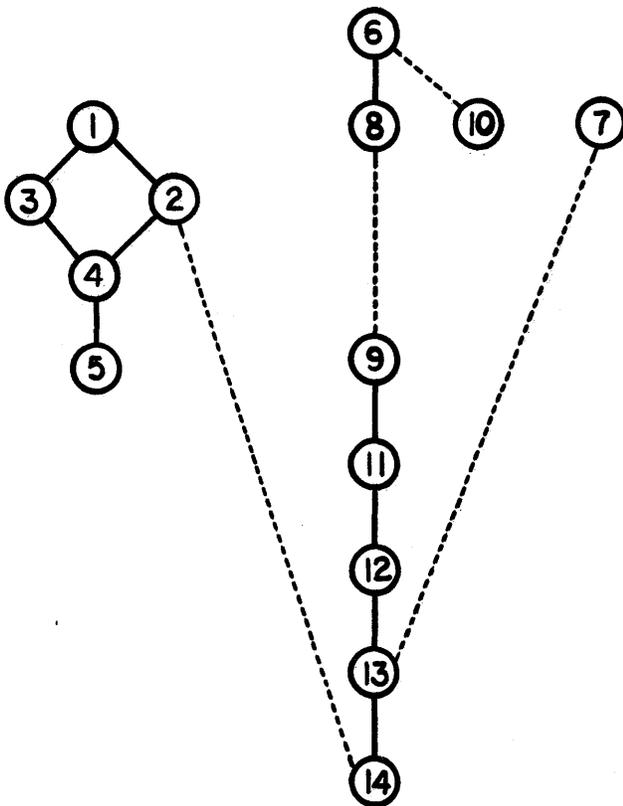


Figure 5—The DAG for trace B1-B2-B3

might be executed for some choice of data. Information about the dynamic behavior of the code is used to concentrate on traces that represent the most likely paths. This information includes estimates of the probability of each conditional jump being taken, which can be obtained from benchmark runs on uncompact microcode.

Figure 4 shows an example microprogram that will be used to illustrate the global compaction technique. Assume that the trace B1-B2-B3 has been chosen for the first compaction. The first step of the trace compaction technique is the construction of a DAG for the entire trace. Conditional jumps require special handling, since moving an MO from below a jump to above it might change a value needed in the block that is the target of the jump. The following must be added to the normal definition of data precedence:

Jump Precedence Rule. Suppose MO J is a conditional jump to block Bt, where Bt is not the immediately following block in the trace. When testing J for data preceding other MOs, any register live at the entrance to Bt is considered to be read by J. The extra reads are not used when testing whether other MOs data precede J.

The DAG obtained from the example trace is shown in Figure 5. The edges between MOs originating in different blocks are shown as dotted lines but do not receive any special treatment during compaction. The edge from M8 to M9 arises from the use of the jump precedence rule.

Once the DAG is formed, the trace is compacted using list scheduling. The microinstructions resulting from the com-

MI	MOs FROM BLOCK:		
	B1	B2	B3
1		M6	
2		M8	M10
3	M1		M9
4	M2	M7	M11
5	M3		M12
6	M4		M13
7	M5		M14

Figure 6—The compacted MIs for trace B1-B2-B3

paction of trace B1-B2-B3 are shown in Figure 6. For illustration purposes, the MOs in each MI have been aligned in columns according to their original block. The actual order of the MOs in an MI depends on the MI format.

The microprogram is not legal until a *bookkeeping* step has been done after the trace has been compacted. In general, the compaction process has moved many MOs according to the motion rules of Figure 3. The motions corresponding to rules R1 and R3 were made without moving the MO to both affected blocks, leaving the off-the-trace block without a copy of the MO. The bookkeeping step is explained as follows.

Consider rule R1 applied to blocks B1, B4, and B2 of Figure 4 (b). Since an execution of block B4 cannot be followed by the execution of any MO originally from B1, the “after B4 finishes” entry point in the new trace must come after the last MO originally in B1. As can be seen in Figure 6, the lowest MO in the B1 column is M5. Thus any MO originally in blocks B2 or B3 and now at or above M5 is above the new entry point and must be copied to block B4. These MOs are above and to the right of the line marked “R1 for M5” in Figure 7.

MI	MOs FROM BLOCK:		
	B1	B2	B3
1		M6	
2		M8	M10
3	M1		M9
4	M2	M7	M11
5	M3		M12
6	M4		M13
7	M5		M14

R1 for M5

R3 for M8

Figure 7—The MIs of Figure 6, with lines delineating MOs affected by the patch-up operation

Table I—A summary of the different approaches to global compaction

	Fundamental algorithm	Composite MOs	MOs can cross block boundary	Recompaction needed after each motion	Implementation
Dasgupta	FCFS source order dependent	no	symmetric pairs only	yes	no
Tokoro	Extended FCFS source order dependent	no	general rules for MO motion	yes	hand simulations
Wood	List Scheduling	yes	no	—	nonproduction implementation, small example
Fisher	List Scheduling	yes	general compaction of multiblock trace	no	hand simulations

Consider rule R3 applied to blocks B2, B3, and B5 of Figure 4b. Any MO originally in block B1 or B2 but now below the jump MO (M8) must be copied into block B5. These MOs are delineated by the line marked "R3 for M8" in Figure 7.

After the bookkeeping step for this trace has been completed, a new trace of uncompact blocks is selected. The process is then repeated until all of the blocks have been compacted. The example microprogram has a main trace of length 12 when individual blocks are separately compacted and has a main trace of length 7 when this global method is used. However, the space used rises from 16 to 20 microinstructions. Tradeoffs can be made between program space and execution time.⁶

In trace scheduling, loops are handled similarly to Wood's approach. Compacted loops may be treated as special MOs in the surrounding blocks. When these special MOs are contained in a trace, the scheduler has the ability to move MOs around the compacted loop to achieve the best result. The major features of the different approaches to global compaction are contrasted in Table I.

EXTENSIONS TO GLOBAL COMPACTION

Several extensions to global compaction techniques are currently being researched.

Compaction Techniques for Loops

A special case of global methods is the compaction of tight, innermost loops of microcode. Tight loops are relatively small (under 50 lines, say), are expected to have very many iter-

ations, and have few jump instructions in the loop body. These loops are frequently found in physical device input/output handlers and in compute-bound scientific code; small improvements in the speed of loops in compute-bound code may be far more significant than other global compactations.

An initial approach to loop compaction using trace scheduling begins by making K copies of the loop body. Each copy except the last is made to jump to the next copy instead of back to its own first instruction. The last copy is made to jump back to the first copy. Optimizing compilers can sometimes use a similar technique, called loop unrolling, to reduce the number of loop tests. The path from the old loop entrance to the last copy of the loop body is loop-free code and may be regarded as a trace. Trace scheduling can then be employed to compact this loop-free code. Since this allows operations that were originally in different loop iterations to appear in parallel, a significant speedup may be achieved.

The above technique suggests the following generalization. Rather than considering as fixed the number of iterations, the code is unwound "as needed" during the compaction process. This unwound code is considered a trace, and a data-precedence DAG for it is updated as the trace continues to expand. Some of the operations in the trace are jumps; extra edges are drawn from them according to the jump precedence rule. The code unwinding continues until a repeating pattern is recognized, i.e. until there is a *cyclic schedule*.

For a schedule to show a repeating pattern, the operations in any microinstruction, $M(c)$, must be identical to the operations in microinstruction $M(c + P)$, where P is the period. Number the copies of the loop body in the unwound code and let the number of the copy that an operation comes from be called its iteration. Then any operation in $M(c + P)$ and its counterpart in $M(c)$ must be the same number of iterations apart, e.g. W. The number of operations in this pattern will

be W times the number of operations in the original loop. Methods for producing schedules with this property are the subject of current investigation.

The reason for seeking a repeating schedule is that it may be rewound into a single loop. Each sequence of P instructions, such as $M(c)$, $M(c + 1)$, \dots , $M(c + P - 1)$, is identical to every other sequence of P instructions. This sequence forms the body of the rewound loop. All of the instructions that were scheduled before the pattern started are placed into a loop header. Both the header and the new loop body may contain conditional jump instructions in addition to the original loop tests, but the bookkeeping of normal trace scheduling assures that the schedule is legal.

Distinguishing Among Memory References

Unless there is absolute knowledge of the addresses of memory references when one is attempting to compact long sequences of code, there will be an implied data dependency between any write to memory and any subsequent read. This is due to the fact that the write may have been to the location being read. If this is the case, the read must follow the write. If there is a conceptually small enough quantity of code, the clever hand coder may know when two memory references are certain to be different.

This is especially important in loops. Loops commonly contain references tied to the loop index, and the above methods would be forced to produce a DAG with little available parallelism in order to assure that the references were done in the stated order. Researchers into multiprocessor systems have considered this problem.¹³ There, the aim is to notice that each loop iteration is data-independent from the others or to apply some transformation to cause that to be the case. Each loop iteration may be thought of as a process and assigned to its own processor. This problem is described as transforming *FOR* loops into *FORALL* loops.

In the loop compaction methods discussed above, it is not necessary to go quite so far. After the periodic schedule is rewound into a new loop, it is expected that there will be some data dependency between elements of different iterations. But the techniques to do a *FORALL* transformation are useful in eliminating many of the edges that would otherwise be obtained, and are applicable to microcode compaction. Similarly, registers are often referred to indirectly. Methods for distinguishing different register references would be useful for eliminating edges from the DAG.

Renaming Variables

Consider the following short section of code:

```
A := B / C
IF A >= 1 THEN
  A := C / B
END
```

The "single identity principle"² states that a variable should only be assigned a value once. Variables in existing code can

be renamed to achieve this. This renaming of variables allows computations to be made whenever the inputs are ready, even if the old value is still live. When code joins, as it does after the *END* of the *IF* statement above, it is impossible to use renaming to its fullest, since the code after the *END* would have to know which A was being referenced. We may, however, write:

```
A := B / C
A' := C / B
IF A >= 1 THEN
  A := A'
END
```

If division takes far longer than assignment and if two dividers (or one pipelined divider) are available, the two divisions may be overlapped to obtain faster execution. The same general technique has the potential of removing many of the DAG edges between operations originating in separate blocks.

Delayed Microoperation Binding

The act of assigning an abstract representation of an object to a concrete representation is referred to as *binding*. Typical bindings on the microcode level include: binding an abstract task description to a sequence of MOs, binding a variable to a memory location, etc. Research has been done on the important questions involving binding on the microcode level. In Davidson et al.¹⁴ high-level microprogramming language constructs may be bound via the use of specific programmer-described model bindings, while other constructs are bound in programmer-independent ways. The related question of how the compaction process might influence binding is considered here. Since microcode is very idiomatic, there are often sharply different sequences of microoperations that can carry out a given task. The extent to which tasks overlap will often depend upon a careful choice of bindings of the tasks. Sometimes choosing a long version of a task realizes the greatest overlap.

Work carried out in this area operates on a sequence of *tasks* rather than MOs. A task may be realized by a short sequence (often 1) of MOs. Before compaction, tasks are not bound to any particular sequence. Instead, attached to each is a set b_1, \dots, b_k of possible bindings. Each binding consists of the leading MO followed by a (possibly empty) sequence of tasks. Scheduling a task in a particular cycle means selecting a binding from b , scheduling the leading MO in that cycle, and placing the rest of the tasks from that binding on the DAG. Delayed binding makes scheduling a much more complex process. For example, simple list scheduling is unlikely to suffice. A list scheduler will always prefer the binding that allows the earliest scheduling of a task. Sometimes this is disastrous.

Register Allocation

If register allocation is done before compaction and there are more microprogram variables than registers, many arbi-

trary bindings will be made that influence the compaction process. For example, two operation streams that happen to use the same register cannot execute in parallel, but parallel activity may be possible with a change of register assignments. Besides register assignment, the choice of which variables are kept in registers at any moment influences compaction. Operations may be unnecessarily delayed in the schedule if variables they reference are temporarily not in a register. Furthermore, the variables needed for an MI become precisely known only at compaction time. Thus register allocation at compaction time may help minimize the number of changes in allocations. Since allocation changes require adding load and store operations to the original code, minimizing them can lead to shorter compactions.

Register allocation can occur as part of compaction by extending the "several version task" concept of the previous section. This is done with two changes to the compaction process. First, each binding that assumes that a piece of data is in a particular register is replaced by a set of bindings. This set is the original binding with the register replaced by a template that extends over all the permissible registers that may contain that data. When the task is bound, the specification of the selected register is inserted. Second, whenever a variable is removed from the registers, all bindings that read that variable are dynamically updated. The new bindings consist of the old ones preceded by appropriate fetches. The fetch is then a template that is only filled in when the variable is bound to a new register. To schedule well, then, the standard heuristics of register allocation will have to be included in the heuristic scheduling techniques.

CONCLUSIONS

The idiomatic nature of microcode makes its production a notoriously difficult problem surrounded by popular myths. Contrary to popular belief, the classical local microcode compaction problem is essentially solved. It is only the global compaction problem, which is in the initial stages of investigation, that still poses major research questions. We believe the most promising global technique is trace compaction, which automatically moves MOs across block boundaries in a structured fashion.

An important extension to trace scheduling is loop unrolling/rolling, which might yield significant improvements in the speed of compute-bound code. Delayed binding may achieve effective code compaction for architectures on which a variety of alternatives are available for carrying out a task. Architectures designed for machine emulation are particularly idiomatic and often offer such a choice. As a special case of delayed binding, register allocation is apparently best done in conjunction with compaction. Other extensions are also promising.

Recent trends in attached processors have included the use of the horizontal microcode level for user programming. While these machines are usually not as idiomatic as those

intended for emulation, they are often considerably more horizontal. Experiments for hardware with a related kind of parallelism have suggested that a significant speedup may be obtained,¹⁵ but we are unlikely to use such machines well without effective compaction techniques.

We believe the well-known advantages of high-level languages have not been widely applied in the microprogramming community because of the mistaken belief that microcode improvement is an impassable obstacle. It is true that for machines with horizontal architectures the added burden of compaction makes compilation more difficult. Nevertheless, development tools such as high-level language compilers are sorely needed in the horizontal environment. Perhaps their development may even serve as impetus to the development of high-level language compilers for machines with vertical architectures. The fact that the lower complexity of these machines permits programming at the microassembler (or lower) level has retarded the development of more reasonable tools.

REFERENCES

1. Davidson, S. and B. D. Shriver. "Firmware Engineering: An Extensive Update." In *Informatik-Fachberichte*, Vol. 31: *Firmware Engineering*, Springer-Verlag, 1980, pp. 25-71.
2. Ramamoorthy, D. V. and M. Tsuchiya. "A High-Level Language for Horizontal Microprogramming." *IEEE Trans. Comput.* C-23, 8 (Aug. 1974), pp. 791-801.
3. Landskov, D., S. Davidson, B. Shriver, and P. W. Mallett. "Local Microcode Compaction Techniques." *Comput. Surv.* 12, 3 (Sept. 1980), pp. 261-294.
4. Fisher, J. A. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling With Resources*. U.S. Dept. of Energy Report, Mathematics and Computing COO-3077-161, New York Univ., Oct. 1979.
5. Tokoro, M., T. Takizuka, E. Tamura, and I. Yamaura. "A Technique of Global Optimization of Microprograms." In *Proc. 11th Annual Workshop on Microprogramming*, ACM, IEEE, New York, 1978, pp. 41-50.
6. Fisher, J. A. "Trace Scheduling: A Technique for Global Microcode Compaction." *IEEE Trans. Comput.*, to appear.
7. Mallett, P. W. "Methods of Compacting Microprograms." Ph.D. dissertation, Univ. of Southwestern Louisiana, Lafayette, Dec. 1978.
8. Davidson, S., D. Landskov, B. D. Shriver, and P. W. Mallett. "Some Experiments in Local Microcode Compaction for Horizontal Machines." *IEEE Trans. Comput.*, to appear.
9. Dasgupta, S. "The Organization of Microprogram Stores." *Comput. Surv.* 11, 1 (March 1979), pp. 39-65.
10. Landskov, D. "The Equivalence of FCFS Compaction to List Scheduling With Source Order." Dept. of Computer Science Technical Report, Univ. of Southwestern Louisiana, Lafayette, Jan. 1981.
11. Aho, A. V., and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.
12. Wood, W. G. "The Computer Aided Design of Microprograms." Ph.D. dissertation, Univ. of Edinburgh, Scotland, 1979.
13. Padua, D. A., D. J. Kuck, and D. H. Lawrie. "High-Speed Multiprocessors and Compilation Techniques." *IEEE Trans. Comput.* C-29, 9 (Sept. 1980), pp. 763-776.
14. Davidson, S. and B. D. Shriver. "A Solution to the Resource Binding Problem." In *Proc. 1981 AFIPS National Computer Conf.* AFIPS Press, Arlington, Va.
15. Riseman, E. M. and C. C. Foster. "The Inhibition of Potential Parallelism by Conditional Jumps." *IEEE Trans. Comput.* C-21, 12 (Dec. 1972), pp. 1405-1411.

V-Compiler: A next-generation tool for microprogramming

by DAVE PATTERSON

University of California
Berkeley, California

and

ROSS GOODELL, MICHAEL D. POE, and SIMON C. STEELY, JR.

Digital Equipment Corporation
Tewksbury, Massachusetts

INTRODUCTION

Microprogramming has always been a difficult task. Related to hardware and software, it seems to have inherited difficulties from both. Microprogramming has the classic reliability and maintainability problems of software and from hardware it has inherited size and speed efficiency as practically the only measure of success. This legacy has made microprogramming a very difficult task.¹ Falk summarized the state of microprogramming today:

At present, microprogramming is an elite activity, performed effectively only by a small number of expert practitioners. The work is detailed, precise, time-consuming, and considerably more expensive than present-day software programming.²

In this environment, it is not surprising that the two characteristics of microprogramming that have the biggest impact on products are development time and microcode errors. Development time has its biggest impact in products that are built from off-the-shelf technology. This occurs occasionally in CPUs and frequently in special purpose devices. Almost by definition these projects are paced by the microcode development time.

The microcode problem that impacts all projects is errors. As microprograms have become more complex, clerical errors in assembly level coding become more expensive to locate and debug.

The current design methodology results in microarchitectures and microprograms that are designed largely by intuition. Although some fine tuning is performed, there is no time for experimentation. It would clearly be advantageous if design ideas are quantitatively investigated before they are incorporated into the final product.

The V-Compiler project began as an investigation into improving the situation described above. The characteristics that distinguish this project are a common format to interface all pieces of the system, the use of production systems to achieve

retargetability, global microword packing, and use of an intermediate level microprogramming language. We intend to build a practical tool for a practical environment.

This paper first presents the goals of the V-Compiler project and then mentions the research on which the V-Compiler is based. The structure and key ideas of the F-Compiler are then explained. The final portion includes the current status, the results of some simple examples, and an analysis of this current system in light of our goals.

SYSTEM GOALS

The long range goals of the system are the following.

Efficiency

This system should produce code for large microprograms by automated means which is within 15% of the size and speed of microcode that would be produced by hand. Important sections could be optimized by hand.

High Level

The microcode should be written in a form that is easy to read and write, simplifying maintenance and improving reliability. This increases the programmer's productivity, and makes it easier to deal with increasingly complex algorithms.

Simple Retargeting

In a practical environment adapting to and learning a new tool for each project is not feasible. It must be easy to make the system produce microcode for different microarchitectures. Our long term goal is to reduce this retargeting to one month for a CPU.

Transportable Microprograms

One of the major advantages of the system is the potential for reducing microcode development time within a family of CPUs, or for experimentations with hardware/firmware trade-offs. If the previous goals are reached, then it is possible to support a microarchitecture independent language. Experience with transportable BLISS programs at Digital indicates that highly efficient programs can be transported, although transportable programs must be written more carefully than nontransportable programs.³ Clearly the development time and the error rate would be reduced if we could reuse microprograms. Perhaps even more important, the ability to modify the microarchitectures without rewriting the microcode would allow designers to see how these modifications affect performance. Experimentation could then check intuition in evolving future microarchitecture designs.

User Directed Compilation

The aim of the V-Compiler is to assist in creating efficient microprograms, but the microprogrammer must be able to override any compiler decision.

RELATED RESEARCH

Space restrictions allow us to only mention research that has had a direct influence on our work. Several recent papers provide an excellent survey of related microprogramming research.^{4, 5, 6, 7} The work by Patterson⁸, Goodell⁹, Fisher⁷, Sager¹⁰, and Cattell¹¹ have directly influenced our research. Patterson^{2,8} demonstrated that machine dependent, high level microprogramming languages can produce efficient microcode. Goodell⁹ pointed out there is not enough time to develop a new compiler during each project development cycle, since microprogramming is so time consuming that it cannot wait. Fisher's⁷ approach to global optimization is presented elsewhere in this proceedings. Sager's¹⁰ work on an internal Digital project proved the usefulness of algorithms for compaction of basic blocks, moving micro-operations past conditional branches, and loop unrolling in the production of microcode. Cattell's¹¹ research is the basis for the retargetability of the Production Quality Compiler project at Carnegie Mellon University.¹² His goal was to automatically derive code generators for algorithmic high level languages from machine descriptions of register oriented instruction sets. The high level language problem was simplified by choosing an appropriate intermediate level language. This research has several important ideas, but the most relevant to V-Compiler are listed here:

1. Distinction between Compile time and Compile-Compile time. An obvious idea, this distinguishes between work done once per compiler (Compile-Compile time) and work done once per compilation (Compile time). The former can be relatively expensive but the latter must be relatively efficient.
2. Compile-Compile time uses machine independent axioms to provide a code sequence for every intermediate

language statement. If an instruction can be found that performs the intermediate language statement exactly, then it is used. If more than one instruction is required, a heuristic search is used to find the matching sequence. For example, his technique found a minimal sequence of operations to perform AND on the PDP-11 (which only has inverted AND) and SUBTRACT on the PDP-8 (which only has ADD and TWO's COMPLEMENT).

3. Compile-time uses the prederived sequences to generate code from the intermediate language programs. Cattell uses the largest match ("maximal munching method") when more than one sequence can be applied.

Cattell¹¹ also uses productions and we have incorporated this idea in the V-Compiler. Productions systems are commonly used in artificial intelligence research. A well-known example of productions is the use of BNF (Backus-Naur Form) to describe the grammars of programming languages. Production systems represent an alternative programming methodology to procedure oriented languages.¹³ A pattern (antecedent) describes the sequence of code or data tokens that invoke the production. A second pattern (consequence) describes what replace the tokens. When a pattern matches the antecedent, a body of code is executed, which may construct variables for the consequence.

Recent work¹⁴ has demonstrated the use of productions to preprocess source text. Complicated strategies are used in artificial intelligence to decide which production is to be applied, when more than one production could be simultaneously invoked.¹⁵ Production systems have been used successfully to synthesize complete programs.¹⁶

V-COMPILER TECHNIQUES

Productions

Productions must be easy to write and must execute efficiently. To simplify the writing of productions, we added the necessary capabilities to the microprogramming source language, rather than introducing a specialized notation. Production systems can be created much as macros would be created in an assembler language. Productions can be interactively written and applied to simplify their development.

Several techniques are used to reduce the overhead of using productions. Although backtracking and production nesting are supported, only a single pass is made over the input for a set of productions. Pattern matching is reduced by only allowing semantic productions which are matched against the parsed input trees, rather than matching the characters of the input syntax.¹⁷ Finally, the productions are precompiled into threaded code to accelerate their interpretation.

Packing

This packer differs from the trace approach by Fisher.¹⁸ A method of ordering the compaction of code blocks¹⁹ allows tight loops to receive the strongest compaction. This approach also avoids the problem found in an ordering method based on a complete trace through the microprogram.¹⁸ As multiway

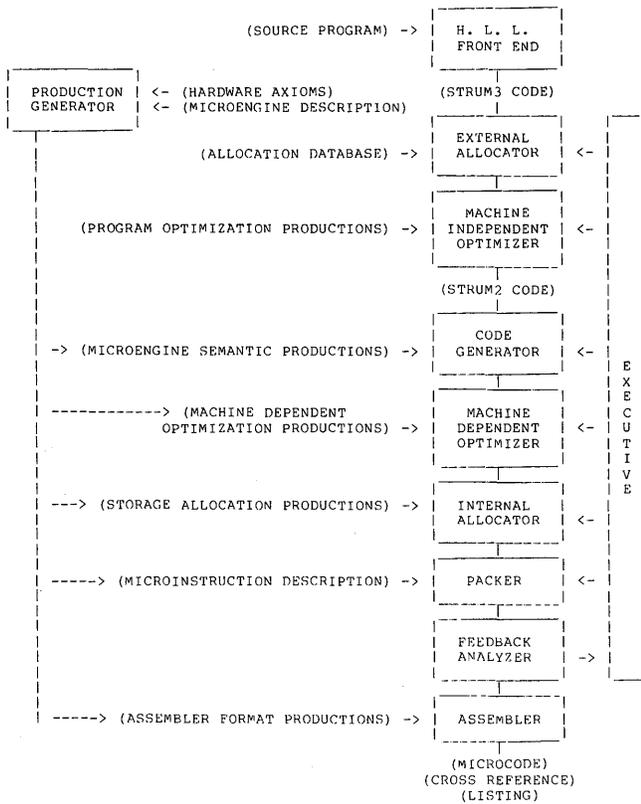


Figure 1—The V-Compiler components

branches are common in microcode investigated by the authors, a single path through a multiway branch is never shortened at the expense of the other paths in that set, when elements of the set are individually less likely to be executed but collectively represent a majority of the execution time.

The packer of the V-Compiler¹⁹, allows incremental binding of resources. This approach can also allow a form of register allocation to be made at compaction time, if identical elements of a register class are allocated as resources. With careful writing of productions, the packer currently is able to reverse the order of control flow fork-join pairs, although it has not been measured how often this optimization is important. The packer implements the write-read group motion mentioned by Fisher.⁷ Loop unrolling and possible rerolling, also mentioned by Fisher⁷, has been performed by hand in a commercial environment.¹⁰ We intend to try this with the executive, packer, and machine independent optimizer.

V-COMPILER SYSTEM

Figure 1 shows the components of the ultimate V-Compiler. Our design philosophy was to make each logical step of the translation process correspond to one V-Compiler component. As each component in Figure 1 requires substantial development time, we decided that we must tackle the most important components first. Our reasoning is that it is not worthwhile to develop the complete system if the critical components do not achieve satisfactory results.

```

begin
FIELD.SIZE := reg[0];           !(defines FIELD.SIZE as register 0)
BIT.POSITION := reg[1];
BYTE.POS := reg[2];
BASE.ADR := reg[3];
FIELD.START := reg[4];
7. := const.reg[7];           !(defines 7. as register with constant value 7)
32. := const.reg[32];
RES.OPND.FLT := xrtm;         !(define RES.OPND.FLT as an exit routine)

if FIELD.SIZE gtru 32. !if more than 32 bits then take error exit
then
  exit(RES.OPND.FLT);         !exit must specify an exit routine (allows common tails)
exit(RES.OPND.FLT);
BYTE.POS = BIT.POSITION srl 3; !get byte offset by shifting right 3
BASE.ADR = BASE.ADR + BYTE.POS; !get byte address of start of field
FIELD.START = 7. and BIT.POSITION; !get relative address of start of field
...;
end;
    
```

Figure 2—Microcode fragment in external (STRUM2) format

The V-Compiler system is being built in phases. Each phase extends rather than replaces the previous system. Before describing the system, it is useful to describe what may be the most important aspect of a multi-phase system; the interfaces between the components. As the components of a hardware system can be interfaced by a common bus, the components of the V-Compiler are interconnected by a common intermediate data representation called the Software Bus (S-Bus).²⁰ All data shared between components uses the S-Bus structure.

Like the GDB intermediate representation used by ISPS²¹ and the TCOL intermediate language of the PQCC¹², the S-Bus provides a common intermediate representation for expression trees. The advantage of a standard interface is that a single set of routines can be used by all programs. Using the idea of data abstraction, the S-Bus extends this concept by making these primitive routines the only way to access S-Bus functions. This allows the S-Bus to use data structures which fit the application without disturbing the rest of the system.^{22, 23} It also carefully separates semantics from syntax to allow unanticipated extensions to be easily added.

In addition to the obvious advantages of a standard interface, the S-Bus supports user conveniences. We have developed a tool which transforms list formatted S-Bus into a more readable form and vice versa. Figures 2 and 3 have been included to give the reader a feeling of the S-Bus syntax as seen by the programmer and program respectively.

Now that we have specified the interface language, we can proceed to describe the components of the V-Compiler shown in Figure 1. The *Executive*, the long block on the right, controls the special purpose components on the left. Starting at the top, we will explain each of these components and comment upon the difficulty of implementation and current status:

The *high level language front end* is a compiler front end which will translate from a high level language into an intermediate level, machine independent S-Bus variant called STRUM3. This language consists of arithmetic logical and field operators, simple assignment statements, jumps, *IF-THEN* and *IF-THEN-ELSE*, and *WHILE*. It also allows the definition of blocks and procedures.

One of our early important decisions was to delay the high level language front end to the very end of the project. We consider translation of high level to intermediate level language to be a well understood problem and certainly not the most difficult problem on the list. It is also possible that future improvements in language design and compiler technology

```

%%BLOCK ( %%DEF ( FIELD.SIZE ,
                REG ( %%FIELD ( 0 ) ) ) ,
          %%DEF ( BIT.POSITION ,
                REG ( %%FIELD ( 1 ) ) ) ,
          %%DEF ( BYTE.POS ,
                REG ( %%FIELD ( 2 ) ) ) ,
          %%DEF ( BASE.ADR ,
                REG ( %%FIELD ( 3 ) ) ) ,
          %%DEF ( FIELD.START ,
                REG ( %%FIELD ( 4 ) ) ) ,
          %%DEF ( 7. ,
                CONST.REG ( %%FIELD ( 7 ) ) ) ,
          %%DEF ( 32. ,
                CONST.REG ( %%FIELD ( 32 ) ) ) ,
          %%DEF ( RES.OPND.FLT ,
                XRTN ) ,
          %%IF ( %%CTRU ( FIELD.SIZE ,
                        32. ) ,
                EXIT ( RES.OPND.FLT ) ) ,
          %%ASSIGN ( BYTE.POS ,
                    %%SRD ( BIT.POSITION ,
                            3 ) ) ,
          %%ASSIGN ( BASE.ADR ,
                    %%SUM ( BASE.ADR ,
                            BYTE.POS ) ) ,
          %%ASSIGN ( FIELD.START ,
                    %%AND ( 7. ,
                            BIT.POSITION ) ) ,
          ; ; ,
          ) ; ; '

```

Figure 3—Microcode fragment converted to internal S-Bus format

will dictate selection of a different language than we might have originally selected.

The *external allocator* is needed to allocate resources to the external variables of separately compiled microroutines. We must eventually support separate compilation in our practical environment. The external allocator has not been implemented and it is not a trivial problem. Our current idea is that it will at least provide the bookkeeping function (normally associated with linking) of which resources are used by each routine and may eventually cause recompilation and reallocation of conflicting components.

The *machine independent optimizer* will perform the classic compiler optimizations, such as constant folding, removing common subexpressions, and elimination of chained jumps. This was not implemented, but we again believe that the techniques are well known. Others are using productions to specify such optimizations.^{11, 24}

The next box is the *code generator* which reads the internal form of the STRUM2 programs and the microengine semantic production table. STRUM2 is generated by the machine independent optimizer and is a lower level language than STRUM3. Figure 2 shows an example of STRUM2 code. The code generator applies the productions to STRUM2 programs to produce microoperations for the machine. This is essentially a translator, but this component is made much more simple by the STRUM2 language. An initial version of the code generator was implemented and translated list structured STRUM2 program (Figure 3) into machine dependent microoperations (Figure 4). We believe the simplicity of STRUM2 will allow a relatively small number of productions to generate efficient sequences of microoperations.

An advantage of this approach, which holds for any higher level translation system, is that the efficiency of the object code can be improved by enhancing the translation process rather than by rewriting the source. The table that drives the

function generator can be improved over the whole microcode development cycle. Since this table contains productions which are to be generated by senior microprogrammers, it will be much easier for this to occur in the V-Compiler than in traditional translators. As the few lines of a production are applied over the thousands of lines of microcode, this technique may be more effective than hand optimization of the object code. Several of the components (the machine independent optimizer, code generator, machine dependent optimizer, and internal allocator) are implemented by productions which are interpreted by a common production handler.¹⁷

The *machine dependent optimizer* takes advantage of microarchitecture idioms to produce better code; for example, incrementing a counter rather than using an adder. This component is analogous to the peephole optimizer found in many compilers. As the optimizations are again specified as productions, there is very little difference between the programs that implement the code generator and this component. By separating these components the code generation productions can be easily changed for a new microarchitecture and only the machine dependent optimization productions would need to be recreated. This optimizer, however, was not part of our initial system.

The *internal allocator* binds any local variables to fixed registers. The resource allocation problem is difficult; this program is not part of our initial V-Compiler. Our approach (incremental binding) is to have the microprogrammer declare variables as certain classes and to have the code generator accumulate a set of constraints about each variable. This information will be used by the internal allocator. Since external use of resources has already been established, we believe this subproblem of the general resource allocation problem will be manageable with simple heuristics.

```

%%BLOCK ( %%VDEF ( 1 ) ,
          %%VDEF ( CC ) ,
          %%VDEF ( PC ) ,
          %%VDEF ( %LEQU ) ,
          %%VDEF ( %REG0 ) ,
          %%VDEF ( %REG1 ) ,
          %%VDEF ( %REG2 ) ,
          %%VDEF ( %REG3 ) ,
          %%VDEF ( %REG4 ) ,
          %%VDEF ( %CONST.REG7 ) ,
          %%VDEF ( %CONST.REG32 ) ,
          %%VDEF ( RES.OPND.FLT ) ,
          %%ASSIGN ( CC ,
                    %%COMPARE ( %REG0 ,
                                %CONST.REG32 ) ) ,
          %%FLOW ( %%ASSIGN ( PC ,
                              %%SKIP.IF ( %%TEST ( CC ,
                                                    %LEQU ) ) ,
                              PC ) ) ,
                    %%EXIT ( RES.OPND.FLT ) ) ,
          %%ASSIGN ( %REG2 ,
                    %REG1 ) ,
          %%ASSIGN ( %REG2 ,
                    %%SRD ( %REG2 ,
                            1 ) ) ,
          %%ASSIGN ( %REG2 ,
                    %%SRD ( %REG2 ,
                            1 ) ) ,
          %%ASSIGN ( %REG2 ,
                    %%SRD ( %REG2 ,
                            1 ) ) ,
          %%ASSIGN ( %REG3 ,
                    %%SUM ( %REG3 ,
                            %REG2 ) ) ,
          %%ASSIGN ( %REG4 ,
                    %CONST.REG7 ) ,
          %%ASSIGN ( %REG4 ,
                    %%AND ( %REG4 ,
                            %REG1 ) ) ,
          ) ; ; '

```

Figure 4—Microcode fragment in machine dependent micro-operations

The *packer* first reads the micro-operation sequences to form a data dependency graph based on reads, writes, and tests of registers. Using this graph, a sequence of micro-operations is scheduled into parallel microinstructions. An initial version of this component was implemented. Figure 5 gives a sample packing of the microoperations in Figure 4 for a fictional microengine with dual input and output ports on the register storage, with a separate ALU, shifter, and constant ROM.

A problem of most microcode improvement systems is providing a system that allows microarchitecture independent compaction. Our pragmatic approach is to require the user to supply two custom subroutines:

1. FIT—The two inputs to this subroutine are a micro-operation and a partially formed microinstruction. This routine simply returns a Boolean value indicating whether micro-operation will fit into the microword.
2. COMBINE—This subroutine uses the same input but returns the microinstruction with the new micro-operation placed within it.

This approach allows complete generality of format of the microword and packer without introduction of an elaborate notation in the initial system. To keep the packer machine independent, we have also created packer directives that relay control flow and therefore the packer does not have to understand the semantics of the microarchitecture. For example, the directive EXIT identifies the micro-operations that cause an unconditional transfer of control.

The next component is the *analyzer* which digests information from the functions above it and feeds it back to the executive in the form of directives for a more optimal compilation of the same code. Perhaps the most difficult component in Figure 1, this was not part of the initial system. The idea is that by providing information from lower levels, the V-Compiler can iteratively produce better microcode. As experience is gained from the earlier V-Compiler, we will better understand how to take advantage of feedback. This is clearly an open research problem and is part of the dissertation research of Poe at University of Massachusetts at Amherst.

The *assembler* takes the internal S-Bus form of the microinstructions with the format production table to produce the final microinstructions. This is a necessary but conceptually simple component. Probably the most difficult part is linking microaddresses from separately compiled microroutines. Although the initial V-Compiler system used an existing micro-assembler, we expect later versions to include assembly in

```

UWORD (  %%ASSIGN(%REG2,%REG1),
          %%ASSIGN(CC,%%COMPARE(%REG0,%CONST.REG32))
        ),
UWORD (  %%ASSIGN(%REG2,%%SRD(%REG2,1)),
          %%ASSIGN(%REG4,%CONST.REG7)
        ),
UWORD (  %%ASSIGN(%REG2,%%SRD(%REG2,1)),
          %%ASSIGN(PC,%%SKIP.IF(%%TEST(CC,%LEQU),PC))
        ),
UWORD (  %%EXIT(RES.OPND.FLT)
        ),
UWORD (  %%ASSIGN(%REG4,%%AND(%REG4,%REG1))
        ),
UWORD (  %%ASSIGN(%REG2,%%SRD(%REG2,1))
        ),
UWORD (  %%ASSIGN(%REG3,%%SUM(%REG3,%REG2))
        ),

```

Figure 5—Microcode fragment in packed microwords

order to avoid a second machine description for the micro-assembler. We would expect that a future version would also provide cross references which map STRUM2 statements into microinstructions and vice versa. It should provide the information needed for microcode simulation programs.²⁵

In the upper left corner of Figure 3 is the production generator. This program runs at compile-compile time. It reads a formal machine description and uses a set of axioms to derive the machine dependent productions for the compiler. This is again an open research problem being studied by Poe and is not part of our initial implementation. The advantage of this approach is that it would take much less time to define a new microarchitecture, facilitating practical experimentation with microarchitectures and higher level microprogramming.

Summarizing the description above, we ultimately rely upon the executive to supply intelligence in the system. As components are implemented, we expect the executive to vary their parameters, and call components repeatedly to achieve the best results. An obvious example is trying variations of register allocation by the internal allocator and recording the resulting number of microinstructions produced by the packer. A final comment is that, as part of the multilevel design philosophy, users may add subroutines to tailor the system to particular microarchitectures.

EXAMPLES

As of this writing, the prototype system does basic compilation and packing of STRUM2 programs. It includes a production driven code generator, a simple "internal allocator" which binds variables to registers as specified explicitly in the input STRUM2 code, a preliminary packer which allows some movement of micro-operations past basic blocks, and a conversion routine (written in productions) which converts packer output to source code for an existing micro-assembler. The sample code in Figures 2, 3, 4, and 5 was used and generated by the components of this current system.

In order to test our ideas, we produced by hand an example for a real architecture before we began implementation. The subroutine selected extracts a bit field from memory. We used both Fisher's¹⁸ height and Sager's¹⁰ heuristic scheduling approaches. The result was that Fisher's schedule was 18% (five extra microinstructions) and Sager's 11% (three extra microinstructions) longer than the hand-coded example. Since the experience of Patterson¹ and Tokoro²⁶ have been that small examples are usually more efficient when done by hand and large examples are usually more efficient when done automatically, these are encouraging results. On the other hand, since these examples were done knowing the hand-coded microcode, the results are still preliminary.

While this is an interim report, it is interesting to compare our current results to the goals for the whole system presented in the first part of the paper.

Efficiency

The example in the figures produced code as good as that produced by hand for a paper machine. An example done by

hand for a real machine was 11% to 18% larger than hand coded microcode. These results appear to meet our goal.

High level language

The human readable version of STRUM2 has proved to be a useful low level microprogramming language and is a considerable improvement over traditional microassembly language.

Simple retargeting and transportability

As we have not tried to retarget the V-Compiler we have no definite results. We do, however, have indications that writing the productions is reasonable. The example in Figure 2 required six productions occupying less than 30 lines of code.

User direction

This initial system allows the microprogrammer to include explicit microinstructions in the STRUM2 language. Actions of the packer can be controlled by including directives in the productions. For example, TIE, specifies that a pair of microoperations must appear in sequential microinstructions.

A second concern was performance of production based translators and packing. Since the primary purpose of the prototype V-Compiler is to establish the feasibility of the V-Compiler, little effort has gone into programming it efficiently. Care was taken, however, to make the basic algorithms efficient, and the code was written in BLISS, which has a highly optimizing compiler. It took the prototype system 35 seconds to compile and pack the program fragment shown in Figure 2 on a lightly loaded VAX 11/780. Precompiling the productions reduced this time to 23 seconds.

SUMMARY

The ideas and current status of the V-Compiler system have been described. Each phase will make the microprogramming task easier with little loss of efficiency. Four basic concepts underlie the V-Compiler effort:

S-Bus

All components of the system use this common structure for input and output. The analogy is made to a hardware bus which allows various devices to communicate easily.

Productions

The V-Compiler achieves retargetability by using a powerful semantic production system for the machine dependent translations required during compilation.

Microword Packing

Beginning with the work of Fisher¹⁸ and Sager¹⁰, we are implementing a system of building microwords out of vertical sequences of microoperations. A key feature of this program is machine independence.

Critical Components First

Our development strategy is based upon solving the most critical problems first. For instance we have ignored the problem of selecting high level microprogramming language. Instead, we have concentrated on developing an intermediate language that simplifies the V-Compiler and which could be produced by a high level language front end.

Simple examples done by hand and by the initial V-Compiler yield promising results. We are encouraged and will continue to evolve the V-Compiler.

ACKNOWLEDGMENTS

The authors are indebted to the reviewers and to Cheryl Wiecek and Tom Eggers for their careful reading.

REFERENCES

1. Patterson, D. A., "An Experiment in High Level Language Microprogramming and Verification," to appear, *CACM*, 1981.
2. Falk, H., "Hard-soft Tradeoffs," *IEEE Spectrum*, Vol. 11, No. 2, February 1974, pp. 34.
3. Brender, R. F., "A Survey of BLISS-16, BLISS-32, and BLISS-36" *Proceedings of the Digital Equipment Computer Users Society*, DEC, April 1978, pp. 1113-1127.
4. Dasgupta, S., "Some Aspects of High-Level Microprogramming," *Computing Surveys*, Vol. 12, No. 3, September 1980, pp. 295-323.
5. Landskov, D., Davidson, S., Shriver, B., and Mallett, P. W., "Local Microcode Compaction Techniques," *Computing Surveys*, Vol. 12, No. 3, September 1980, pp. 261-294.
6. Davidson, S., and Shriver, B. D., "Firmware Engineering: An Extensive Update", in *Firmware, Microprogramming, and Restructurable Hardware*, Chroust, G. and Mihlbacker, J. R. (Eds), North-Holland Publishing Co., Amsterdam, 1980, pp. 1-36.
7. Fisher, J. A., Landskov, D., and Shriver, B., "Microcode Compaction: Looking Backward and Looking Forward", this proceedings, 1981.
8. Patterson, D. A., "STRUM: Structured Microprogramming System for Correct Firmware," *IEEE Transactions on Computers*, Vol C-25, No. 10, October 1976, pp. 974-985.
9. Goodell, R., "An ISPS Micro-assembler," *Proceedings Computer Hardware Description Languages*, Palo Alto, California, October 8-9, 1979, pp. 62-68.
10. Sager, D., private communication, 1977.
11. Cattell, R. G., "Formalization and Automatic Derivation of Code Generators," PHD thesis, Department of Computer Science, Carnegie-Mellon University, April 1978.
12. Leverett, B. W., et al., "An Overview of the Production Quality Compiler-Compiler Project," Department of Computer Science, Carnegie-Mellon University, February 1979.
13. Davis, R., and King, J., "An Overview of Production Systems," *Machine Intelligence*, 8, 1977, pp. 300-332.
14. Greenwood, S. R., "Macro: A Programming Language," *SIGPLAN Notices*, Vol. 14, No. 12, December 1979, pp. 80-91.
15. Nilsson, N. N., *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, California, 1980.

16. Barstow, D. R., "Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules," Stanford Artificial Intelligence Laboratory, Memo AIM-308, November 1977.
17. Goodell, R., "Using Semantic Productions to Compile Microcode in the V-Compiler", in preparation, 1981.
18. Fisher, J. A., "The Optimization of Horizontal Microcode Withing Basic Blocks and Beyond," PHD thesis, Courant Institute, New York University, New York, New York, October 1979.
19. Poe, M. D., "Heuristics for the Global Optimizations of Microprograms," *Proceedings of the 13th Annual Workshop on Microprogramming*, Colorado Springs, Colorado, November 30—December 3, 1980, pp. 13-22.
20. Goodell, R., "The Software Bus," unpublished DEC Internal Document, December 1979.
21. Barbacci, M. R., Barnes, G. E., Cattell, R. C., and Siewiorek, D. P., "The ISPS Computer Description Language," Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.
22. Liskov, B. H., "A Design Methodology for Reliable Systems", *Proceedings FJCC*, 1972, pp. 191-199.
23. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems Modules", *Comm. ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
24. Sethi, R., "Semantic Directed Compiler Generation", SICPLAN meeting, Boston Chapter, October 2, 1980.
25. Wiecek, C. A., and Steely, S.C., Jr., "Performance Simulation as a Tool in Central Processing Unit Design," *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, University of Colorado, Boulder, Colorado, August 13-15, 1979, pp. 41-47.
26. Tokoro, M., et al., "An Approach to Micro-program Optimization Considering Resource Occupancy and Instruction Formats," *Proceedings 10th Annual Workshop on Microprogramming*, 1977, pp. 92-100.

Adaptable pipeline system with dynamic architecture*

by SVETLANA P. KARTASHEV
University of Nebraska
Lincoln, Nebraska

and

STEVEN I. KARTASHEV
Dynamic Computer Architecture, Inc.
Lincoln Nebraska

ABSTRACT

This paper describes a pipeline system with dynamic architecture that performs cost-effective adaptations to the algorithm being executed. The system performs the following pipeline adaptations: (1) the number of stages in the pipeline changed to allow each instruction to activate the number of stages that matches the number of operations it realizes; (2) the operation sequences in the pipeline modified to allow any sequence of operations to execute without reconfiguration and thus eliminate the time overhead caused by this reconfiguration; and (3) the operation time in each stage adjusted to the minimum required for that operation because it may shorten the time of the total operation. This paper also discusses fast and flexible information exchanges between pipeline stages that can be done while the pipeline is working. Namely, each pipeline stage C_i may obtain during pipeline computations a temporary result that was computed by any other stage C_j . It is shown that such a pipeline may be organized from DC groups and thus be amenable to LSI implementation.

A. INTRODUCTION

Pipeline systems may employ two kinds of pipelines—instruction and arithmetic. In an instruction pipeline, the fetching of instructions and operands is fragmented into several short overlapped processes (instruction address generation, instruction fetch, instruction decode, operand address generation, and operand fetch).

The main problems faced in an instruction pipeline are those of time overheads caused by conditional branches and of

variations in the number of operand addresses and addressing procedures used in instructions.^{1,2}

In existing instruction pipelines reconfiguration is used mostly to offset the time overheads caused by the last mentioned factor. For instance, in the instruction pipeline of the MU5³, instructions may bypass unneeded pipeline stages. This, however, creates dummy time intervals associated with resolving conflicts when the instruction encounters operands prepared for preceding instructions as a result of such bypassing.

In an arithmetic pipeline an instruction containing several consecutive operations propagates through pipeline stages so that each stage executes one operation of this instruction. Since the operation phase of the i th instruction overlaps the phase of operand fetch for the next ($i + 1$ st) instruction, the time required to propagate an instruction through one stage equals the time of the operation assigned to the stage. All the times for operand fetches are therefore eliminated from the time of program execution. This is the source of the executional speedup achieved by an arithmetic pipeline.

The major problem with arithmetic pipelines is the time overhead introduced because of the disparity between the pipeline(s) and the algorithm being executed. As a result, pipeline systems tend to become dedicated to certain types of computations and usually have a limited applicability.

To broaden the range of their cost-effective application, arithmetic pipelines offer various software controllable reconfigurations of the available hardware resource. The general idea is to reconfigure the resource by means of software to reduce the dissimilarity between the arithmetic pipeline and the sequences of operations assigned to different instructions.

Let us look at some existing systems and consider their use of reconfiguration. In the TI ASC⁴, reconfiguration of the arithmetic pipeline means bypassing unneeded pipeline stages, i.e., the instruction propagates through a stage only if it implements the operation encountered in the instruction. Otherwise this stage is bypassed. This is similar to the way the MU-5 reconfigures its instruction pipeline. The weakness of

* This work was supported by Subcontract No. 481-79-35 of the Prime Contract No. DASG60-78-C-0058 between General Research Corporation and U.S. Army Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama.

this technique is in the time lost solving conflicts among instructions implementing different sequences of arithmetic operations.

In the CRAY-1⁵, 12 functional units are organized into dedicated pipelines. These are partitioned into four categories: address, scalar, vector, and floating point. Although each pipeline cannot reconfigure, reconfiguration is used to chain several pipelines together to form pipelines with a larger number of stages. This allows the pipelined execution of instructions with long sequences of operations to be organized. Reconfiguration is thus used in existing pipeline systems in a limited sense, as a means of bypassing or chaining pipeline stages.

Several authors have proposed theoretical pipeline systems with a deeper level of reconfiguration. Reddi and Feustel⁶ describe a restructurable pipeline system that may reconfigure into different sequences of resource units. The architecture of such systems provides that programs to be executed be decomposed into separate program blocks, and the compiler determines which interconnections must be established among the operational units assigned to the execution of each block. This pattern of interconnections is then encoded into a program instruction that activates the required interconnections among hardware units during execution of said program block.

Thomasian and Avizienis⁷ proposed a reconfigurable pipeline system consisting of an array of pipelined arithmetic processors that can employ different configurations between processors. To carry out pipelined computations, the tasks requiring the same configuration are arranged together. After executing all tasks appropriate for that configuration, the system reconfigures and starts execution of a new block of tasks in the next configuration.

B. PROBLEMS OF PIPELINE COMPUTATIONS

Since many programs have different sequences of operations following each other, a pipeline that executes such a program must reconfigure each time it switches from one sequence of operations to another. If Δt is the time required for each reconfiguration and N is the number of different operation sequences, then the pipeline loses time $\Delta t \cdot N$ reconfiguring its resource. If Δt or N is large, then the speed advantage of pipelined computation may be lost and there is no sense in computing this program in a pipeline. To reduce N , programs are sometimes rearranged into tasks where each task may be computed by a single configuration of the pipeline. But this requires special programming which may again restrict the class of programs that can cost-effectively be pipelined.

Next it is quite difficult to compute by means of pipeline programs that require broad exchanges of temporary results between tasks. Indeed, each stage of a pipeline has to receive two operands to execute an operation: one from the preceding stage and another one from the local memory attached to the stage. Each local memory is therefore engaged in continuous fetches of operands required by that stage while the pipeline is working. Thus during computation it is impossible to load this memory with temporary results needed by the stage, otherwise access to operands stored in the local memory stops.

Therefore all local memories may be loaded with data words only before computation begins. Hence, if a pipeline stage needs a temporary result computed earlier, it must fetch this result from another memory designated for temporary results. Since a pipeline stage may require a temporary result computed by any other pipeline stage, it has to be provided with fast information exchanges between stages that do not degrade pipeline performance. Since such exchanges are poorly developed in existing systems, many programs are eliminated from consideration for pipeline computation. This again narrows pipeline applicability. Limited applicability is therefore the most severe drawback of all pipeline computations.

Their other drawbacks are associated with the following causes. Frequent disparity between the number of consecutive operations in the instruction and the number of pipeline stages connected into a pipeline creates additional dummy time intervals associated with instruction propagation through unneeded stages or with conflict resolution when an instruction bypasses the unneeded stages and encounters operands prepared for some of its predecessors.^{3,4}

In existing pipelines, the time for processor dependent operations (addition, subtraction, $>$, $<$, etc.) is permanent and does not depend on operand size; however, selection of a permanent operation time in each stage requires that it be selected as the time of the longest operation (addition involving words of maximal size). It follows that all faster operations (processor dependent operators handling smaller word sizes, Boolean operations, etc.) are slowed down because they are given the same length of time as the longest processor dependent operation.

A general weakness of pipeline architectures is pipeline drains due to conditional branching. A pipeline may have dummy time intervals when no processing is performed if, as a result of a conditional test, the program switches to another instruction sequence that was not already being processed by the pipeline stages. The problem of conditional branch may be solved if one uses the solution adopted in the IBM 360/91,⁸ based on the use of two pipelines. For this case true (incremental) and false (specified by a jump address) program sequences may be computed by two independent pipelines, main and subsidiary, where the subsidiary pipeline is switched into operation only during a conditional branch instruction, that is, its instruction memory replicates a portion of the program. If the conditional branch is made to the instruction sequence computed in the subsidiary pipeline, it transfers all computational results necessary for further computation to the main pipeline and stops.

C. INTENT

Since dedication is the main shortcoming of pipeline computers, a major thrust of an architectural research in pipelines has to be directed at broadening their applicability. Ideally, a pipeline system must be able to compute any program as cost-effectively as a general purpose computer.

Let us now show some architectural properties that may expand the applicability of pipeline systems, speed up computations, and simplify programming.

1. *Universality of a pipeline stage:* Each pipeline stage must be capable of executing any operation encountered in the instruction.
2. *Variable number of stages in the pipeline activated by different program instructions:* Each instruction must propagate through the number of consecutive stages in the pipeline that matches the number of operations it realizes. Consequently two instructions that realize a and b operations respectively must activate a and b consecutive stages in the same pipeline.
3. *Connection of each pipeline stage with main memory:* Inasmuch as the instructions may activate a variable number of stages in the pipeline, each pipeline stage may function as the end stage of some instruction. Therefore each stage must be capable of sending its computational results not only to the next stage but also to the main memory.

Properties 1, 2, and 3 together allow one to obtain pipelines capable of executing any sequence of operations in a program with no reconfiguration of pipeline stages.

This will eliminate all problems associated with pipeline reconfiguration such as time overhead for reconfiguration; conflict resolution during bypassing; programming difficulties associated with regrouping instructions into separate blocks executed by different pipeline configurations; synchronization of these blocks and sequencing them; etc.

4. *Fast access of one pipeline stage to temporary results computed by other pipeline stages:* In order to have rapid access to temporary results each pipeline stage must be equipped with a register memory that stores temporary results needed by the stage. Data exchanges need to be developed that will allow each pipeline stage to send its computational results not only to the next stage but also to the register memory of any other pipeline stage.
5. *Update of the local memory of a pipeline stage with new data, when it fetches the operands it needs:* Each local memory assigned to a pipeline stage must be capable of performing two concurrent actions: fetch operands needed by the stage and receive new data words from the main memory. This requires partitioning the local memory M into two levels $1-M$ and $2-M$ and separately connecting each level with the processor of the stage. If $1-M$ fetches operands for the stage, the second memory, $2-M$, may receive data words from the main memory. Subsequent fetches of operands from $2-M$ are accomplished via changes in operand addresses. A local memory M of a pipeline stage may in general be partitioned into r submemories, $1-M, \dots, r-M$. Each memory $i-M$ may then update its contents during the period $T_{UP} = (r - 1)T_M$, where T_M is the time when each $i-M$ fetches operands needed for the stage. By increasing r , one increases T_{UP} . One may therefore connect each $i-M$ to a slower memory storing initial data and still cause no degradation in pipeline's performance.

Architectural Properties 4 and 5 will solve the problem of pipeline degradation caused by the absence in a stage of tem-

porary results or initial data words that it needs for uninterrupted performance. Programmers will then be able to work with both the registers and local memories assigned to a stage just as they work with these memories in conventional computers.

6. *Variable time intervals for operations assigned to a stage.* Each stage must be capable of executing its operation during the minimal time. To implement this feature, the processor of a stage has to reconfigure into the word size that matches the maximal word size of an operand or computed result, and the control unit in the stage has to generate a variable time interval that depends on the word size for processor dependent operation and that assumes a minimal permanent duration for processor independent operations. For example, if the i th instruction in a pipeline stage adds 64-bit words, its processor has to reconfigure into a 64-bit size and the control unit has to generate the minimal time required for a 64-bit addition. If the next instruction subtracts 16-bit words in the same stage, the processor assumes a 16-bit word size and the control unit generates the time for a 16-bit subtraction.

The performance gain obtained from the variable time interval property is due to the ability of the pipeline to work at a variable rate and to fan out results much faster if it is filled with short operations.

A pipeline system that implements Architectural Properties 1 through 6 has a widely expanded area of applicability compared to existing systems, requires no special programming, and speeds up pipelined computations.

Such a novel pipeline organization may be created by assembling it from Dynamic Computer Groups (DC-groups) discussed in Kartashev and Kartashev.^{9,10} The pipeline system assembled from DC groups was called a universal dynamic pipeline architecture or simply dynamic pipeline. It was briefly discussed in Kartashev et al. (1979)¹¹ as an illustration of the adaptation properties shown by dynamic architectures in general.

This paper describes the generic concepts of a dynamic pipeline architecture and introduces the architectural organizations that allow the implementation of Properties 1 through 6.

D. DYNAMIC PIPELINE: BASIC CONCEPTS

The dynamic pipeline is assembled from $F + 1$ stages C_0, C_1, \dots, C_F , where each stage is implemented as a single DC group (Fig. 1). The initial stage C_0 stores instructions in memory M_0 and fetches them to processor P_0 . Its size matches that of one instruction. Each pipeline stage C_i ($i = 1, \dots, F$) has memory M_i for storing initial data words and addresses, processor P_i and general register memory RM_i that stores temporary results required by processor P_i . These are either computed by stage C_i or by other stages. Since each C_i is a single DC group, the word size of its processor varies in h -bit

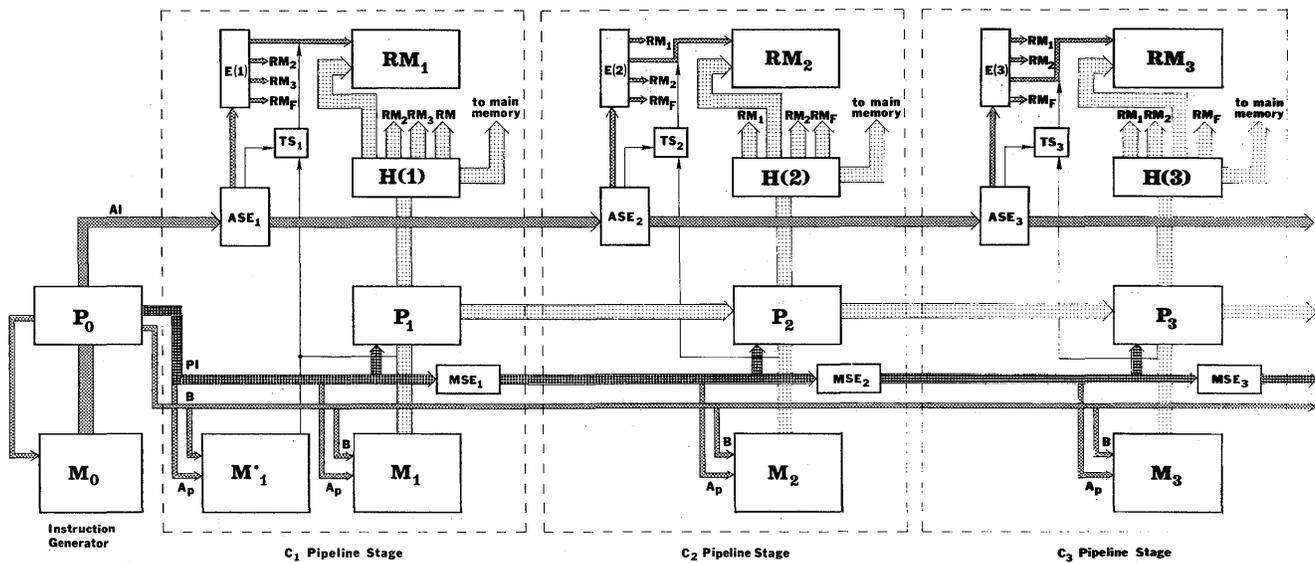


Figure 1—Hardware diagram of a dynamic pipeline

increments from h to $h \cdot n$ bits. For example, for $h = 16$ and $n = 4$, each processor P_i may assume one of the following word sizes 16, 32, 48, 64 bits, i.e., it changes in 16-bit increments.

Each instruction fetched from M_0 to P_0 of the initial stage C_0 includes two portions: the pipeline instruction, PI , and the address instruction, AI . The instruction PI propagates through a bus made of $(F - 1)$ memory connecting elements, MSE , and activates one operation in each pipeline stage. Concurrently instruction AI propagates through the bus made of F address connecting elements, ASE , so that each ASE_i may generate the address of any register memory RM that should receive the temporary result computed in stage C_i .

Each MSE_i separates two pipeline stages C_i and C_{i+1} and may assume two modes of transfer: *right* or *no transfer*. For right transfer, MSE_i propagates the pipelined instruction PI to the next stage to the right, C_{i+1} , with a delay of one interval. For no transfer, MSE_i blocks further instruction propagation in the pipeline. The MSE element is a universal module UM , that is used as the unique building block of a DC group.^{12,9,10}

Each ASE element may modify the addresses stored in the instruction AI . It includes a universal module UM equipped with the memory AM that stores constants needed for address modification. Each time a pipeline stage C_i receives PI instruction, the address element with the same position i , ASE_i , receives the instruction AI shifted from ASE_{i-1} . The ASE_i is synchronized by stage C_i and stores the AI instruction during the time that C_i executes an operation. If the result of this operation is a temporary result for another stage C_j , to use in the future it is written to its register memory RM_j . To do this, ASE_i generates the address for RM_j so that the computational result obtained by C_i may be sent not only to the next stage C_{i+1} but to any other stage that may use it in the future. The logical circuits $E(i)$ and $H(i)$ respectively, broadcast the address and the temporary result obtained in stage C_i to any register memory RM .

E. ADAPTATION CAPABILITIES OF A DYNAMIC PIPELINE

Let us now discuss how a dynamic pipeline adapts to an executing algorithm. This is accomplished by several adaptation codes stored in the PI instruction.

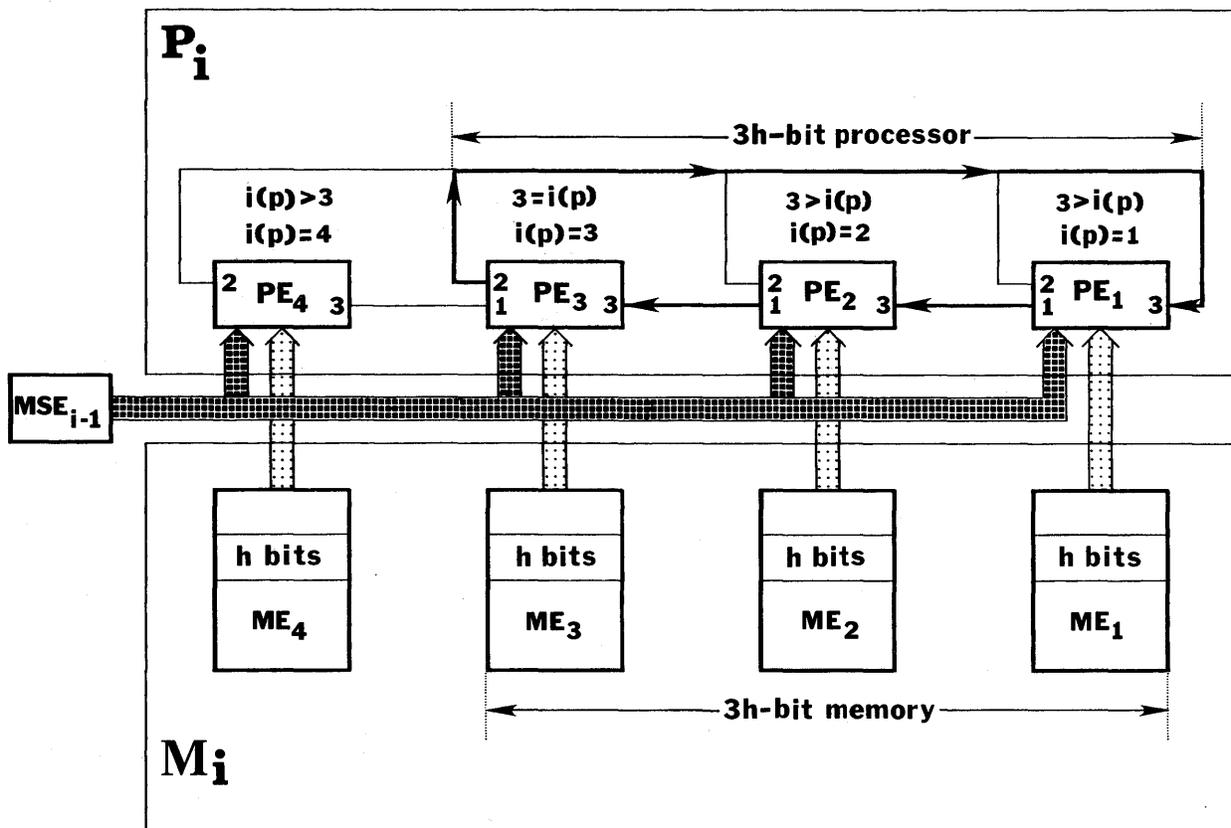
1. Adaptation to a Sequence of Operations

Since each PI must be capable of executing any sequence of operations, it has to store the code of an operation sequence, SOP . When a SOP propagates through the pipeline stages, however, it will activate the same operation in each stage. In order that each stage, C_i , execute an individual operation, its processor P_i should store a position code d_i that is the binary value of its position within the pipeline. These two codes, SOP and d_i , achieve the selective activation of the operation assigned to stage C_i by the instruction PI .

Example. Suppose instruction PI has been assigned the sequence of operations $(+, -, \wedge, \times, +)$ encoded as SOP_{25} . This instruction is executed in the pipeline stages C_1, C_2, \dots , and C_5 . As the instruction passes through the pipeline it encounters position code $d = 001$ in the first stage, C_1 , and together $d = 001$ and SOP_{25} activate addition; in the second stage, C_2 , $d = 010$ and SOP_{25} activate subtraction; etc.

2. Dynamic Variation of a Pipeline's Length

Since each PI instruction may be assigned a variable number of consecutive operations, a dynamic pipeline must change the number of pipeline stages activated. This means that if instruction PI realizes a sequence of w operations, it

Figure 2—Reconfiguration of the P_i processor

must propagate only through w consecutive pipeline stages C_1, C_2, \dots, C_w and obtain its computational result in stage C_w . Such a variation in the number of pipeline stages activated by the PI instruction is accomplished by another adaptation code, w , stored in PI that shows the number of operations that PI realizes. When the PI instruction propagates through each connecting element MSE_i , containing position code d_i , w is compared with d_i . If $d_i < w$, MSE_i propagates this instruction to the next stage with a delay of one time interval. If $d_i \geq w$, MSE_i stops further instruction execution and sends this PI and its result to the next C_{w+1} . In the next time interval, the computational result received by C_{w+1} is sent to the main memory. The two adaptations mentioned above allow a dynamic pipeline to execute quite different sequences of operations one after the other without requiring reconfiguration between pipeline stages. Therefore eliminated are (a) the time overheads introduced by pipeline reconfiguration, when no processing can be performed, and (b) the need for special programming to restructure programs into tasks that can be assigned to single pipeline configuration.

Example. Suppose a dynamic pipeline executes consecutive instructions PI_1, PI_2 , etc. Instruction PI_1 calls for three operations (+, -, \times) and stores code $w = 3$. PI_1 's computational result is obtained in C_3 and is transferred first to C_4 and then to the main memory. The next instruction, PI_2 , realizes five operations (-, +, \wedge , +, \times) and stores $w = 5$. Thus its result computed by C_5 , is sent to the main memory via C_6 , etc.

3. Variable Time Intervals for a Pipeline Stage

All operations executed in the processor of a pipeline stage are divided into two groups: (1) The operations that include the carry propagation microoperation (add, subtract, etc.). The time for such operations depends on the word size. (2) The operations that do not include the carry propagation microoperation, (addition module 2, logical addition, etc.). The time of such operations does not depend on the word size and for most operations can be very short.

To speed up operations of the first group, the PI instruction in each stage C_i must reconfigure its processor into the minimal word size and cause the control unit to generate the minimal time interval compatible with the selected processor size. Speeding up operations of the second group requires that the control unit in a stage generate the minimal time interval of permanent duration. Consider now speed up of the first group of operations.

Reconfiguration of the processor

This can be easily accomplished inasmuch as each pipeline stage is realized as one DC group and its processor P_i contains n processor elements PE_1, PE_2, \dots, PE_n (Fig. 2). Each PE handles h -bits. Selection of the minimal processor size for P_i can be performed with adaptation code k , stored in instruc-

tion PI. The code k shows the number of PE elements that are to be connected into the $k \cdot h$ -bit processor of the stage. For instance, for $k = 1$, the processor is formed from a single PE and handles h -bit words; for $k = 2$, it is formed from two PE's and handles $2h$ -bit words, etc.

Such reconfiguration of the resource into $k \cdot h$ -bit processors may be accomplished if the carry signal α produced by the local adder in each PE_{*i*}, is routed either to carry-out pin 1 connected to carry-in pin 3 of the next more significant PE or to overflow pin 2 if it is the overflow in a $k \cdot h$ -bit adder. All 2-pins are connected together with pin 3 of the least significant PE so the α overflow is routed as the end-around carry to the LSB of $k \cdot h$ -bit adder.

Selective activation of the α signal to pins 1 or 2 of each PE is accomplished by comparison of two codes stored in each PE: the processor position code $i(p)$ which shows the relative position of this PE inside a $k \cdot h$ -bit processor and adaptation code k brought to the PE by the PI instruction.

If $k > i(p)$, the α signal is routed to pin 1.

If $k = i(p)$, it is routed to pin 2.

If $k < i(p)$, the α signal is routed to no pin.

Example. Let the processor resource be PE₁ through PE₄ storing the following processor position codes: in PE₁, $i(p) = 1$, in PE₂, $i(p) = 2$, etc. (Fig. 2). Suppose that the PI instruction requires that a $3 \cdot h$ -bit processor be formed from the available PE's. It includes adaptation code $k = 3$. When PI is fetched to each PE of the stage, it compares code k with the local processor position code $i(p)$. In PE₁, $i(p) < k$ ($1 < 3$) and signal α is routed to pin 1. Similarly in PE₂, $i(p) < k$ ($2 < 3$) routes α to pin 1. In PE₃, $i(p) = k$ ($3 = 3$) and α is fanned out to overflow pin 2 connected with the LSB of the $3h$ -bit adder. In PE₄, $i(p) > k$ ($4 > 3$) and α is routed to no pin. A correct end-around-carry path has therefore been formed.

Organization of a minimal time of operation

Since a DC group uses the modular control organization,¹² it allows one to obtain a variable time of operation in a pipeline stage. For the pipelined mode of operation, however, the modular control organization has to be modified. Indeed, when a DC group works as a multicomputer system, each instruction is entirely executed in one computer based on the interaction of two sequencers, CAD-I and CAD-M, where CAD-I activates a sequence of operations and CAD-M specifies the time of each operation.^{9, 10, 12}

For instance, if an instruction activates the sequence $(D + B - H) < F$, the first state of CAD-I fetches the instruction, the second state executes $D + B$, the third state performs $(D + B) - H$, and the fourth state executes the comparison $(D + B - H) < F$. All of these operations are distributed among separate stages, however, in a pipeline so that stage C_0 fetches the instruction, stage C_1 executes $D + B$, stage C_2 executes $(D + B) - H$, and C_3 stage compares $(D + B - H)$ and F . Each stage therefore executes only one operation out of the whole sequence. Furthermore, two stages C_i and C_{i+1} may concurrently execute two operations activated by instructions PI_j and PI_{j+1} respectively. The CAD-I sequencer of each stage must therefore acquire any of its

states for the time interval it keeps instruction PI. At the next time interval a new PI instruction will be written to this stage, and the CAD-I must establish another state that corresponds to that instruction.

It then follows that for noniterative operations (addition, subtraction, Boolean) the CAD-I functions as a decoder, CAD-ID, and for iterative operations (multiplication, division, etc.) it works as a sequencer, CAD-IS (Fig. 3). The functioning of the CAD-I and CAD-M is controlled by several codes either stored in PE or brought by instruction PI, and they may lead to changing the time, T , assigned to an operation executed in a stage. Let us see how this may be accomplished.

The time of operation, T is $T = t_o \cdot b$, where t_o is the time of h -bit addition in one PE, and b depends on the opcode SOP and processor code p specifying the time of processor dependent operations. Since t_o is the minimal time interval, it is assumed that it is the period of a synchronization sequence. Then code p shows the number of clock periods, t_o , that a processor dependent operation needs to execute in a $k \cdot h$ -bit processor.

For linear carry propagation, $p = k$, for non-linear carry propagation with CLA circuits, $p < k$. If CLA circuits are used in a $k \cdot h$ -bit processor, two codes, k and p , are required to organize the minimal time of a processor-dependent operation: k reconfigures the processor into the minimal size needed, and p generates a minimal time interval. To reduce the bit size of the PI instruction, it will be taken as accepted that a set of all possible p codes is stored in a register Y_1 of each PE, so that the k code brought by the PI instruction selects the necessary p code (Fig. 3).

The variable time T for a processor dependent operation is generated as follows: $T = p \cdot t_o$, i.e., $b = p$. If $p > 1$, it initiates the CAD-M into passing a loop containing p states. Each state lasts one t_o . During this time, the decoder CAD-ID maintains microcommand MIC which activates operation in the processor. When CAD-M completes its loop it issues a completion signal, CS. This terminates the microcommand. If no CLA circuits are used, then $T = k \cdot t_o$, and CAD-M passes a loop containing k states. Therefore by changing the adaptation code k stored in the PI instruction, one changes the time T of a processor dependent operation executed in a stage.

If the operation is independent of the processor, then it is activated via decoder CAD-ID only, i.e., CAD-M is not initiated and the operation takes the time $T = t_o$ of one small clock period, i.e., $b = 1$. If stage C_i executes iterative operation (multiplication, division), CAD-IS sequencer executes a sequence containing several states. If a state in this sequence has to last $p \cdot t_o$ or $k \cdot t_o$, then the CAD-IS initiates CAD-M and performs transition to the next state under the completion signal CS issued by the CAD-M.

Example. Let the processor of the C_i stage reconfigure its size in 16-bit increments from 16 to 64 bits. Assume that the $k \cdot h$ -bit adder uses 4-bit CLA circuits. This gives the time $t_o = 12t_d$ of one clock period (16-bit addition), where t_d is one gate delay. Suppose that the C_i stage executes three consecutive instructions PI_1 , PI_2 , PI_3 , which activate respectively 64 bit addition (PI_1) 16-bit subtraction (PI_2) and 32-bit logical multiplication (PI_3). The PI instruction stores adaptation code $k = 4$, ($64/16 = 4$) that selects processor code $p = 3$ (more de-

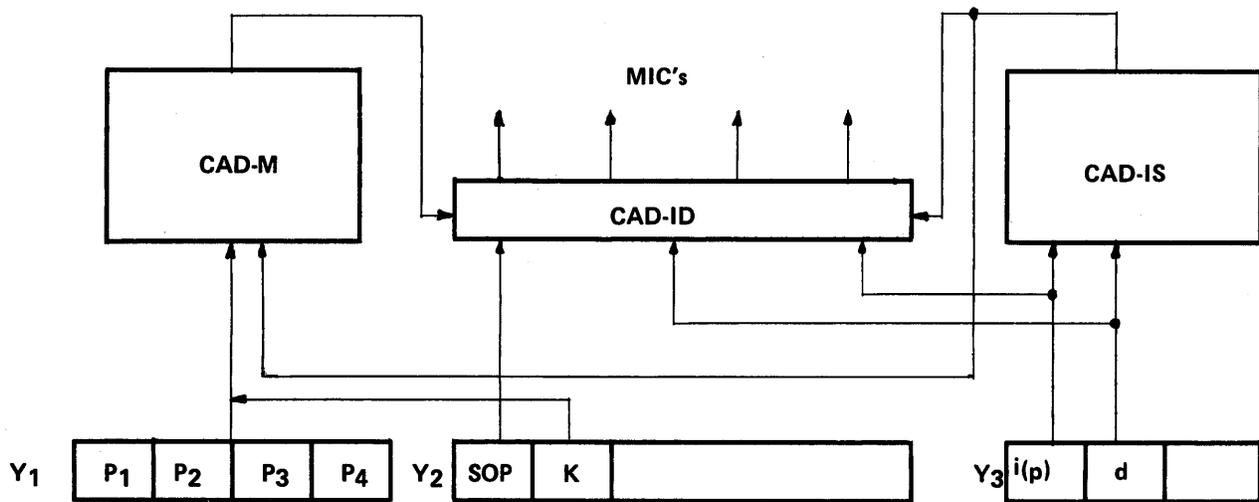


Figure 3—Control unit in pipeline stage C_i

tails on finding p are given in Kartashev and Kartashev (1979)¹⁰ and Kartashev et al. (1979)¹¹; the p code activates CAD-M into passing a loop containing three states so the 64-bit addition is executed during the time $T = 3t_o$. The next PI_2 instruction stores $k = 1$ ($16/16 = 1$) that selects $p = 1$ and the 16-bit subtraction takes the time $T = t_o$. The PI_3 instruction activates logical multiplication independent of the processor size. The instruction stores no k and the operation takes the time $T = t_o$.

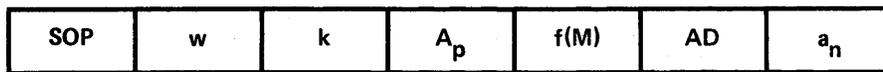
F. ORGANIZATION OF A PIPELINE STAGE

Generally, computation in a stage C_i performed by PI instruction is similar to that of a 3-addressed instruction in a conventional computer. Symbolically it can be presented as *, $A1, A2 \rightarrow A3$, where * stands for an operation assigned to stage C_i ; $A1, A2, A3$ are the addresses respectively of first and second operands and computational result. Beginning second stage, i.e., for $i > 1$ the first operand is transferred to

stage C_i , from the preceding stage C_{i-1} , the second operand is fetched via address $A2$ from data memory M_i if it is initial data word, or from register memory RM_i , if it is a temporary result. For this case M_i stores address of the second operand which then accesses the RM_i . The computational result of stage C_i may be sent in two directions: to the next stage C_{i+1} and to the register memory RM_i of some other stage C_j . For this case the connecting element ASE_i generates the $A3$ address of RM_j .

For the stage C_1 ($i = 1$), both operands must be fetched from memory since computation of the PI instruction begins since C_1 . Further, since RM_1 memory is small, it cannot store all first operands for C_1 stage, otherwise it will be required that its size be equal to that of data memory M_1 . Thus the P_1 processor in stage C_1 (Fig. 1) must be connected in parallel with two data memories M_1 , and M_1^* ; so that M_1 stores the first operand accessed by address $A1$. M_1^* stores either the second operand or its address, if the second operand as a temporary result is stored in RM_1 .

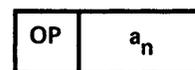
Such organization of computations in one stage of a dynam-



Basic instruction



PI instruction



A1 instruction

Figure 4—Organization of pipeline stage C_i

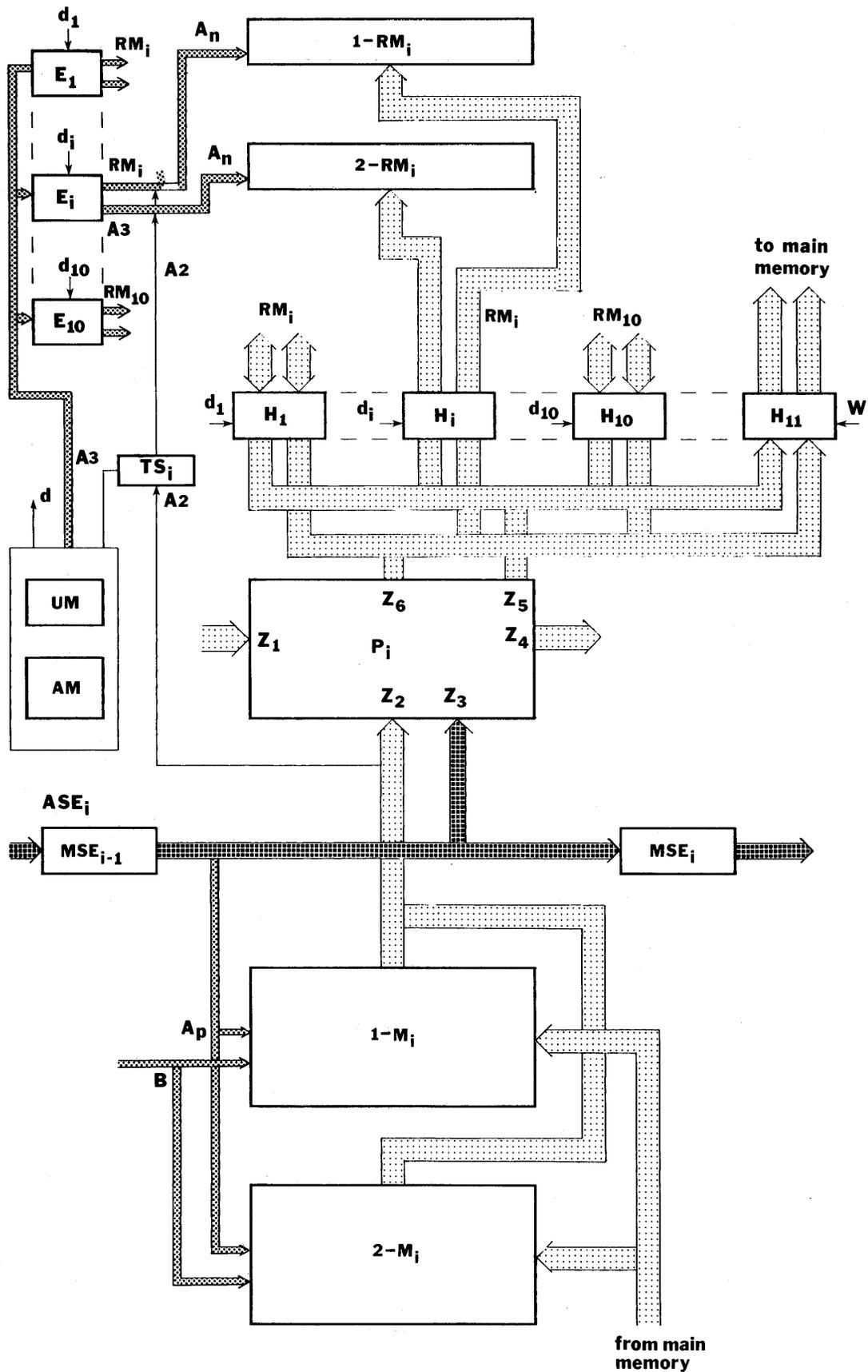


Figure 5—Registers in P_i used for pipelined mode of operation

ic pipeline leads to the following hardware diagram of each C_i (C_i in addition to all hardware units contained in any stage C_i must contain an additional data memory M^*). One pipeline stage C_i contains the processor P_i (Fig. 4), data memory M_i for storing initial data words, register memory RM_i for storing temporary results, the MSE_i connecting element for propagating PI instruction, the ASE_i connecting element for propagating AI instruction, the H and E logical circuits for broadcasting respectively temporary results and their addresses to register memories RM , and the logical circuit TS_i which activates the address transfer from M_i to RM_i memory.

The P_i processor is assembled from n h -bit processor elements PE_1, \dots, PE_n and may assume $h, 2h, \dots, n \cdot h$ -bit sizes. To organize pipelined mode of operation, a P_i should contain the following registers (Fig. 5): instruction PI is received to $R5$ register through pins Z_3 ; the first operand computed in the previous stage is received through pins Z_1 to $R3$ register; $R4$ register stores the second operand if it is fetched from data memory M_i through Z_2 pins; $R1$ and $R2$ registers may store alternatively second operand if it is fetched from register memory RM_i (partitioned into two levels) through Z_6 and Z_5 pins respectively. (For the stage C_1 , when fetched from data memory M^* the second operand may be written to $R1$ or $R2$ registers.)

The data memory M_i stores either initial data words for the P_i processor or the addresses $A2$ of second operands provided these are fetched from RM_i . The logical circuit TS_i broadcasts the $A2$ address fetched from M_i to RM_i register memory. It is activated by a special tag bit e stored in the AI instruction that shows which memory M or RM stores the second operand needed by P_i processor. In order to be capable of updating its content without degrading the pipeline performance, each M_i is partitioned into two levels $1-M_i$ and $2-M_i$; each level is connected in parallel with the processor P_i of the same stage and the common main memory of the pipeline. Consequently, when one level ($1-M$ or $2-M$) is used for fetching addresses or operands, another level updates its content by receiving new block of initial words from the main memory. (In stage C_1 , both M_1 and M_1^* memories are partitioned into two levels $1-M_1, 2-M_1, 1-M_1^*, 2-M_1^*$). Thus a dynamic pipeline does not stop due to the absence of initial data words.

To increase the update time during which the memory M_i may update its content, M_i may be partitioned into r levels, $1-M_i, 2-M_i, \dots, r-M_i$. Then if T_M is the time when one level is working in the pipeline, it may update its content during the time $(r-1)T_M$. A switch from one level to another is performed by generating a new base address B .

The effective address of the M_i memory is $E = B + A_p$, where the base address B is sent continuously from initial stage C_0 to all data memories of the pipeline, A_p is a relative address stored in PI instruction. The memory M_i that receives $E = B + A_p$ either fetches the second operand directly ($E = A2$) or indirectly, i.e., E fetches $A2$ which then accesses second operand from RM .

The RM_i register memory stores the temporary results for P_i that may be computed by a local P_i or by any other processor P_j in the pipeline. During the same time interval, one RM_i may be accessed twice: when PI instruction writes temporary result into this RM and the next PI fetches the second operand from the same RM . It then follows that the register

memory in a stage must be also partitioned into two levels $1-RM_i$ and $2-RM_i$ to perform these accesses in parallel.

In stage C_i data exchanges between the P_i processor and register memories, RM_1, RM_2, \dots, RM_F belonging to stages C_1, C_2, \dots, C_F respectively are performed via logical circuits H_1, \dots, H_F respectively so that H_j connects P_i with RM_j ($j = 1, \dots, F$). The address $A3$ of a memory cell in RM_j that has to store a temporary result computed by the P_i processor is sent to RM_j as follows. The ASE_i element which stores the AI instruction during the time P_i processor executes its PI portion generates position code d_j and destination address $A3$ in RM_j . This address is sent concurrently to F logical circuits E_1, \dots, E_F where each E_j ($j = 1, \dots, F$) broadcasts the destination address to RM_j . Selection of E_j is performed with position code d_j . Thus the same position code d_j selects E_j and H_j for broadcasting both address and data word to RM_j .

Example. Let PI instruction compute the formula $(A + B - D)^2 - K$ and the temporary result $(A + B - D)$ be sent to register memory RM_1 of stage C_1 into a cell, say #57, because stage C_1 will use this result for another instruction. The PI instruction executes $A + B$ in stage C_1 and $A + B - D$ in stage C_2 . The $A + B - D$ result is sent in two directions: to stage C_3 which performs $(A + B - D)^2$ and to RM_1 of stage C_1 . To send a temporary result to RM_1 , the ASE_2 connecting element generates position code $d = 1$, (i.e., signal d_1 is activated) and destination address $A3 = 57$. Signal d_1 activates E_1 and H_1 circuits in stage C_2 which connect C_2 with RM_1 via address and data bus respectively. Thus RM_1 receives both the address #57 transferred via E_1 and the result $(A + B - D)$ transferred via H_1 .

The final result of each PI instruction may be written to the main memory of the pipeline system. This will require that the PI instruction in addition to relative address A_p of M_i data memories store the destination address of the main memory. To eliminate storage of this address from the PI instruction, the following organization is proposed. A PI that implements sequence of w operations obtains the final result in stage C_w . Therefore the next stage C_{w+1} has an empty cell in address $B + A_p$ of M_{w+1} accessed by this PI. It then follows that a programmer may store in the $B + A_p$ address of M_{w+1} the destination address of main memory that has to receive the final result of this PI. When the PI instruction is transferred to C_{w+1} , this address is fetched from M_{w+1} to the $R4$ register of P_{w+1} processor. Thus P_{w+1} receives both the final result into its $R3$ register and the main memory address of this result into the $R4$ register. Therefore P_{w+1} may transfer both data word and its address to the main memory through pins Z_6 and Z_5 and logical circuit H_{F+1} (in Fig. 3, $H_{F+1} = H_{11}$, since $F = 10$). This circuit is activated by equality signal w produced in the last connecting element MSE_w which propagates the PI instruction to stage C_{w+1} . The last MSE_w which propagates PI instruction is recognized via equality $d = w$ between its position code d_w and the code w stored in PI which shows how many stages execute PI.

G. ADDRESSING PROCEDURES FOR TEMPORARY RESULTS

Each instruction PI realizing w operations may obtain w temporary results. To preserve flexibility of programming a pro-

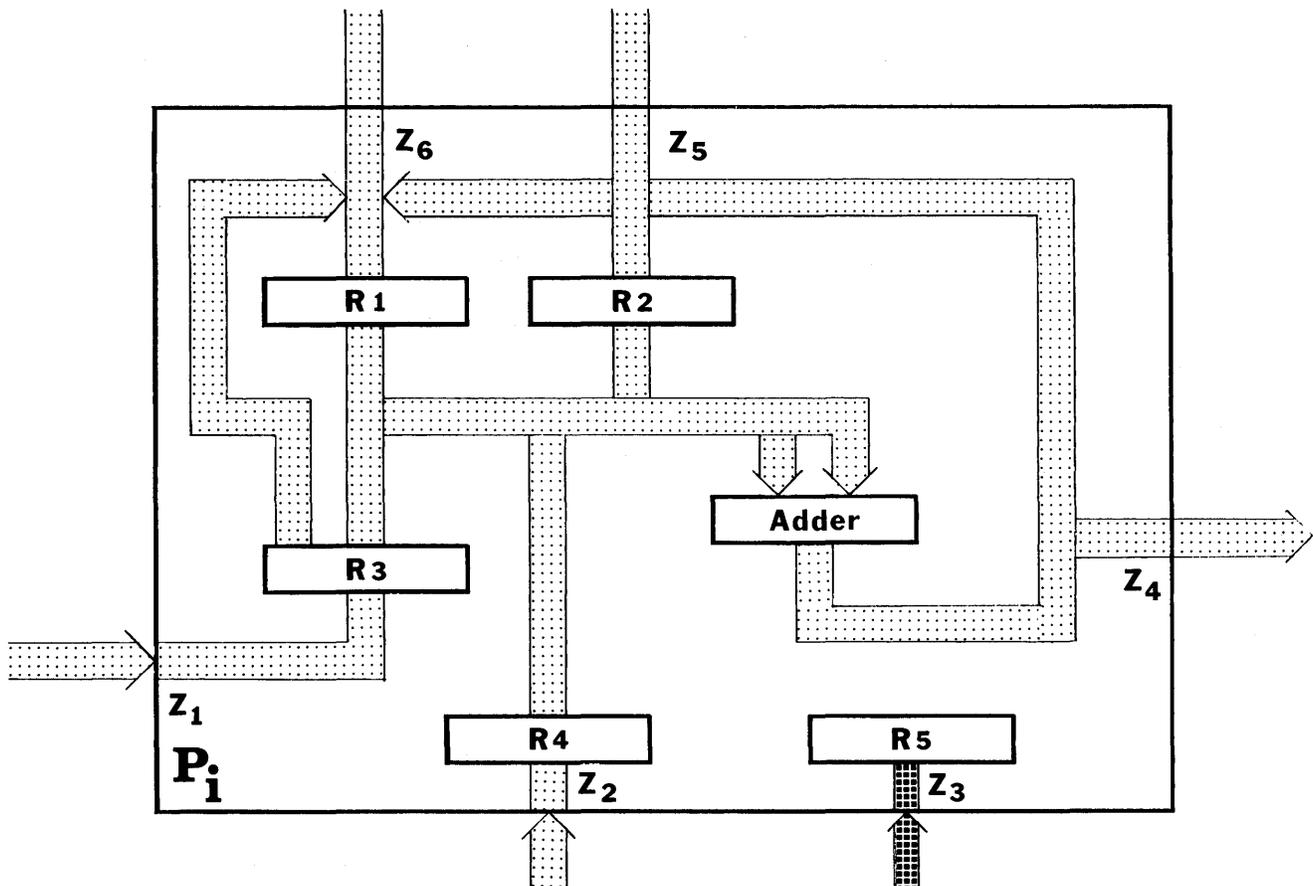


Figure 6—Address elements ASE

grammer should be allowed to send these results to any register memory RM. Since a separate destination address A3 and position code d_j of RM_j may be required for the temporary result computed in each stage C_i , to send all w temporary results of instruction PI to register memories will require an addresses field in $d \times w \times A3$ bits. For instance, in a pipeline with 10 stages (code $d = \log_2 10 = 4$ bits) and with a register memory of 256 cells per stage ($A3 = 8$ bits), the instruction PI containing 10 operations (code $w = 4$ bits) will require an address field of $d \times w \times A3 = 4 \times 4 \times 8 = 128$ bits.

Let us introduce one organization that eliminates the storage of these addresses from the instruction fields (AI and PI). Instead each instruction AI stores only one 8-bit address a_n , but the capability of each stage to generate new meanings of A3 and d_j is preserved.

Each stage C_i of a dynamic pipeline is provided with the address element ASE_i . The ASE_i stores AI instruction during the time stage C_i stores the PI instruction; i.e., the propagation of the AI instruction through the connecting element ASE is synchronized by the propagation of the PI instruction through pipeline stage C . Each ASE consists of a microprocessor UM and memory AM (Fig. 6) partitioned into two levels 1-AM and 2-AM. One level works with the pipeline and the second updates its own contents. A switch from one level to another is accomplished by changing the base address D

produced by stage C_0 . This results in a new effective address $E = D + a_n$.

The relative address a_n is stored in the AI instruction and accesses the cell of AM memory (1-AM or 2-AM) that stores the following information.

1. The position code d_j of register memory RM_j that is to receive the temporary result.
2. An address modification constant AMC selected by the programmer as follows: If A3 is the address of the temporary result then $A3 = a_n \pm AMC$.
3. An indexing increment $f(RM)$ of register memory RM for the case that the PI instruction in stage C_i performs vector computations and C_i computes an array of temporary results that should be sent to several addresses A3 of RM_j differing from each other by an indexing increment $f(RM)$. Namely, each next address $A3 = A3^* + f(RM)$, where $A3^*$ is the current address.
4. Two tag bits $e(A2)$ and $e(A3)$. Bit $e(A2)$ shows whether memory RM_i or M_i stores the second operand needed by stage C_i ; i.e., it activates broadcast of the A2 address fetched from M_i through circuit TS_i . Bit $e(A3)$ shows whether or not stage C_i should generate addresses d_j and A3 for the temporary result.

When the instruction AI is received by the ASE element it is written to UM, and the address a_n stored in the instruction field fetches one cell of memory AM. As a result, UM generates d_j and destination address A3 as $A3 = a_n \pm AMC$ for scalar computations or $A3 = A3^* + f(RM)$ for vector computations. By writing different values of CMA, d , e , and $f(RM)$ to the same cell a_n of each memory AM accessed by instruction AI one accomplishes variation in the addresses d_j and A3 generated by each ASE element. The instruction PI implementing w operations may therefore send all its temporary results to w different addresses d_j and A3.

Since the minimal operation time in every stage is the time of 16-bit addition, whereas the A3 address modification procedure in each ASE takes the time of an 8-bit addition (inasmuch as each A3 is an 8-bit address), the access time of any AM memory should not exceed the time of an 8-bit addition.

For vector computations executed over an array of data words having array dimension AD, the C_o computer blocks further instruction fetch while the same instruction propagates through the pipeline AD times to perform the vector computation. Each time its PI portion begins to travel through the consecutive pipeline stages, the P_o processor generates a new meaning for the relative address $A_p + f(M)$, where $f(M)$ is an indexing increment for the data memories M . This allows memory to be accessed by a new effective address $E = B_o + A_p + f(M)$. The AI portion likewise obtains new meaning of address A3 in the ASE connecting element as $A3 = A3^* + f(RM)$, where $f(RM)$ is an indexing increment for memory RM.

Example. Let instruction PI, computed in stages C_1 through C_4 , send computational results of stages C_2 and C_4 to RM_2 (address A3 = 127) and RM_1 (address A3 = 98) respectively. The remaining results are not sent to register memories. Let the AI portion of this instruction store relative address $a_n = 71$. Then in ASE_1 , cell #71 of memory AM stores $e(A3) = 0$, i.e., ASE_1 generates neither A3 nor d_j . In ASE_2 , cell #71 stores $AMC = 56$, $d_j = 3$, $e(A3) = 1$, because $A3 = 127$ and $a_n = 71$, give $AMC = A3 - a_n = 56$. ASE_3 has $e(A3) = 0$ in cell #71, and no A3 and no d_j are generated. ASE_4 has cell #71 of its AM storing $AMC = 27$ and $d_j = 1$, because $AMC = A3 - a_n = 98 - 71 = 27$.

H. INSTRUCTION FORMATS

In a dynamic pipeline there are two instruction formats: basic and auxiliary. A basic instruction performs computations, and an auxiliary instruction brings to the pipeline new base addresses B_o , D , and some other values. Every instruction (basic or auxiliary) occupies one cell of the memory M_o . The basic instruction is partitioned into three portions PI, AI, and VI, where PI propagates through the MSE bus; AI propagates through the ASE bus; VI does not propagate in the pipeline but is stored in processor P_o . It is used for vector computations.

Instruction PI stores adaptation codes (SOP, w , k) and relative address of data memories, M_i . Instruction AI stores a two-bit opcode OP and relative address a_n of AM memories assigned to ASE elements; Instruction VI stores array dimension AD and indexing increment $f(M)$ for memories M_i . We

will need to find the bit size of the basic instruction format. If the dynamic pipeline realizes an instruction set containing not more than 256 instructions, $SOP = \log_2 256 = 8$ bits. The code $w = \log_2 F$ where F is the total number of stages; for $w = 4$, one obtains pipelines with up to 16 stages. The code k is determined by the number n of processor elements PE assembled into a single processor P_i , i.e. $k = \log_2 n$. If the P_i contains four 16-bit PE's, $k = \log_2 4 = 2$. Such a P_i may assume word sizes ranging from 16 to 64 bits in 16 bit increments (16, 32, 48, 64). The address A_p is the relative address in one page of memory M_i . If $A_p = 12$ bits, this will allow the use of pages containing 2,048 cells. The indexing increment $f(M) = 7$ bits; and the array dimension $AD = 12$ bits. This will allow one to compute with one instruction a data array occupying an entire page. The address a_n is an 8-bit address that allows computation of 8-bit addresses in RM memories. The two bit code OP is not included into the basic instruction since it is formed by opcode SOP for vector computations. Thus the overall bit size BI of a Basic instruction is: $BI = 8 + 4 + 2 + 12 + 7 + 12 + 8 = 53$ bits. Of those bits $PI = 8 + 4 + 2 + 12 = 26$ bits and $AI = 10$ bits.

In spite of their insignificant bit sizes, instructions PI and AI bring to each stage all the information about operands and operations needed to accomplish effective pipeline adaptations to the algorithm executing and to perform fast transmission of temporary results computed by pipeline stages without degrading pipeline performance.

I. SYNCHRONIZATION OF COMPUTATIONS IN A DYNAMIC PIPELINE

At each time interval pipeline stage C_i performs two phases of the computation concurrently: (a) basic phase, aimed at execution of the operation provided by current PI_i instruction; and (b) preparatory phase, aimed at preparing operands for the next PI_{i+1} instruction in the same pipeline stage.

Let us consider the actions executed in pipeline stage C_i by basic and preparatory phases respectively.

1. Basic Phase

This phase executes the operation assigned by instruction PI to stage C_i . The PI instruction is stored in register R5 of the P_i processor. It was written there during the clock pulse marking the end of the preceding time interval. The operation is performed over two operands stored in registers of processor P_i . The result of the operation may be sent to any combination of the following destinations:

- P processor of the next stage:* By passing through the Z_i pins of processor P_i , the result is written to the R3 register of the next processor, P_{i+1} . This occurs when processor P_{i+1} completes its basic phase.
- Register memory RM of any pipeline stage:* This transfer is performed through the pair of circuits H and E described above and activated by position code d .
- Main memory of the pipeline:* The transfer is made through the circuit H_{F+1} .

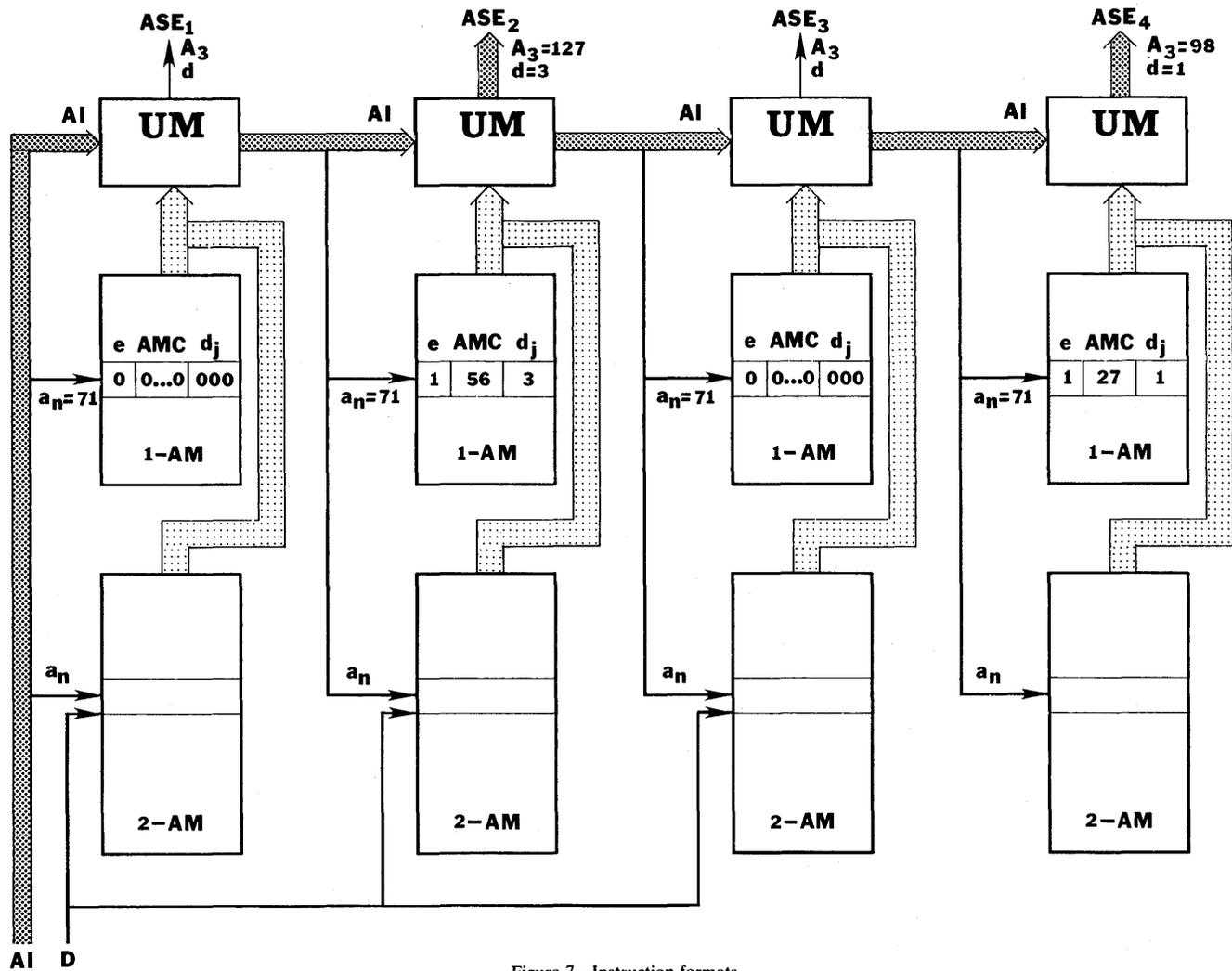


Figure 7—Instruction formats

2. Preparatory Phase

In each pipeline stage, the preparatory phase of the next instruction is executed concurrently with the basic phase of the present instruction. The objective of the preparatory phase is to prepare operands for the next PI instruction to be executed in the same stage. Let us establish the actions executed in pipeline stage C_i during the preparatory phase:

- a. Fetch of an operand from data memory M or register memory RM attached to the stage. An operand needed for stage C_i may be fetched either from data memory M or register memory RM . To this end memory M is accessed by the effective address $B + A_p$ where A_p is the relative address stored in instruction PI in the beginning of the preparatory phase. If the operand is stored in M , it is fetched to the $R4$ register by using the address $B + A_p$ (Fig. 5). If it is stored in RM , then the $B + A_p$ address of M fetches the address $A2$ and address $A2$ accesses memory RM via logic TS_i . Writing of the operand to a register of processor P_i ($R4, R1, R2$) is

performed at the clock pulse that signals the end of the preparatory phase.

- b. Writing the next PI instruction. Every PI instruction is written to register $R5$ of stage P_i from the left connecting element MSE_{i-1} at the last clock pulse of the preparatory phase.

- c. Writing the result of the operation executed in stage C_{i-1} . Processor P_i receives this operand to its register $R3$. Since the actions (a) to (c) prepare all the information necessary for the next basic phase, the preparatory phase ends on completion of these actions.

3. Synchronization of Basic and Preparatory Phases

Since each pipeline stage is equipped with the modular control organization, it may generate the minimal operation times for all the operations it executes. However, implementation of a variable operation time in a stage requires

synchronization of the preparatory and basic phases of two consecutive instructions PI_j and PI_{j+1} . Any next PI_{j+1} and its two operands can be written to processor registers of a pipeline stage only at the moment of time when the present instruction PI_j has ended its basic phase in this stage. Therefore, for any pipeline stage, the preparatory phase for instruction PI_{j+1} cannot be shorter than the basic phase for the preceding instruction PI_j .

Furthermore, since the preparatory phase in a stage prepares two operands for the next basic phase, its time cannot be shorter than the times of preparing both operands. One operand to be fetched from memories M or RM can be made available during the basic phase of instruction PI_j provided the time of direct access for M or indirect access for RM is shorter than the time of the shortest basic phase in a stage (16-bit addition). This is the basis for selecting the memory access times for M and RM . As for another operand being received from the preceding stage, C_{i-1} , it may be written to stage C_i only when C_{i-1} completes its operation. It then follows that if C_{i-1} executes a shorter operation than C_i , the result from C_{i-1} appears before C_i completes its basic phase. For this case the preparatory phase in C_i coincides with its basic phase since the operands prepared by the preparatory phase may be written to C_i only when it completes its basic phase.

Therefore, for two pipeline stages C_{i-1} and C_i executing shorter and longer operations respectively, the time for the preparatory phase in C_i coincides with the time of the basic phase in the same C_i . If, on the other hand, stage C_{i-1} executes a longer operation than C_i , then the operand from stage C_{i-1} appears after C_i completes its operation. Then the preparatory phase in C_i is determined by the basic phase in C_{i-1} and will last longer than the basic phase in C_i . Hence, for two pipeline stages C_{i-1} and C_i executing longer and shorter operations respectively, the time of the preparatory phase in C_i matches that of the basic phase in C_{i-1} .

It then follows from the above that the preparatory phase for stage C_i either coincides with the basic phase of the same C_i or with the basic phase of the preceding C_{i-1} . The preparatory phase thus introduces no additional delay into the rate of pipeline operation as determined by consecutive durations of its basic phases, provided the memory access times for M and RM do not exceed the time of the shortest basic phase.

Thus we have shown that for a pair of stages C_{i-1} and C_i , the right stage, C_i , must synchronize its left neighbor C_{i-1} if C_i executes a longer operation than C_{i-1} , and the left stage, C_{i-1} , must synchronize C_i if C_{i-1} executes a longer operation than C_i does.

Consider how such synchronization may be accomplished. In each pipeline stage, the completion of its basic phase occurs when the operation sequencer, $CAD-M$ ends its loop and performs a transition to its initial state recognized by completion signal, CS_i . If stage C_i synchronizes C_{i-1} , it must delay transition to the initial state of $CAD-M$ in C_{i-1} until $CAD-M$ in C_i establishes the state that immediately precedes the initial one. At the next clock period both $CAD-M$'s perform concurrent transitions to their initial state, which will mean concurrent with the end of basic phases in C_{i-1} and C_i . Namely, any transition of $CAD-M$ in C_{i-1} to the initial state has to be activated by signal PR_i produced in $CAD-M$ of C_i by the state which precedes the initial one. This means that in the pipeline

each right processor P_i has to be connected with its next left processor P_{i-1} via a one-line connection, sending signal PR_i produced by $CAD-M$ to P_i . This signal activates transition of each left $CAD-M$ to the initial state. This will accomplish synchronization of shorter and longer basic phases executed in C_{i-1} and C_i respectively.

If C_{i-1} and C_i execute longer and shorter basic phases respectively, then the PR_i generated in $CAD-M$ of C_i cannot be used for synchronization, since $CAD-M$ in C_i finished its operation much earlier than that in C_{i-1} . For this case the time when C_{i-1} completes execution and issues an operand for C_i can be determined by the PR_{i-1} signal generated by $CAD-M$ in C_{i-1} . Indeed, at the next clock period C_{i-1} will send the operand to register $R3$ of C_i causing completion of the preparatory phase in C_i . This can be accomplished if each left processor P_{i-1} is connected with its next right neighbor P_i via a one-line connection sending signal PR_{i-1} produced by $CAD-M$ in P_{i-1} . This signal will enable writing of two operands and PI instruction to registers of processor P_i .

Thus the synchronization of phases considered here requires that every pair of neighbors P_{i-1} and P_i be connected with two lines, so that using one line P_i synchronizes P_{i-1} with signal PR_i generated in P_i and using another line P_{i-1} synchronizes P_i with signal PR_{i-1} generated by P_{i-1} .

J. CONCLUSIONS

A dynamic pipeline performs a high degree of architectural adaptation toward executing algorithms. It eliminates the need for pipeline reconfiguration and its immediate consequences—time overheads and program restructuring. Programming for a dynamic pipeline is very simple and, practically, it does not deviate from programming for conventional computers.

One can state that a dynamic pipeline is a new type of general-purpose computer that may perform pipelined computations. It allows elimination of the time of memory accesses from the time of program execution, bearing no penalty for dedication, which is the price all existing pipelined systems pay for the performance gains they achieve.

REFERENCES

1. Ramamoorthy, C. V. and Li, H. F., "Pipeline Architecture," *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 61-102.
2. Irwin, M. J., "Reconfigurable Pipeline Systems," *Proceedings 1978 ACM Annual Conference*, Vol. 1, pp. 86-92.
3. Ibbett, R. N., and Capon, P. C., "The Development of the MU5 Computer System," *Communications of the ACM*, Vol. 21, No. 1, January 1978, pp. 13-24.
4. Watson, W. J., "The TI ASC—A Highly Modular and Flexible Super Computer Architecture," *In AFIPS 1972 Fall Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, pp. 221-228.
5. Russell, R. M., "The CRAY-1 Computer System" *Communications ACM*, Vol. 21, January 1978, pp. 63-72.
6. Reddi, S. S. and Feustel, E. A., "A Restructurable Computer System," *IEEE Transactions on Computers*, Vol. C-27, No. 1, January 1978, pp. 1-20.
7. Thomasian, A., and Avizienis, A., "A Design Study of a Shared-Resource Computer System," *Proceedings of the Third International Symposium on Computer Architecture*, 1976, pp. 105-111.

8. Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M., "IBM System 360 Model 91, Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development*, January 1967, pp. 8-24.
9. Kartashev, S. I. and Kartashev, S. P., "Dynamic Architectures: Problems and Solutions," *Computer*, Vol. 11, July 1978, pp. 26-40.
10. Kartashev, S. I. and Kartashev, S. P., "Multicomputer System with Dynamic Architecture," *IEEE Transactions on Computers*, Vol. C-28, No. 10, October 1979, pp. 704-721.
11. Kartashev, S. I., Kartashev, S. P. and Ramamoorthy, C. V., "Adaptation Properties for Dynamic Architectures," 1979 National Computer Conference, *AFIPS Conference Proceedings*, AFIPS Press, 1979, Vol. 48, pp. 543-556.
12. Kartashev, S. I. and Kartashev, S. P., "A Microprocessor with Modular Control as a Universal Building Block for Complex Computers," *Proc. 3rd Euromicro Symposium on Microprocessing and Microprogramming*, Amsterdam, 1977, pp. 210-216.

Modular crossbar switch for large-scale multiprocessor systems—structure and implementation*

by BERNHARD QUATEMBER

Johannes Kepler University
Linz, Austria

ABSTRACT

This paper describes the architecture of an innovative modular crossbar switch for large-scale multiprocessor systems; its modular design principle is given a particularly thorough treatment. The crucial points of this design principle are the utilization of bit-serial buses and an integration of a combination of crosspoints with its own microprogrammed controller, (which is designated as "configurational processor") into a single building block, preferably an IC. This innovative concept allows the implementation of full crossbar switches for systems up to about 128 processors at relatively low cost. The expense of a crossbar for a system with approximately 1024 processing elements is then comparable to that of the other parts of the whole system.

A particular implementation of such a crossbar switch is embedded in a large-scale multimicroprocessor system being built at the Johannes Kepler University in Linz, Austria. This implementation is discussed in detail; emphasis is given to the complexity of its functional elements and its hardware expense. Since no ICs have yet been produced for the realization of the new building block for the crossbar mentioned above, this building block is realized on a printed circuit board using standard LSI-circuits.

INTRODUCTION

Up to the present, most multiprocessor systems have been built with relatively small numbers of processors. However, there is substantial interest in systems with a larger number of processors. Bearing in mind the decreasing cost of hardware, it has become reasonable to consider the implementation of large-scale multiprocessor systems wherein the most challenging problem would now be the realization of a high performance interconnection subsystem (processor-to-memory switch). This problem has not yet been solved satisfactorily. There are very few concepts that could be applied for large-scale systems. Probably the best known among them and the only one which can be regarded as efficient is the architectural concept of the CM*-system.¹⁻⁴ The usefulness of this concept

is based mainly on the assumption that memory references proceeding across cluster boundaries occur very infrequently. This assumption is of course justified in numerous fields of application. Nevertheless, there are further opportunities for possible utilization of a large-scale multiprocessor system where this assumption is not true.⁵ In these cases, the processor-to-memory switch of the CM*-system would not suffice to fulfill the requirements. Examining the known design concepts for interconnection networks,^{1-4,6-19} we have reached the conclusion that they do not form a useful basis for the realization of a large-scale multiprocessor system with a very wide range of applicability. In our latest investigations we have focused on the availability of a crossbar switch, since this type of processor-to-memory switch would have definite advantages, particularly for large-scale systems. Unfortunately, its realization for large-scale systems based on the known design concepts¹⁰⁻¹⁷ would be much too complex and expensive, as shown below. It is the purpose of this paper to present an innovative design concept that allows an inexpensive implementation of crossbar switches for multiprocessor systems with several hundreds of processing elements.

STATE-OF-THE-ART CROSSBAR SWITCHES AND THE APPLICABILITY OF THEIR DESIGN PRINCIPLES FOR LARGE SCALE MULTIPROCESSOR SYSTEMS

The known design principles for the extraordinarily well-suited crossbar switch have been thoroughly described in the literature.¹⁰⁻¹⁷ These descriptions are confined to systems with a relatively small number of processing elements (up to about 16 processing elements). The question naturally arises, whether these design principles could be utilized for large-scale systems. So far, there are several remarks in the current literature¹⁸⁻²⁰ to the effect that the use of these principles as a basis for the implementation of a large-scale crossbar would result in much too complex and expensive hardware. However, the reasons for this difficulty have not been clearly pointed out. For this reason we have carried out a thorough and careful analysis of this topic.²¹ We were able to show that the hardware expense is caused not only by the complexity of functional elements (switches, decoders, etc.) but also (and mainly) by the complexity of wiring and cabling (as summarized below), particularly for the architectures of the

*This work was supported by Grant No. 3896 from Fonds zur Forderung der wissenschaftlichen Forschung (Austrian Research Council).

C.mmp-system and the Multi-Interpreter-System of Burroughs. The well-known design concept of the C.mmp-system¹²⁻¹⁴ requires an extraordinarily high number of switches due to the considerable bus width and the quadratic complexity of crosspoints. Even worse would be the complex wiring needed for realization of the buses and for the requisite large number of interconnection lines between the control units of the switches and the switches themselves, since in both cases expensive "off-chip/on-board"-interconnection lines and even more expensive "off-board"-interconnection lines would be predominant. The design principle of the Multi-Interpreter-System of Burroughs^{15,16} makes full use of bit-serial buses provided for the crossbar. Nevertheless, the quadratic complexity of the required interconnection lines between the single control unit for the switches and the switches themselves, which, unfortunately, have to be very expensive "off-board"-interconnection lines, precludes this concept from the implementation of large-scale systems.

As both the C.mmp-system and the Multi-Interpreter-System of Burroughs were proposed several years ago, it is reasonable to ask whether the progress in IC technology in recent years and the progress predicted for the future could influence the implementation of large-scale crossbars. An earlier discussion of this²¹ shows that the gigantic progress in IC technology cannot be fully exploited when the known design principles for crossbars are applied, mainly for the following reasons:

1. due to the fact that the number of pins on an IC is limited (the maximum is about 60 pins), it is impossible to implement the whole crossbar on a single chip. Hence it is necessary to subdivide the crossbar into smaller building blocks; and
2. there is no conceivable way of subdividing the crossbar in an advantageous manner so that the expense of wiring and cabling would remain within a reasonable limit.

Thus the system organization has become the only area in which major improvements are conceivable, and the following discussion should be regarded as an attempt in this direction.

RATIONALE OF THE INNOVATIVE, MODULAR CROSSBAR SWITCH

Design Goals

- (1) The prime design goal is to guarantee the feasibility of the crossbar switch for large-scale multiprocessor systems. It has been shown that the manner in which the required subdivision into smaller building blocks is carried out plays an important role in connection with the applicability of the respective design concept for large-scale systems and there is no doubt that to achieve this goal (feasibility) one must solve the problem of discovering a new organization that will be based on a more appropriate subdivision of the crossbar.
- (2) As in all computer systems, a high degree of modularity

with a minimal number of types of building blocks would be desirable. Thus, a further goal of this design concept is to allow the implementation of a crossbar with a minimal number of types of building blocks, preferably a single type (that is, with a multitude of identical building blocks).

- (3) Another design goal, which is to some extent related to those mentioned above, is the good extensibility of the crossbar, since in large-scale multiprocessor systems we have to reckon with varying numbers of processors or processing elements. Thus the ease of growth without major changes of the hardware design (particularly without changes in the arrangement of card frames and the housing of these frames in the cabinets) is of great importance.
- (4) As seen above, predictions for future improvements within the framework of known design principles in the implementation of crossbar switches by LSI technology are not optimistic. Nevertheless, it is important to look for ways to implement the advances in LSI technology in order to reduce the size of the crossbar switch and the expense of wiring and cabling. It is therefore a further important goal of our proposed design concept to achieve major improvements in this way, in contrast to the known concepts.

Design Philosophy

The innovative concept presented here is mainly based on the present advances in LSI (VLSI) technology.

Before presenting the details of this new design approach for large-scale crossbar switches, it should be noted that even with the new concept it will of course be impossible to avoid the quadratic complexity of essential functional elements (switches) and of interconnection lines that is typical of the crossbar scheme.

The pivotal points of the new design approach, on the contrary, are:

1. to endeavour to distribute the above-mentioned complexity, which is inherent in every large scale crossbar system, among the separate elements of the physical structure so that more complex chips can be employed and correspondingly less expensive wiring and cabling for the "on-board/off-chip" and especially the "off-board" interconnections can be used;
2. to keep down the amount of "on-board/off-chip" and especially "off-board" interconnection lines and of certain functional elements at the expense of introducing additional functional elements and complex functional units that have not been used before in crossbar designs;
3. to implement these new functional elements and units on LSI chips as far as possible, together with the requisite functional elements that are normally contained in conventional crossbar designs and are also necessary in our design, in such a manner that the expense of these additional new elements and units will be considerably lower than the corresponding savings from the reduced amount of costly wiring and cabling.

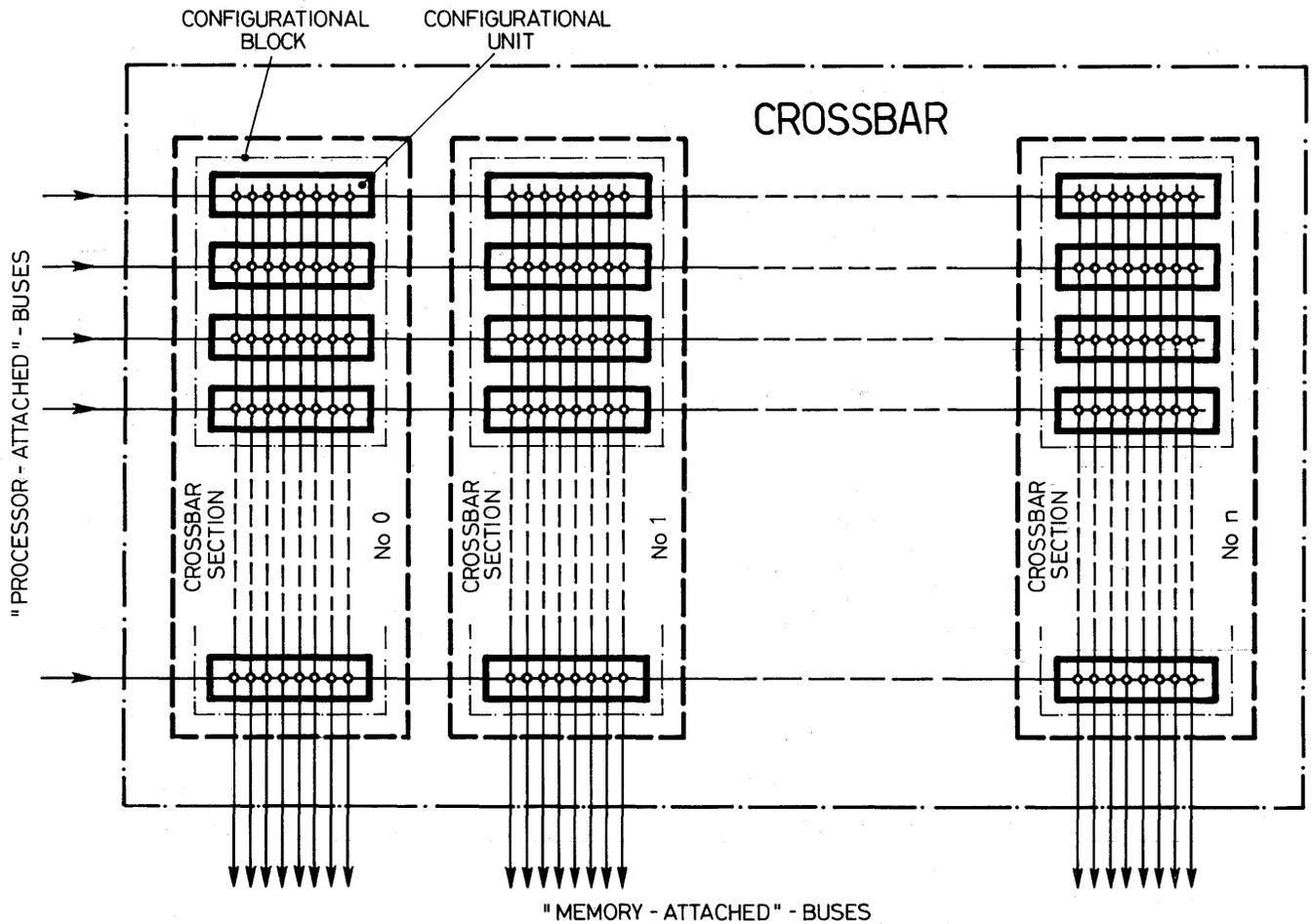


Figure 1—Canonical crossbar switch structure

The prime measure for achieving these central aims is to drastically decrease the hardware complexity—the number of switches and, in part, the amount of “off-board” wiring inherent in conventional crossbar designs—by employing bit-serial buses at the expense of more complex functional units. However, this measure does not suffice. As already remarked, there is a second reason for hardware complexity, namely the complexity associated with the realization of the required interconnection lines for control of the switches. Conventional crossbar designs provide a single control unit (e.g., Burroughs Multi-Interpreter-System) or several control units (e.g., C.mmp-system) for control of the switches. This unit (or these units) is (geographically) separated from the array of crosspoints. In addition, the control information for setting and opening the switches (configurational information) is transferred to the array of switches in a decoded form and in a bitparallel manner, resulting in a large number of interconnection lines that are, owing to their considerable distance, very expensive “off-board”-interconnection lines, in many cases even extremely expensive inter-cabinet cables. Returning to the above main points of the design approach as a basis, the hardware expense caused by the necessary control of the switches can be reduced to a minimum by the following two measures:

1. introducing new functional units with inherent intelligence to control the switches²²⁻²⁵ (we call them “configurational processors,”* and they are geographically distributed among the whole crossbar);
2. transferring the configurational information from the processors or the processing elements to the configurational processors in binary coded form and in a bit-serial manner via the bit-serial “processor-attached” buses of the crossbar switch discussed above.²²⁻²⁵

To apply these measures for reduction in wiring expense, the crossbar has to be subdivided in an adequate manner.

Figure 1 depicts how this subdivision is carried out. Several crosspoints have been combined to a unit. The number of these crosspoints is, in principle, arbitrary, although powers of two are preferable. For the number of combined crosspoints, we shall now introduce the term “subdivision-number.” The 8 crosspoints shown in Figure 1 seems to be a reasonable choice for the first implementation of a crossbar

* We have chosen the term “configurational” (e.g., configurational processor) due to the capability of such processors to establish data paths so that the totality of established data paths can be regarded as configurations of the crossbar or configurations of the whole multiprocessor system.

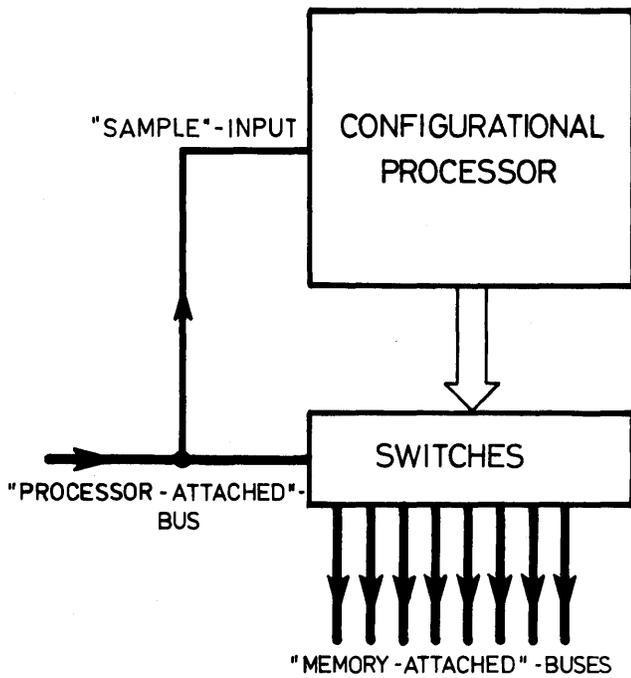


Figure 2—Block diagram of a configurational unit

based on the proposed design concept (this implementation will be discussed below in detail). However, in other circumstances it could be more advantageous to combine 16 or 32 crosspoints. Each of these combinations of crosspoints is provided with its own configurational processor, which is closely attached to the switches in the crosspoints.

This integration of switches and configurational processors produces a functional unit, which is depicted in Figure 2 and will be referred to as a configurational unit. In this concept, such a configurational unit is the smallest unit that could be taken into consideration for realization as a single building block. However, it is desirable to combine suitable numbers of configurational units to form a building block (c.f. Figure 1), preferably an IC. We refer to such an element as a configurational block. Figure 3 shows the structure of a configurational block. For the number of configurational units in such a configurational block we shall now introduce the term "blocking number."

Thus as we see from Figures 1 and 4, the whole crossbar switch can be realized with building blocks of a single type. It is in this implementation with one type of identical building block, which can, in addition, be reproduced with ease, that the highest conceivable degree of modularity is attained, thus fulfilling optimally our design goal of good modularity.

To weigh the advantages of the new design concept on the wiring expense, we have to remember:

1. that the transfer of the configurational information takes place via the "processor-attached" buses, so that no separate wiring has to be provided for this purpose; and
2. that the only interconnection lines that have to be realized are those between the configurational processor and switches. This wiring is of course subject to quadratic complexity, but the interconnection lines will be either

"on-chip"-, or in the most unfavourable case, "off-chip/on-board"-interconnection lines (depending on the particular form of implementation), in contrast to the mainly "off-board" interconnection lines for the purpose of control of the switches in a conventional crossbar.

As the cost and also the size of the "on-chip"- or "off-chip/on-board"-interconnection lines, which are the only ones now required, are essentially lower than the "off-board"-interconnection lines that are mainly required otherwise, our goal of strongly reducing the expense of the wiring for this purpose seems to have been attained.

The basic hardware structure of a configurational processor is depicted in Figure 5 (certain less important details for the assessment of the practicability of the new design have been omitted, e.g., circuitry for the solution of the bus/memory port-contention problem, extension to bidirectional switches,

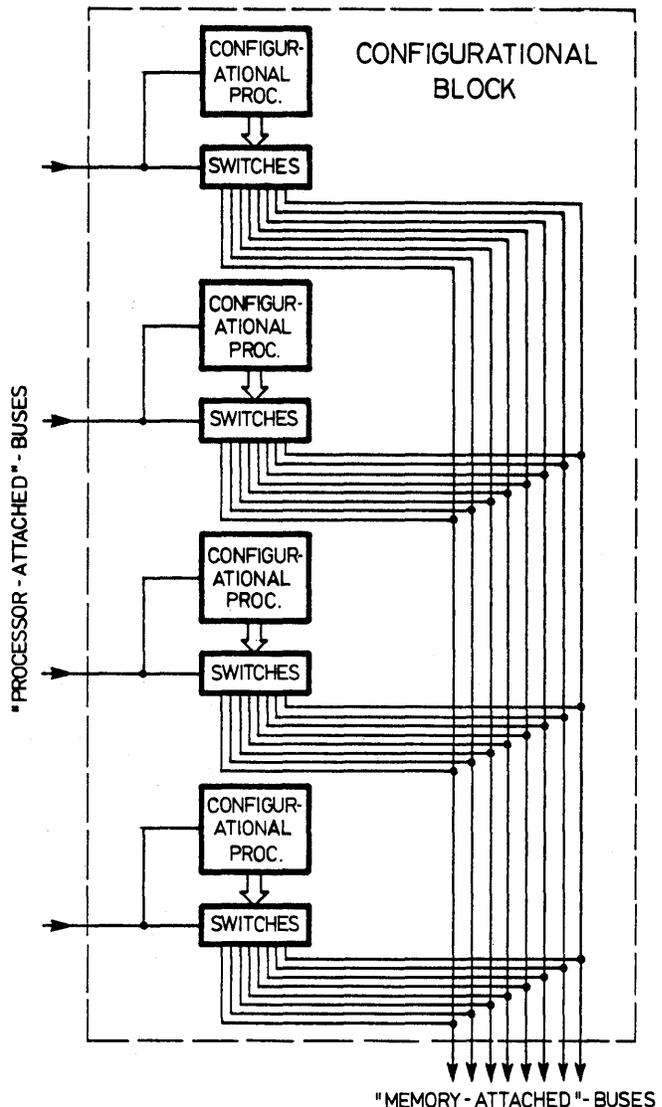


Figure 3—Structure of a configurational block

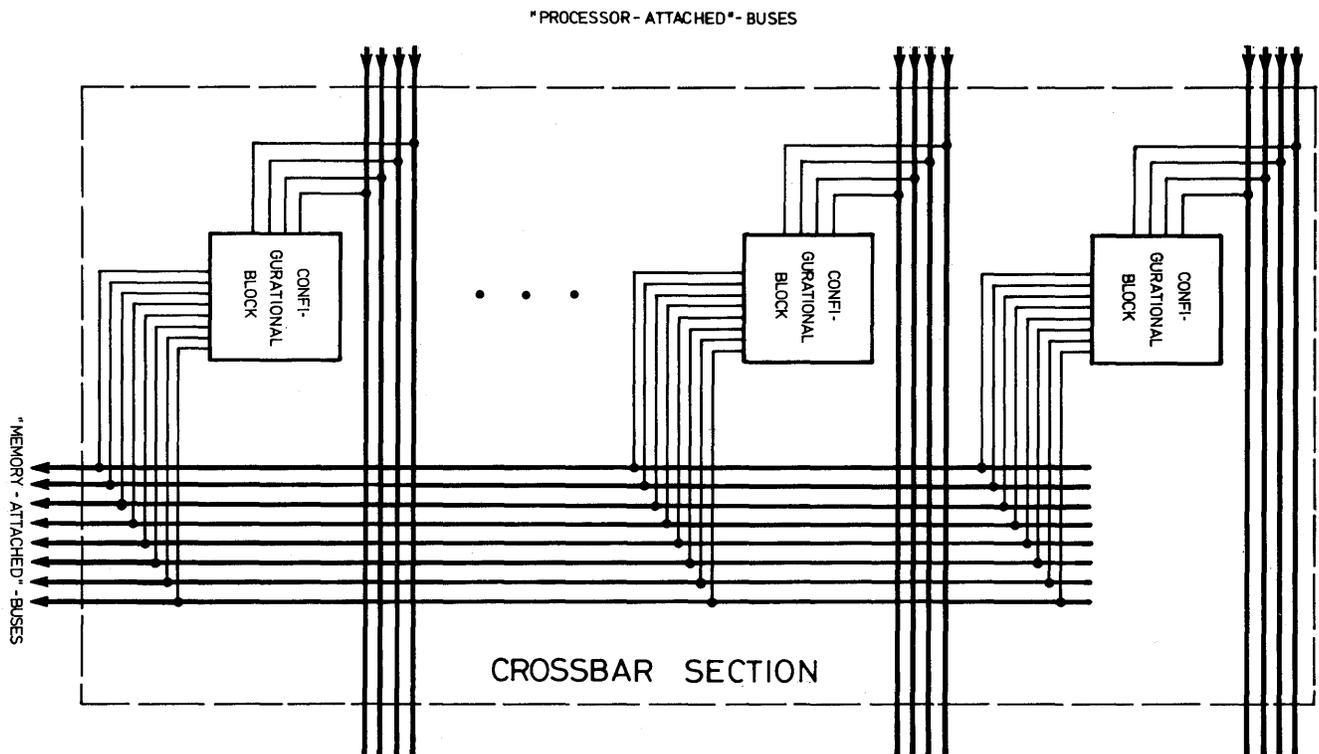


Figure 4—Structure of a crossbar section

adaptions to the special features of the particular multiprocessor system).

As shown in Figure 5, the configurational processor has all the essential parts that are usually provided in a micro-programmed processor. However, these parts are not as sophisticated as, for example, those of the processor of a general purpose computer because of their intended purpose as components of a relatively simple controller for the closely attached switches. As also shown in Figure 5, for every control input of the switches in the configurational processor a separate output is provided, which delivers a signal to these control inputs via a separate line. By means of these output signals the switches are set or opened, resulting in each case in the establishing or removal of a communication path (or possibly—as we will later show in more detail—several communication paths) for a memory reference across the crossbar.

In contrast to conventional crossbar designs, we have to distinguish between two consecutive phases (sequences of operations) in every communication via the crossbar:

1. the establishment of a communication path,
2. the transfer of the address and data required for a memory reference to a memory module (memory bank) across the crossbar, and
3. the removal of the communication path after use (which involves operations similar to those in 1).

It should be noted that once a communication path has been established, more than one memory reference to the same memory bank can be effected.

The communication via the buses of the crossbar occurs in an asynchronous, bitserial manner. Therefore no central clock is needed. Every processing element and every configurational processor has its own clock. These clocks are not synchronized; however, they do have to be calibrated.

In order to establish or remove a particular communication path, the relevant information (configurational information), which is provided by the respective processing element, has to be transferred via its "processor-attached" bus to the configurational processors attached to this bus. During this process only the configurational processor that is concerned will accept this information by setting or closing the switch.

The communication via the bus in the course of the transfer of the configurational information takes place on the basis of a relatively simple protocol. Among other things that this protocol provides is a data format of fixed length with a start bit and a stop bit framing a header with the control information, a body that specifies the crosspoint (in binary coded notation), and, as an option, a trailer for error detection and correction (the latter will not be considered in this paper).

The configurational information that is transferred to the configurational processor is transmitted to its "sample" input (Figure 5). The configurational processor works with a clock rate that is considerably higher than the baud rate of the information transferred via the bus. Thus, at the ends of successive time intervals, which are short compared with the length of the bits of the information transferred via the bus, the configurational information transmitted to the conditional input of the control unit decides the branching behavior of the (running) microprogram. Hence it is possible to identify a

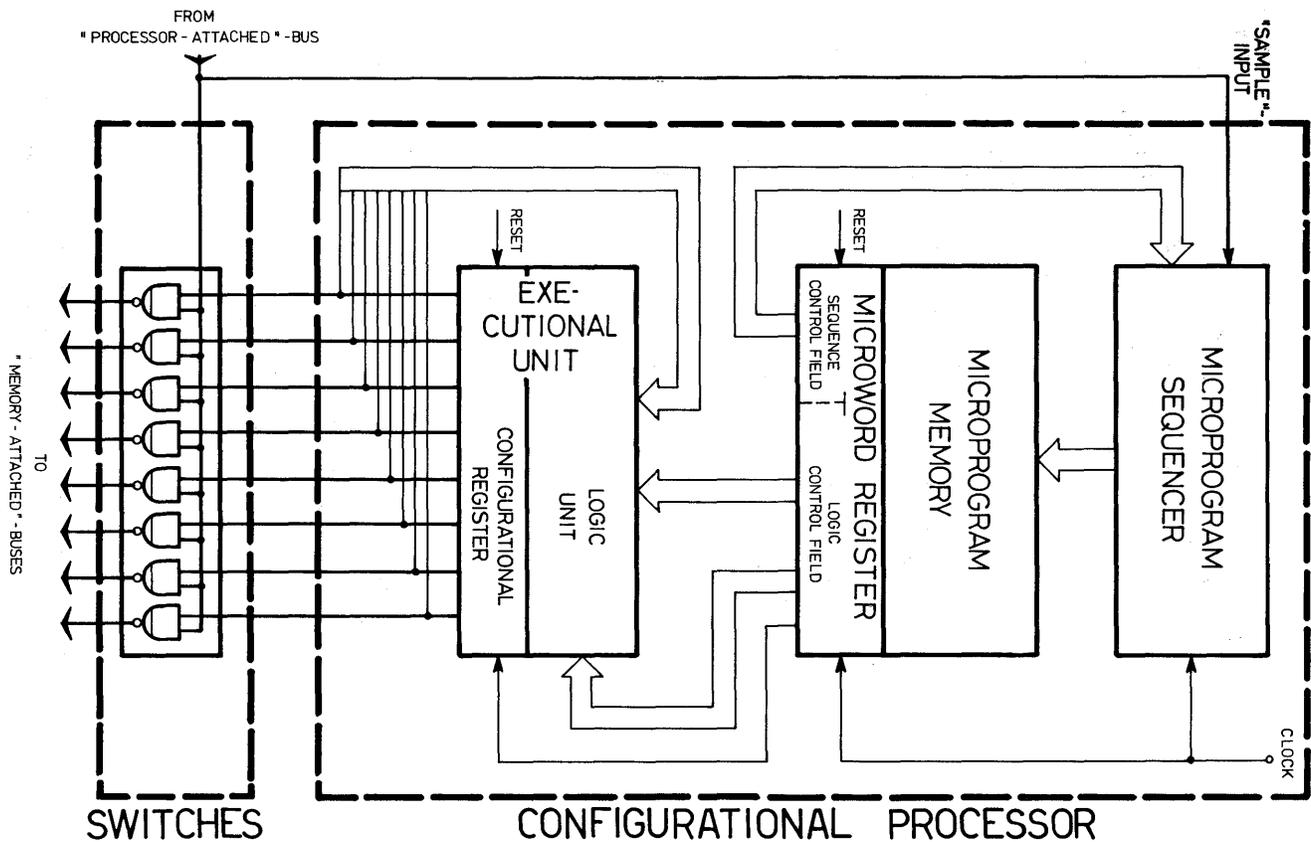


Figure 5—Details of a configurational unit (configurational processor)

start bit, establish synchronization, and thereby sample the subsequent bits by firmware means.

The sequence of bits involved in the transfer of information via the bus is sampled and then interpreted by the microprogram. At this point, the header in this configurational information enables the microprogram to distinguish, for instance, between a transfer of configurational information for the establishment (removal) of a communication path and other kinds of information transfer. In addition, the header also contains the information concerning the establishment or removal of a communication path. Suppose that the interpretation of the header implies that a communication path is to be established or removed. Then the body in the data format of the configurational information is also interpreted. First the microprogram examines whether there are crosspoints attached to the respective configurational processor that should be activated or not. If such a crosspoint is to be set or opened, a sequence of operations in its executional unit are put into action. These operations will be treated after the following short discussion of the structure of this unit. As represented in Figure 5, the executional unit contains a so-called configurational register. In this register a single flip-flop is provided for each crosspoint. The outputs of these flip-flops are also outputs of the whole executional unit. They are connected with the switches in the crosspoints belonging to them. To establish (remove) a single data path, the respective flip-flop has to be set (reset). An advantage of this new design is

the ease with which one can design the configurational processor so that arbitrary bit patterns can be stored in this register. Thus data paths can be established in which one "processor-attached" bus is connected with several "memory-attached" buses, allowing the processing element to write in several memory modules (memory banks). This feature will be very useful in certain applications and is completely new in the field of multiprocessor systems. Thus a comparatively complex interconnection structure is established (removed) step by step. At each step, a "memory-attached" bus is connected (removed) by means of a single item of configurational information with the data format discussed above. The command to carry out a connection or a disconnection has to be contained, as already mentioned, in the header of the corresponding item of configurational information. The body of the configurational information contains in binary coded form the information which of the crosspoints should be activated. After interpretation of this body by the microprogram, the control unit delivers this information in a bitparallel and decoded form to the logic unit represented in Figure 5. This unit carries out a suitable boolean operation (for example "or" for a connection) on this information and the current content of the configurational register. The configurational register is then updated, whereby the connection or disconnection of the respective bus is directly attained. For the important cases where all of the crosspoints are set (opened), two separate special procedures, which require only a single item of config-

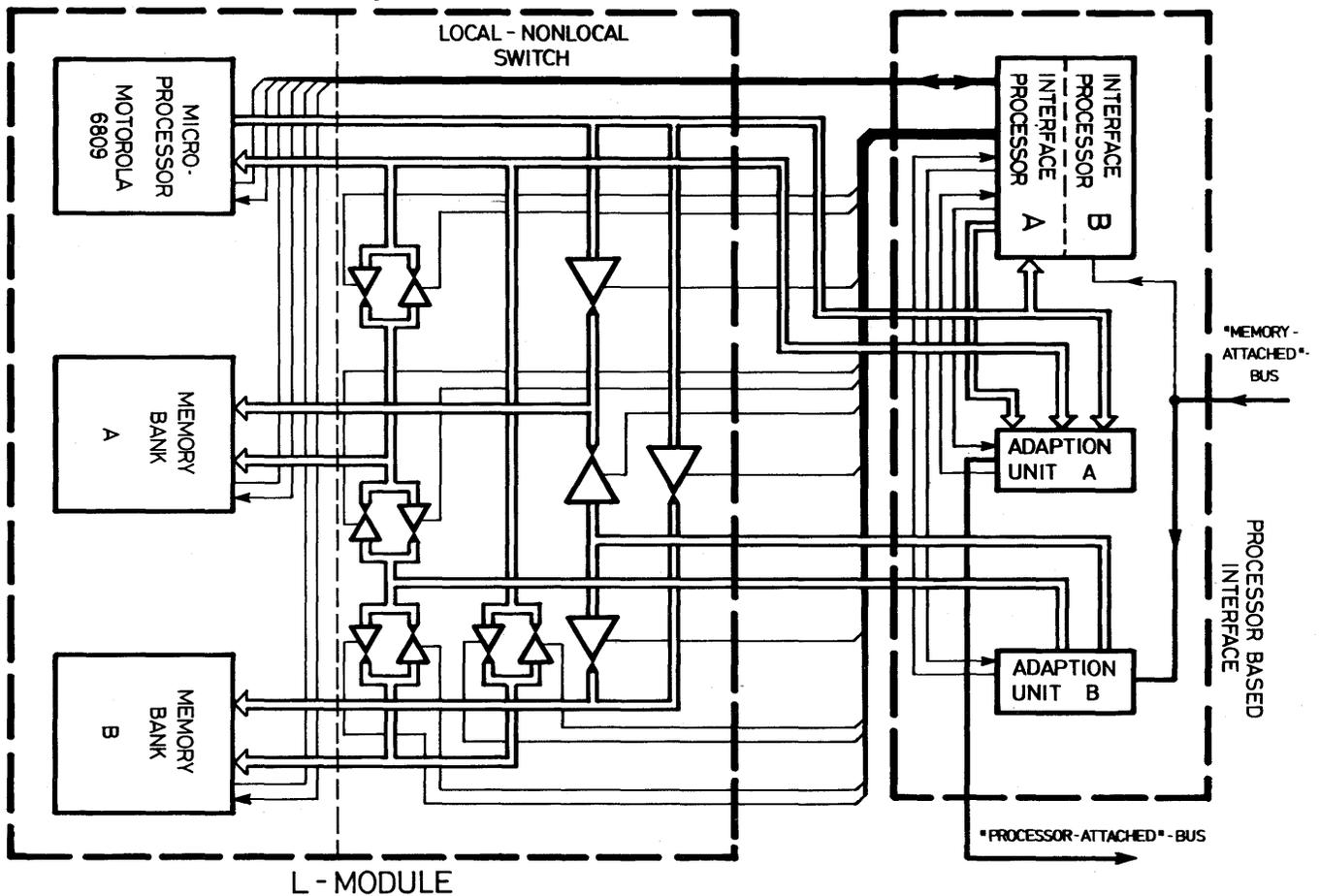


Figure 6—Details of a processing element with an L-module in the multiprocessor system at Johannes Kepler University

urational information with the data format discussed above, are provided. The information concerning the occurrence of two special cases is provided in the header. Since information about particular crosspoints is no longer necessary, the body in the data format is of no significance. Once a data path has been established, the transfers in the course of memory references are carried out based on a protocol which is similar to the one for the transfer of the configurational information. It should be noted that both the address and the data for a memory reference are transferred via the same (physical) communication path.

Implementation Approaches

Our construction forces us to realize a configurational block as a single building block. In this way it is possible to construct the complete crossbar from such identical building blocks (c.f. Figure 1)

In order to achieve an advantageous solution for the construction of a multiprocessor system with this new crossbar, a crossbar section (c.f. Figure 1 and 4) should be housed together with those processing elements that are interconnected with it by its "memory-attached" buses, together with a suitable portion of shared memory within the same cabinet. Hence a suitable subdivision-number should be chosen so that

a maximal number of processing elements can be housed together with portions of the above-mentioned subsystems. We use the term "canonical construction" to indicate that a configurational block is implemented as a building block and that the above-mentioned method of housing and subdivision of the crossbar are applied.

As mentioned above, it is a goal of this research to exploit LSI technology for the implementation of the crossbar switch in order to keep down expenses and size. Hence it would be desirable to investigate whether the implementation of such a building block (configurational block) utilizing this technology is practicable. Figures 1, 3 and 4 indicate that in a crossbar produced by an arrangement of such building blocks, these components will have high circuit complexity and low external connecting wiring requirements. These design characteristics are regarded as the most important technical criteria for a promising exploitation of LSI technology. Another criterion is the possibility of production at a tolerable cost level. There is a good chance of fulfilling this criterion, since only one type of basic building block will be required for the whole crossbar. In addition, large numbers will be needed, which would justify mass production of this component, the principal prerequisite for low-cost production. For this reason, our research is based on the expectation that integrated

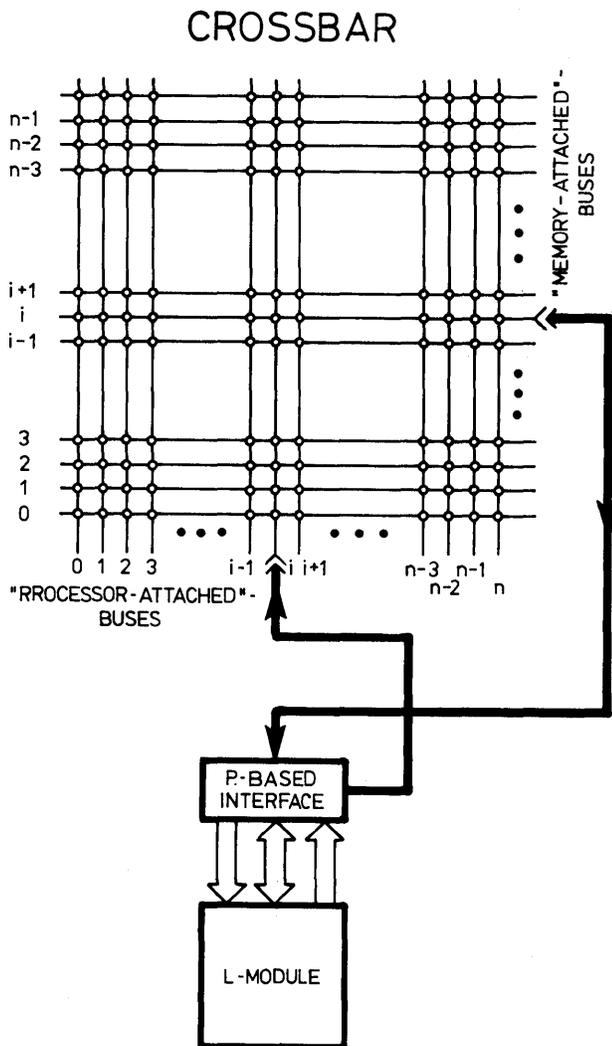


Figure 7—Organization of the multiprocessor system at Johannes Kepler University: arrangement of a processing element

circuits for this purpose will be produced in the future. This implementation approach aims at a realization of the configurational blocks with the highest possible subdivision number and blocking number. However, there are limitations due to

1. the limited number (60) of pins;
2. the limited degree of complexity of a chip;
3. the upper limit on the subdivision number imposed by the number of processing elements in a cabinet.

Taking present-day LSI technology as a basis, limitation 2 is the (still) crucial one. However, it is already possible (and reasonable) to realize an IC with a subdivision number of 16 and a blocking number of 4 in a 24-pin DIP. Supposing that such an IC is available, a multiprocessor system (employing single-chip microprocessors) with 256 processing elements could be implemented in 8 cabinets.

No ICs have yet been produced for the realization of the configurational block of the new crossbar. Nevertheless, this building block can be realized on a printed circuit board using standard ICs (mainly standard LSI ICs). This implementation

approach would allow a cost-effective implementation of a multiprocessor system with up to 64 processing elements. But the practicability of this implementation approach does not affect the importance of the previous goal of producing a special purpose IC, since a volume reduction by a factor of about 25 could be achieved.

AN IMPLEMENTATION OF THE INNOVATIVE, MODULAR CROSSBAR SWITCH

In order to demonstrate (among other things) the usefulness of the innovative design concept of the crossbar, a large-scale multiprocessor system is now being built at the Johannes Kepler University in Linz, Austria.^{22-24,26-28} This system is a pure research vehicle and is intended as a testbed for further developments.

Characterization of the Multiprocessor System in which the Implemented Crossbar is Embedded

There is no fundamental limit on the size of the crossbar. However, a particular realization has to be based on the maximal size of the system. In this realization, the size of the system is limited to 256 processing elements on account of the software-management and the interfacing to the crossbar. The mechanical construction, the packaging, the wiring and cabling, though, are designed only for a modular extensibility of up to 64 processing elements.

In addition to the new crossbar, the multiprocessor system consists of the processing elements (whose memory is a shared memory) and a synchronization logic. The synchronization logic is described elsewhere and will not be treated here.^{5,28} The processing elements contain an L-module and a processor-based interface. The logical concept of an L-module has been introduced by B. Buchberger.⁵

As illustrated in Figure 6, the L-module contains a single-chip microprocessor (Motorola 6809), two memory banks, and a local-nonlocal switch. The whole memory of an L-module can be regarded as a shared memory; both local and nonlocal references can be made for both memory banks. In this organization, the following advantageous mode of operation is also possible: The processor of an L-module can refer to memory bank A and a remote L-module can simultaneously refer to memory bank B. The L-module is closely attached adjacent to a processor-based interface for the crossbar.

In Figure 7 the arrangement of such a processing element with respect to the crossbar is shown. Further details concerning the L-module have been given in earlier studies.^{5,27} The software management in the system is adapted in the following way to take into account the operational behavior of the new crossbar: In a nonlocal reference, we have to distinguish between the following two independent phases, each of which is produced by a separate instruction:

1. the establishment or removal of a communication path in the crossbar, and
2. the transfer of information to the corresponding remote processing element across the crossbar.

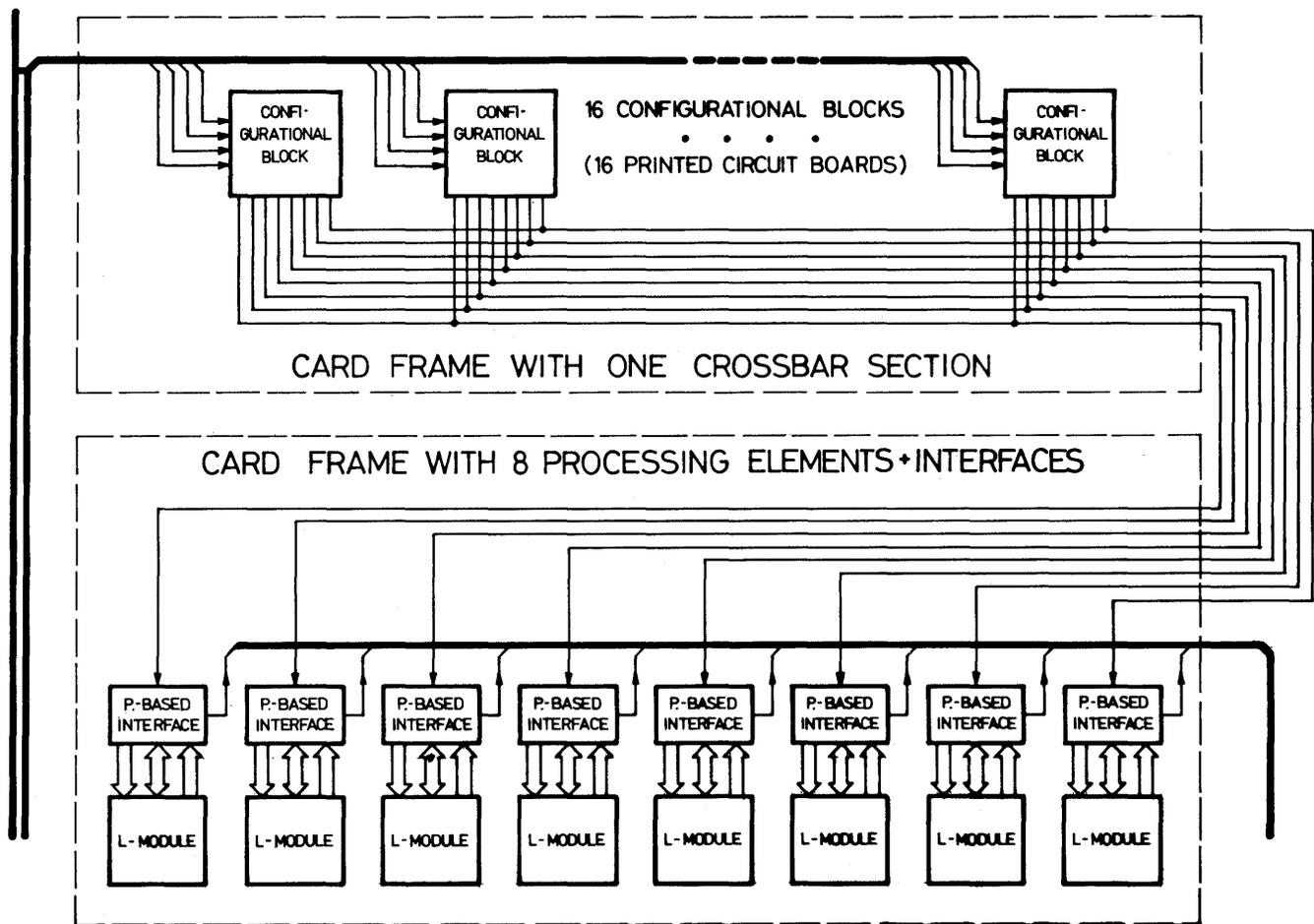


Figure 8—Wiring and cabling between card frames in a cabinet of the multiprocessor system at Johannes Kepler University

Each item of information transferred across the crossbar contains the data and an address referring to the destination memory bank. Thus the addressing of the shared memory uses the mailbox principle.

The Implementation of the Basic Building Block of the Crossbar Switch

The basic building block in this implementation is a configurational block with the subdivision number 8 and the blocking number 4.

As no special ICs for the new crossbar have yet been built, this basic building block is implemented on a printed circuit board (233.4 × 220 mm) with standard LSI circuits and a few SSI circuits. The design is mainly based on the employment of LSI circuits of the family Am 2900 (Advanced Micro Devices). In addition, a bipolar field programmable logic sequencer (Signetics 82S105) was used for the realization of the executional unit (c.f. Figure 5). This had the advantage of allowing a reduction in volume compared with alternative implementations (for example, with MSI and SSI circuits). However, simple SSI-circuits were used for the switches.

Physical Configuration

The complete multiprocessor system consists of 8 identical cabinets. The above-mentioned canonical design is applied. Each cabinet contains 8 processing elements together with their attached crossbar section. This crossbar section consists of 16 printed circuit boards in a separate card frame. The processing elements are packaged in an adjacent card frame. These two card frames and the requisite wiring and cabling in this area are shown in Figure 8.

In Figure 9, the physical structure of a cabinet is depicted. The "out-card frame/in-cabinet" wiring is clearly represented. The numbers above the diagonal strokes indicate the numbers of pairs of conductors (twisted pairs, two conductors in a flatband cable). We indicate pairs, not single conductors, since it is preferable to use pairs of conductors in the realization of buses in the crossbar.

Figure 10 depicts the required intercabinet cabling in the complete system. Once again, the numbers above the diagonal strokes represent the required numbers of pairs of conductors. Figures 9 and 10 allow an assessment of the wiring and cabling expense, which appear to be very low compared with the expense of wiring in other design concepts for multiprocessor systems.

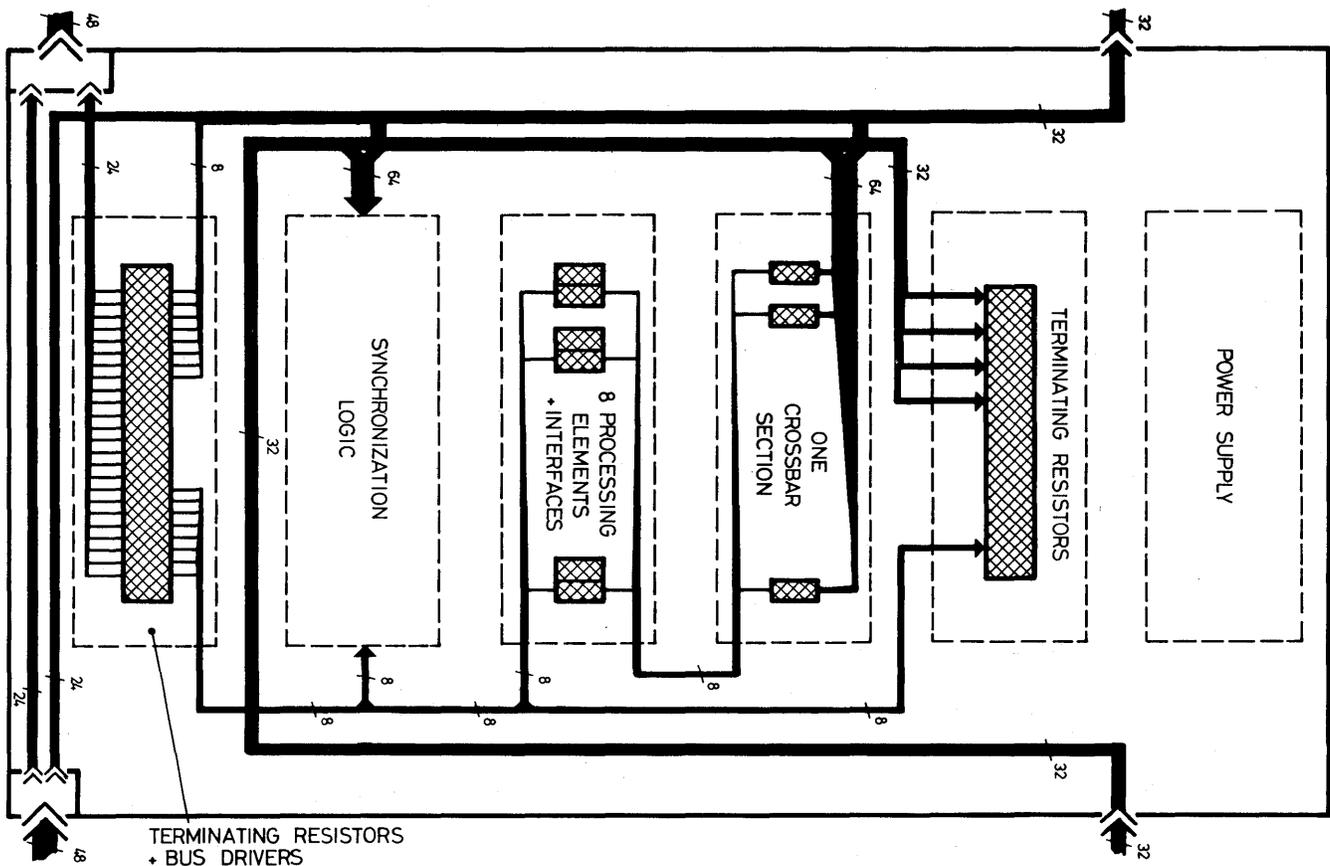


Figure 9—"Out-card frame" wiring and cabling in a cabinet of the multiprocessor system at Johannes Kepler University

PERFORMANCE

The access time in a nonlocal reference in this implementation is about 6 microseconds but could easily be improved at the cost of more expense and effort, which did not seem justified at this experimental stage.

REFERENCES

1. Swan, R.J., S.H. Fuller, and D.P. Siewiorek. "Cm*—A modular, multi-microprocessor." *AFIPS Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 637-644.
2. Swan, R.J., A. Bechtolsheim, K. Lai, and J.K. Ousterhout. "The imple-

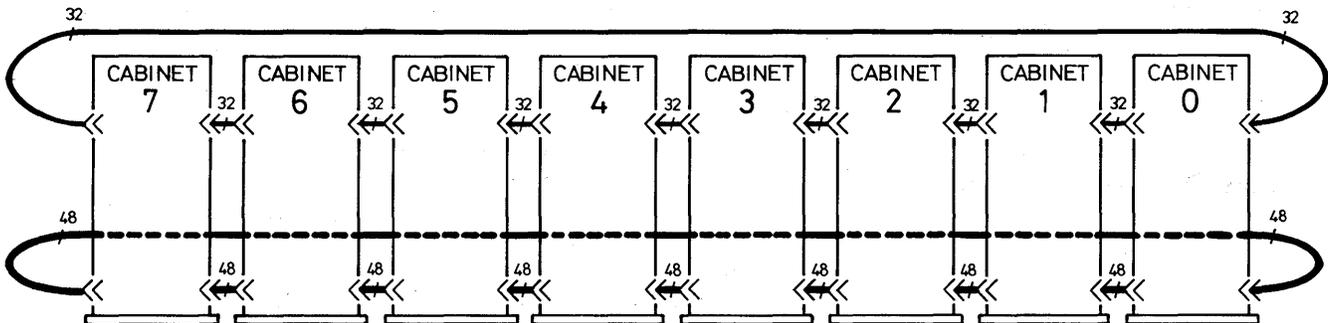


Figure 10—Intercabinet cabling of the multiprocessor system at Johannes Kepler University

- mentation of the Cm* multimicroprocessor." *AFIPS Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 645-655.
3. Jones, A.K., R.J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans. "Software management of Cm*—A distributed multiprocessor," *AFIPS Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 657-663.
 4. Fuller, S.H., J.K. Ousterhout, L. Raskin, P. Rubinfeld, P.J. Sindhu, and R.J. Swan. "Multi-Microprocessors: An Overview and Working Example." *Proceedings of the IEEE* (Vol. 66), No. 2, February 1978, pp. 228.
 5. Buchberger, B., and K. Aspetsberger. "A Universal Variable-Topology Multi-Microprocessor-System." *Proceedings of the 6th International Symposium on Mini- and Microcomputers and their Applications*, Budapest, 1980, pp. 136-140.
 6. Enslow, P.H., Jr. *Multiprocessors and Parallel Processing*. New York: John Wiley & Sons, Inc., 1974.
 7. Enslow, P.H., Jr. "Multiprocessors and other Parallel Systems—an Introduction and Overview." In W. Händler (Ed.), *Computer Architecture, Workshop of the Gesellschaft für Informatik, Erlangen, May 1975, Informatik-Fachberichte*. Berlin, Heidelberg, New York: Springer-Verlag, 1975.
 8. Enslow, P.H., Jr. "Multiprocessor Organization—A Survey." *Computing Surveys* (Vol. 9), No. 1, March 1977, pp. 103-129.
 9. Weitzman, C. *Distributed Micro/Minicomputer Systems: Structure, Implementation, and Application*. Englewood Cliffs: Prentice-Hall, Inc., 1980.
 10. Porter, R.E. "The RW-400—a new polymorphic data system." *Datamation* (Vol. 6), January/February 1960, pp. 8-14.
 11. Anderson, J.P., S.A. Hoffman, J. Shifman, and R.J. Williams. "D825—A Multiple-Computer System for Command and Control." *AFIPS Proceedings of the Fall Joint Computer Conference* (Vol. 22), 1962, pp. 86-96.
 12. Bell, C.G., and P. Freeman. "C.ai—A computer architecture for AI research." *AFIPS Proceedings of the Fall Joint Computer Conference* (Vol. 41), Part II, 1972, pp. 779-790.
 13. Wulf, W.A., and C.G. Bell. "C.mmp—A multi-mini-processor." *AFIPS Proceedings of the Fall Joint Computer Conference* (Vol. 41), Part II, 1972, pp. 765-777.
 14. Wulf, W., and R. Levin. "A Local Network." *Datamation* (Vol. 21), February 1975, pp. 47-50.
 15. Davis, R.L., S. Zucker, and C.M. Campbell. "A building block approach to multiprocessing." *AFIPS Proceedings of the Spring Joint Computer Conference* (Vol. 40), 1972, pp. 685-703.
 16. Davis, R.L., and S. Zucker. "Structure of a multiprocessor using micro-programmable building blocks." *Proceedings of the National Aerospace Electronics Conference*, Dayton, Ohio, May 1971, pp. 186-200.
 17. Bell, C.G., and A. Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill Book Company, 1971.
 18. Patel, J.H. "Processor-Memory Interconnections for Multiprocessors." *Proceedings of the 6th Annual Symposium on Computer Architecture*, April 1979, pp. 168-177.
 19. Sullivan, H., and T.R. Bashkow. "A large scale, homogeneous, fully distributed parallel machine, I." *Proceedings of the 4th Annual Symposium on Computer Architecture*, March 1977, pp. 105-117.
 20. Giloi, W.K. "Rechnerarchitektur." *Informatik-Spektrum* (Vol. 3), No. 1, January 1980, pp. 3-18.
 21. Quatember, B. "A Hardware Realization of a variable-topology Multiprocessor System—The Implementation Principles." *Proceedings of the 6th International Symposium on Mini- and Microcomputers and their Applications*, Budapest, 1980, pp. 141-147.
 22. Quatember, B. *Rekonfigurierbares Verbindungsnetzwerk mit bitserieller Übertragung der Konfigurationsinformation und über seine räumliche Ausdehnung verteilten Konfigurationsprozessoren*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1980.
 23. Quatember, B. *Kreuzschienenschalter mit über seine räumliche Ausdehnung verteilten Konfigurationprozessoren*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1980.
 24. Quatember, B. *Konfigurationsprozessor für digitale Verbindungsnetzwerke*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1981.
 25. Quatember, B. *Als Integrierte Schaltung ausführbare Baueinheit zum Aufbau von rekonfigurierbaren Verbindungsnetzwerken, insbesondere von Kreuzschienenschaltern*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1981.
 26. Buchberger, B., and B. Quatember. *Rekonfigurierbares Multiprozessor-system*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1980.
 27. Buchberger, B., and B. Quatember. *Universelle Funktionseinheit mit einem Prozessor und einer Speicheranordnung mit mindestens zwei Speicherbänken und mindestens zwei Speichereingängen*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1981.
 28. Quatember, B. *Einrichtung zur bitseriellen Übertragung, Speicherung und zur bitseriellen Abfrage von Statusinformation im Bereiche der digitalen Informationsverarbeitung*. Patent Application, Österreichisches Patentamt (Austrian Patent Office), 1981.

Some potential deadlocks in layered communications architectures*

by JOSEPH HELLERSTEIN and WESLEY W. CHU

University of California
Los Angeles, California

ABSTRACT

Since their introduction in the early 1970s, layered communications architectures have become widely used. Unfortunately, the limitations of these architectures have not been generally recognized. Specifically, if caution is not exercised in their design, deadlocks can occur. This paper presents several examples of such errors.

INTRODUCTION

The use of layered communications architectures first became popular with the development of the ARPA network.⁴ Since then, layering has been employed by CYCLADES⁸, Digital Equipment Corporation's DECNET⁵, and IBM's System Network Architecture.⁶ This almost universal acceptance of layered architectures has resulted from their ease of understanding, service sharing, and modification.

Partly because of the widespread success of layered communications architectures, there has been a lack of understanding of their limitations. In particular, we have frequently encountered the mistaken belief (in both industry and academia) that if two protocols contain no deadlock, then when these protocols are layered, no deadlock will be present. The consequences of this fallacious belief can be dire.

In this paper, we present examples to illustrate some areas in which caution must be exercised when protocols are layered. However, to put these examples in proper perspective, we first define layering and then explain the underlying cause of layering induced deadlocks.

Given a set of services to be performed, a layered architecture can be obtained by partitioning this set into a number of distinct layers.⁷ Each layer provides services to the layers above it and is in turn served by the layers below it. For any given layer, there may be a number of service providing entities. Communication between entities at the same layer is accomplished by the use of protocols.

A fundamental principle in layered architectures is that of data hiding. According to this precept, a layer knows nothing about the format or content of the messages supplied to it by

higher layers. Thus, control information exchanged between layer $n+1$ entities (such as connects, disconnects, and acknowledgments) is treated as data by layer n entities.

The reader should note that it is because of service partitioning and data hiding that so many of the benefits of layering result. However, service partitioning makes higher layers dependent on the services of lower layers. Data hiding extends the scope of this dependency by prohibiting lower layers from distinguishing between higher layer control and data messages. It is from these dependencies that the layering of deadlock free protocols can result in a deadlock.

The next three sections present examples of deadlocks introduced by layering.[†] In each case, we give possible resolutions for these errors.

DEADLOCK 1: INTERLAYER RESOURCE CONTENTION

Here we address the implications of two protocol layers contending for the same set of resources. Instances of such resources are buffers, processes, and logical names. It is quite common for such contention to exist, since by having two layers share a resource pool, throughput can be increased. (For example, sharing buffer pools can greatly improve throughput over that achievable by having separate pools.²)

Below we illustrate how a deadlock can occur when such resource contention exists. Consider a two layer architecture as one might find in a network communications processor, like the ARPA IMP. The lower layer handles the communications line (we ignore the electrical interface), and the higher layer controls the transport protocol. We assume that each layer can permit multiple outstanding transmissions before an acknowledgment is required. Furthermore, we assume that the protocol handlers share the same buffer pool so as to maximize buffer utilization. Finally, note that if the architecture is truly layered, then the link protocol will treat acknowledgments returned to the transport protocol as data.

[†]The reader familiar with protocol verification literature will note that some of the errors that we classify as deadlocks are technically instances of tempo blocking.¹ Because we are interested in addressing a broad audience and this distinction does not impact our paper, we have chosen to ignore the difference between these two categories of protocol errors.

*This research is supported by the U.S. Army Contract No. GASG 60-70-C-0087.

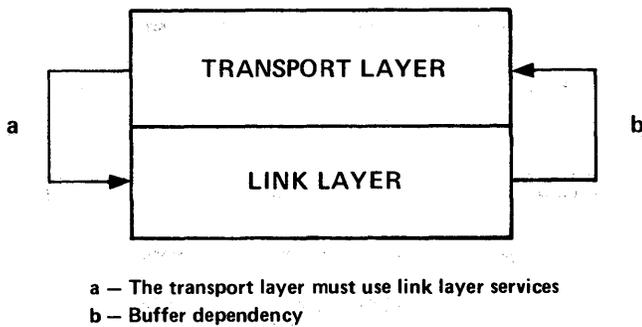


Figure 1—Dependencies in Deadlock 1

Now consider the following scenario. The transport protocol has a large number of unacknowledged messages in the buffer pool. Since its handler shares the same pool as the link protocol handler, the link protocol goes into a “flow control” mode, in which no more data messages are accepted from adjacent nodes. But, the buffer shortage will never be relieved until a transport layer acknowledgment is received, which is a data message to the link protocol. Hence, a deadlock exists.

We suggest two ways of dealing with deadlocks resulting from interlayer resource contention: prevention and minimization of occurrence.

With regard to prevention, since layered architectures cause higher layers to be dependent on lower ones, the goal of prevention is to insure that resource contention does not also make a lower layer dependent on a higher one. The use of directed graphs can aid in this examination.³ Figure 1 is an instance of such a graph. A directed cycle is indicative of a deadlock. To eliminate the cycle in Figure 1, one might design the system so that the transport layer will always (1) have a receive buffer available for the link layer or (2) provide one within a finite period of time.

While proper use of prevention will guarantee that deadlocks never occur, the analysis required can be complicated. An alternative for those resources that are explicitly allocated is to minimize the probability of a deadlock. This can be accomplished by employing usage thresholds. With specific reference to the problem of shared buffer pools, we may want to give priority to buffer requests made by the link receive handler over those made by the transport layer output handler. This policy can be enforced by insuring that once the free pool has fewer than B buffers remaining, further requests by the transport layer output handler will be rejected. In this way, one could minimize the possibility of outstanding transport layer transmissions preventing the receipt of a transport layer acknowledgment by the link layer.

DEADLOCK 2: LOWER LAYER RESOURCE TIMEOUT

This section discusses the effect of a higher layer being critically dependent on a resource allocated and deallocated by a lower layer. A common example of such a resource (and the attendant dependency) is that of transport layer connection ports (sockets in ARPA terminology). The dependency on transport layer connection ports results from the fact that only

by establishing a pair of connection ports can two processes communicate.

The manner in which transport layer connection ports are allocated and deallocated varies a great deal. Here, we assume that a transport layer connection abides by the following rules:

Rule 1. When a connection request arrives for a particular host and/or process, the transport protocol allocates a port and forwards the request with a unique port identifier to the appropriate host.

Rule 2. When the host and/or process responds to a connection request, the transport layer completes the connection and forwards the response to the requesting party.

Often, the transport protocol handler is in a separate computer from the host(s) it serves. So, there is considerable concern about insuring that a host failure or degradation does not have a severe impact on the communications network. With regard to connections, in particular, the communications network should guard against the host or host process being in an infinite loop. The standard approach to detecting failure in a communications environment is by use of a timeout. Thus, the communications network might add to its transport layer connection protocol the following rule:

Rule 3: If the host does not respond within a timeout period, the connection port is deallocated and a disconnect is returned to the requesting party. A host reply after the port has been deallocated is treated as an error.

Suppose we have a terminal that makes use of a communications network to access a host for time sharing services. We assume that the communications network transport layer connection protocol operates according to Rules 1–3. Furthermore, we assume that the time sharing system processes new logon requests by placing them in a low priority queue, thereby assuring users already logged on of a certain level of responsiveness. (If the system is too heavily loaded, new logons will not be performed.)

Now consider the following scenario. A user logs on to the time sharing system via the communications network. He starts a job and then logs off. This job forks several processes, each of which makes heavy CPU and I/O demands on the time sharing system. Even worse, because of a bug, each process goes into an infinite loop of these resource demands.

Realizing his mistake, the user tries to log on and kill the job. However, to do so he must use the services of the communications network (i.e., we have the layering service partitioning dependency). But, whenever the user attempts to log on to the network, the time sharing system will fail to reply before the transport layer connection timeout occurs, since the host is heavily loaded due to the user’s job. Thus, the user cannot log on until the time sharing system becomes less loaded. However, system load will not be reduced until the user can log on.* So, a deadlock exists.

*In commercial time sharing systems, it is often possible to set time limits on job execution. However, this is clearly an unsatisfactory way of resolving the deadlock, since the time requirements of jobs vary a great deal.

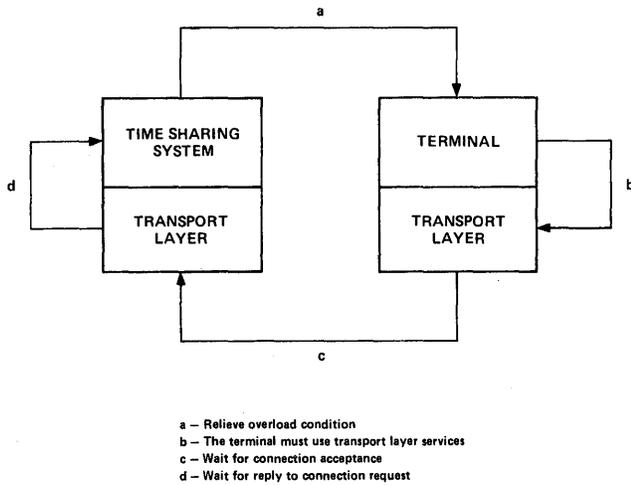


Figure 2—Dependencies in Deadlock 2

Solutions to this deadlock are not easy if the design goals of the time sharing system and the communications network are to be maintained, and layering is not to be violated. One approach would be to provide the user with a network command repertoire that includes the ability to override the standard transport layer connection timeout.

DEADLOCK 3: HALF DUPLEX LOWER LAYER

In addition to the problems presented in the previous two sections, if a lower layer protocol is half duplex, then still other difficulties may arise. By half duplex, we refer to protocols in which at one instant only a single party may transmit data messages, although either party may send control messages. The party permitted to transmit data is referred to as the line owner. A line owner relinquishes control of the line by performing a line turn around sequence.

With the proliferation of interactive systems, half duplex protocols have become less popular. However, half duplex protocols frequently have the advantages of simplicity of control (especially if the send and receive handlers share a lot of data) and often require fewer resources (particularly buffers and processes). This simplicity and need for fewer resources may be particularly attractive for microprocessor-based applications.

To show why caution should be employed when a half duplex lower layer protocol is used, we examine an architecture with two layers. The higher layer consists of a number of batch streams. The lower layer is a half duplex link protocol used by these batch streams. We assume that the batch stream protocol permits N outstanding transmissions before an acknowledgment is required. Also, if no acknowledgment is received within a timeout period, the unacknowledged messages are retransmitted. We further assume that the half duplex link protocol operates under the rule that line ownership (i.e., permission to transmit data) is not relinquished until all data messages have been transmitted. Of course, link layer acknowledgments are expected, and if they fail to arrive in a timely manner, link level retransmissions will be invoked.

Now consider the following scenario. Initially, only a few batch streams are present. So, once they transmit up to N messages, the link protocol releases control of the line so that batch stream acknowledgments may be received. (Recall that if this is truly a layered architecture, then batch stream acknowledgments are treated as data by the link handler.) However, as the number of batch streams approaches some number, say M , then the time to transmit $M*N$ messages becomes greater than the batch stream retransmission timeout, thereby causing batch stream retransmissions. Even worse, these retransmissions further delay performing a line turn around. Indeed, if M is sufficiently large, the link handler will never yield control of the line. In other words, the batch streams will persist with their retransmissions until batch stream acknowledgments are received. Since a batch stream acknowledgment is a data message to the link handler, a line turn around is required. However, a line turn around will not take place until the batch streams end their retransmissions. Hence, a deadlock exists.

One way to resolve this deadlock is to require the link protocol to release control of the line periodically. While this may reduce throughput under situations that would not lead to deadlocks, it preserves the independence of the two layers.

CONCLUSIONS

In this paper, we have presented some examples to demonstrate that caution must be used when layered communications architectures are designed. Specifically, since layering introduces a dependency in higher layers for the services of lower layers, care must be exercised to avoid deadlocks. Despite this potential problem area, we believe that layering remains an effective method for structuring large systems. However, its usage requires some caution.

REFERENCES

1. Bochmann, Gregor V., "Finite State Description of Communication Protocols," *Proceedings of Symposium on Computer Network Protocols*, Liège, Belgium, February 13-15, 1978, pp. F3-1-F3-11.

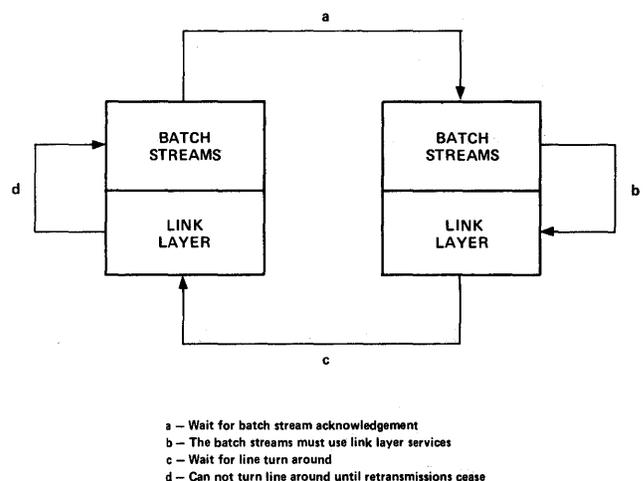


Figure 3—Dependencies in Deadlock 3

2. Chu, W. W., "Dynamic Buffer Management for Computer Communications," *Proceedings of the Third Data Communications Symposium*, Tampa, Florida, November 1973, pp. 68-72.
3. Chu, Wesley, and G. Ohlmacher, "Avoiding Deadlock in Distributed Data Bases," *ACM National Symposium*, Vol. 1, November 1974, pp. 156-160.
4. Crocker, S. D., et al., "Function-Oriented Protocols for the ARPA Computer Network," *AFIPS Proceedings*, SJCC, 1972, pp. 271-279.
5. Digital Equipment Corporation, "DECNET," 1976.
6. Gray, J. P., and C. R. Blair, "IBM's Systems Network Architecture," *Data-mation*, April 1975, pp. 51-56.
7. ISO TC97/SC16/N34, "Provisional Model of Open-Systems Architecture," March 1977.
8. Zimmerman, H., and N. Naffah, "An Open Systems Architecture," *Proceedings of the Fourth International Conference on Computer Communication (ICCC-78)*, Kyoto, September 1978, pp. 669-674.

General-purpose integrated indexing circuits—a proposal

by A.C.D. DE FIGUEIREDO

Universidade de Coimbra
Coimbra, Portugal

ABSTRACT

For applications requiring irregular forms of addressing, the high speed of operation of present day random-access memories and large-scale-integrated arithmetic structures can hardly lead to any significant improvements over conventional techniques unless appropriate means are used for addressing the operands at comparable speeds. This paper proposes the architecture for an integrated indexing unit intended for high-speed generation of the addressing patterns required for a wide range of such applications. Reference is made to the usefulness of the proposed unit for the organization of fast array processing attachments to conventional minicomputers and microcomputers.

INTRODUCTION

Whatever the approach one takes when organizing a processor intended for operating on arrays of data, it is helpful to differentiate two basic types of functions:

- *Data processing functions*, which relate to the arithmetic and logic operations performed upon or between arrays
- *Data manipulating functions*, which concern the organization or reordering of array elements that may be required before or after processing

A large body of knowledge has been accumulated over the past few years on the speed-critical aspects of data processing, such as computer arithmetic; and this has led to a number of successful implementations at the integrated-circuit level.^{5,7} The same cannot be said, however, of data-manipulating functions. Although one may find in the literature the proposal of hardware-controlled techniques for the organization of information,^{1,2} the general trend is still to relegate this task to the software, which makes it depend for many applications on rather involved and time-consuming address calculations.

The integrated indexing circuit proposed in this paper, which evolved from a hardware data-shuffling principle presented in de Figueiredo,² enables the generation, without any address calculations, of a large range of addressing patterns that permit the manipulation of an array to take place while the array is stored into or fetched from memory.

PRINCIPLE OF OPERATION

The principle of operation of the proposed indexing circuit can be easily described in terms of the simplified four-bit unit depicted in Figure 1, where the four multiplexers are used to permute the counter outputs according to some binary pattern that is applied to the multiplexer select inputs. If the counter is reset or preset, the select inputs of the multiplexers are driven with an appropriate control word, and the counter is then forced to count up or down, a sequence of binary numbers is obtained that may be used to address a random-access memory. For instance, if the outputs of the binary counter are permuted as shown in Figure 2 and the counter is forced to count from 0 to 15, the resulting addressing sequence causes the elements of a 16-word array to be stored or fetched in shuffled order.

The application of this *counter/multiplexer principle* to the generation of a variety of shuffled addressing sequences has been discussed in detail in de Figueiredo,² where it has been shown that a very significant saving of control words can be achieved if the higher-order outputs of the circuit can be masked so as to permit the use of a single control word for a number of different manipulations. The ability to mask one, two, three or more of the higher-order outputs is then one of the desirable features of an indexing circuit based upon this principle.

PROPOSED ARCHITECTURE

Figure 3 illustrates a block diagram description of the proposed indexing circuit, which is intended for accessing data memories with up to 64K words. Apart from a 16-bit counter and a set of sixteen 16-input multiplexers, the unit possesses a mask circuit that controls the inhibition of the multiplexer outputs and a set of eight 8-bit registers that hold the current control word. These registers are loaded from the outside through a common 8-bit bus, W_7 – W_0 , which is also used to carry preset information to the counter. The selection between each one of the eight registers and between the two 8-bit halves of the counter is achieved by means of a 4-bit address, S_3 – S_0 . The loading of information into the registers or into the counter is clocked by a LOAD signal. The mask circuit receives a 4-bit word, M_3 – M_0 , and decodes it into com-

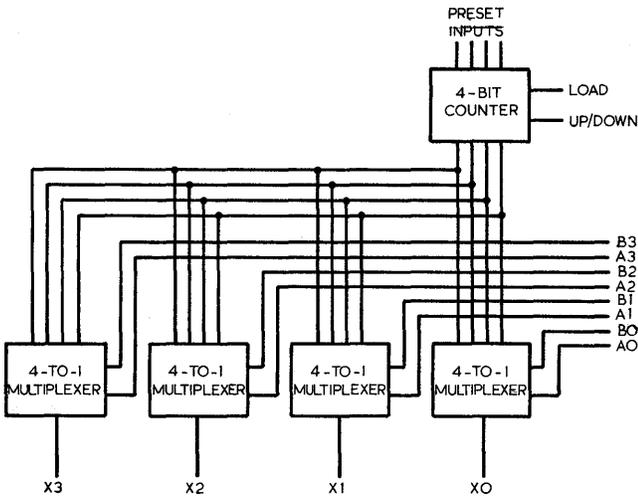


Figure 1—Simplified 4-bit indexing circuit

mands that inhibit output X_{15} , outputs X_{15} and X_{14} , outputs X_{15} to X_{13} , ..., outputs X_{15} to X_2 , all outputs, or none of the outputs.

Taking into account the inputs mentioned above plus the 16 outputs of the unit and the CLOCK, RESET and UP/DOWN inputs to the counter, the total number of signal pins for a large-scale-integrated implementation of this circuit amounts to 36, which means that the circuit can be accommodated within a standard 40-pin package.

A discrete TTL version of the proposed circuit has been implemented and has shown that the successive addresses in a sequence can be produced at a rate of over one address every 100 nanoseconds. Compared with conventional addressing techniques that rely upon the calculation of the successive addresses, this represents an improvement that may exceed two orders of magnitude.

The setting up of the proposed circuit may require the loading of eight registers, and in some cases the presetting of the counter. When the time available for this operation is not critical, the control words may be supplied by a general control microprocessor. Otherwise, a fast ROM lookup table may hold locally all critical control words, and a three-bit counter

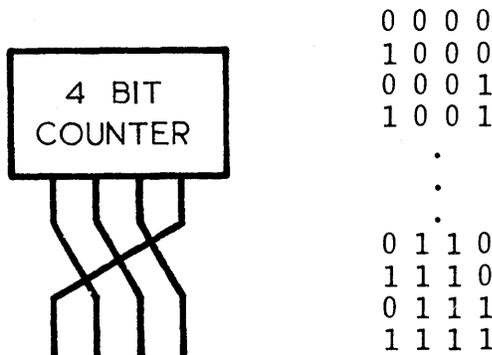


Figure 2—Generation of shuffled addressing patterns

must be used to supply the three least significant bits of the addresses to the ROM while simultaneously driving S_2 , S_1 , and S_0 .

APPLICATION

The use of the addressing patterns resulting from the permutation of the outputs of a binary counter constitutes a subject that remains to be fully understood and formalized. Apart from the aspects relating to the sequential generation of shuffled patterns, which have been the object of a study inspired by the parallel processing properties of the perfect shuffle,² very little seems to have been investigated so far. It is not surprising, therefore, that most of the known advantages of a unit that implements the counter/multiplexer principle result at present from its ability to produce shuffled addresses.

The applicability of the shuffle function is nevertheless extremely general. As shown by Stone for the case of parallel processors,⁶ the shuffle function can be exploited in a variety of applications, including the computation of the fast Fourier transform, polynomial evaluation, sorting, and matrix transposition. All these applications can be dealt with without much difficulty in a sequential processing environment, provided the proposed unit is used for the generation of the required shuffled addressing patterns.

One important application of the unit relates to the design of high-speed processing peripherals that upgrade the performance of general-purpose minicomputers and microcomputers by taking over the highly repetitive, computationally intensive array processing tasks. In these processing peripherals the counter/multiplexer principle can be used, not only for controlling the read and write transfers between fast data memory and high-speed arithmetic/logic unit, but also for implementing refined forms of direct memory access that enable the manipulation of a block of data to take place on the fly while a block is transferred between host computer and processing peripheral.^{3,4}

The number of indexing circuits that should be incorpo-

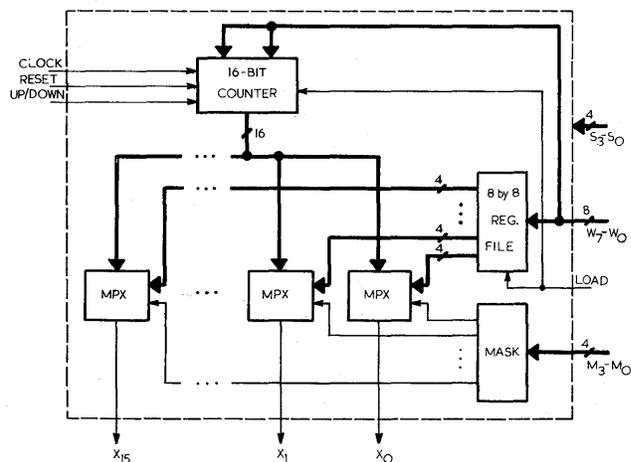


Figure 3—Block diagram description of a 16-bit integrated indexing circuit

rated in a processor is mainly dictated by the required performance. Although it can be stated, on the basis of experience acquired, that two units—one for fetching the operands and another for sending the results back to store—suffice for producing all forms of data manipulation required in standard array processing applications, it can easily be figured that much higher performances are likely to be obtained from a parallel type of architecture that employs a large number of such units.

CONCLUSIONS

A proposal has been presented for the architecture of an integrated indexing circuit that generates at very high speed a variety of addressing patterns required for array processing applications. Some suggestions have been put forward concerning the use of this circuit as a functional part of array processing attachments to small general-purpose computers.

Although the circuit can be used very effectively in applications requiring the shuffled manipulation of array elements, it is believed that the scope of its use can be enlarged considerably if some effort is put into the investigation of all relevant addressing patterns that result from the permutation of the outputs of a binary counter.

ACKNOWLEDGMENT

This work was supported by Instituto Nacional de Investigação Científica under Grant EC1 and by Junta Nacional de Investigação Científica under Research Contract No. 108.79.53.

REFERENCES

1. Feng, T. "Data manipulating functions in parallel processors and their implementations." *IEEE Trans. Computers*, C-23 (1974), pp. 309-318.
2. de Figueiredo, A.C.D. "High-speed addressing technique for digital signal processing." *Proc. of the 1979 Int'l. Symp. on Circuits and Systems*, Tokyo, Japan (New York, USA: IEEE 1979), July 1979, pp. 977-978.
3. de Figueiredo, A.C.D. "Towards a generalized concept of direct memory access." *Abstracts of the 1980 ACM Comp. Sc. Conf.*, Kansas City, USA, Feb. 1980, p. 46.
4. de Figueiredo, A.C.D., E. Sá Marta, and J.G.C. e Silva. "Fast transposition technique for microprocessor array processing attachments." *Proc. of EUROMICRO 80*, London, Sept. 1980, pp. 25-30.
5. Schirm, L., IV. "Packing a signal processor onto a single digital board." *Electronics*, Dec. 20, 1979, pp. 109-115.
6. Stone, H. "Parallel processing with the perfect shuffle." *IEEE Trans. Computers*, Feb. 1971, pp. 153-161.
7. Waser, S. "High-speed monolithic multipliers for real-time digital signal processing." *Computer*, Oct. 1978, pp. 19-29.

The VALI (Variable Language Interpreter)

by JAMES D. MOONEY

West Virginia University
Morgantown, West Virginia

ABSTRACT

The Variable Language Interpreter (VALI) is a high-level language computer architecture. The language definition is not fixed, but can be easily changed to process many popular languages. The languages may be complete, so portable programs can be handled.

VALI makes use of parallel processors to achieve its objectives. Parsing is carried out by an array of identical units called token processors. These generate a high-level intermediate form called an execution tree. This tree is then interpreted by additional processors, exploiting its parallelism whenever possible.

A version of this design has been simulated. Suitable implementation methods are discussed.

INTRODUCTION

The Variable Language Interpreter (VALI) is a high-level language machine (HLLM) in which the language definition can be easily changed. HLLMs make use of hardware solutions to improve program throughput, especially in compiler-intensive applications. Greatly simplified system software and more opportunities for run-time checking are usually added benefits. The cost is not negligible, but it can be justified if the system is of benefit in a wide range of applications. Current HLLMs have been criticized in part for the rigidity of the languages they can process. With VALI, the language is defined by tables in memory, and the definition can be changed or extended at any time.

HLLM architectures have been studied frequently. Bashkow et al.¹ designed an early FORTRAN machine. The SYMBOL² machine was built and used commercially. Chu³ has done extensive work and published a textbook. Carlson⁴ surveys the field through 1974. A recent conference⁵ has produced additional proposals. This is but a small sample of the work in this area.

Proposals exist for many languages. However, most machines have been designed to implement a single language, and they are not easily adaptable to others. Moreover, the language accepted is usually a specially designed one or a

subset of a popular language. Thus the application areas are limited and portability of programs is not available.

In 1975 Fournier⁶ proposed a system in which the language is defined by an interpretive microprogram structured as a syntax network. In 1977 the author⁷ proposed a design with passive tables as the language representation. VALI is an evolution of this previous, unpublished system. By use of a flexible language representation, a variety of popular languages can be interpreted. Implementation of FORTRAN and ALGOL 60 was described earlier.⁷ A complete PASCAL definition has been prepared, and an ADA definition is under development.

The main components of the VALI system are shown in the block diagram of Figure 1. A set of language definition tables (LDTs) is initially loaded to specify the language to be used. Program source text then enters the analysis section as an input stream. The heart of the analysis section is an array of identical processing units called Token Processors (TPs). These units scan and parse the text in parallel, generating a high-level intermediate form called the execution tree or E-tree. The E-tree is interpreted directly by the execution section. An array of Operand Evaluators (OEs) traverses the tree, extracting semantic actions to be performed. A collection of Execution Processors (EPs) carries out these actions, thereby executing the program.

Each of these sections communicates with a main store managed by the Storage Control Unit. This unit facilitates access to stacks, queues, associative tables, and other high-level data structures.

The remaining sections of this paper will discuss the parsing mechanism, language representation, E-tree generation and processing, and implementation and use of the system.

SYNTAX AND PARSING

The job of the analysis section is to scan and parse the program text, using the LDTs, and generate an E-tree. In the following we describe the representation of syntax and the parallel parsing method of the TPs.

Sequential parsing algorithms have been studied extensively. Well-known methods include general backtracking, a

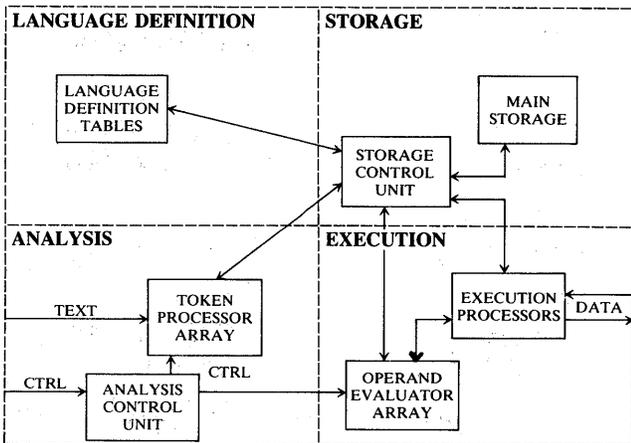


Figure 1—VALI system block diagram

recursive descent,⁸ predictor-corrector,⁹ LL(k),¹⁰ LR(k),¹¹ and others. Each method is applicable to certain broad subsets of context-free grammars (CFGs). They differ in the specific subset, processing speed, complexity of the tables that must be produced, etc.

The parsing process must deal with the fact that at any point in the input stream there may be several interpretations of the current symbol that could lead to a valid program. Sequential methods either try the possibilities one at a time, backing up in case of failure, or look ahead some number of symbols and require the grammar to contain enough information in those symbols to resolve the meaning of the current one. The latter method is often favored; e.g., an LR(1) grammar is derived for the language, the required parse tables are generated, and the LR algorithm is used. This has been feasible for most popular languages.

Although LR(1) grammars can usually be found, they are not necessarily the most natural grammars for a language nor those which most easily model the desired semantics. They must also be represented by large tables whose relation to the language is obscured. Backtracking and recursive descent can often use very natural grammars directly. However, these methods are not used because they are very slow, and the complexity of backing out of all the effects of a wrong try can be unmanageable.

Parallel parsing provides an additional option. Instead of trying possibilities and backing up, we may try *all* possibilities at the same time. This is the option explored here for VALI. In this approach, one processor keeps track of each possible parse at any time. At most one of these parses will survive, but we do not greatly care how long the others keep trying.

The syntax of a language for VALI is described initially by a set of syntax graphs (SGs) similar, for example, to the transition diagrams described by Aho and Ullman.¹² Each arc is stored in memory as an ordered pair of nodes and an associated symbol. No further manipulation is needed for parsing purposes. Terminals in this grammar are tokens in the language. The tokens in turn are described by separate tables that have all the information necessary for lexical scanning.

During parsing each active TP is given a goal token. Its task

is then to see if such a token appears next in the input stream. The token may be a single character or a string. Each TP maintains a stack of information describing its current position in the syntax graphs. If the token is found, the TP uses this stack and the LDTs to determine what next transitions are possible. For each such transition it activates a new TP with the appropriate goal symbol. There is no problem if several of the transitions have the same goal. An updated control stack is passed to each new TP. These stacks are linked with pointers to their common base to avoid unnecessary copying.

If the original token was not found, the TP fails and becomes inactive. The process continues until no TPs are active or until one TP reaches the final node of the root syntax graph.

The logic required of a TP seems complex; however, because most actions are table-driven, a rather simple processor is sufficient, and all can be of the same design. The basic TP flowchart is shown in Figure 2. This flowchart includes the semantic functions discussed below. A possible TP implementation is discussed in a later section.

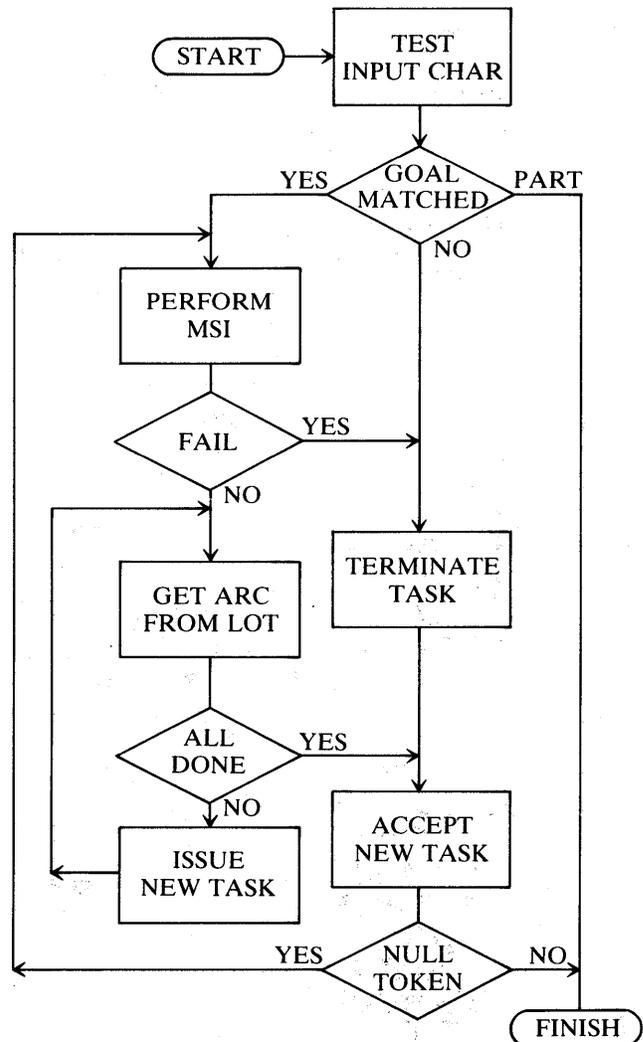


Figure 2—Token processor flowchart

We are thus providing a variation of recursive descent, yet it is recursive descent in which the correct productions will always be (among) the first chosen. The method does not backtrack, yet it has the same power as general backtracking. It accepts the same class of grammars: any CFG that is not left-recursive. Left recursion is both easier to discover and easier to correct than a non-LR(1) property. It can always be removed. In contrast to the exponential time characteristics of backtracking, VALI will accept—or reject—any string in linear time.

The tradeoff concerns the maximum number of TPs that may be needed. This number depends on the grammar, not on the program text. Formal grammars can be constructed for which the number of TPs is unbounded. For known programming languages, however, the limit has always been small ($< < 100$). By extending the power of the TPs, the limit may be made smaller still; however, as discussed below, this may not always be desirable.

THE EXECUTION TREE

The output of the analysis process is an E-tree which represents the semantics of the program. We now describe the tree structure and the way it is generated by the TPs.

An E-tree is a connected tree of cells with no cycles and a single root. A cell is a sequence of fields. The contents of a field may be a value, a control code, or a link to another cell. The tree may be augmented with links to groups of cells that will be processed as a table.

Cells are of two semantic types. Active cells represent actions to be performed. They contain a function designator in the first field and operands in subsequent fields. In general, the operands may be evaluated in any order. Special function types—e.g. SEQUENCE, CONDitional—are introduced to control the order or choice of evaluation when required. Passive cells represent data objects. They contain an object class designator as their first field and component information in subsequent fields.

As generated by the analysis section, an E-tree embodies the static semantics of the program text. A simplified E-tree is shown in Figure 3. This tree models a program to select the largest element in a one-dimensional array. A complete tree contains aspects such as name tables and typed data items that are not shown in the figure.

To generate the tree, each arc in the SGs may be augmented with one or more metasegment instructions, whose function is to create and update cells in a subtree. As far as possible, the grammar is chosen so that distinct token sequences always correspond to specific developments in the tree. Each path through a particular SG leads to generation of a distinct subtree; this subtree is attached to one at a higher level when and if the SG is exited after successful recognition.

An E-tree in this form can be saved and archived. This would be appropriate, for example, for a library of subroutines. More often, however, the tree is passed directly to the execution section for interpretation.

Execution of the program is achieved by evaluating the root cell of the tree. This leads to recursive evaluation of other

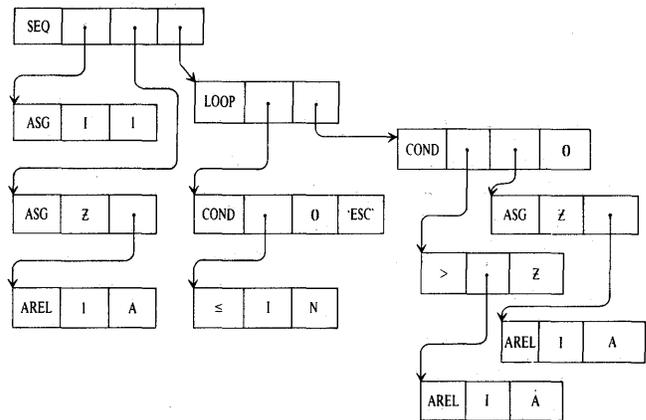


Figure 3—Example of an execution tree

cells, and this process is controlled by the OE array. An active cell is evaluated by first evaluating each of its operands, then applying the specified function to yield a result. A passive cell is evaluated by first evaluating each of its components, then composing the object as required.

Initially, an OE is assigned to the root cell. This OE attempts to evaluate the operand fields, which it can do for simple values and links to passive cells. When a link to an active cell is encountered, the OE assigns its evaluation to another OE, then waits for the result. Meanwhile, it continues to process the available operands.

All the OEs are identical. The number of OEs required, in contrast to the TPs, depends on the depth of the tree and thus on the complexity of the particular program.

When all operands are available for an active cell, the function is applied. This is a job for an EP, which is specialized to handle the particular function. The EP applies the necessary algorithm and passes the result to the OE for propagation up the tree. When a result is achieved for the root cell of the tree, the program has been fully executed.

During execution, the E-tree changes dynamically: Name tables are developed, user data are attached and modified, etc. The tree thus models the complete environment of the program at any time while it is executing.

IMPLEMENTATION

The VALI system components discussed here consist primarily of the TP and OE arrays and the EPs. The first two of these share many characteristics. They consist of moderate numbers of identical processors, operating cooperatively but independently on the same or related data. They must communicate with each other and with external units. The degree of interconnection can be traded off against processing speed.

One version of each of these units has been defined to the level of register and signal equations. These definitions are available elsewhere.⁷

These arrays can be constructed initially from available logic or perhaps microprocessor chips. However, they are

reasonable candidates for VLSI implementation. In this case we would favor the simplest logic in each unit, although a larger number may be required. Several tradeoffs of this nature are available. Several TPs or OEs could reside on a chip; eventually a complete array could be placed on a single chip, greatly easing the pinout problem. A generalized parser and interpreter would have broad application that should make this approach feasible.

The EPs require a different approach. They are intended to perform a variety of specialized functions. The functions vary in complexity—e.g., arithmetic, name table processing, I/O control. The collection should be extendible from time to time as additional hardware functions become cost-effective. For generality, a means of providing arbitrary algorithms is required. All of this leads to a mixed implementation approach, in which the functions are provided by means ranging from specialized chips to conventional software simulation. Despite this, all functions can be accessed by common interface conventions; therefore the implementation may be changed without affecting the rest of the system.

A memory system with many interesting features is a goal for the future. Initially, the memory can be based on well-known techniques.

CONCLUSIONS

We have presented an architecture for an HLLM that differs from its predecessors in degree of parallelism and of generality. It can be configured to interpret portable programs in most popular programming languages.

The system has been simulated for a simple language to demonstrate its feasibility. This simulation is described in detail elsewhere.⁷ A more extensive simulation implementing PASCAL and ADA is in progress.

High-level-language machines can give considerable gains in processing speed for compiler-intensive applications, e.g., program development, one-shot problem solving. The cost can be most easily justified when the possible application area is broad. One way to insure this is to provide processing for many languages, especially those in common use. VALI takes a step in this direction.

The per-copy cost of implementing the TP and OE arrays can be low if VLSI methods are used. The chip design costs, currently high but being attacked in various ways, can be

offset by the broad applications of the device. The implementation costs of the EPs span a spectrum, in which performance may be traded for economy.

Finally, the analyzer is driven by a straightforward and uniform type of language specification, which incorporates syntax and semantics and leads to a common intermediate form. This will motivate the definition of a variety of languages in this form, which may then be of interest in other investigations.

ACKNOWLEDGMENTS

I am grateful for fruitful discussions with Bill Wulf of Carnegie-Mellon University and with my colleagues Frances Van Scoy and Malcolm Lane, which helped to clarify some of the ideas in this paper.

REFERENCES

1. Bashkow, T. R., A. Sasson, and A. Kronfeld. "System Design of a FORTRAN Machine." *IEEE Transactions on Electronic Computers*, EC-16 (1967), pp. 485-499.
2. Rice, R., and W. R. Smith. "SYMBOL—A Major Departure from Classic Software Dominated von Neumann Computing Systems." *AFIPS Proceedings of the Spring Joint Computer Conference*, Vol. 38 (1971), pp. 575-587.
3. Chu, Y. (Ed.). *High Level Language Computer Architecture*. New York: Academic Press, 1975.
4. Carlson, C. R. "A Survey of High-Level Language Computer Architecture." In Y. Chu (Ed.), *High Level Language Computer Architecture*. New York: Academic Press, 1975.
5. University of Maryland. *Proceedings of the International Workshop on High Level Language Computer Architecture*. Department of Computer Science, University of Maryland, 1980.
6. Fournier, S. "The Architecture of a Grammar-Programmable High Level Language Machine." Ph.D. dissertation, Ohio State University, Columbus, Ohio, 1975.
7. Mooney, J. "Design of a Variable High Level Language Computer Using Parallel Processing." Ph.D. dissertation, Ohio State University, Columbus, Ohio, 1977.
8. Conway, M. E. "Design of a Separable Transition Diagram Compiler." *Communications of the ACM*, 6 (1963), pp. 396-408.
9. Earley, J. "An Efficient Context-free Parsing Algorithm." *Communications of the ACM*, 13 (1970), pp. 94-102.
10. Lewis, P. M., and R. E. Stearns. "Syntax-Directed Transduction." *Journal of the ACM*, 15 (1968), pp. 465-468.
11. Knuth, D. "On the Translation of Languages from Left to Right." *Information and Control*, 8 (1965), pp. 607-639.
12. Aho, A. V., and J. D. Ullman. *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley, 1977.

The architecture of MANIP—a parallel computer system for solving NP-complete problems

by BENJAMIN W. WAH

Purdue University
West Lafayette, Indiana

and

Y. W. MA

University of California, Berkeley
Berkeley, California

ABSTRACT

In this paper, we study the network architecture of MANIP, a parallel Machine for processing Non-deterministic Polynomial complete problems. The most general technique that can be used to solve a wide variety of NP-complete problems on a uni-processor system, optimally or sub-optimally, is the branch and bound algorithm. We have adapted and extended the branch and bound algorithm for parallel processing. The parallel branch and bound algorithm requires a combination of sorting and merging. A common memory to sort for a large number of processors can become a bottleneck in the system. We have proposed a system with distributed intelligence so that sorting can be carried out in a distributed fashion. A uni-directional ring network is proved to be the optimal and most cost-effective inter-processor communication network when sorting is done by a hardware priority queue in each processor.

I. INTRODUCTION

A class of common, deterministic problems defined in computer science, operations research, and other application areas is the *NP-complete problems*.³⁵ This class of problems is characterized by a deterministic algorithm that computes a function from a countable domain into a countable range, and it generally involves the optimization of an objective function. The computation time for all known optimal algorithms for this class of problems increases exponentially with the problem size, i.e., if n represents the size of the problem, then the computation time goes up as k^n where $k > 1$. There is a subclass of NP-complete problems called *strong NP-complete problems*¹⁶ such that there is no "pseudopolynomial" algorithm which solves the problem in a time bounded by a polynomial in the input length and the magnitude of the largest number in the given problem instance. The implication of a problem being strongly NP-complete is that there is no *fully*

polynomial time approximation scheme which solves the problem in a time bounded by a polynomial in the input length and the reciprocal of the prescribed degree of accuracy. Many problems in areas like deterministic scheduling, graph theory, routing, database, mathematical programming, automata and language theory, image processing, microprogram optimization, etc., have been proved to be either NP-complete or strongly NP-complete.¹⁷ The set of NP-complete problems therefore spans a wide spectrum of application areas.

We are presenting in this paper the architectural design of a parallel computer system that can be used to solve *NP-complete problems without fully polynomial time approximation schemes*. Since the time complexity to solve these problems optimally is exponential, the common approach is to solve optimally for small problems and to solve sub-optimally using heuristics for large problems. The most general technique that can be used to solve a wide variety of these problems, optimally or sub-optimally, is the branch and bound algorithm.⁴⁴ The branch and bound algorithm will be discussed in detail in Section II. Conventionally, the branch and bound algorithm has generally been studied with respect to limited memory space, the selection and bounding criteria, the theoretical behavior, and the adaptation to a single computer system. What little work that has gone on from the viewpoint of parallelism has been directed toward a general purpose computer network. The problem of the necessary parallel computer architecture and its associated operating system to provide an execution environment for a branch and bound algorithm has been little studied or less understood. The significance of this study therefore lies in two aspects. First, it can result in the design of a special purpose VLSI parallel computer system to execute the parallel branch and bound algorithm. The number of computers can be designed to fit the need of the applications. Second, with a better understanding of the parallel branch and bound algorithm, it can be designed into existing computer networks and distributed computer systems.

The feasibility of this study has greatly increased with re-

cent changes in the state of the art in memory, VLSI, and communication technologies. The cost per unit of memory is decreasing and a wide variety of new storage devices, such as CCD memory, bubble memory are available. A number of other technologies are undergoing intensive study, including holographic, laser and other optical, and magneto-optical, and have the potential for commercial development within the next decade. At the same time, the number of components per chip is doubling each year and there is a trend of increased specialization in the functions of the VLSI chip^{15, 40} and the design of a single chip computer system.⁵⁵ A conference was held in Caltech in 1979 to investigate the potential of VLSI technology.⁶² Lastly, the improvement in wideband communication technology allows local or remote computers to be interconnected together using optical fibers and satellites.

With the economic feasibility and consequent existence of these new technologies, more powerful search strategies can be used in the branch and bound algorithm. Traditional implementation of branch and bound algorithm is faced with the problem of limited memory size. With larger and inexpensive secondary storage, the branch and bound algorithm can be designed with a virtual backing store. Candidate problems unlikely to lead to the optimal solution can be stored in the secondary storage. The conventional virtual memory system does not work very efficiently here because the access characteristics of a branch and bound algorithm are significantly different from the access characteristics of a program. The complexity of the problem is compounded as parallel computers are used. Another problem faced in the efficient implementation of branch and bound algorithm is sorting. In order for the execution time to be minimum, single processor implementation sorts the intermediate sub-problems by the lower bounds in ascending order and the sub-problem with the minimum lower bound is picked up for expansion (best first search). Other heuristic search strategies may also involve searching through a set of values generated by heuristic functions.²⁹ In a parallel computer system, the requirement that the global set of subproblems are completely sorted by lower bounds can be relaxed. Suppose there are n processors, it is sufficient to place one of the n sub-problems with minimum lower bounds in each processor and not important which one of these n sub-problems is evaluated in a particular computer. This relaxed sorting requirement can be incorporated into the design of a more efficient architecture than conventional architectures that perform complete sorting.

In this paper, we study the network architecture of MANIP, an architecture using VLSI technology to implement a parallel branch and bound algorithm. We want to design special purpose processors for evaluating the bounds and simple interconnection network for interconnecting the processors. The system is designed with the following design objectives: First, the system should be modularly expandable to include a very large number of processors. Second, the design must have high performance and the cost should be low by replicating simple cells. Third, the system should use distributed control so that there would not be a controller that becomes the bottleneck in future system expansion. Fourth, efficient load balancing strategies should be implemented so that the processors can be kept busy most of the time. Lastly, the system should be recoverable from hardware failures.

This paper is divided into five sections. Section II presents the branch and bound algorithm and the parallel version of the branch and bound algorithm and discusses the previous work on parallel computer architecture for branch and bound algorithm. Section III identifies the architectural alternatives in implementing the parallel branch and bound algorithm. Section IV presents the network architecture and its optimality. Lastly, section V provides some discussions on the problem of implementation, the performance of the network, and gives some concluding remarks.

II. PREVIOUS WORK

A. Parallel Branch and Bound Algorithm

An NP-complete problem is usually put into the form of a constrained optimization problem*:

$$\begin{array}{ll} \text{minimize} & C_0(x) \\ \text{subject to} & g_1(x) \geq 0 \\ & g_2(x) \geq 0 \\ & \vdots \\ & g_m(x) \geq 0 \\ \text{and} & x \in X \end{array}$$

where X represents the domain of optimization defined by the m constraints, normally an Euclidean n -space, and x denotes a vector (x_1, x_2, \dots, x_n) . A solution vector that lies in X is said to be a *feasible solution* and a feasible solution for which $C_0(x)$ is minimal is said to be an *optimal solution*.

Many methods exist to solve for the optimal solution in the aforementioned optimization problem. Some of these are specially designed techniques like Gomory's cutting plane method for solving integer programming problems. However, the most general algorithm, although sometimes not the most efficient, is the branch and bound algorithm. In this section, we describe the branch and bound algorithm and expand the algorithm into a parallel version so that it can be implemented on a parallel computer system.

1. Previous work on branch and bound algorithm

The branch and bound algorithm is an organized and intelligently structured search of the space of all feasible solutions. It has been extensively studied in areas such as artificial intelligence and operations research.^{27, 44, 48, 53} It has been applied extensively to solve problems in scheduling,^{41, 46} knapsack,^{33, 34} traveling salesman,^{18, 24, 25} facility allocation,^{11, 59} integer programming,^{19, 20} and many others. Dominance relation similar to that used in dynamic programming has been used to prune search tree nodes.

Theoretical properties of the branch and bound algorithm have been developed in several studies.^{28-32, 37, 53} One study²⁹ shows that depth-first search, breadth-first search and best-

* There are also problems which are not NP-complete and are put into this form.

first search are special cases of heuristic search. In heuristic search, an evaluation function $f(n)$ for a sub-problem n is computed as the sum of cost of an optimal path from a given start node to n and cost of an optimal path from n to a goal. An ordered search algorithm picks up a sub-problem with the minimum value of f for expansion each time. Any general heuristic functions can be included in the computation and the choice of a heuristic function depends on the application.

2. Essential features of branch and bound algorithm

In branch and bound algorithm, the space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets and both the lower and upper bounds are calculated for the cost of solutions within each subset. After each partitioning, these subsets with a lower bound (in the case of minimization) that exceeds either the cost of a known feasible solution or the least upper bound of all the subsets, are excluded from all further partitioning. The partitioning process continues until a feasible solution is found such that the cost is no greater than the lower bound for any subset. The state of the partitioning process at any time can be represented as a partial tree (Figure 1). Each node in the tree represents a partition and is termed *sub-problem*. The partitioning process selects a partition and breaks up this partition into smaller partitions which in essence extends the node in the partial tree representing this partition by one level and using the sons to denote the smaller partitions. In Figure 1, node j is expanded in the partitioning process into k other partitions which are represented as sons of node j in the partial tree.

There are two essential features of a branch and bound algorithm, namely, the branching rule and the bounding rule. Let us discuss these with respect to the tree in Figure 1. Each node in the partial tree has two numbers associated with it—the upper bound and the lower bound of the sub-problem. The leaf nodes in the partial tree are candidates for partitioning. We say that a leaf node of the partial tree whose lower bound is less than both the value of a known feasible solution and the greatest upper bound of all leaf nodes is *active*; otherwise it is designated as *terminated*, and need not be considered in any further computation.

The branching algorithm examines the set of active leaf nodes and selects one for expansion based on some pre-defined criterion. If the set of active nodes is maintained in a first-in-first-out (FIFO) list, the algorithm is called a *breadth-first search*. If the set is maintained in a last-in-first-out list, then the algorithm is termed *depth-first search*. Lastly, if the node selected for expansion is one with the minimum lower bound, then the search algorithm is called a *best-first search*. In a breadth-first search, the nodes of the tree will always be examined in levels; that is, a node at a lower level will always be examined before a node at a higher level. This search will always find a goal node nearest to the root. However, the sequence of nodes examined is always predetermined and therefore the search is “blind.” The depth-first search has a similar behavior except that a sub-tree is generated completely before the other sub-trees are examined. In both of these algorithms, since the next node to be examined is known, the state of the parent node leading to the next node

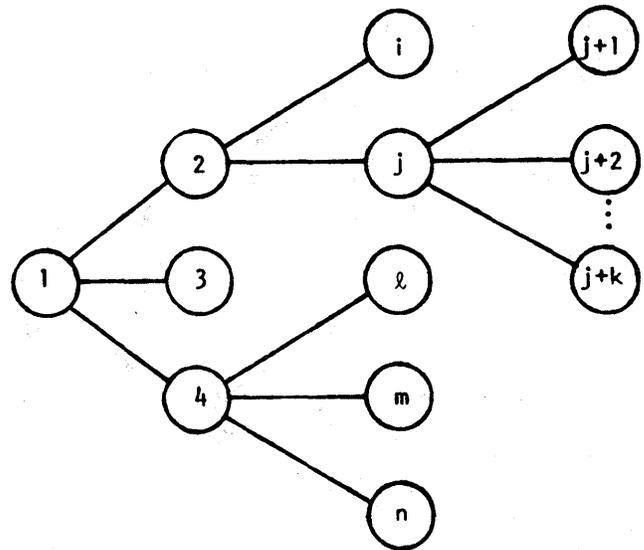


Figure 1—A branch and bound tree

does not have to be kept because the path to the next node from the root node is easily found and unique. These two algorithms are therefore somewhat space-economical. On the other hand, the best-first search is space consuming because all the active sub-problems must be stored as intermediate data in the computer. However, the total number of nodes expanded is minimized in the sense that any branching operation performed under this policy must also be performed under other policies, provided that all the bounds are unique.⁴⁴ Since time is a critical factor in evaluating large NP-complete problems, we will implement the best-first branching algorithm in MANIP. The large intermediate storage problem can be solved by moving sub-problems with large lower bounds to the secondary storage.

Once the sub-problem has been selected for partitioning, the next task is to select some undetermined parameters in the sub-problem in order to define alternatives for these parameters and create multiple sub-problems. For example, in the traveling salesman problem, the undetermined alternatives are the set of untraversed edges. In expanding a sub-problem, an untraversed edge (i, j) is selected and two alternatives can be created, namely, the edge is traversed and that the salesman goes directly from city i to city j and vice versa. The parameter chosen to be expanded is usually done in a rather ad hoc fashion.

After new sub-problems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a sub-problem. In general, only the lower bound is evaluated because the merit of using the upper bound is very small. The bounding algorithm designed is highly dependent on the problem. For example, in an integer programming problem, a linear program with the integer constraints relaxed can be used as a lower bound;⁴³ in a traveling salesman problem, an assignment algorithm¹⁰ or a spanning tree algorithm can be used as the bounding algorithm. We present an example of an NP-complete problem, the vertex covering problem¹⁷ in order to illustrate the parallel branch and bound algorithm.

In the *vertex covering problem*, the problem is to find, in an

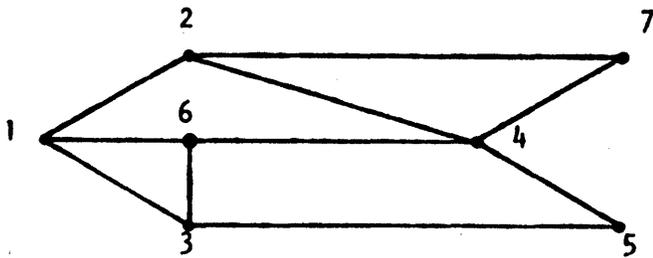


Figure 2a—An example graph

undirected graph, the minimum number of vertices that are needed to “cover” all the edges in the graph. (Cover means that all the edges in the graph emanate from at least one of the included vertices.) The branching rule uses the best-first search and branches on an unselected vertex with the largest out-degree. Two sub-problems can be created, one including this vertex in the set and one excluding it. The lower bound in the bounding rule is chosen to be the minimum number of unselected vertices such that the total out-degree is greater than or equal to the number of uncovered edges. Notice that edges emanating from different vertices in the lower bound calculation may overlap and therefore this vertex does not necessarily cover all the uncovered edges. Further, if a vertex has been excluded in a previous stage and there are uncovered edges emanating from this excluded vertex in the current sub-problem, the unselected vertex covering these edges must be included in the minimal set first. As an example, the branch and bound tree for the graph in Figure 2a is shown in Figure 2b.

3. The parallel branch and bound algorithm

We identify three sources of parallelism in the branch and bound algorithm.

a. *Parallel evaluation of subproblems.* Since multiple sub-problems are available, they can be evaluated simultaneously. Due to overheads in inter-processor communications and sorting, and because some sub-problem evaluations are unnecessary, the improvement in execution time is usually less than n times (n is the number of processors). For example, Figure 2c shows the parallel evaluation of branch and bound algorithm on the graph in Figure 2a using two processors. It is seen that the parallel evaluation of node 2 in Figure 2c is not useful, since the corresponding node 3 in Figure 2b is not evaluated. When the problem size is large, the parallelism will contribute to better improvement in execution time.

b. *Parallel sorting of subproblems.* In the best-first search, the list of sub-problems must be maintained in a sorted order by the lower bounds. This sorting can be done by parallel architecture such as Batcher’s sorting network.² In Section III we give a discussion on the type of interconnection network required for parallel sorting.

c. *Parallel execution of the bounding algorithm.* Specially designed architecture can be used to implement a bounding algorithm. For example, if the simplex algorithm is used, then matrix manipulation hardware is helpful. However, the archi-

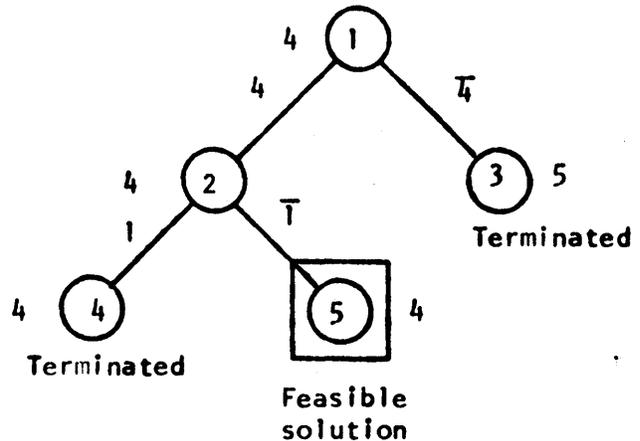


Figure 2b—The branch and bound tree for Figure 2a. (The number in the node indicates the order of evaluation; the number outside the node indicates the lower bound, the number on the edge indicates the included or excluded node.)

ture is designed for solving general NP-complete problems, therefore the bounding algorithm has to be changed for different problems. In this case, software implementation of the algorithm is more cost effective.

4. Efficiency considerations

Many results have been proved for the non-parallel version of the branch and bound algorithm.^{28, 31, 37} It has been shown that the best-first search is the best branching rule and minimizes the number of sub-problems expanded.⁴⁴ Furthermore, the branch and bound algorithm can be used as a general purpose heuristic to compute solutions that differ from the optimum by no more than a prescribed amount.⁴⁴ Suppose it was decided at the outset that a deviation of 10% from the optimum is tolerable. If a feasible solution of 150 is obtained, then all sub-problems with lower bounds of 136.4 or more ($= 150/1.1$) will be terminated. This technique significantly reduces the amount of intermediate storage and the time to

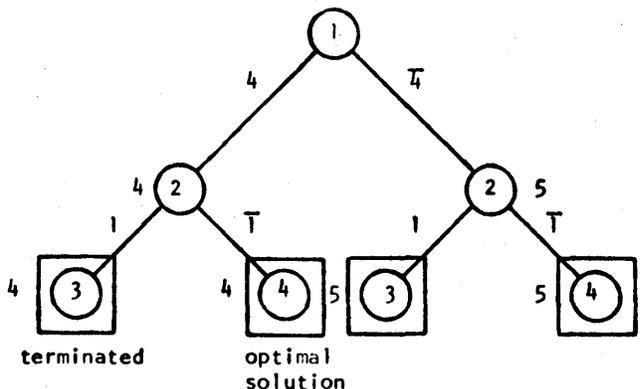


Figure 2c—The parallel branch and bound tree for Figure 2a with two processors

arrive at a sub-optimal solution. Technique is also available to find the best solution in a given length of time.⁴⁴ It consists basically of searching for an optimal solution for a length of time equal to $T/2$. If one is not found, then search is continued for a sub-optimal solution that differs from the optimal by no more than 5% in time of length $T/4$. The time for searching is halved each time while the precision of the solution is reduced until a solution is found. All these can be incorporated into the parallel branch and bound algorithm.

Unfortunately, very little can be said about the efficiency of the parallel branch and bound algorithm. It was found in one study⁴⁴ that only those sub-problems with lower bounds smaller than the optimal solution will be evaluated in a branch and bound algorithm. For a parallel branch and bound algorithm, the improvement in execution time will be n times (n is the number of processors) if the number of sub-problems in the intermediate list with lower bounds smaller than or equal to the optimal solution is always greater than or equal to n . However, this number is highly dependent on the problem and the partitioning being carried out earlier. Simulations are used to find the speed improvement using n processors.

B. Parallelism in NP-Complete Problem Evaluation

Many studies have been made to design multiple computers to speed up problems in searching. Kuck³⁹ has provided a survey on using parallelism to evaluate arithmetic expressions and linear recurrences, and execute programs. Tree structured architectures are proposed to solve problems in searching⁶ and database.⁶³ One of the tree architectures proposed to solve a wide variety of problems is the X-tree.⁷ Alpha-beta algorithm has been proposed to be evaluated on a tree architecture¹ and a general purpose network computer.¹⁴ Decision tree evaluation is also speeded up by using associative processors.⁴⁷ A variety of SIMD and MIMD interconnection networks have been proposed for processor-processor communication or processor-memory communication. Examples of these include Benes,⁵ indirect binary n -cube,⁵⁶ banyan,²¹ STARAN's flip network,³ Omega,⁴⁵ data manipulator,¹³ ILLIAC IV's mesh,³⁸ perfect shuffle,⁶⁴ PM21,⁶² delta,⁵⁴ reverse exchange network,⁶⁷ etc. However, these networks are usually designed for general purpose applications and therefore the necessary features for processing NP-complete problems are not identified. Our study identifies the necessary architectural features and therefore would provide insights to evaluate NP-complete problems on these computers.

Harris and Smith²² proposed a tree architecture to solve the traveling salesman problem. Basically, the system dedicates one subproblem to each processor and this processor reports to its parent processor when the evaluation is complete. Because of the limited degree of communication, some processors may be working on tasks that can otherwise be eliminated if a better interconnection network is designed. Desai^{8,9} also proposed a staged MIMD system to solve an 0-1 integer program using implicit enumeration. Nevertheless, implicit enumeration is time consuming and wasteful, and for NP-complete problems, the critical issues of exponential space or exponential time must be addressed in the algorithm.

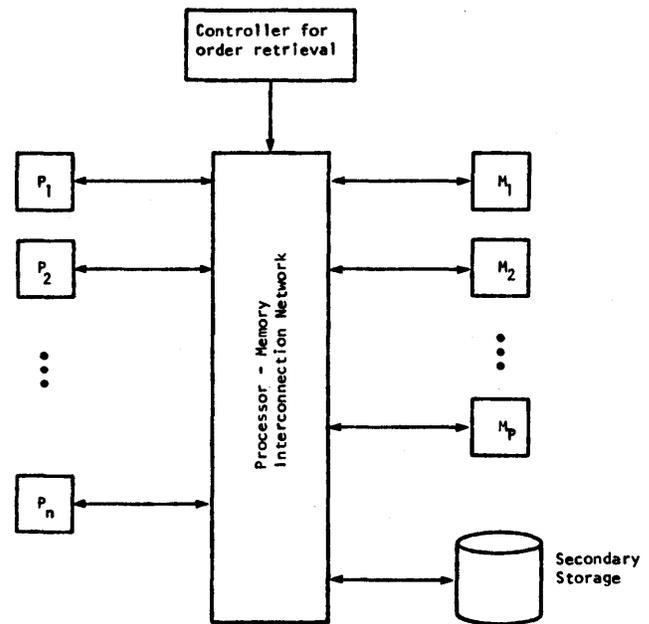


Figure 3a—Common memory

The only published work on applying branch and bound algorithm to solve NP-complete problems is by El-Dessouki and Huen.¹² A general purpose network computer is assumed. Due to memory space limitation, depth first search is used to evaluate sub-problems. Because the network is assumed to be slow and possibly distributed geographically, extensive inter-processor communication cannot be done. No performance results are given on the evaluation of example NP-complete problems. However, depth first search has been shown to be sub-optimal in minimizing the execution time of the branch and bound algorithm.⁴⁴ In the light of VLSI technology, larger and inexpensive memories, and faster communication media, the consideration of reducing the execution time (at the expense of larger memory space requirement) is a more critical problem.

In the next two sections, we present the architecture required to support the parallel branch and bound algorithm. We first compare two architectural alternatives and prove that the uni-directional ring network is the optimal interconnection network.

III. ARCHITECTURAL ALTERNATIVES SUPPORTING THE PARALLEL BRANCH AND BOUND ALGORITHM

There are basically two architectural alternatives to implement a parallel branch and bound algorithm, that is, the parallel processors can be interconnected either through a common memory or directly to each other.

A. Common Memory (Figure 3a)

In this implementation, the memory to store the sub-problems is separated from the processors. Because a single

memory would become a bottleneck in the accesses, multiple memory modules would have to be used. There is a processor-memory interconnection network that connects the processors and memories together. The number of memory modules used depends on the frequency of accesses from the processors which in turn depends on the complexity of the bounding algorithm. Sub-problems generated by the processors are stored through the interconnection network in the memories. A secondary storage is also connected to the network for extended storage.

Since it is required to order the sub-problem in ascending order by their lower bounds, the memory must be capable of order retrieval of the sub-problems. This means that each memory module must be capable of order retrieval of the sub-problems, and an external interconnection network must be capable of merging the extrema obtained from each module. The memory modules can be implemented with associative memory,⁵⁸ or they can be implemented as VLSI priority queues.⁴⁰ The processor-memory interconnection network can be designed for merging the sub-problems with minimum lower bounds from each memory module. Suppose there are n processors and p memory modules, then n sub-problems with minimum lower bounds in each memory module are fed to the sorting network. Sorting algorithms have been developed on mesh computers,^{52,65} perfect shuffle,⁶⁴ Batcher's odd-even merging network,^{2,61} and others.^{42,49} Optimal sorting networks have been investigated by Muller and Preparata,⁵¹ Baudet and Stevenson,⁴ Hirschberg,²⁶ and Preparata.⁵⁷ It was found that sorting of n numbers can be done in time $O(\log n)$ with n^2 intermediate processors. Since sorting networks do not allow intermediate results to be used until the sorting is completed, the maximum speed improvement that we can have is $O(\frac{n}{\log n})$, assuming that the number of iterations is improved by a factor of n .

The above sorting process is carried out in a decentralized fashion in the common memory. On the other hand, it is possible to perform sorting in a distributed fashion by sorting the sub-problems locally in each processor and exchanging messages among the processors. This leads to the second architectural alternative.

B. Private Memory (Figure 3b)

In this alternative, each processor has a private memory and is implemented as a unit. Sorting of sub-problems by lower bounds is carried out locally within each processor. Since it is not sufficient for the processor to work on local minima only, a processor-processor interconnection network is used so that local minima from different processors can be sent over the network and distributed. The objective of the distribution is to distribute the n global minima so that each processor has one

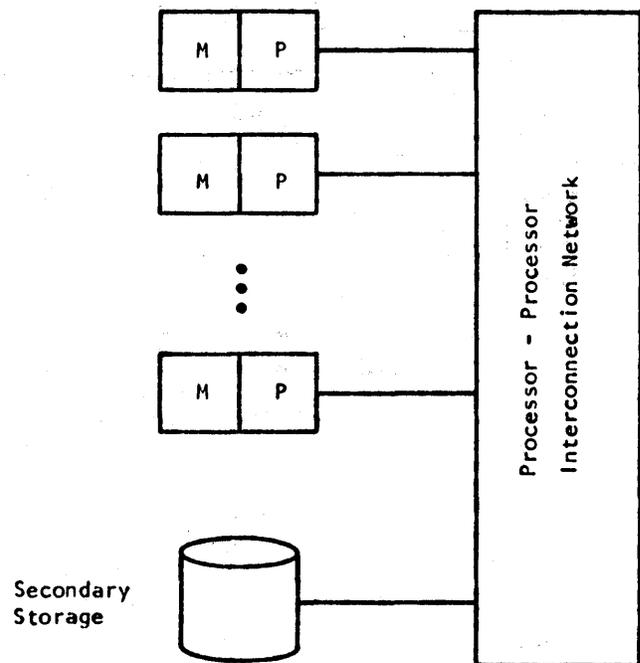


Figure 3b—Private memory

of the global minima. Expansion of the global minima, as mentioned earlier in Section II, is the most effective criterion of the parallel branch and bound algorithm.

The parallel branch and bound algorithm is implemented in this architecture as follows. Each processor keeps a list of sub-problems that are sorted by the lower bounds in ascending order. At the beginning of a cycle, each processor picks up the sub-problem with the minimum lower bounds in its list and expands it into two or more sub-problems. The lower bounds for the expanded sub-problems are evaluated and the sub-problems are inserted back into the list. The local minima from each processor are sent to neighboring processors and inserted into the local lists there. This process repeats until the n global minima are distributed one to each processor. The cycle then starts anew until each processor has one of the n global minima. This procedure can be improved by overlapping the distribution with the expansion of the sub-problems. Two possibilities can occur. First, the distribution time can be smaller than or equal to the lower bound evaluation time. So, although the distribution is completed, the distribution must be carried out again when the lower bounds for the currently expanded sub-problems are available. Second, the distribution time can be greater than the lower bound evaluation time, so the processors remain idle until the distribution is complete. In both of these cases, complete overlap is not attained due to the different processing and distribution times. A compromise can be made by overlapping the sub-problem expansion with the sub-problem distribution. In the case that the distribution is completed first, a local sorting can be performed when the sub-problems are evaluated and the processors expand the local minima without waiting for a complete distribution. In the case that the sub-problems evaluation is completed first, the next sub-problem in the local list can be evaluated immediately without waiting for the distribu-

* A VLSI priority queue is a distributed logic device that maintains the sub-problems in a sorted order. The logical structure is a two input, two output device (deque) such that tags can be input or output from the top or bottom. Comparators are inserted between consecutive elements in the queue. For any two consecutive elements, if the top element is greater than the bottom element, these two elements are exchanged. By this means, larger elements are "dropped" to the bottom of the queue and smaller elements "float" to the top of the queue. Further, elements can be inserted into the queue continuously without waiting for the previous element to be sorted in the queue.

tion to complete. It is shown in Section IV that this strategy is actually very effective.

C. Discussion

There are advantages and disadvantages associated with each of the architectural alternatives presented in this section. The first approach can either be fast (the processor-memory interconnection network is a hardware sorting network such as Batcher's network³⁶) and expensive or slow and not quite expensive (an external software sorter is used). Nonetheless, the interconnection network available today generally possesses properties of substantial delays, high cost and is difficult to evolve. Furthermore, the sorting network orderly retrieves the n sub-problems that have the minimum lower bounds. Since this is not required by the system, this may lead to unnecessary degradation in performance. Another characteristic of the sorting network is that sorting has to be completed before the list is available. On the other hand, the second alternative can utilize the current VLSI technology to implement the processor and memory on a single chip. Although the processor-processor interconnection network may be expensive and incurs substantial overhead in sub-problem distributions, it is shown that sorting does not have to be completed before sub-problem expansion can begin and this causes a relatively small degradation in performance. This, together with a few other nice properties, make this a more cost-effective design. We therefore select the second alternative in our design.

IV. NETWORK ARCHITECTURE

The objective of the network is to have a *complete distribution*; that is, to distribute the sub-problems in the local memories of the processors so that the n global minima can be distributed, one to each of the n processors. The locations of these n global minima are not known *a priori*; otherwise the problem is very simple and the processor with more than one global minima can send one of these sub-problems to processors without any. Since predetermined distribution operations are unknown, we can allow all the processors to carry out the same distribution operations (e.g., distribute to the nearest neighbor), or to carry out different distribution operations (e.g., one processor may be distributing to its nearest neighbor while the others are not). The former type of distribution possesses the property that each processor is connected to and from the same number of neighboring processors and has the *state preserving property*. That is, if the global minima have been distributed to the processors, continual redistribution would not disturb the state and the global minima would remain distributed to the processors. On the other hand, each processor may be connected to and from a different number of neighboring processors in the latter case and it is rather difficult to preserve the state. For this reason, we choose to investigate the former case only.

The design of the interconnection network ranges from a simple uni-directional ring network where each processor can communicate with one of its neighbors to a fully connected

network where communications can be carried out simultaneously with all the processors. Analysis in this section shows that a simple uni-directional ring network is the optimal interconnection network. In order to do this, an urn model must first be developed.

A. The Urn Model

The n processors in the system are represented as n urns that contain n white marbles which stand for the global minima and $S - n$ yellow marbles where S is the total number of active sub-problems. The white marbles are originally distributed randomly to the urns. The distribution process moves the marbles around so that eventually, one white marble is distributed to each urn. The white marbles are always "lighter" than the yellow marbles so that they always "float" to the top of the urn. During the distribution process, one or more marbles are taken from each urn and distributed to one or more urns in the system. If a white marble exists in the urn, it is always distributed first. The ordering of the yellow and white marbles in the urns models the ordering of the sub-problems by lower bounds in ascending order in the processors. If one of the n global minima (white marble) exists in a processor (urn), it is always ordered before the other sub-problems (yellow marbles) and is always distributed first. It should be noted that this model does not take into account the ordering of the white marbles which is important in a conventional sorting and merging problem. It is sufficient for exactly one white marble to be distributed to each urn whereas in a sorting and merging problem, the white marbles are ordered before they are distributed to the urns. It is hoped that the relaxation induced in this problem can help to reduce the amount of marble movements.

We investigate distribution strategies that correspond to different degrees of interconnection. The first strategy shifts a white marble, if there are any, to the urn on the right. This corresponds to a *uni-directional ring network* (Figure 4a). A more general strategy distributes the j th marble ($j < k$) in the i th urn to the $([i + j] \bmod n)$ th urn in parallel. This corresponds to a *k-connected network* (Figure 5a). When $k = 1$, this becomes the uni-directional ring network. In Figures 4 and 5, we have also shown the state of the system after a number of distributions.

In evaluating the interconnection network, all the overheads must be accounted for in the distribution process. The overheads in a distribution include the time to shift and the time to let the white marbles "float" to the top (which corresponds to merging the newly arrived sub-problems into the original list). The lower bounds for the number of distributions to achieve complete distribution are shown in the next section.

B. Lower Bound for a Complete Distribution

In evaluating the lower bound, the sorting method in each processor must be taken into account. The overhead for sorting depends on the implementation. The complexity is $O(m \log m)$ for sorting m numbers by software (e.g., heap

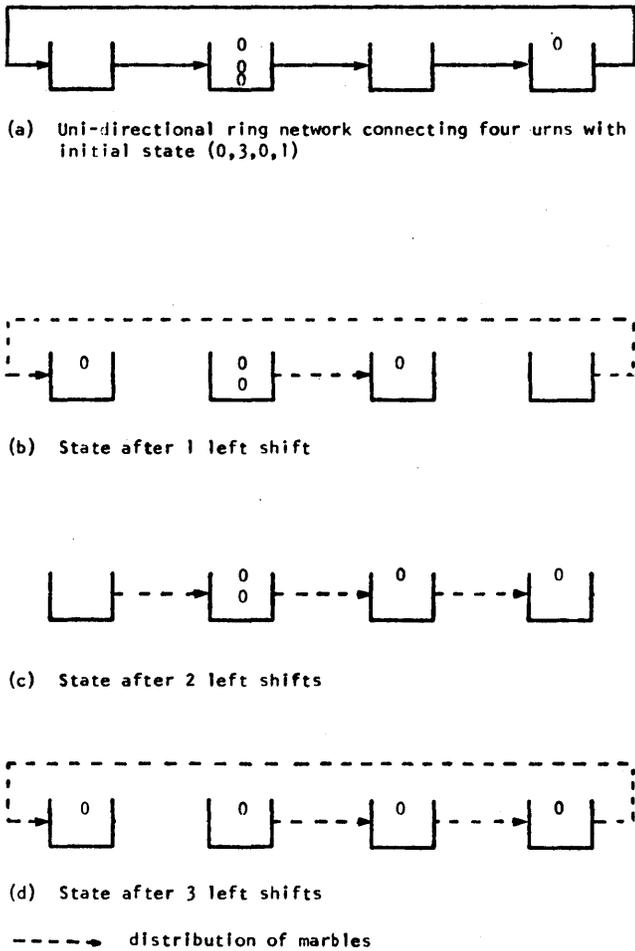


Figure 4—Uni-directional ring network connecting four urns

sort,³⁶ $O(m)$ for sorting by a hardware priority queue, and $O((\log m)^2)$ for sorting by Batcher's odd-even merging network.³⁶ In the following theorem, we evaluate the lower bound of a complete distribution for the three sorting methods.

Theorem 1

Let K be the communication time to transfer one or more sub-problems in parallel to the other processors, m be the maximum number of sub-problems that can be stored in a processor, and n be the number of processors in the system. Depending on the degree of connection, the lower bounds on the number of operations for a complete distribution is between $O(K + n \log m)$ and $O(nK + n \log m)$ for sorting by software, is between $O(K + n)$ and $O(Kn)$ for sorting by hardware priority queues and is between $O(K + \log^2 m)$ and $O(Kn + n \log^2 m)$ for sorting by Batcher's networks.

Proof

Suppose each urn is connected to n^x other urns ($0 \leq x \leq 1$), that is, urn i is connected to urn $(i + k) \bmod n$ ($1 \leq k \leq n^x$). The maximum delay to transfer a marble from one urn to another is n^{1-x} . Assuming all the n marbles reside in one single urn and transfers can be made in parallel to n^x other

urns, it would take n^{1-x} transfers to take all the marbles out from this urn. Since all the transfers are carried out simultaneously, each urn would be receiving n^x marbles in a time interval K .

After each transfer, the marbles must be inserted into the local lists before another distribution can take place. For sorting by software, the time needed is the time to insert n^x numbers into a priority tree that may contain as many as m numbers. The total sorting overhead is therefore $O(n^x \log m)$. For sorting by a hardware priority queue, each insertion takes constant time and n^x numbers can be inserted into the priority queue in time $O(n^x)$. For sorting by Batcher's network, all the m numbers in a processor are connected to a network. The sorting overhead is therefore $O(\log^2 m)$.

To summarize, n^{1-x} iterations are needed, and each iteration takes time K for communication and additional overhead for sorting. The lower bound on the total overhead is therefore $O(n^{1-x}[K + n^x \log m])$ for sorting by software; $O(n^{1-x}(K + n^x))$ for sorting by hardware priority queues; and $O(n^{1-x}(K + \log^2 m))$ for sorting by Batcher's network. For an n -connected network, $x = 1$ and the lower bounds are $O(K + n \log m)$, $O(K + n)$, $O(K + \log^2 m)$ for sorting by software, hardware priority queue and Batcher's network respectively. For a uni-directional ring network, $x = 0$ and the corresponding lower bounds are $O(Kn + n \log m)$, $O(Kn)$, and $O(Kn + n \log^2 m)$.

From the above theorem, it is obvious that sorting by hardware priority queues is better than sorting by software and Batcher's network for a uni-directional ring network. However, depending on the relative sizes of K , m , and n , sorting by priority queues may be better than sorting by Batcher's network or vice versa in an n -connected network. In general, K is very small because of the advances in communication technologies; n is usually large because it governs the degree of parallelism; m is also large, and in an n -connected network, $m \geq n$. Taking these into account, the lower bounds for an n -connected network are $O(n \log m)$, $O(n)$ and $O(\log^2 m)$ for sorting by software, hardware priority queues, and Batcher's network. Batcher's network has less overhead if $m \leq O(c^{\sqrt{n}})$, where $c > 1$. On the other hand, Batcher's net-

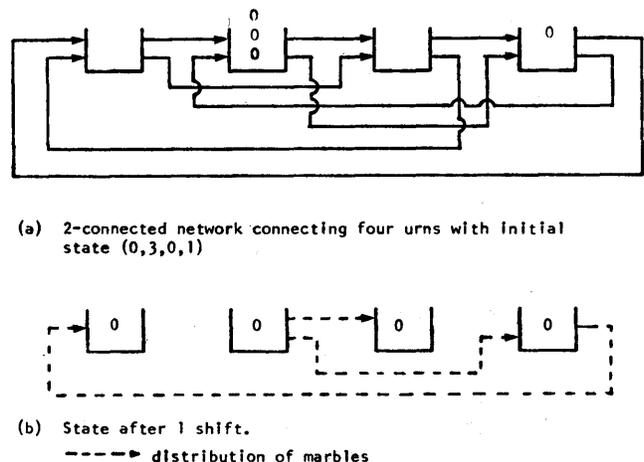


Figure 5—2-connected network connecting four urns

work uses $O(m \log^2 m)$ hardware,³⁶ as compared to $O(m)$ for a hardware priority queue. For the present time, we favor the use of a hardware priority queue because of its reduced hardware complexity. In the next section, we prove that the uni-directional ring network is the optimal network if hardware priority queues are used.

C. The Optimal Interconnection Network with Hardware Priority Queue

In this section, we show that the uni-directional ring network is the optimal interconnection network by showing that the amount of work needed for a complete distribution is $O(n-1)$ and therefore equals the lower bound evaluated earlier.

Theorem 2

The number of distributions for a complete distribution in a k -connected network ($1 \leq k \leq n$) is at most $n-1$.

Proof

We first prove the case of the uni-directional ring (1-connected) network. The proof is by contradiction. Suppose a white marble cannot get to the top of urn i in $n-1$ distributions and remains in the second position, that is, after $n-1$ distributions, urn i still contains at least two white marbles and the distribution is not complete. An urn that starts with 0 or 1 marbles can never get more than one marble after $n-1$ distributions. Hence urn i must have started with at least two marbles. And in $n-1$ distributions, $n-1$ distinct white marbles must have passed over the top of urn i , because if not, the second white marble in urn i would have a chance to get to the top of urn i . This implies that there are altogether $n-1+2 = n+1$ white marbles in the system, which contradicts the original assumption that there are n white marbles in the urns. Complete distribution can always be achieved in $n-1$ distributions. The proof for a k -connected network with $k > 1$ is similar and will not be repeated here.

The overall amount of work is therefore $(n-1) \times (\text{sorting overhead})$. Since the sorting overhead is the smallest in a uni-directional ring network, the overall complexity to achieve a complete distribution is therefore $O(n)$. As we have proved in Theorem 1 that the lower bound of distributions using hardware priority queues is $O(n)$, the uni-directional ring network is the optimal interconnection network. Although the number of distributions to achieve a complete distribution in a k -connected network ($k > 1$) may be smaller, as evidenced in the simulation results shown later. The performance can only be improved by a constant factor because the lower bound is also $O(n)$. Furthermore, the number of network links in a k -connected network ($k > 1$) is n^k , as compared to n in a uni-directional ring network. We conclude that the uni-directional ring network is the optimal and most cost-effective way of implementation.

In the remainder of this section, we present some results on the average fraction of urns containing white marbles using the k -connected network and try to answer the question we raise in Section III—namely, what is the degradation in per-

formance if a complete distribution is not attained before the processors pick up sub-problems for expansion. The evaluation results are obtained by generating all the possible combinations of n white marbles in the n urns as initial distributions. It is seen in Figure 6 that the increase in the average fraction of urns containing white marbles due to increasing k is rather small. Furthermore, the sorting overhead is not included in the evaluation. The final performance for $k > 1$ is expected to be less than the performance of the uni-directional ring network. In Figure 7, the fraction of urns containing marbles for the different number of distributions in a uni-directional ring network is shown. It is seen that these curves approach different asymptotic values as the number of urns is increased. The asymptotic average fraction of urns containing white marbles as the number of urns is increased for no distribution ($s = 0$) has been shown to be 0.5. The analyses for cases where $s > 0$ are similar but more difficult. It is also seen that the improvement is significant for the first few distributions, but the improvement is diminishing as the number of distributions is increased. This implies that the fraction of urns containing white marbles is significantly improved by a small number of distributions. In general, less than half of the urns do not contain white marbles for an incomplete distribution.

D. Technology Dependent Considerations

We have assumed in the urn model that each marble represents a sub-problem. Actually, a sub-problem is characterized not only by a lower bound, but also by the state of the problem. For example, let a graph of p nodes be represented in the form of a p by p connectivity matrix, and each sub-problem include a partial assignment of the nodes and edges. In a distribution, the partial assignment must also be transferred with the lower bound of the sub-problem. If p is large, the transfer time can be in the order of milliseconds or seconds. On the other hand, sorting in the processors has a relatively small overhead as compared with the distribution time. If we examine the complexity measure of the uni-directional ring network again, we discover a more serious problem. The overheads for complete distribution is $O(n)$. Suppose the number of cycles in a parallel branch and bound algorithm improves by a factor of n , and in each cycle, there is an overhead for distribution of $O(n)$; this implies that there is no overall improvement in performance as far as complexity measure is concerned. These observations imply that it is necessary to design additional hardware or strategies in order to reduce the distribution overhead so that distribution can be overlapped completely with sub-problem expansion. There are several alternatives.

The first alternative considers sending the tags (each consisting of the urn number and the lower bound) instead of the white marble (the entire sub-problem) in a distribution. After $O(n)$ distributions, complete distribution is obtained. These tags are then gated to an external controller which counts the number of white marbles in each urn and decides on the optimal transfer sequence of white marbles from one urn to another. A k -connected network may be used in order to allow k parallel sub-problem transfers to be made from each

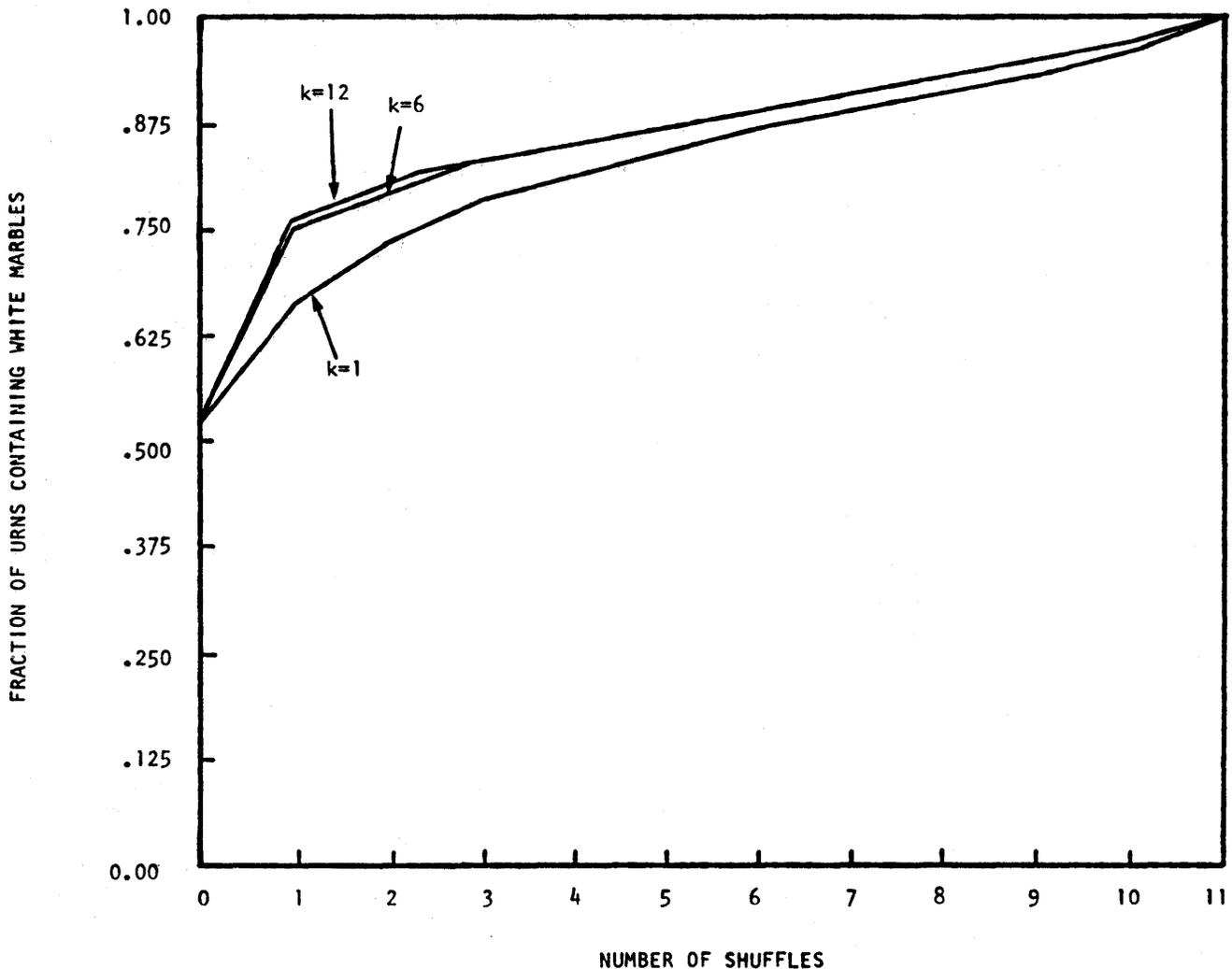


Figure 6—Performance of the k -connected network for 12 urns
($k = 1$ for uni-directional ring network)

urn. Of course, the value of k has to be determined so that the response time requirement is satisfied.

The second alternative considers sending n marbles from each urn and passing them through a sorting network such as Batcher's sorting network. The first n marbles coming out from the sorting network must be white, and they are returned in parallel to the n urns. As we know, the complexity for sorting n^2 numbers using Batcher's network is $O(4 \log^2 n)$. The overall improvement of the parallel branch and bound algorithm is at most

$$O\left(\frac{n}{\log^2 n}\right)$$

Further, this scheme uses extensive hardware and may not be practical when n is large.

None of the above schemes is perfect and requires additional hardware support. However, when hardware becomes sufficiently inexpensive, it may be possible to use more processors and allow them to operate at over 50% efficiency (the average number of urns containing white marbles without any distribution is over 50%). Furthermore, we have assumed so

far that the system operates in a coupled fashion; that is, the sub-problems are evaluated while the distributions are made and the evaluation of the next set of problems does not start until part or all of the distributions are done. In practice, the sub-problems have different sizes and different processing times and therefore it would be inefficient for the system to wait until all the processors are finished. Each processor would behave independently and execute the lower bound evaluation function in its local memory. When this evaluation is finished, it picks up a sub-problem with the minimum lower bound from its local list of sub-problems. Since the time when one processor picks up a sub-problem to the time when another processor picks up a new sub-problem can be relatively short, the distribution process may not be completed and the system would be operating at less than optimal performance.

Finally, if an urn does not contain a white marble (one of the first n global minima), it may contain a marble of different color (which may correspond to one of the $(j+1)n$ th global minima, $j > 0$, and this is distributed accord-

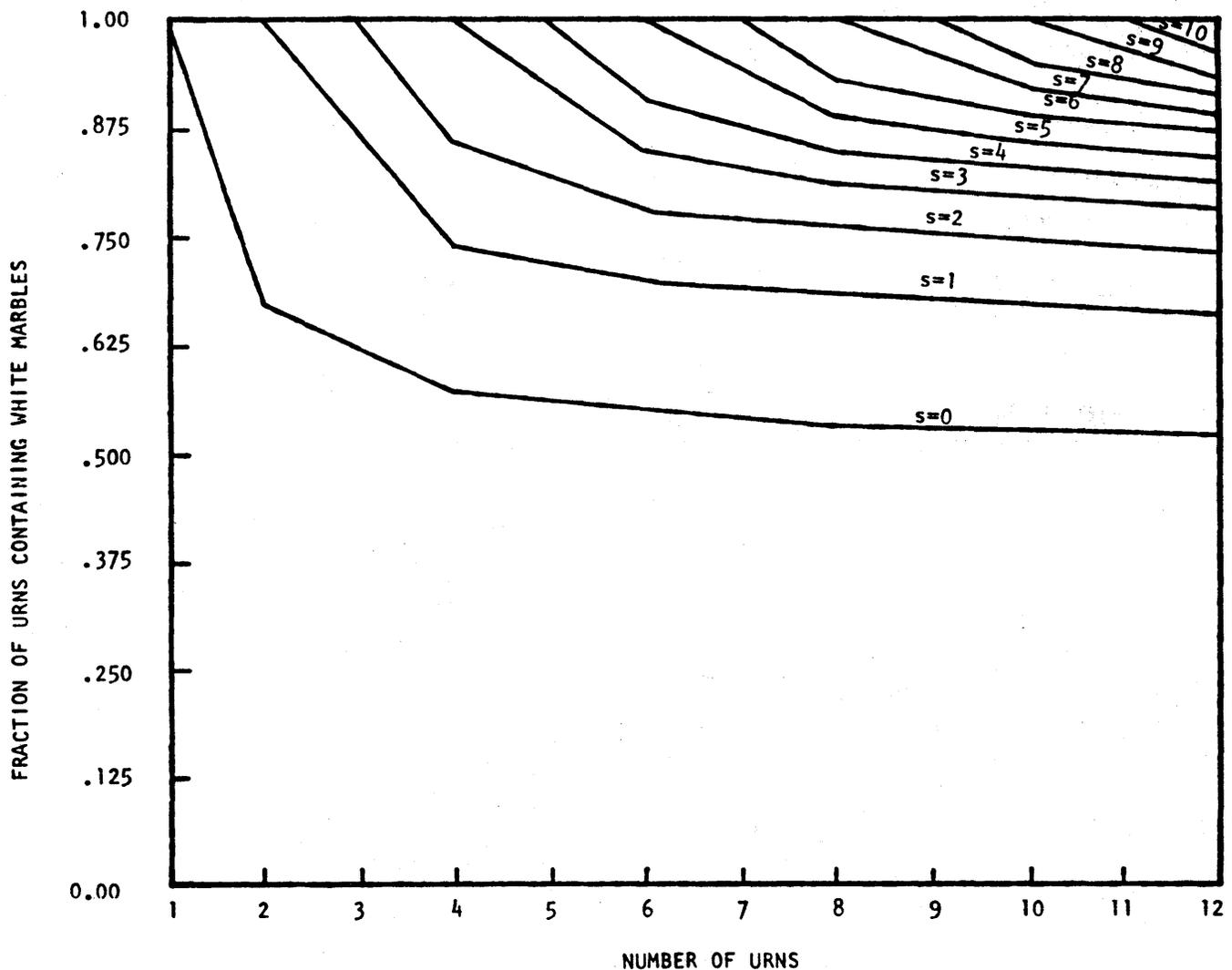


Figure 7—Performance of the uni-directional ring network ($k = 1$) for different number of urns ($s =$ distributions)

ingly. So although a processor may not be working on one of the n th global minima, the expansion of a sub-problem with the minimum lower bound may still contribute to the speedup.

One interesting point to notice is when a small number of distributions are made, the distribution of the first global minima improves; that is, the number of urns containing white marbles increases. However, the distribution of the $n + 1$ th to $2n$ th global minima which are represented as black marbles may be worse. This distribution is important because it governs the distribution of the white marbles in the next iteration (when the black marbles in this iteration become the white marbles in the next iteration). It is shown that the average number of urns containing black marbles after a complete distribution is actually smaller than the average number of urns if the marbles were distributed randomly. This phenomenon is illustrated in Figure 8. Fortunately, the difference between these two average numbers for large n is insignificant. The simulation results are not included here.

The problem discussed in this section for speeding up the

distribution time can be solved by faster technology. The ring network can run up to several hundred mega-bits per second and can be used as a "barrel" as in Control Data Corporation's 6000 and 7000 series computers.⁶⁶ It can be realized with sub-nanosecond emitter coupled logic that can operate at rates up to 100 MHz. Off-the-shelf parts, such as Fairchild's F100K, are available to implement the uni-directional ring network.

V. CONCLUSION

In this paper, we have proposed and studied the network architecture of MANIP, a parallel computer system for processing NP-complete problems. NP-complete problems have the unique property that the computation time for all known optimal algorithms increases exponentially with the problem size. Thus a small increase in the problem size may cause a very large increase in the problem space needed for the opti-

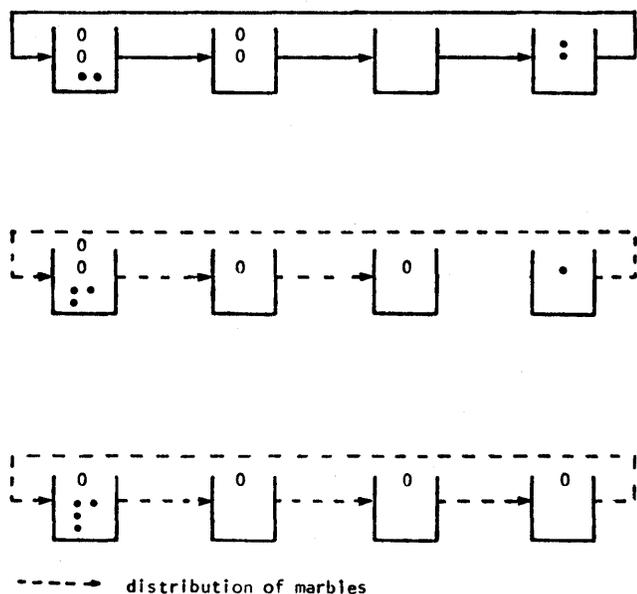


Figure 8—The decrease in the number of urns containing black marbles when the shuffle is complete (the white marbles represent the first n global minima; the black marbles represent the $n + 1$ st to $2n$ th global minima)

mal algorithm to complete the examination. Due to the inherent difficulty in solving NP-complete problems, parallelism in processing is proposed to expand the size of solvable problems. The most general technique that can be used to solve a wide variety of NP-complete problems on a uni-processor system, optimally or suboptimally, is the branch and bound algorithm. We have studied in this paper a parallel version of the branch and bound algorithm which can be executed efficiently on a parallel computer system.

The parallel branch and bound algorithm requires a combination of sorting and merging. The sub-problems are evaluated to produce new sub-problems which are inserted into a list of previously created sub-problems. This list is maintained in a sorted order by the lower bounds of the sub-problems so that the minima can be picked up for expansion in the next cycle. The process is terminated when a feasible solution is found with a value smaller than the lower bounds of all the sub-problems in the list. Since it is important to maintain a global sorted list of sub-problems, a common memory shared by all the processors can be used. However, this can become a bottleneck when the number of processors is large. We have proposed an alternative such that each processor has a local memory and the processors communicate with each other through an inter-processor communication network. When the processors have created new sub-problems, they are first inserted into the local list, and then sub-problems with minimum lower bounds from each processor are distributed until a set of n global minima are obtained. These n global minima are distributed to the n processors in the system for processing (complete distribution).

We have proved that the lower bound for the amount of work to achieve a complete distribution is $O(n)$ when sorting is done by a hardware priority queue within each processor. We have also shown that the uni-directional ring network is the optimal and most cost-effective way of implementing the

inter-processor communication network. Sorting by other sorting methods gives different performance. Sorting by software, such as heap sort, has a worse performance, while sorting by Batcher's odd-even merging network has a better performance at the expense of increased hardware complexity. The proposed interconnection network is reliable because of its simplicity and its reconfigurability. Faulty processors can be switched off the network without affecting the performance of other processors. Redundant rings can also be used to increase the reliability of the network.

The architecture of the processors and the performance evaluation of the system is given in a different paper. The simulation results there show that with complete distributions, the number of iterations reduce by a factor of n using n processors. With no distribution, the performance is very poor. However, when one or more distributions are applied in each iteration, the total number of iterations is the same as if a complete distribution is used (with very small variations). One major problem encountered in the simulations is the problem of insufficient memory space. The branch and bound algorithm has to switch from best-first search to depth-first search when memory space is exhausted. This significantly degrades the memory performance. A method to increase the virtual space of the branch and bound algorithm is to use a virtual memory management system. This will be presented in a future paper.

REFERENCES

1. S. G. Aki, D. T. Barnard, and R. J. Doran, "Simulation and Analysis in Deriving Time and Storage Requirements for a Parallel Alpha-Beta Algorithm," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 231-234, 1980.
2. K. E. Batcher, "Sorting Networks and their Applications," *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32, pp. 307-314, Apr. 1968.
3. K. E. Batcher, "The Flip Network in STARAN," *Proc. of 1976 Int'l. Conf. on Parallel Processing*, Michigan, pp. 65-71, 1976.
4. G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp. 84-87, Jan. 1978.
5. V. E. Benes, "Optimal Rearrangeable Multistate Connecting Networks," *Bell System Technical Journal*, Vol. 48, No. 4, pp. 1641-1656, July 1964.
6. J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *Proc. of 1979 International Conf. on Parallel Processing*, Michigan, pp. 257-266, 1979.
7. B. C. Desai, "The BPU, A Staged Parallel Processing System to Solve the Zero-One Problem," *Proc. of ICS78*, Taipei, Taiwan, pp. 802-817, Dec. 1978.
8. B. C. Desai, "A Parallel Microprocessing System," *Proc. of 1979 Int'l. Conf. on Parallel Processing*, p. 136, 1979.
9. A. M. Despain and D. A. Patterson, "X-tree: A Tree Structured Multiprocessor Computer Architecture," *Proc. of 5th Symp. on Comp. Arch.*, Palo Alto, CA, 1978, pp. 144-151.
10. W. L. Eastman, "A Solution to the Traveling Salesman Problem," presented at the American Summer Meeting of the Econometric Society, Cambridge, Mass., Aug. 1958.
11. A. Efromyson and T. C. Ray, "A Branch and Bound Algorithm for Plant Location," *Operations Research*, Vol. 14, pp. 361-368, 1966.
12. El-Dessouki and W. H. Huen, "Distributed Enumeration on Network Computers," *Proc. of 1979 Int'l. Conf. on Parallel Processing*, Michigan, pp. 137-146, 1979. Also published in *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp. 818-825, Sept. 1980.
13. T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implications," *IEEE Trans. Computers*, Vol. C-23, No. 3, pp. 309-318, Mar. 1974.

14. J. P. Fishburn, R. A. Finkel, and S. A. Lawless, "Parallel Alpha-Beta Search on ARACHNE," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 235-243, 1980.
15. Foster, M. J. and H. T. Kung, "Design of special-purpose VLSI chips," *IEEE Computer*, Vol. 13, No. 1, pp. 26-40, 1980.
16. M. R. Garey and D. S. Johnson, "Strong NP-completeness Results: Motivations, Examples, and Implications," *JACM*, Vol. 25, No. 3, pp. 499-508, July 1978.
17. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, San Francisco, 1979.
18. R. Garfinkel, "On Partitioning the Feasible Set in a Branch and Bound Algorithm for the Asymmetric Travelling Salesman Problem," *Operations Research*, Vol. 21, No. 1, pp. 340-342, 1973.
19. R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, John Wiley and Sons, Inc., New York, 1972.
20. A. M. Geoffrion and R. E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey," *Management Science*, Vol. 18, No. 9, pp. 465-491, May 1972.
21. L. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proc. 1st Annual Comp. Architecture Conf.*, pp. 21-28, Dec. 1973.
22. J. A. Harris and D. R. Smith, "Hierarchical Multi-processor Organizations," *Proc. 4th Annual Symp. on Comp. Arch.*, pp. 41-48, 1977.
23. M. Held and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," *Jr. of SIAM*, Vol. 10, pp. 196-210, 1962.
24. M. Held and R. Karp, "The Travelling Salesman Problem and Minimum Spanning Trees," *Operations Research*, Vol. 18, pp. 1138-1162, 1970.
25. M. Held and R. Karp, "The Travelling Salesman Problem and Minimum Spanning Trees, Part II," *Math. Prog.*, Vol. 1, pp. 6-25, 1971.
26. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," *CACM*, Vol. 21, No. 8, pp. 657-601, Aug. 1978.
27. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.
28. T. Ibaraki, "Computational Efficiency of Approximate Branch and Bound Algorithms," *Math. of Oper. Research*, Vol. 1, No. 3, pp. 287-298, 1976.
29. T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch and Bound Algorithms," *Int. Jr. of Comp. and Info. Sci.*, Vol. 5, No. 4, pp. 315-344, 1976.
30. T. Ibaraki, "On the Computational Efficiency of Branch and Bound Algorithms," *J. of Oper. Res. Soc. of Japan*, Vol. 20, No. 1, pp. 16-35, 1977.
31. T. Ibaraki, "The Power of Dominance Relations in Branch and Bound Algorithms," *JACM*, Vol. 24, No. 2, pp. 264-279, 1977.
32. T. Ibaraki, "Depth-m Search in Branch-and-Bound Algorithms," *Int. Jr. of Comp. and Inf. Sci.*, Vol. 7, No. 4, pp. 315-343, 1978.
33. G. Ingargiola and J. Korsh, "A Reduction Algorithm for Zero-one Single Knapsack Problems," *Management Science*, Vol. 20, No. 4, pp. 460-663, 1973.
34. G. Ingargiola and J. Korsh, "A General Algorithm for One Dimensional Knapsack Problems," *Operations Research*, Vol. 25, No. 5, pp. 752-759, 1977.
35. R. M. Karp, "Reducibility Among Combinational Problems," *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85-104, 1972.
36. D. E. Knuth, *The Art of Computer Programming, Sorting, and Searching*, Vol. 3, Addison-Wesley, 1973.
37. W. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch and Bound Algorithms for Permutation Problems," *JACM*, Vol. 21, No. 1, pp. 140-156, 1974.
38. D. J. Kuck, "ILLIAC IV Software and Application Programming," *IEEE Trans. on Comp.*, Vol. C-17, pp. 746-757, Aug. 1968.
39. D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Survey*, Vol. 9, No. 1, pp. 29-59, 1977.
40. H. T. Kung, "The Structure of Parallel Algorithms," *Advances in Computers*, Vol. 19, Yovits, M. C. ed., Academic Press, New York, 1980.
41. B. Lageweg, J. Lenstra and A. Rinnooykar, "Job-shop Scheduling by Implicit Enumeration," *Management Science*, Vol. 24, No. 4, pp. 441-400, 1977.
42. E. A. Lamagna, "The Complexity of Monotone Networks for Certain Bilinear Forms, Routing Problems, Sorting and Merging," *IEEE Trans. on Computers*, Vol. C-28, No. 10, pp. 773-782, Oct. 1979.
43. A. H. Land and A. Doig, "An Automatic Method for Solving Discrete Programming Problems," *Econometrica*, Vol. 28, pp. 497-520, 1960.
44. Lawler, E. L. and Wood, D. W., "Branch and Bound Methods: A Survey," *Operations Research*, Vol. 14, pp. 699-719, 1966.
45. D. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, Vol. C-24, No. 12, pp. 215-255, Dec. 1975.
46. J. Lenstra, "Sequencing by Enumerative Methods," *Math. Centre. Tract 69*, Mathematisch Centrum, Amsterdam, 1976.
47. D. D. Marshall, "A Parallel Processor Approach for Searching Decision Trees," *Proc. of 1977 Int'l. Conf. on Parallel Processing*, Michigan, pp. 199-201, 1977.
48. L. Mitten, "Branch and Bound Methods: General Formulation and Properties," *Operations Research*, Vol. 18, pp. 24-34, 1970.
49. H. P. Moravec, "Fully Connecting Multiple Computers with Pipelined Sorting Nets," *IEEE Trans. on Computers*, Vol. C-28, No. 10, pp. 795-798, Oct. 1979.
50. T. Morin and R. Marsten, "Branch and Bound Strategies for Dynamic Programming," *Operations Research*, Vol. 24, pp. 611-627, 1976.
51. D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and Switching," *JACM*, Vol. 22, No. 2, pp. 195-201, Apr. 1975.
52. D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh Connected Parallel Computer," *IEEE Trans. on Computers*, Vol. C-27, No. 1, pp. 2-7, Jan. 1979.
53. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw Hill, New York, 1971.
54. J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," *6th Annual Symposium on Computer Architecture*, pp. 168-177, 1979.
55. D. A. Patterson and C. H. Sequin, "Design Considerations for Single Chip Computers of the Future," *IEEE Trans. on Computers*, Vol. C-29, No. 2, pp. 108-116, Feb. 1980.
56. M. C. Pease, "The Indirect Binary n-cube Microprocessor Array," *IEEE Trans. on Computers*, Vol. C-26, No. 5, pp. 458-473, May 1977.
57. F. P. Preparata, "Parallelism in Sorting," *Proc. of 1977 Int'l. Conf. on Parallel Processing*, Michigan, pp. 202-206, Aug. 1977.
58. C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *IEEE Trans. on Computers*, Vol. C-27, No. 9, pp. 800-814, Sept. 1978.
59. G. Sa, "Branch and Bound and Approximate Solutions to the Capacitated Plant Location Problem," *Operations Research*, Vol. 17, No. 6, pp. 1005-1016, 1969.
60. S. Sahni, "General Techniques for Combinational Approximation," *Operations Research*, Vol. 25, No. 6, pp. 920-936, 1977.
61. R. Sedgewick, "Data Movement in Odd-Even Merging," *SIAM Journal of Computing*, Vol. 7, No. 3, pp. 239-272, Aug. 1978.
62. C. L. Seitz, *Proceedings of the Caltech Conference on Very Large Scale Integration*, California Institute of Technology, Jan. 1979.
63. S. W. Song, "A Highly Concurrent Tree Machine for Database Applications," *Proc. of 1980 Int'l. Conf. on Parallel Processing*, Michigan, pp. 259-268, 1979.
64. H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. on Computers*, Vol. C-20, No. 2, pp. 153-161, Feb. 1971.
65. C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *CACM*, Vol. 20, No. 4, pp. 263-271, Apr. 1977.
66. J. E. Thornton, *Design of a Computer: The Control Data 6000*, Scott, Foresman and Company, Glenview, Illinois, pp. 141-153, 1970.
67. C. L. Wu and T. Y. Feng, "The Reverse-Exchange Interconnection Network," *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp. 801-811, Sept. 1980.

Parallel sorting machines: their speed and efficiency

by LEON E. WINSLOW and YUAN-CHIEH CHOW

Wright State University
Dayton, Ohio

ABSTRACT

A unified approach for analyzing and classifying parallel sorting machines is discussed. The approach accounts for a number of important factors that have significant impacts on the measure of the size and complexities of parallel sorters. These factors include the sorter architectures, sequential/parallel input, single/multiple passes, and the base of the comparitors. An efficiency measure is also introduced to describe the optimality and other general characteristics of the sorters. Finally, a new sorter with the time complexity of $O(c \cdot \log_b N)$, where b is an arbitrary base, is proposed. Its efficiency is independent of the number of processors, the size of the list, and the maximum value in the list.

INTRODUCTION

There are sorting algorithms, such as Quicksort, which require sort times of $O(N \cdot \log_2 N)$ on a single-processor system. Ideally, using parallel processors to perform a sort should reduce the sorting time by a factor equal to the number of processors used. No published parallel-processor sorter achieves this ideal. This paper presents two parallel-processor sorters whose speed is directly proportional to the number of parallel processors used, and it shows how sorters with sort times of $O(N)$, $O(\log_b N)$, and $O(1)$ can be obtained.

Published parallel-processor sorters fall into two categories, the ladder² and the network.¹ The ladder uses N parallel comparitors and sorts a list of N items in the time $O(N)$. The bitonic sorter, the fastest of the network sorters, uses $O(N \cdot \lceil \log_2 N \rceil^2)$ comparitors and sorts in time $O(\lceil \log_2 N \rceil^2)$. Increasing the number of comparitors in either the ladder or bitonic sorter increases the size of the list that can be sorted, but it has no effect on the time required; that is, the time is in some sense independent of the number of comparitors.

If P processors are working in parallel for time T , then they should produce PT units of work. Information theoretic arguments show that sorting a list of N items is equivalent to computing $N \cdot \log_b N$ digits where the base b is arbitrary. A single processor can sort a list in time $O(N \cdot \log_2 N)$; so the efficiency, the ratio of the actual work produced to the theoretical work that can be produced, is essentially 1. The ladder, however, requires N units working for time N , so the efficiency is

$$\frac{N \cdot \log_2 N}{N^2} = \frac{\lceil \log_2 N \rceil}{N}$$

By a similar calculation, the efficiency of the bitonic sorter is $\lceil \log_2 N \rceil^{-3}$. In both the ladder and the bitonic sorter, the efficiency decreases as the size of the list and/or the number of units increases; that is, the units are doing useful work a smaller and smaller fraction of the time. An optimally designed sorter has an efficiency of 1, which is independent of the list size, the values in the list, and the number of processors used. The single-processor sort is essentially optimal, but neither of the parallel-processor sorters even approach optimality.

The next section uses the radix sorter to consider where and how parallelism can be introduced so that a range of sort times, from $O(N \cdot \log_b \text{MAX})$ to $O(1)$, can be obtained where MAX is the largest value in the list. The efficiency of the sorter is independent of the number of processors used; that is, doubling the number of processors doubles the sort space.

The third section presents a new sorter with sort times ranging from $O(cN \cdot \log_b N)$ to $O(c \cdot \log_b N)$, where c is a constant between 1.0 and 1.5 and the efficiency is $(1/c)$, which is independent of the number of processors, the size of the list, and the maximum value in the list.

THE RADIX SORTER

Before introducing parallelism, we consider the standard radix sorter in some detail to illustrate where and why parallelism is used. The mechanical punched card sorter is a radix sorter with base 12; that is, the sorter is set to a particular card column, and, as each punched card enters the sorter, it uses the hole in the prespecified card column to determine which of the 12 buckets to place the card in. To sort a deck of cards, a series of passes is necessary. During each pass the prespecified card column is shifted forward one position in the columns used for sorting (the units position is used for the first pass, the tens position for the next pass, and so forth), all the cards are sorted (or placed) into the correct bucket, and the cards are collected for the next pass. Obviously, for a deck of N cards, each pass requires time $O(N)$, and the number of passes is equal to the number of card columns necessary to store the largest value.

To construct a more general radix sorter, we assume the existence of the "base b element." A base b element inputs

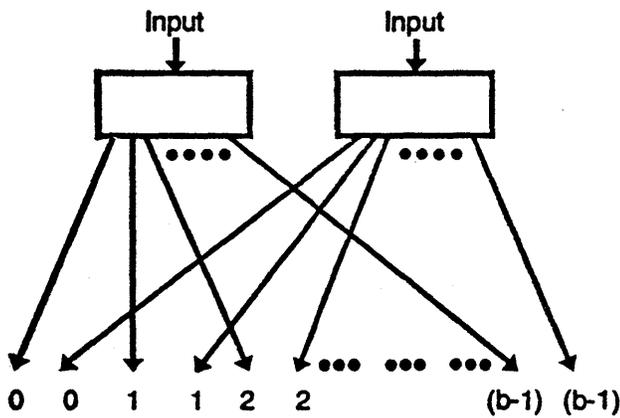


Figure 1—A parallel radix sorter

an item and outputs the item into one of b buckets using a prespecified digital position in the item to determine the correct bucket. One can visualize, if desired, a base b element as a memory with b words or buckets of storage. Incoming data is stored in the bucket, whose address is determined by some digit in the incoming data.

A single base b element can be used as a base b radix sorter. During each pass all the items in the list are entered, one item at a time, and stored in the proper bucket; then the bucket contents are collected for the next pass. For a list of N items, the time required for a single pass is $O(N)$, and the number of passes is $\lceil \log_b \text{MAX} \rceil$, where MAX is the value of the largest item in the list. The total time required for a sort is then $O(N \lceil \log_b \text{MAX} \rceil)$. The speed of a base b radix sorter is limited by the time required for a pass and the number of passes required. The number of passes required is a function of b and MAX; increasing b decreases the number of passes until the limiting value of $b = \text{MAX}$ is reached, at which point the number of passes is one and the total time for a sort is $O(N)$.

Fundamental speed increases in the time required for a pass are limited by the need to enter the whole list sequentially. Introducing parallel (simultaneous) input of the items and using parallel processors or elements is the only way to obtain fundamental speed increases.

One can use, say, two base b elements in parallel and process two list items simultaneously; but either both elements must use the same buckets or there must be some way to merge the two sets of buckets. The first option introduces resource conflicts, so we choose the second. We consider first the design of this merging capability and then return to the design of a parallel sorter.

Figure 1 shows two parallel base b elements with their outputs lined up so that the 0s are together, the 1s are together, and so forth. Assuming each of these outputs is a bucket capable of storing a single item, after the two items have been processed by the elements, two of the buckets contain values and the remaining $(2b - 2)$ buckets are empty. To merge the contents of the two nonempty buckets onto the first two output lines, we use the network shown in Figure 2. In this network there is one line and one series (or level) of switches for each bucket. If a bucket is empty, its corresponding level of switches are all set to the right. If a bucket is nonempty, the switches are vertical. A standard inductive argument can be

used to show that if the switches are set as soon as the buckets are filled, and then the buckets' contents are output onto the lines, the contents of the two nonempty buckets are output on the two zero lines with the smaller value to the left.

Assuming two base b elements working in parallel followed by the network of Figure 2, it is clear that the time for a single pass is halved. In general, assuming P base b elements working in parallel followed by a similar network, the time for a single pass is divided by P ; that is, the time required is directly proportional to the number of parallel elements or processors. Since increasing the number of elements has no effect on the number of passes required, the total time for a P element radix sort is $O(\lceil N/P \rceil \lceil \log_b \text{MAX} \rceil)$, where MAX is the value of the largest item in the list. In the limiting case, when P equals N , the total sort time is $O(\log_b \text{MAX})$. If both P equals N and b equals MAX, the sort time reduces to $O(1)$, the ultimate limit.

To compute the efficiency of a parallel radix sorter with P elements, we note that the actual work performed is still $N \lceil \log_b N \rceil$. Dividing this by the potential work, P processors working in parallel for time $\log_b \text{MAX}$, gives an efficiency of

$$\frac{N \log_b N}{P \lceil N/P \rceil \log_b \text{MAX}} = \log_{\text{MAX}} N$$

which depends only on the ratio of N and MAX and is independent of the number of processors; that is, the useful work produced by each element is independent of the number of elements.

THE BALANCED TREE SORTER

A basic difficulty with the radix sorter is the dependence of the sort time upon the maximum value in the list rather than the number of items in the list. If the items in the list are 20-character employee names, even a short list requires many passes or large values of b . This section presents a sorter with the number of passes depending upon N rather than the maximum value in the list. We first consider the concept of sorting by using a balanced tree and then the design of a balanced tree sorter.

In a balanced tree all subtrees have the same height. Assume each node in a balanced tree contains $(b - 1)$ values and has b sons. An item in the list to be sorted is compared to $(b - 1)$ values, and the nearest value is used to determine which son node is used for the next comparison. As the item descends the tree, always using the nearest son for the next comparison node, sooner or later it reaches a node where a match is found or it reaches the bottom of the tree. Assuming that the node values have been correctly chosen so that the tree remains balanced, when all the items in the list have been inserted in the tree, the sorted list can be obtained from the node values. Since the tree is balanced, it contains $\lceil \log_b N \rceil$ levels.

The assumption that the exact node values are known in advance can be avoided by using a variation on Quicksort.³ Starting with original list and choosing $(b - 1)$ values at random from the list as the node values for the root node of the tree, this node can be used to insert each item in the list into a bucket corresponding to the correct son. The results of the first pass are then a set of b buckets or sublists with every item in one sublist preceding every item in the next sublist; that is,

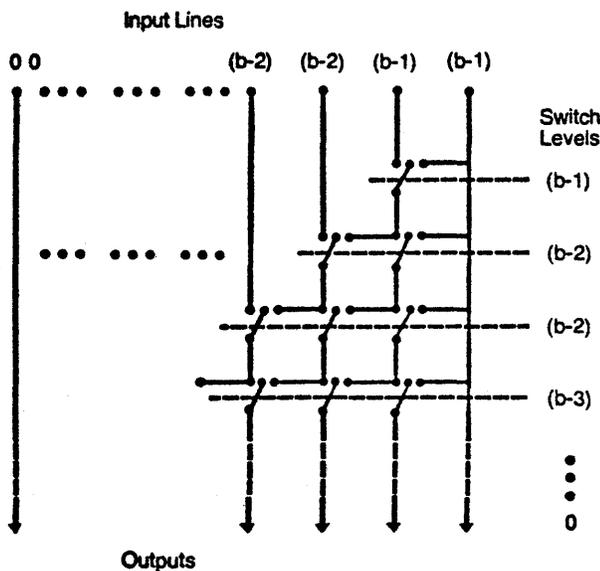


Figure 2—A network to collect the contents of the nonempty lines onto the beginning lines

the first pass generates b sublists with a sort ordering between the sublists.

Each sublist (the contents of one bucket) is now processed the same way the original list was processed with $(b - 1)$ node values chosen at random from the sublist. This generates b^2 sublists with a sort ordering between all the sublists. Continuing this process reduces the size of the sublists at each pass until they all contain a single item, at which point the list is sorted. (In practice, the process usually stops when each sublist contains less than b items.) Note that this process assumes that the b node values are themselves sorted by some means; we assume in the following that this is done and that b is small enough in comparison to N that this additional sort time is negligible.

This is essentially Quicksort, using $(b - 1)$ comparisons at each step rather than 1. As in Quicksort, choosing node values at random rather than using the exact values affects the time required less than one might assume. If the exact values were used so that the tree remained balanced, then $\lceil \log_b N \rceil$ levels are required to store the tree and $\lceil \log_b N \rceil$ passes are required. If the node values are chosen at random from the sublists, then simulation studies show that the mean number of passes becomes $c \lceil \log_b N \rceil$ where the value of c depends upon the value of b and the precise way the node values are chosen. If the node values are simply the first $(b - 1)$ values chosen at random from the list and b equals, say, 20, then c is about 1.3; in the worst case, for b equals 2, $c = 1.39$ and c decreases as b increases. While these values of c are mean values and the exact number of passes or comparisons depends upon the list values, the standard deviation in the number of comparisons is typically a few percent of the mean, so the time variation from sorting one list to the next is typically a few percent of the mean time.

To implement a balanced tree sorter, we use a b way comparator, a device that inputs an item, compares the value of the item to all $(b - 1)$ previously stored node values, and outputs the item to the bucket or position corresponding to the nearest node value. If a single b way comparator is used,

the $c \lceil \log_b N \rceil$ passes are necessary, so, assuming the time to sort the node values themselves is negligible, the total time to sort the list is $O(cN \lceil \log_b N \rceil)$. Although the constant c can be adsorbed in the big O , including the c makes explicit the dependence of the sort time on the method of choosing the node value.

As in the radix sorter, fundamental speed increases in the single-element balanced tree sorter can be obtained by increasing b or introducing parallel elements processing parallel input. Increasing b to the limiting value of N reduces the number of levels or passes to 1, but the time to sort the values used in the nodes is no longer negligible; indeed, it becomes the sort time. The time required to sort the values used in the nodes sets a practical upper limit on the value of b that requires further study.

Parallel elements can be introduced the same way they were introduced in the radix sorter, with the merging network of Figure 2 to collect the output. In the first pass, a single set of $(b - 1)$ node values is chosen and all the comparitors use this set of node values. In the second and later passes, the comparitors can use the same node values and the same sublists, or each comparator can be working on a different sublist with node values chosen from that sublist. Obviously during the first pass (and, on the average, during all passes) the time for the pass is reduced by the number of parallel comparitors. On the average the sort time is $O(\lceil cN/P \rceil \log_b N)$ where P is the number of comparitors. The efficiency is $1/c$, which is always greater than 0.5 and is independent of the number of comparitors and the list to be sorted; that is, this sorter approaches the optimal possible design.

Increasing the value of P to the limiting value of N reduces the sort time to $O(c \log_b N)$. Note that to attain this limiting value, after the first pass each comparator must be working on the sublist containing the item assigned to the processor. This requires some careful assignment of the workload.

CONCLUSION

The complexities of parallel sorting machines depend heavily on the sorter architectures, processors, data, and the software algorithm used. In this paper we have shown how sorters with sort times of $O(N \log_b \text{MAX})$, $O(\log_b \text{MAX})$, $O(1)$, $O(N \log_b N)$, and $O(\log_b N)$ can be achieved. The efficiency measure of the parallel sorters is proved to be useful in describing the behavior of the sorters. A new sorter with sort time of $O(c \log_b N)$ and efficiency $(1/c) > 0.5$ is proposed. This sorter could achieve near optimum with software support. Some hardware for the implementation of the new sorter is discussed. More implementation detail is reported in a separate paper.⁴

REFERENCES

1. Batcher, K. E. "Sorting Networks and Their Applications." *Proc. 1968 SJCC, AFIPS*, Vol. 32, pp. 307-314.
2. Chen, T.C., V.Y. Lum, and C. Tung. "The Rebound Sorter: An Efficient Sort Engine for Large Files." *Proc. 1978 International Conf. on Very Large Data Bases*, pp. 312-318.
3. Knuth, D. E. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. Addison-Wesley, 1973.
4. Winslow, L. E., and Y. C. Chow. "The Analysis & Design of Some New Sorting Machines."

NETWORK TECHNOLOGY

Packet communication of online speech*

by DANNY COHEN

USC/Information Sciences Institute
Marina del Rey, California 90291

ABSTRACT

The rapid progress of both communication and processing technologies in recent years constitutes a challenging technological revolution, epitomized by the photographs sent by Voyager from Saturn.

This revolution offers new opportunities for many avenues of technology. Real-time speech communication is one of them. This paper discusses the use of packet switching, the emerging computer communication technology, for online speech application.

The paper reviews the computer communication technologies such as packet and circuit switching, datagrams, and virtual circuits. It mentions the voice encoding (vocoding) algorithms and their implementation and discusses the interaction of vocoding and packet-switching technology. It argues about the issues of network voice protocols and introduces applications of online speech.

After describing the experience with packet speech, the paper concludes with several ideas about the future of the field.

INTRODUCTION

The rapid revolution in computer communication technology in recent years has enabled computer networks to offer a new means of communication for online speech† in addition to the conventional data communication for which they are typically designed.

To communicate speech over computer networks, it is necessary to digitize the speech. Until recently the process of digital speech encoding ("vocoding") was either prohibitively expensive or resulted in data rates that were impractically high for the available computer communication networks. There-

fore, online digital speech communication was possible but not practical until several breakthroughs took place in communication technology, in processing technology, and in the mathematical understanding of the signal processing necessary to support it.

The cost of digital speech is decreasing at a remarkable rate, matched only by processing power and communication. In this inflationary world these may be the only two commodities whose prices keep dropping. Therefore, even though the cost of digital speech may still appear prohibitive, the situation is bound to change in the near future.

The advantages of digital speech are many. Basically, digital speech can be processed by computers for a wide variety of applications, such as speech understanding, storage and retrieval, sophisticated editing, and security. Digital speech can be reamplified indefinitely without any loss of signal fidelity, and it also can benefit from the use of many error correction schemes for overcoming low-quality communication lines.

Traditional voice communication (telephone) networks have failed to provide the required services for computer communication, giving rise to the existence of computer communication networks. Now that these computer communication networks exist, the dual question arises: Are computer communication networks adequate for online speech?

The positive answer to this question is probably due more to the fact that data networks are more modern than voice networks and benefit from all the experience accumulated so far in the field of communication, especially from the modern concept of augmenting the conventional online communication subsystems with processing (and storage) capabilities.

In the second section, circuit switching and packet switching are compared. This issue is akin to the comparison of voice and data networks.

The process of digitizing voice provides several opportunities for engineering tradeoffs, such as processing vs. quality, communication data rate vs. quality, and processing vs. data rate. For any given communication data rate, the more processing applied, the higher the quality. The more data communicated with a given amount of processing, the higher the quality. Consequently, since more processing is applied, fewer data must be communicated to achieve a certain voice quality. Hence, high-quality voice communication requires a high rate of communication or high-level processing.

*This research is supported by the Defense Advanced Research Project Agency under Contract No. DAHC15-72-C-0308. Views and conclusions contained in this report are the author's and should not be interpreted as representing official opinion or policy of DARPA, the U.S. government, or any person or agency connected with them.

†Throughout this note online speech is understood as interactive speech communication between people such that they have the ability to interrupt each other at any time—unlike push-to-talk systems.

This gives rise to the question of how much to compress the speech to optimize the overall speech communication system. One extreme is minimizing the processing requirements by using PCM (pulse code modulation). This yields approximately a 64Kbps data rate for achieving telephone quality. The other extreme is minimizing the data rate while achieving similar quality. This results in computational requirements of about a million instructions per second (1Mips) and a data rate of about 2Kbps. Between these two extremes there are many points best suited to certain system requirements—for example, CVSD (continuously varying slope delta modulation) technique at 8, 16, or 32Kbps.

To make packet speech practical, it is necessary to have vocoding algorithms that guarantee high-quality speech communication at a reasonable data rate and that may be implemented by practical hardware. Issues about both the vocoding algorithms and the hardware for their implementations are discussed in the third section.

Packet communication has cost and reliability advantages when compared with circuit switching. This is due to the ability to augment the communication subsystem with processing capabilities that can increase the reliability of the total end-to-end communication beyond that of any of its individual subsystems, as well as to the ability to exploit the statistical properties of the activity distribution, thus catering to the averages, not to the maximal (worst-case) levels.

Unfortunately, the same processing capabilities that increase both the system reliability and the cost effectiveness tend also to introduce variable delays into the end-to-end communication. The variability of the delay is due to the dependence on the total level of activities ("load") in the network, which depends on the quantity of traffic caused by all other users of the same network.

The ability to cope with these delays and their variability is one of the major issues in handling packet speech. The interaction of the packet-switching technology and speech communication is described in the fourth section.

One of the most important lessons learned has been that software is as complicated and important to the total system as hardware. In the early days of computing many believed that once the computer hardware was operational, the rest was just a small matter of programming. We have learned that this is not so, and that the software of significant systems is as difficult (and expensive) as the hardware.

A similar lesson was learned about communication. The mere delivery of bits from one computing system to another is only the beginning, just as computer hardware is only the beginning. After accomplishing this delivery one has to make sure that the higher-level meaning is communicated and that information is transferred, not just data.

The technique of getting the meaning (information) across the communication system is handled by the communication protocols, which may be described as the software, or the programming language, of the communication system.

Since data-oriented protocols are not suited for online voice communication, new network voice protocols have to be devised. Such protocols are described in the fifth section, with the rationale leading to their development.

The ability of computers to handle online speech, together

with the large body of already existing capabilities of computers, may be used for a variety of new applications, discussed in the sixth section.

The seventh section describes some of the experience gained and some of the milestones in the short history of packet speech.

The last section states several conclusions about the future of packet speech communication.

ON COMPUTER COMMUNICATION NETWORKS

The ability to communicate data (i.e., bits) over communication lines is relatively old. Modems for a variety of communication lines (especially telephone lines), at a variety of speeds, have existed for many years.

The switching capability is what makes a collection of such communication lines a communication network. This switching allows the interconnection of N users without having to use facilities of N^2 size. The total capabilities of a communication network depend on both the communication capabilities of the individual links and the processing capabilities of the switching nodes.

There are two basic approaches to the switching issue. One is to select a set of physical links and to dedicate them to the desired communication path as long as they may be needed. The other is to dynamically timeshare the communication links as required. The former approach is called circuit switching (CS), and the latter is called packet switching (PS).

Hence, circuit switching requires that communication paths be set up before they are actually used. However, once such a path is established, it is guaranteed to be available for the entire duration of the communication, even in periods when it is not in use.

These circuits are not necessarily implemented by distinct hardware. They could be implemented in that way (as in old manual telephone exchanges) or be multiplexed on the same hardware by using any of the many available multiplexing techniques, such as frequency- or time-multiplexing (e.g., TASI). For this discussion they appear as if they are distinct.

An important feature of these circuits is that they consume certain resources, namely a portion of the communication bandwidth (actually data rate). As mentioned before, this has the advantage that once a path is set up, there is no delay associated with each further use; but it also has the disadvantage that these dedicated resources are consumed when there is no need for them. Hence, resource management and utilization always appear as a worst-case situation.

On the other hand, packet switching employs a totally dynamic resource allocation computed according to the actual demand. This has the advantage that the resource allocation can adapt to the dynamic performance and, for example, give all the available resources to a certain user if he/she is the sole active one at the time. Hence, under favorable statistics, such as a high peak-to-average ratio, packet switching can offer much better performance than circuit switching.

In addition, in situations where communication lines may change their performance at a very rapid rate—failures included—packet switching has the advantage that it can recover faster than circuit switching and reestablish connection in alternate routes.

The choice between circuit and packet switching depends on the traffic distribution and the characteristics of the resources in each particular situation. Typical computer communication is bursty and requires a low-duty cycle of high-data-rate communication. Therefore, in nearly all circumstances packet switching is the preferred technology for implementing cost-effective computer communication.

There are two ways of providing communication services on top of either CS or PS systems. One is by virtual circuits (VC) and the other by datagrams (DG).

Note that even though the services of virtual circuits and of datagrams are akin to the implementation schemes of circuit switching and packet switching, respectively, it is possible to implement either service (VC and DG) on either implementation scheme (CS and PS).

Virtual circuit communication is a discipline of communication where the communicating processes set up a virtual circuit between themselves, which has all the properties of a reliable error-free circuit. However, performance, as measured by data rate and delay, may vary.

The processing power of the switching nodes of the communication subsystem is used to guarantee the error-free reliable delivery of the data at the receiving end. This is accomplished by using a variety of techniques, such as software checksums, sequence numbers, and exchange of acknowledgments. In this mode of communication there is no concept of data granularity, bits, bytes, or records of any arbitrary size.

Datagram communication is the discipline in which the end-to-end communication is packaged in units of datagrams. The processing power of the switching nodes may be used to guarantee the integrity of each individual datagram, but it does not guarantee that all of them are delivered necessarily in the proper order. As a matter of fact, both loss and duplication of datagrams are possible under this discipline.

One of the important characteristics of datagram communication is the totally independent treatment of each individual datagram, without the use of any stored *state information* regarding the *connection* to which this datagram may belong. This differs from virtual circuit systems, which require that (in addition to the data communicated) the system interface must support the handling of *a priori* information about the connection, such as the connection setup.

Note the similarity between the virtual circuit concept and circuit switching and between the datagram concept and packet switching. Note also that in spite of these similarities, in many situations virtual circuits are implemented either by lower-level datagrams² or directly by packet switching,³ which in turn may be implemented by lower-level circuit switching.

This may sound confusing, but in practice it is very simple because of the modularity of the system design, which separates processes from details and idiosyncrasies of the processes on the other side of these well-defined interfaces.

A complete description and thorough treatment of virtual circuits and packet switching can be found in the *IEEE Proceedings*, Special Issue on Packet Communications,¹ which includes many articles on the subject.

As mentioned above, the use of processing makes it possible to achieve reliable end-to-end communication through software checksums, acknowledgments (both positive and negative), retransmissions, and sequencing. The key concept

is that the receiving node does not deliver to its customer any data unless it is absolutely sure about the data's integrity; at the same time, the sending node keeps retransmitting the data until it is convinced that the data have arrived properly at the receiver.

It is possible to prove mathematically that if the probability of a packet to traverse the communication link successfully is positive, even arbitrarily small, then its probability of being correctly delivered to the end receiver may be made arbitrarily high by using checksums, acknowledgments, and retransmissions. Unfortunately, this augmentation of the communication subsystem with processing introduces random delays, and distribution is highly dependent on the overall network performance.

In any well-designed network the performance parameters are tuned so that the disadvantages of the extra delay, introduced to increase the communication reliability, are outweighed by the advantages of the achieved reliability.

It is typical of most computer data transactions that no communication errors can be tolerated. In some applications the communication is decoded sequentially so that any error may cause a chain of undesired effects. In other applications, such as file transfer, the use of the data may be most sensitive to any error, such as missing a pointer in a database or jumping to a wrong location in a program.

Therefore, data integrity is typically the dominant factor in the design of computer communication networks. Performance is considered a secondary factor.

IMPLEMENTATION: ALGORITHMS AND HARDWARE

To make packet voice communication practical, it is necessary to perform real-time vocoding with hardware of reasonable size and cost. Until recently this task was not well understood mathematically, and it required computations at rates that could be provided only by super-computers.

As a result of the work of many researchers in several institutes, a better algorithmic understanding of the vocoding task has recently been achieved. Recent developments in computer technology and architecture as well as in the VLSI field allow the implementation of these algorithms by a compact hardware configuration.

The major classification of the vocoding techniques are the time and the frequency domain techniques. The former class includes techniques like PCM and CVSD and is designed for the reproduction (at the receiver's end) of voice signals that *look* as close as possible to the original input signals (at the transmitter's end).

On the other hand, the frequency domain techniques, also called the acoustic or spectral techniques, are designed for the reproduction of voice signals that *sound* as close as possible to the original input signals. A variety of such vocoding techniques may be found in Flanagan⁶ and Rabiner and Schafer.⁸

In recent years linear prediction coding (LPC) (Markel⁷) has emerged as the dominant vocoding technology for (relatively) low-data-rate (narrowband) applications in general and packet speech in particular. LPC may be viewed as a hybrid between the time domain and the frequency domain

techniques. It has some features that are best understood in the time domain, like the prediction, and some that are best understood in the frequency domain, like its logarithmic spectral matching.

The companion paper⁴ describes the highlights of the efforts of a cooperative group of researchers, spread across the country from Massachusetts to California, collaborating on the algorithmic issues for packet speech.

The computation rate required for most of the vocoding techniques is of the order of magnitude of millions of instructions per second. The development of special- and general-purpose peripheral-array processors made the real-time pursuit of these algorithms feasible.

Recent development of VLSI components further reduced the size of the vocoder to just a few chips. The Speak-&Spell™ toy (made by TI) was the first entry of this technology into the consumer market. Since then it has been followed by other devices by various vendors, implementing LPC and other speech synthesis systems. Chips with both full LPC synthesizers and storage of many canned words are available on the market at low cost.

The companion paper⁵ describes the architectural considerations for using modern technology in the implementation of a practical (i.e., small and low-cost) packet voice terminal.

SPEECH AND PACKET SWITCHING NETWORKS

In the second section, packet communication was discussed. It explained how computer communication networks achieve a high degree of data integrity by using processing power.

These systems typically are designed for computer data where timeliness (i.e., delays and data rate) is secondary to data integrity. Unfortunately, this is not always the right priority for online speech communication. In online speech communication the loss of a short burst of speech may be less harmful to overall voice quality than the time that may be required for retrieval of a better copy across the network. This important difference between data and online speech communication is also reflected in the way the flow control should be applied. This example is an instance of the difference between real-time and non-real-time applications. (Note: real-time, not online. Some discussions about these real-time issues may be found in Cohen's work.^{11,12})

These differences necessitate the use of communication protocols designed to cater to real-time applications, rather than the more conventional protocols, such as those used for file transfer and terminal connections. Such a protocol is described below in the fifth section.

The variability of the delay may cause intermittent gaps in the reconstruction of speech at the receiver's end. Such discontinuities have a negative effect on the quality of speech communication. Techniques for overcoming this problem are discussed by Cohen.⁹

A frequent question is: How long can the absolute delay be between the instant an utterance is spoken and the instant it is heard?

The answer is that for a noninteractive (e.g., push-to-talk [PTT]) speech communication system any practical delay can be tolerated. But for interactive speech communication it is

important to minimize this delay. A delay of up to about 200 milliseconds is unnoticed. However, noticeable delays degrade only slightly the interactive nature of the communication. There is no absolute critical value beyond which the system fails to be interactive. This may be up to several seconds!

By using more resources the total end-to-end delay can typically be reduced. The engineering decision about the value of this delay is a simple matter of economics, equivalent to the question of quantitative evaluation of the cost of the delay (or: How much are you willing to pay to reduce the delay further?).

As mentioned before, the delay variance (not its value) is the more difficult issue to deal with. This delay variance is introduced into communication systems by the packet switching nodes, which are time-shared between several communication tasks and whose performance depends on the total traffic served.

Multihop communication systems with several communication nodes tend to introduce more delay variance than single-hop systems, independent of the actual value of the delay itself.

Both local networks (e.g., the Ethernet¹⁰) and satellite channels provide high data rates with fixed delays (the former very small and the latter very large). On the other hand, terrestrial networks typically have lower data rates but tend to introduce delays with significant variance as a result of the multitude of processing nodes used by these networks.

In situations where communication resources are plentiful and processing is limited,¹⁰ there is no need to use any speech compression, and raw samples can be transmitted. However, in most practical situations this is not the case, and vocoding is essential.

The difficulties associated with handling the variable delay may suggest the use of circuit switching instead of packet switching for voice communication, or some hybrid combination of the two. The rationale behind this suggestion is that by establishing virtual circuits the dynamics of datagram handling (by the communication nodes) may be reduced, hence improving the delay variance.

The question of the proper mix of pure packet switching and virtual circuits for packet speech is very controversial. The results of any scientific treatment of this subject are sensitive to the choice of assumptions, e.g., communication parameters and traffic distribution. Several such discussions are cited in the reference section.²¹⁻²⁶

A detailed analysis of the subject was performed by NAC¹³ for the expected data and voice communication requirements of DOD (consistent with Autodin and Autovon). For the assumptions chosen for that study, probably typical of some terrestrial networks, the results favor pure packet switching over hybrid configurations with virtual circuits. This analysis shows a very low sensitivity to the exact traffic distribution.

However, under the assumption of very high setup cost (measured in units of time), such as exists in networks based on satellite channels, it is not clear that the same result still holds.

For that reason experiments are being conducted now in order to identify (and verify) the optimal strategy for satellite-based communication networks under a wide variety of traffic

loads, especially when these networks are operating in conjunction with terrestrial nets.

Currently it is expected that a mixed mode will be used, one that employs pure packet switching over the ground and some reservation scheme for the satellite links.

PROTOCOLS FOR ONLINE PACKET SPEECH

Protocols may be viewed as the programming languages used by communicating processes in order to program each other. It is the task of the protocols to convey the information to be communicated by using the bits actually delivered across the network.

Protocols, like programming languages, are used to offer virtual services on top of existing ones of lower levels. For example, by using the right protocols a lower-level datagram service may become a higher-level virtual circuits service.

It is advantageous to fit the protocols (like programming languages, again) to the specific problems they are expected to solve. However, protocols very efficient for a certain problem may be less efficient, or even totally unfit, for a different class of problems.

Counter to the move to tailor protocols to specific problems, there is a move toward the use of general-purpose protocols, for exactly the same reasons that programming languages are designed as general-purpose rather than special-purpose tools. Protocols should be designed in a layered and modular way that lends itself well to a variety of applications without much loss of efficiency, while being able to take advantage of a large body of existing solutions to previous problems.

As mentioned above, it is the task of protocols to offer higher-level services on top of existing ones of lower level. For example, TCP² and X.25¹⁴ are protocols that provide reliable virtual circuits on top of less reliable packet switching systems. Both these protocols are very important, because most of the transactions typical to computer communication cannot tolerate any damage to the data integrity.

Figure 1 shows the three most important attributes (dimensions) of computer communication: reliability, data rate and end-to-end delay. These attributes are shown as a triangle, rather than three-dimensional Cartesian coordinates, in order to emphasize that it is not possible to achieve all these attributes simultaneously and that in order to improve one, any of the other attributes may be compromised.

Typical online computer file transfer requires both high reliability and high data rate, as depicted by Point A in the figure. On the other hand, other online applications, like terminal communication, require high reliability and low delay, as depicted by Point B.

Speech communication, like most real-time applications, requires both high data rate and a low delay, even at the possible expense of reliability, as depicted by Point C.

As Cohen¹¹ notes, the typical property of real-time communication is that once newer information is available, older information is of less importance and is not necessarily worth pursuing at any delay (and data rate) cost. This is not so for the non-real-time, more conventional computer communication. Any information that is sequentially decoded ("un-

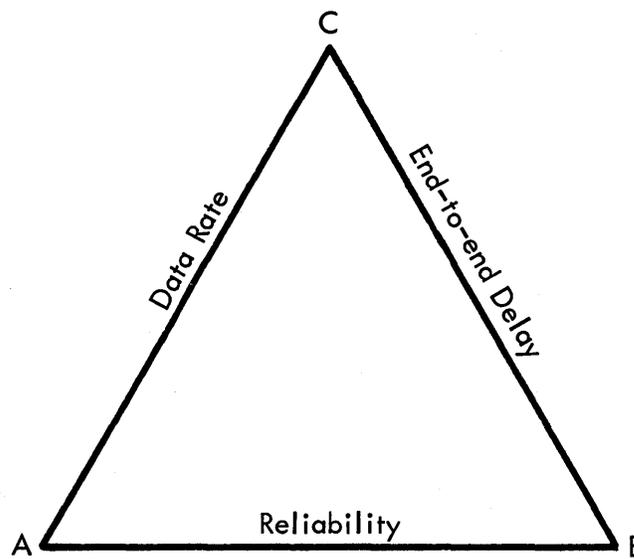


Figure 1—The major attributes of computer communication

derstood") must be delivered reliably and in the proper sequence. For such a communication, newer information is totally worthless until all the previous information is properly received.

In typical non-real-time communication one process sends another process information that changes its state. Any damage to such information may yield loss of synchronization between these processes and lead to severe undesired results. Therefore, when the integrity of any part of the communication is not assured (e.g., when any error is discovered), it is necessary to halt progress until the missing part is reliably recovered. However, for real-time applications (such as weather reporting) it is wrong to delay updated information in order to retrieve old information.

Speech communication is of the real-time type. Damaged short speech bursts may be less harmful to the total communication process than the retrieval of an undamaged version of them, especially through networks with relatively long delays.

Real-time speech communication, unlike non-real-time applications, requires certain communication data rates. Lower data rates cannot support the task, and access rates do not provide any advantage. File transfer, on the other hand, has no intrinsic requirements for bandwidth and data rate except that higher data rates obviously yield a more rapid transfer of large files. No data rate is too low for file transfer, and any access rate is always useful.

This intrinsic bandwidth has important implications to the flow control schemes used for such communication, as described by Cohen.¹¹

Because of these important differences between non-real-time computer communication and real-time speech communication, the traditional standard computer communication protocols were not suitable for speech applications. This fact gave rise to the development of special speech-oriented protocols.

Since speech is a particular instance of the more general class of real-time applications, a typical network voice proto-

col should be viewed as another layer on top of a more general real-time communication protocol layer.

A network voice protocol, NVP, should have the following features:

- Separation of the handling of data and control
- Independence from vocoding techniques
- Real-time binding of vocoding data format
- Support of higher-level application protocols
- Ability to adapt to variable network performance

A network voice protocol designed according to these objectives is described by Cohen.¹⁵

Any network voice protocol has to deal with the following issues: call progress, real-time communication, and handling of speech data. These issues are discussed below.

Control deals mainly with call establishment, monitoring, and termination. It is also responsible for basic decisions about addressing and routing, choice of communication modes and parameters (if any), and the support of higher-level protocols for various applications—conferencing, for example. These issues are independent of the particular choice of vocoding technique in use, of the details (e.g., formats) of the underlying supporting communication network, and of the specific real-time communication issues.

An important attribute of voice protocols is their ability to support a variety of local features through different systems. For example, different styles of user-interface schemes may be used by different systems while being supported by the same protocol.

The loss of control messages causes severe problems. But the loss of voice data messages causes only intermittent (transient) glitches. Damage to control messages can cause major problems, from which the system may not recover by itself. For example, errors in addressing or in communicating the choice of vocoding technique can make communication impossible over that connection.

The integrity of control messages is more important than that of voice data messages. On the other hand, the timeliness of the control messages is less critical than that of the voice messages. Therefore, it is advantageous to separate control and data-handling processes (or protocol layers), using reliability mechanisms (like acknowledgments and retransmissions), even at the cost of significant delays, for assuring the integrity of the control while using other faster real-time mechanisms for the voice data.

Several vocoding techniques may be used for packet speech. Each vocoding technique may have several dialects, which differ in the particular values of certain parameters, such as sampling rate (for PCM and CVSD) or a number of poles (for LPC). The incompatibility of vocoding techniques and their parameters is as severe as the incompatibility of languages: A perfect match is necessary to establish intelligible communication.

Modern voice terminals are capable of using more than just one vocoding technique with several parameter settings. It is the task of the control protocol (actually, the control portion of the voice protocol) to assure the compatibility of the vocoding technique in use.

Even with the variety of vocoding techniques in use, the

control protocol is essentially the same. Therefore, it is important that it be designed so that it is independent of the vocoding technique.

The actual binding of the vocoding technique should be deferred to the time when the communication is established (runtime). This allows the use of higher-data-rate vocoding, for better quality, when the communication system can support it, and of lower-data-rate vocoding when so dictated by lack of communication resources. It is interesting to note that the same control protocol can manage not only voice connections but also connections of different modalities, such as graphics.

One important part of the voice protocol is the interface to the lower-level communication system. The same control protocol should be able to operate on top of several different communication disciplines while using possibly different parameters for each. The binding of the actual communication system used should also be deferred to runtime.

When the network interface is kept as a separate module, the transitions from the use of a particular network to another one are (relatively) easy to implement, since the details of the lower-level transport mechanism do not have to affect the other issues handled by the protocol, such as call progress and handling of vocoding data. Network voice protocols can and should be designed for operation in a general internetworking environment by using specific networks through a layer of internet protocol.^{29,30}

Since voice communication is an instance of real-time communication, the voice protocol should be real-time-oriented. This includes issues like time-stamping, the ability to recover from damage and loss of data (also known as robustness), and flow control. Since real-time communication is typically rate-based, rather than data-based (like file transfer), a different flow control strategy should be used. This flow control should be based on maintaining timely levels of resources (like buffers) rather than on quantities ("windows").

The flow control mechanism may occasionally fail to protect the communication data. Typically this failure is caused by unforeseen changes of the total system load on the communication, which are due to the activities of the other users. Any flow control mechanism that never fails must be over-protective, hence limiting the overall performance of the network by treating all situations as worst cases. (Note that this is a typical type-1/type-2 error tradeoff.)

When such intermittent failures occur, data must be discarded. In the case of real-time communication it is the older data that are discarded in favor of newer data.

Generally, the performance that packet-switching networks offer to their users may fluctuate as a result of the changes of activities by other users. Since voice communication depends on network performance, it is important that the voice protocol be capable of monitoring this performance and adapting to it dynamically. For example, under heavier network load it may be advantageous to use fewer, longer messages in order to achieve a certain data rate—at the possible cost of delay. Under a heavier load the proper thing may be to switch to a vocoding scheme of a lower data rate, if possible.

An important task of the voice protocol is to extract as much performance as possible from the supporting communication networks and to make it available for its users. This

may require the introduction of a multiplexing capability to the protocol in some cases, and the establishment of connection over virtual circuit networks. A new real-time protocol capable of performing these tasks is now under development.¹⁶

In summary, the function of a network voice protocol is to augment the native capability of packet-switching networks in order to provide the support of real-time packet speech. Such a protocol should be designed and implemented in a structured way by modules that independently handle the separate issues of control, real-time communication, and vocoding technology.

APPLICATIONS

There are many applications for speech, ranging from person-person communication to person-machine and machine-person interfaces.

Person-person communication consists mainly of interactive speech between people. However, in some situations offline speech is required. Since such voice messages are not communicated in real time, it is possible to employ reliable communication data transfer protocols, since the resulting additional delay does not degrade the (already lost) interactive nature of the communication.

In some applications, voice messages may be used just like any other conventional data file. In these applications, a voice message is recorded locally into a file; then this data file is communicated across the network, using any standard reliable file transfer protocol, and finally it is played back by the receiver's system. However, if either the originator or the receiver, or both, are not capable of storing the entire voice message, there is the need to communicate it in real time, even though the voice communication is no longer interactive.

A protocol for a network voice message system (VMS)¹⁷ was developed and interfaced to another all-text network electronic mail system.

Multiuser real-time voice conferencing over a packet-switching communication network is another application. Unfortunately, it is not possible to add narrowband vocoded data to get the combined voice, as is done for analog conferencing. LPC and other efficient speech compressing techniques are not linear, and their data are not additive. This causes some difficulties in devising narrowband packet speech conferencing schemes, which can be circumvented by using appropriate conferencing protocols.¹⁸ Experiments with such narrowband packet conferences have been successfully conducted across the United States by using the ARPAnet and across the Atlantic by using the ARPA Atlantic Satellite network.¹⁹

In addition to conferencing and to voice messages used between people, packet voice may be used for the cross-network support of advanced person-machine (and machine-person) interfaces.

In order to support person-machine voice communication ("voice input"), it is necessary to communicate the voice with such quality that the voice recognition process can still be applied in spite of the transformations occurring to the voice. Many voice recognition systems²⁰ perform the recognition task by using the very same parameters communicated by

LPC vocoders. Hence, the packet communication of voice does not degrade its recognition capabilities, if LPC is used.

The dual-application, machine-person communication is the easier of the two. This interface is typically a computer-controlled scheme of playing back stored (canned) voice data, and the computer task is to choose which portions of this stored voice should be played and in which order. Such applications are already in the consumer market, from spelling aids and chess-playing machines to voices in cars.

Because of the efficiency (compressedness) of vocoded voice it is typical to find that stored speech is LPC-vocoded and is sent via any available communication lines to an integrated (single-chip, typically) synthesizer for playback. The same speech, of course, could just as well be communicated across packet-switching networks to remote devices.

In summary, voice applications that can be implemented on local systems can also be implemented across packet-switching communication networks.

EXPERIENCE

In October 1973 ARPA initiated a packet speech program. Its objective was to develop the technology required for practical narrowband digital packet voice. The program pursued several research avenues: the development of the vocoding algorithms for packet speech,⁴ the development of hardware for implementing these algorithms,⁵ and the packet communication technology required for this task.

In order to develop the packet technology support for online speech applications, several experiments were conducted with medium-band vocoding technology. In August 1974 real-time packet voice communication was demonstrated over the ARPAnet between ISI (California) and Lincoln Laboratory (Massachusetts) by using CVSD at 8Kbps. In December 1974 narrowband LPC was demonstrated over the ARPAnet between CHI (Santa Barbara, California) and Lincoln Laboratory. In January 1976 online conferencing capabilities were demonstrated between ISI, CHI, SRI and Lincoln Laboratory. In November 1979 online conferencing capabilities were demonstrated between Washington D.C., Norway and London, over the ARPA Atlantic Satellite network.²⁷ Since then many other applications, such as voice messages, were developed and demonstrated by several organizations.

One of the most popular media for packet speech is a high-band local network.^{28,10} Since highband local networks typically can support data rates one to two orders of magnitude higher than what is needed for telephone-quality non-compressed speech (64Kbps), many of these systems do not use any speech compression techniques for their own local voice traffic.

From all the experience gathered so far, it is evident that packet-switching computer communication networks are capable of supporting real-time speech application.

CONCLUSIONS AND SUMMARY

Packet-switching communication technology has proved capable of supporting real-time speech applications. Recent devel-

opments in signal processing, vocoding algorithms, and hardware have made the implementation of packet voice terminals a very practical means of voice communication. The interest that several telephone companies have shown in packet speech and the amount of effort they are investing in it are a significant testimony to its importance and potential. In the near future we can expect to see more applications of voice, not only for human communication but also in many new aspects of person-machine communication. We expect applications like multimedia conferencing, multimedia electronic message systems, and voice (telephone) based access to databases. We also expect packet voice to be carried by integrated networks that will carry both data and voice.

REFERENCES

1. Kahn, R.E., ed., Proceedings of the IEEE, November 1978. Special Issue on Packet Switching Networks.
2. Postel, J.B., ed., "DOD Standard Transmission Control Protocol", IEN 129, RFC 761, USC/Information Sciences Institute, NTIS ADA082609, January 1980. Appears in Computer Communication Review, Special Interest Group on Data Communications, ACM, 10:4, October 1980.
3. Opderbeck, H. and Hovey, R.B., "Telenet—Network Features And Interface Protocol", NTG-FACHBR (Germany) Vol. 55, 1976. pp. 145-56.
4. Markel, J.D., "Highlights of a Group Effort in Algorithmic Development for Packet Switched Voice Networks", in these proceedings.
5. Blankenship, P., et al. "Hardware Modularity for Packet Voice Terminals", in these proceedings.
6. Flanagan, J.L. *Speech Analysis, Synthesis, and Prediction*, New York, Springer-Verlag, 1972.
7. Markel, J.D., and Gray, A.H., Jr., *Linear Prediction of Speech*, New York, Springer-Verlag, 1976.
8. Rabiner, L.R., and Schafer, R.W., *Digital Processing of Speech Signals*, Englewood Cliffs, N.J., Prentice-Hall, 1978.
9. Cohen, D., "Issues in Transnet Packetized Voice Communication", Proceedings of the 5th Data Communication Symposium, September 1977.
10. Shoch, J.F., "Carrying voice traffic through an Ethernet local network—a general overview," Xerox Parc Technical Report, Aug. 1980. Proceedings of the IFIP WG 6.4 International Workshop on Local-area Computer Networks, Zurich, August 1980.
11. Cohen, D., "Flow Control for Real-Time Applications", Computer Communication Review, January 1980.
12. Cohen, D., "The Oceanview Tales", ISI Report ISI/RR-79-83.
13. Frank, H., and Gitman, I., "Economic Analysis of Integrated DOD Voice and Data Networks", a NAC report for ARPA, September 1978.
14. Rybczynski, A., et al., "A New Communication Protocol for Accessing Data Networks—the International Packet Mode Interface" NCC, 1976, pp. 477-482.
15. Cohen, D., "A Protocol for Packet-Switching Voice Communication" Computer Networks, September 1978.
16. Forgie, J.W., "ST—A Stream Oriented Protocol", an unpublished memorandum.
17. Cohen, D. "VMS—Voice Message System", Proceedings of the International Conference on Computer Message Systems, Ottawa, April 1981.
18. Cohen, D. "Network Voice Conferencing Protocol—NVCP", NSC note 113.
19. Chu, W.W., et al., "Experimental Results on the Packet satellite Network", Proceedings of the NTC, Washington, D.C., November 1979.
20. Reddy, R., ed., *Speech Recognition*, Academic Press, New York, 1975.
21. Ross, M.J., and Sidlo, C.M., "Approaches to the Integration of Voice and Data Telecommunication", Proceedings of the NTC, Washington, D.C., November 1979.
22. Forgie, J., and Nemeth, A., "An Efficient Packetized Voice/Data Network Using Statistical Flow Control", Proceedings of the ICC, Chicago, June 1977.
23. Ross, M., Tabbot, A., and Waite, J., "Design Approaches and Performance Criteria for Voice/Data Switching", Proceedings of the IEEE, September 1977.
24. McAuliffe, D., "An Integrated Approach to Communication Switching" Proceedings of the ICC, Toronto, June 1978.
25. Rudin, H., "Studies on the Integration of Circuit and Packet Switching", Proceedings of the ICC, Toronto, June 1978.
26. Forgie, J.W., "Speech Transmission in Packet-Switched Store-and-Forward Networks", Proceedings of the NCC, 1975, pp. 137-142.
27. Mills, D.L., "SATNET Demonstration", Proceedings of the NTC, Washington, D.C., November 1979.
28. O'Leary, G.C., "Local Access Area Facilities for Packet Voice", Proceedings of the ICC-80, Atlanta, October 1980, pp. 281-286.
29. Postel, J.B., ed., "DOD Standard Internet Protocol", IEN 128, RFC 760, USC/Information Sciences Institute, NTIS ADA079730, January 1980. Appears in Computer Communication Review, Special Interest Group on Data Communications, ACM, 10:4, October 1980.
30. Boggs, D.R., Shoch, J.F., Taft, E.A., Metcalfe, R.M. "Pup: An Inter-network Architecture" IEEE Transactions on Communication 28:4, April 1980, pp. 612-624.

Highlights of a group effort in algorithmic development for packet-switched voice networks

by J.D. MARKEL

Signal Technology, Inc.
Santa Barbara, CA 93101

INTRODUCTION

Since 1973 the Defense Advanced Research Project Agency (DARPA) has sponsored several research groups engaged in a program of developing new and improved algorithms for demonstrating low bit-rate speech transmission across a packet-switched network. The first real-time packet-switched voice system was demonstrated by this group in 1974.¹

The purpose of this paper is to present a few of the highlights of the program from the perspective of one of the participants. It is based upon inputs from a number of researchers listed in the acknowledgments section. The opinions expressed in the conclusions section, however, are strictly those of the author.

The focus of this presentation will be on linear predictive coding (LPC) approaches since these have become dominant during the 1970s and are likely to remain so during the 1980s.

After a brief review of the structure of an LPC analyzer/synthesizer system, various components of the system will be discussed. Due to the amount of research and publications that have been developed from this work and others, the focus here will be on major aspects that relate either to the packet-switched characteristics of the system or to those that have not been discussed in detail elsewhere.

Finally, the author ventures an opinion on the status of the field: its successes, failures, and future directions.

STRUCTURE OF ANALYSIS/SYNTHESIS SYSTEM

The primary focus during the 1970s was towards linear predictive coding (LPC) systems, due to their perceived promise of better quality and ease of hardware implementation as faster processors were developed. This focus is likely to remain unchanged through at least the early 1980s since hardware has been developed for both government² and commercial³ use.

A simplified block diagram of an LPC voice-coder (vocoder) is shown in Figure 1. The incoming speech is first processed by speech conditioning algorithms to produce a signal more closely matching the idealized speech analysis

model. In many systems this topic is not considered, since the input signal is assumed to be speech of good quality with little or no degradation (due to carbon-button telephone distortion, and additive background noise, for example). However, these idealized conditions are seldom met in practice. Parameter analysis consists of linear predictive spectral parameter and gain analysis, and excitation analysis.

The spectral analysis algorithms process the speech signal to produce a small set of coefficients (usually 8–12 in number) and amplitude or gain term which represent a spectral model of a short speech segment (such as 10–20 msec). The excitation analysis algorithms produce estimates of whether the speech segment is “voiced or unvoiced” and then pitch period estimates for voiced segments.

Finally, these results are compressed and coded for transmission through the packet network. The compression utilizes a unique variable frame rate transmission feature that allows efficient low-bit rate interfacing into a packet-switched network (rates not exceeding 2400 bits per second [bps]).

Each of these areas will be discussed in more detail in the following sections. Completing the vocoder system is the decoding and synthesis process, illustrated in Figure 1b. The received codes are decoded and decompressed, and the synthesis filter driving function is reconstructed. The filter coefficients are updated according to the received code at a uniform rate and used for LPC speech synthesis.

SIGNAL CONDITIONING

During the DARPA program, two major directions in signal conditioning have been explored with the goal of improved intelligibility and naturalness in a realistic packet-switched communications environment:

1. Conditioning of signals with noise corruption, e.g., acoustic ambient noise.
2. Conditioning of signals with frequency distortion, e.g., connection into a packet voice network via the switched telephone network with its local loop and carbon button telephone frequency domain distortion.

Degradation of speech by noise arises in a variety of contexts. For example, the speech of a pilot in a plane commu-

* This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Office of Naval Research under Contract N00014-78-C-0214.

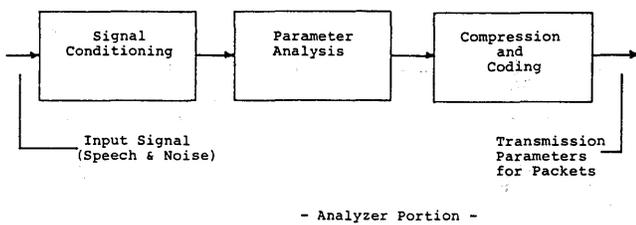


Figure 1a—Block diagram of LPC analysis system

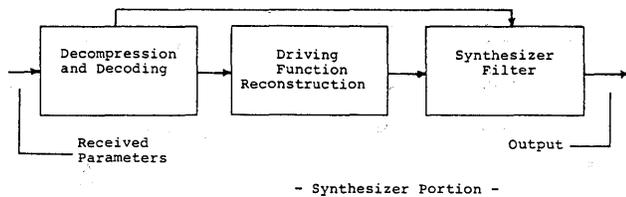


Figure 1b—Block diagram of narrowband speech processor

nicating with the ground control is degraded by the airplane noise. Another example is the speech of a lecturer recorded in a noisy lecture hall. The corrupting noise generally reduces both the intelligibility and quality of speech. Furthermore, the performance of many narrowband speech communication systems such as a linear prediction vocoder degrades quickly in the presence of corrupting noise. Thus, the enhancement of degraded speech has a variety of practical applications.

In the past few years, the speech enhancement problem has received considerable attention in the literature and a variety of algorithms have been developed.^{4,5} Most of these algorithms attempt to capitalize on some aspects of our knowledge about speech and the corrupting noise. For example, algorithms such as Wiener filtering and spectral subtraction are based on the notion that the short-time amplitude of speech is very important for speech intelligibility and that the characteristics of the corrupting noise do not vary rapidly. Algorithms such as comb filtering and speech enhancement by harmonic selection capitalize on the periodicity of voiced speech.⁶ Algorithms based on all-pole modeling of speech capitalize on the notion that speech can be reasonably well modeled as the response of a quasi-stationary all-pole or pole-zero system representing the human vocal tract excited by pulse-like or noise-like sources.

Due to efforts involved in the performance evaluation of speech enhancement systems, only a few algorithms have been carefully evaluated and then only for some selected environments.⁵ The results obtained so far indicate that the various speech enhancement systems that have been developed improve speech quality, but do not improve speech intelligibility when the speech is degraded by additive wideband random noise. When the speech enhancement systems are used as preprocessors for bandwidth compression systems such as a linear prediction vocoder, however, some improvement in intelligibility as well as quality has been observed over the bandwidth compression systems that do not include such a pre-processor.⁵

Although users of the telephone are generally insensitive to the distortion introduced by local loop or subscriber loop

mismatches and the telephone handset (the carbon button mouthpiece in particular), LPC algorithms are not. It has been demonstrated that in the absence of a "robust pitch and voicing detector," severe degradation occurs in the synthetic speech.

Since it is believed that end terminations in packet speech networks of the future will typically be telephones, proper interfacing between them is an important area of research.

A recent study⁷ has shown that as much as 15 dB variation in the voice spectrum can occur due to the local loop telephone characteristics, without considering the guard-band filters that suppress the 0–300 Hz region for T-carrier transmission. A blind-deconvolution problem is defined here since neither the speaker nor telephone channel characteristics are known to the system. However, some success has been demonstrated in attempting to recover a "non-telephone voice" as input to an LPC processor by making use of long term spectral averages over a population of speakers.⁶

A second problem presently being studied is the effect of echo on vocoded speech in a packet-switched environment where the "tails" are connected to the switched telephone network. Due to imperfect 2–4 wire conversion of analog hybrids, a rather strange synthetic quality can result from the analysis and resynthesis of synthetic recirculating echos. Substantial success in this area has been achieved with the use of different echo cancellation algorithms.⁸

PARAMETER ANALYSIS/RESYNTHESIS

In narrowband speech processing, there are three primary parameter sets of interest. The first set is of the spectral parameters that define a mathematical model of the spectrum for a short interval of speech. Typically 8–12 parameters obtained by linear prediction analysis are used. The second parameter set consists of just the gain parameter, which defines the amplitude or energy of the speech segment being analyzed. The third set is the pitch and voicing parameters.

Initial parameter analysis is usually performed at a rate of 40–100 frames or segments per second. For synchronous communication, lower values are usually chosen (the government-sponsored system for synchronous communication² has a 44.4 frames/sec. analysis rate), whereas for packet transmission, a higher rate such as 100 frames/sec is recommended⁹ since a variable frame rate compressor (described in the next section) can follow. After the transmission parameter sets are extracted, they are quantized and coded into an appropriate bit stream for insertion into the transmission packets. At the far end, a reverse process takes place whereby variable frame rate parameter sets are used to update the synthesizer filter coefficients and generate the synthesizer excitation signal.

Several of the major results obtained by the participants in this area are as follows: (1) discovery of the rich mathematical structure of the linear prediction equations¹¹ and development of alternate synthesis filter structures,¹² (2) an understanding of the fixed point implementation characteristics of the most important numerical algorithms in the system,¹³ and (3) the development of robust pitch and voicing detection algorithms for complementing the signal conditioner algorithms.^{14,15}

A large number of contributions by many researchers have

been published in the general area of LPC analysis and synthesis. Interested readers are referred to the *IEEE ASSP Transactions*, the *IEEE Proceedings* paper by Makhoul,¹⁶ and the book by Markel and Gray.¹¹

COMPRESSION AND CODING

Data compression systems resulting in nonuniform transmission of parameters are generally able to achieve lower data rates than uniform-rate parameter transmission. However, for transmission on synchronous lines, the necessary buffering would cause excessive delays due to the time varying nature of speech.

Since packet-switching systems readily accept asynchronous data, there is substantial interest in variable frame rate parameter compression algorithms. During the course of the DARPA project, this topic was initially explored at SRI¹⁷ and later examined in detail at BBN.^{9,10}

Briefly, the compression algorithm determines, using some distortion measure, when it is necessary to update the LPC (reflection, or log area ratio) coefficients. The distortion measure compares the ratio of two different one-step prediction error energies. The numerator is the error energy of the one-step prediction using the last transmitted set of LPC coefficients (or their equivalent). The denominator is the error energy of the optimal one-step prediction. The performance measure, which must be greater than or equal to one, is compared with a threshold value that determines whether new LPC coefficients need to be transmitted or not. If the threshold is exceeded, new values for the LPC coefficients are transmitted. The higher the threshold value, the greater the compression—but the greater the degradation in the synthesized speech.

In one variable frame rate (VFR) scheme,⁹ the input speech is analyzed at a fixed rate (e.g., 100 frames per second [fps], or once every 10 ms) to extract the vocoder model parameters: spectral coefficients (log area ratios or LAR's), pitch, voicing and gain.

However, the model parameters are transmitted only when the properties of the speech signal have changed sufficiently since the preceding transmission; the parameters for the untransmitted frames are regenerated at the receiver through linear interpolation between the parameters of the two adjacent transmitted frames. For example, speech parameters may be transmitted less often during steady-state portions of speech and more often during rapid speech transitions. Thus, the VFR scheme minimizes the redundancy in parameter transmission by minimizing the average frame rate of parameter transmission, under the constraint that the speech quality of the VFR system is preserved at the level achievable from the unreduced (100 fps) fixed rate system. The transmission data rate of the VFR system thus varies in accordance with the changing properties of the input speech; this property of asynchronous data rates makes the VFR scheme ideally suited for a packet-switched communication channel.

In the above experiment, the VFR scheme produced an average transmission rate of 1650 bps for continuous speech (i.e., not including silences). In a formal subjective speech

quality test, the speech quality of this VFR system was found to be equivalent to that of the 5650 bps fixed-rate (100 fps) reference system.¹⁸

Coding, as defined here, is the process of taking a single frame of analyzed parameters and quantizing them into a specified number of bits for transmission across the channel. During this research program major new results were obtained towards understanding how a number of diverse characteristics such as sampling frequency, pre-emphasis, and the types of speech sounds relate to the number of bits allocated to each parameter.^{19,20} These studies in scalar quantization (single parameter at a time) have also led to a new area of research called vector quantization, which holds promise for even further bit-rate reductions for a specified "quality" in low bit-rate systems.²¹

EXTENSIONS FOR VARIABLE BIT-RATES

In a telecommunications system there is a finite probability of having more channel requests than instantaneous available channel capacity. The standard flow control mechanism for accommodating this situation is to block call requests during overload. However, advanced speech coding techniques offer several flow control alternatives in the form of multirate or variable rate algorithms. The ability to automatically select a rate at dial-up or to vary the rate during a conversation in a manner that is transparent to the user (other than detectable changes in quality) provides an additional degree of freedom in the design of future integrated networks.

There are several advanced network design problems that may be best served by multirate or variable rate speech coders. These include the flow control of voice traffic by means of network-induced bit-rate adjustments and the almost universal problem in interoperability between users with different fixed bit-rate systems. For the latter, multirate algorithms offer very attractive alternatives to tandem connections of two sets of analyzers and synthesizers, which typically result in unacceptably poor speech quality and compromised security measures.

To illustrate the vocoder concepts implied by these advanced approaches to network design problems, we shall describe an extension of the LPC system described above.

The approach described here involves a Residual Excited Linear Prediction Vocoder (REL_P)²² as an extension of the LPC vocoder idea, wherein a coded version of the LPC error signal (residual) is used as the excitation for wideband operation instances and a standard LPC vocoder configuration with pitch or noise excitation is used for narrowband operation.

Figure 2 depicts a simple block diagram of an LPC/REL_P form of multirate voice coding. LPC analysis is carried out in the same manner as for the LPC source coder described earlier, wherein the coefficients necessary for characterizing the synthesis (or vocal tract) filter and pitch and voicing decision parameters are transmitted to the receiver as in narrowband voice coding. If the unfiltered residual signal were PCM coded directly, a transmission rate of approximately 20,000 bps would be required. Instead, the residual is low-pass filtered to

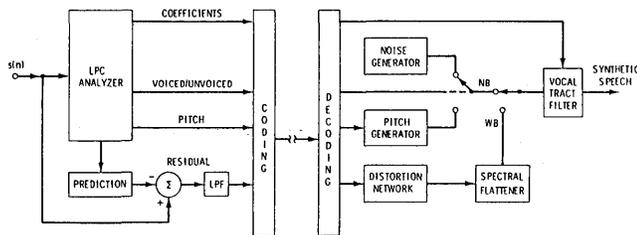


Figure 2—LPC/RELTP multirate voice coding block diagram

approximately 800 Hz to limit the required coding rate to approximately 5,000 bps while preserving the baseband pitch harmonics of the excitation signal. At the receiver, the low pass filtered residual signal must be processed by a non-linear distortion operation to regenerate the high frequency harmonics. For narrowband operation, the coded residual signal would not be transmitted and the synthesizer would operate in a pitch-excited mode. For wideband operation, the transmission rate would be increased to accommodate the coded residual signal and the synthesizer would operate in a residual excited mode.

Other related approaches being studied are a combined channel vocoder and sub-band coder,²³ and a multirate adaptive transform coder (ATC) approach that can collapse to the standard LPC system.²⁴

CONCLUSIONS

Accomplishments

In the author's opinion, the principal pioneering accomplishments in algorithm research during this program were that

- The first real-time software implementation of packet-switched voice was demonstrated in 1974, using LPC algorithms implemented in commercially available array processors on the Arpanet.¹
- A fundamental understanding of the fixed-point arithmetic characteristics of major algorithm components such as autocorrelation calculations, matrix solutions, and synthesis filter structures was developed.¹³
- Concepts of variable-frame-rate algorithms for efficient interface into packet voice systems were thoroughly investigated and demonstrated to provide substantial bit savings with minimal loss in perceived quality.^{9,10,17}
- Concepts of a higher level structure for trading off bit rate against quality were developed for variable and multirate voice coders.^{22,24}
- During this program, experimental and later theoretical investigations were initiated which provided a fundamental understanding of the process of scalar parameter quantization in linear predictive coding.^{19,20}
- Analytical and experimental procedures for various signal conditioning methods were thoroughly investigated.^{4,8}

Lessons Learned and Future Directions

Among the most valuable aspects of research are not only its accomplishments but also the difficulties and the future directions it discovers.

The most sobering news from the last decade of research is that the achievement of consistently good quality speech synthesis through the use of narrowband voice processors has proved elusive. Impressive quality speech demonstrations are possible when processing occurs in a laboratory environment with high signal-to-noise, read text samples, analysis with floating point hardware, and possibly hand-corrected pitch and voicing decisions. Experience has shown, however, that the same system will often generate fair to poor quality speech output when it is used for processing casual or "sloppy" speech in a noisy environment or "telephone speech" after telephone transmission by way of the local subscriber loop.

The bandwidth and usage statistics of the Arpanet have yet to conclusively demonstrate the acceptability of packet-switched voice, because of the packet delay characteristics of the Arpanet test-bed. Users could only reliably communicate in a half-duplex mode due to occasional 4-5 second packet delays. Thus, one of the future research efforts should focus on defining what the realistic parameters of a network have to be to satisfy both packet data and voice requirements, and then on verifying the assumptions through actual operation.

It is clear that the spectral subtraction algorithms (or variations of them) lead to a substantial reduction in perceived noise background in the processed (but not vocoded) speech signal. What is not clear is whether the resultant vocoded signal is more acceptable to the listener. In place of the spectral artifacts due to the noise background, a music-like (although discordant and seemingly random) quality appears. Although the mechanism of this anomaly is now well understood, there does not appear to be a complete solution to the problem.

The area of parameter analysis has gained substantial research success because of the fact that linear prediction modeling turns out to be very rich analytically, with many useful properties. Elegant formulations for optimal scalar parameter quantization, various solution algorithms, indefinite expansions of synthesis structures, and even new generalized filter structures with superior roundoff noise characteristics²⁵ have been developed. In one sense, whatever problems remain in this area can be shrugged off by stating that any modeling errors will show up in the residual signal. It is the business of the pitch and voicing mechanism to extract whatever else must be extracted. However, this approach does not shed much light on the problem of "quality."

One of the serious and apparently fundamental difficulties in spectral modeling of speech is that small variations in spectral structure, i.e., variations from an absolutely flat trend characteristic in the residual or error spectrum cannot be extracted by the model, but are easily extracted by the human ear.

Sophisticated attempts such as pole-zero modeling have met with a resounding failure when one estimates benefit gained versus complexity added.

The principal difficulty in estimating pitch and voicing parameters is that even though research has gone on for many

years before this project and substantial research effort was expended during this project, our accomplishments were more towards efficient real-time implementation than towards "solving the pitch and voicing problem." Realistically, with our current (and foreseeable future) knowledge of modeling speech, we have an inherently unsolvable problem, since when we are asking for "pitch periods" of speech, we are attempting to estimate abstractions of a physical process that does not precisely exist. It is simply as a result of many years of experience that the model we use appears to be quite adequate for many applications.

Overall, it has been a very fruitful prior decade of research into algorithm development for narrowband voice processors and the area of linear predictive coding. We have witnessed changes from processing one 20 msec frame of speech on a PDP 11/20 in 20 sec to real-time processing (1,000 times faster) from devices no larger than a cigar box.

Our knowledge in algorithms has increased to the point where we can specify how many bits should be assigned to a spectral model coefficient to achieve an average dB spectral distortion level. The precursor to commercial packet-switched voice systems that will probably exist in the mid to late 1980's was developed and demonstrated. Nonetheless, there are still many unsolved or unresolved issues, as discussed above, that will challenge us in the decade ahead.

ACKNOWLEDGMENTS

Several colleagues contributed material for this summary paper. The contributors are as follows: Dr. Steven Boll, University of Utah; Dr. A.H. (Steen) Gray, Jr., Signal Technology, Inc.; Dr. Tom Magill, Stanford Research Institute, Int'l.; Dr. John Makhoul, Bolt Beranak and Newman, Inc.; Mr. Al. McLaughlin, Lincoln Laboratory; Prof. Alan Oppenheim, MIT; Dr. Vishu Vishwanathan, Bolt Beranak and Newman, Inc. In addition, the careful reading and constructive criticism of the text from Dr. D.Y. Wong is greatly appreciated.

REFERENCES

1. Cole, R., "A Proposed LPC Analysis System for the SPS-41," NSC Note No. 46, Oct. 1974, unpublished memorandum.
2. Tremain, T., et al., "Implementation of Two Real-Time Narrowband Speech Algorithms," *IEEE Eastcon*, Sept. 1978, pp. 698-708.
3. International Communication Sciences, Chatsworth, Ca. and Time and Space Processing, Inc., Palo Alto, Ca.
4. Lim, J.S., and A.V. Oppenheim, "Enhancement and Bandwidth Compression of Noisy Speech," invited paper, *Proceedings of IEEE*, Vol. 67 (Dec. 1978), pp. 1586-1604.
5. Boll, S.F., "Suppression of Acoustic Noise in Speech Using Spectral Subtraction," *IEEE Trans. ASSP*, Vol. ASSP-27, April 1979.
6. Frazier, R.H., et al., "Enhancement of Speech by Adaptive Filtering," *Proceedings of IEEE*, Conf. on ASSP, Philadelphia, Pa., April 1976.
7. Davis, S.B., and J.D. Markel, "Telephone Channel Equalization for Narrowband Speech Processors," submitted for publication, 1981.
8. Parikh, D., et al., "Study of Echo Cancelling Algorithms for Full Duplex Telephone Networks with Vocoders," *Proceedings of IEEE*, Conf. on ASSP, Atlanta, Ga., April 1981, to be published.
9. Makhoul, J., et al., "Natural Communications with Computers: Speech Compression Research at BBN," *BBN Report No. 2976*, (NTIS No. AD/A 003478/5GA, Vol. II, Dec. 1974).
10. Viswanathan, R., et al., "Variable Frame Rate Narrowband Speech Transmission Over Fixed Rate Noisy Channels," *Proc. EASCON '77*, Paper 23, Washington, D.C., Sept. 1977.
11. Markel, J.D., and A.H. Gray, Jr. *Linear Prediction of Speech*. Springer-Verlag, Berlin, 1976, Ch. 3.
12. Op. cit., Ch. 9
13. Markel, J.D., and A.H. Gray, Jr., "Fixed-Point Truncation Arithmetic Implementation of a Linear Prediction Autocorrelation Vocoder," *IEEE Trans. ASSP*, Vol. ASSP-22, No. 4 (Aug. 1974), pp. 273-281.
14. Paul, D.B., "Homomorphic Pitch Detection," MIT/LL Tech. Note No. 1978-32, 1978.
15. Juang, F., and J.D. Markel, "Spectrally Based Pitch and Voicing Estimation with Statistical Assistance," NSC Note No. 140, Signal Technology, Inc., Santa Barbara, Ca., unpublished memorandum, Oct. 1979.
16. Magill, D.T., Adaptive Speech Compression for Packet Communication Systems, *National Telecommunications Conf.*, 1973, Atlanta, Ga., *Conf. Record*, Vol. 2, pp. 29D-1 through 5.
17. Huggins, A.W.G., et al., "Speech-Quality Testing of Some Variable-Frame-Rate (VFR) Linear-Predictive (LPC) Vocoders," *J. Acoust. Soc. Amer.*, Vol. 62 (Aug. 1977), pp. 430-434.
18. Viswanathan, R., and J. Makhoul, "Quantization Properties of Transmission Parameters in Linear Predictive Systems," *IEEE Trans. Acoust., Speech and Signal Processing*, Vol. ASSP-23, June 1975, pp. 309-321.
19. Gray, A.H. Jr., and J.D. Markel, "Quantization and Bit-Allocation in Speech Processing," *IEEE Trans. ASSP*, Vol. ASSP-25, No. 6 (Dec. 1976), pp. 459-473.
20. Buzo, A., A.H. Gray, Jr., R.M. Gray and J.D. Markel, "Speech Coding Based Upon Vector Quantization," *IEEE Trans. ASSP*, Vol. ASSP-28, No. 5 (Oct. 1980), pp. 562-574.
21. Un, C., and D.T. Magill, "The Residual Excited Linear Prediction Vocoder with Transmission Rate Below 9.6 Kbs," *IEEE Trans. on Communications*, Vol. COM-23, No. 12, Dec. 1975.
22. Gold, B., "Multiple Rate Channel Vocoding," *EASCON '78 Record*.
23. Berouti, M., and J. Makhoul, "An Embedded-Code Multirate Speech Transform Coder," *Int'l. Conf. Acoustics, Speech and Signal Processing*, Denver, Co., April 1980, pp. 356-359.
24. Markel, J.D., and A.H. Gray, Jr., "Roundoff Noise Characteristics of a Class of Orthogonal Polynomial Structures," *IEEE Trans. Acoust., Speech and Signal Processing*, Vol. ASSP-23, No. 6 (Dec. 1975), pp. 473-486.

A modular approach to packet voice terminal hardware design*

by G. C. O'LEARY, P. E. BLANKENSHIP, J. TIERNEY, and J. A. FELDMAN

Massachusetts Institute of Technology
Lexington, Massachusetts

ABSTRACT

In addition to encoding and decoding speech, packet voice terminal functions include packetization and depacketization; the application and interpretation of voice, internet, and local network protocols; dialing, ringing, ancillary signaling; and interface to local network transmission media. In this paper we describe a modular packet voice terminal design in which the partitioning of the hardware is effected along these basic functional lines. Interfaces are provided between the speech processing, protocol handling, and local network interfacing functions such that each of these is implemented in a physically independent module with considerable freedom for modifying or replacing any one module independently of the others. This allows for a broad choice of voice coders on the one hand, and a variety of local network types on the other. Specific modules discussed include a microprocessor-based protocol-handling section, an interface module to a local area coaxial cable network which uses a contention-based mechanism for packet voice communication, and a variety of voice coder/decoder designs covering the rate spectrum from 2.4 to 64 kbps.

INTRODUCTION

The success of packet networks for transmitting data has led to a great deal of interest in the potential of using packet techniques for the transmission of voice. The feasibility of transmitting speech on a packet network has been demonstrated by experiments performed on the ARPANET^{1, 2} and in the Atlantic Packet Satellite Experiment (SATNET).³ In both these experiments the limited channel bandwidth available restricted the class of packet voice experiments that could be performed. In particular, it was not possible to experimentally demonstrate the statistical multiplexing possible with a large number of voice circuits. A new experiment, which will provide a wideband packet satellite channel, is now being planned.⁴ The 3 Mbps capability of this channel will be sufficient for many voice connections even if 64 kbps PCM is used.

* This work was sponsored by the Advanced Research Projects Agency. The U.S. government assumes no responsibility for the information presented.

A wideband packet voice network should have a number of advantages over a circuit-switched network. A recent study⁵ showed that the transmission and switching costs of a packet voice network should be lower than those of a circuit-switched network. This saving is obtained mainly because in the packet system the channel capacity is used only when a speaker is actually talking, which is typically about half of the time. By multiplexing many users on a common wideband channel a factor approaching two in bandwidth saving is obtained. In addition, other benefits are obtained with packet voice. Since a packet network has the flexibility to asynchronously accept packets of arbitrary sizes, it becomes easy to mix terminals with different frame and data rates. If vocoders are used to further reduce the bandwidth utilization, each vocoder would use only the channel capacity necessary to transmit its information rather than a fixed minimum bandwidth increment as is used in circuit-switched networks. It is also possible to design systems in which the transmission rate varies dynamically with channel conditions.⁶

As part of the wideband satellite testbed, Lincoln Laboratory is developing a packet voice terminal as the mainstay of integrated voice/data network experimentation. This terminal is designed to access the long haul satellite network through a local access network. The local access aspect of the problem has been described elsewhere.^{7, 8, 9} In this paper we concentrate on the terminal functions as opposed to the transmission functions.

TERMINAL REQUIREMENTS

The voice terminal must provide the interface between the user and the network much as the ordinary telephone instrument. Besides converting between acoustic and electrical signals, the terminal must provide the full range of signaling capabilities such as dialing, ringing, dial tone, busy signal, and the like, that the user has come to expect from the telephone. In addition, the voice terminal on a digital network should be able to include a variety of special features. The voice terminal design must include a great deal of flexibility in order to meet the requirements of the wide range of applications in which it might be used. Because of the experimental nature of the program, the terminals must be able to accommodate evolving system requirements as the experiment progresses.

In addition, the terminal should be able to deal with a variety of voice coders. Choices range from very simple but high bit-rate PCM-based coders to the more elaborate narrowband or multiple rate systems. The lack (thus far) of a universal voice digitization algorithm that can function optimally in all acoustic and/or channel environments imposes a requirement for terminals that can accommodate a variety of algorithm types. In the long run the potential interoperability problems posed by this multiplicity of bit rates and speech algorithms might be solved via programmable or multirate coders. Speech might then be digitized using coding strategies and bit rates most appropriate to local conditions, while receiving and decoding in accordance with the algorithm and bit rate combinations that best suit the remote end of the connection. The apparently simple requirement of a digital I/O format for telephones thus introduces in the terminal a host of problems relating not only to the basic speech digitization process, but to the formatting of input and output data streams and the inclusion of control mechanisms for algorithm selection, voice bit rate, etc.

TERMINAL DESIGN

The experimental terminal was designed to provide a 64 kbit/sec PCM capability, and in addition provide a flexible interface for any of a number of single card narrowband speech processors being designed to explore some of the possible voice processor options. Also provision has been made for connecting to external vocoders or programmable speech processors.

The required flexibility is achieved in several ways. First, the terminal hardware is partitioned along functional boundaries with clearly defined interfaces so that changes in one section would have minimal impact on other parts of the terminal. Second, each major functional block is controlled by a microprocessor. This permits changes to be made by adapting only the software. Figure 1 shows the basic partitioning of a general terminal. There are three functional units. The voice processor performs the conversions between the analog speech and digits but is independent of the transmission process. The protocol processor forms the packets and provides the necessary layers of protocol to insure that the packet can be delivered to a distant network and be played out at the proper time. It is independent of the transmission medium. The access controller accepts packets from the protocol processor and transmits them across the network to the access controller in another terminal. It deals only in packets and knows nothing about voice or higher-level protocols. The detailed functions of each module as realized for our experimental network are described below.

Access Control

The access controller provides the network-dependent packet transport mechanism for the terminal. The access controller receives packets from the protocol processor and transmits them over the network. The access controller in our terminal is designed for a local access area which funnels

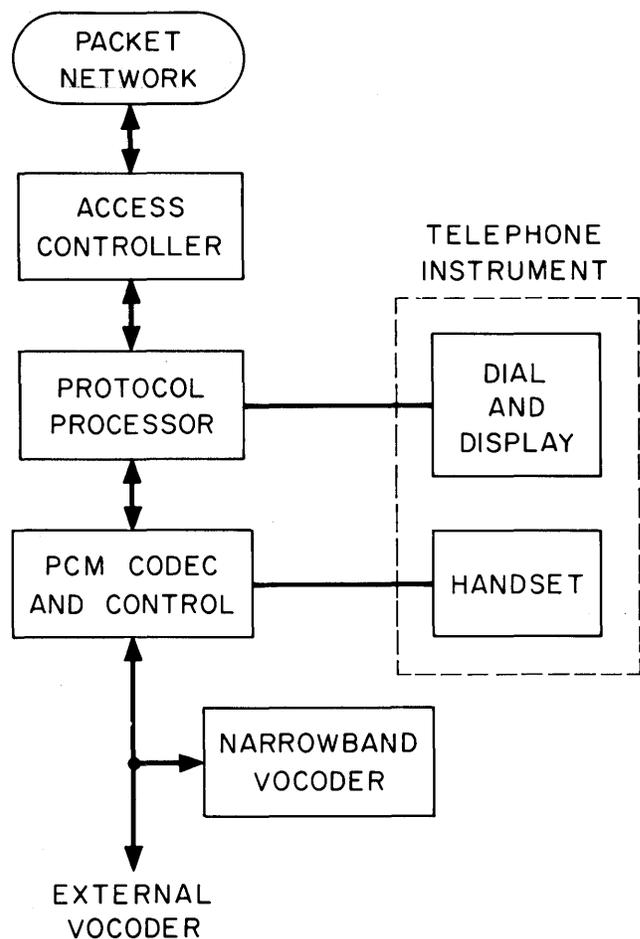


Figure 1—Modular packet voice terminal architecture

packets to a traffic concentrator. The details of this design can be found elsewhere.^{7, 8, 9} A generalized packet interface is provided between the access controller and the protocol processor, and it is possible to connect to other packet networks such as the packet radio network by changing the design of the access module.

Protocol Processor

The protocol processor contains the main intelligence in the terminal. It has the task of forming the outgoing packets and interpreting the incoming ones. It also handles transfers to and from the speech processing devices. The protocol processor has two main functions. The first is to establish the call. Thus the processor must include an interface with the user input (key-pad) and must be able to generate and interpret packets necessary for establishing the call. Once the call is established, the protocol processor must generate and receive the speech packets. On transmission, it forms packets from the input voice bit stream whenever the voice processor indicates the speaker is active. To this it adds an address header, protocol information, and a time stamp according to a Network Voice Protocol (NVP).¹⁰ The completed packet is then

transferred to the access controller. On reception, the header is checked and the time stamp is used to put the packet in its proper place in the queue of packets going to the voice processor. If no packet arrives to fill a particular output frame, then it is assumed that a silence interval has occurred, and the voice processor will be signaled to generate a silent interval in its output.

The protocol processor is a microprocessor-based card consisting of an 8085 CPU, a data memory, a program memory, a DMA controller, and a USART port. An extension memory card to handle the protocol programs is connected. The total program memory could be as large as 40K bytes if the cards were fully populated. The current protocol program requires about 12K bytes. The DMA controller is used to pass data in both directions to the access controller and to the speech processor. The USART is used as a serial channel to the telephone instrument for signaling and display.

Communication between the protocol processor and the speech processors is via two pairs of byte parallel channels. The first pair is used for the speech data and is controlled by the protocol processor's DMA chip. The second pair has several possible uses depending on the speech processor. At start-up the speech processor presents an identification code on the control input channel. The protocol processor uses this code to determine which speech algorithm is active and to select the program parameters necessary for the selected vocoder. The control channel can also be used for passing control parameters to variable rate vocoders or for loading programs into programmable voice processors.

Telephone Instrument

The telephone instrument provides the user interface to the terminal. The interface between the telephone and the protocol processor was designed to allow the addition of a wide range of special user functions and displays.

The telephone instrument is a standard touch-tone phone modified for this application. Internally, the phone contains a small micro-processor system consisting of an 8085 CPU, ROM memory, keypad, and a USART. Communication with the protocol processor is via an RS-232 serial interface.

The drivers and receivers for the analog signals to and from the handset are contained in the telephone set. Tone generators in the set are used to add dial tone, busy signals, and ringing signals into the analog signal sent to the earpiece. These tones are controlled by commands transmitted over the serial line from the protocol processor. The transmit and receive analog signals are kept separate throughout, providing a 4-wire end-to-end connection capability and eliminating the need for hybrids and echo suppressors in the system.

Voice Processor Control Card

The Voice Processor Control Card has three functions. First, it provides each terminal with a 64 kbit PCM capability so that all terminals on the network will have at least one common back-up voice digitization mode. Second, it provides a switching function between the PCM and any other speech

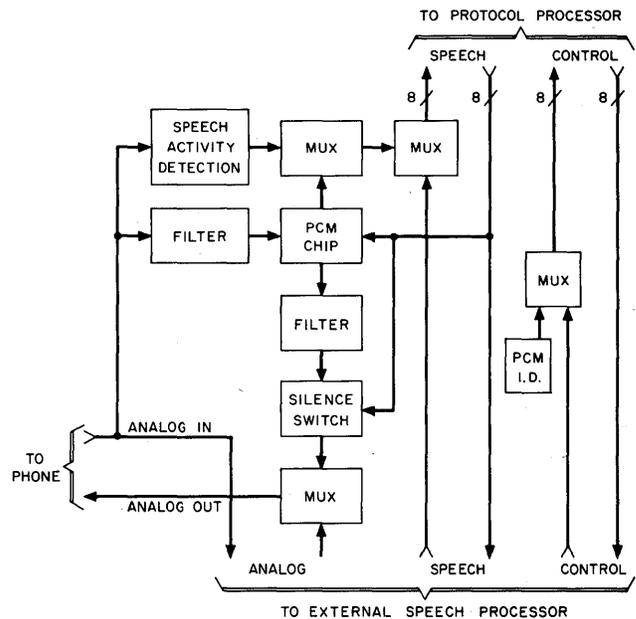


Figure 2—PCM subsystem and external speech processor interface

processor that is connected. Finally, it handles the analog signals coming from the telephone instrument. The analog signals are also switched between the PCM Codec and the peripheral speech processor. A block diagram is shown in Figure 2.

The speech processors send and receive speech data in groups of 8-bit bytes called parcels. The number of bytes in a parcel is dependent on the speech processor. Typically, a parcel represents about 20 ms worth of speech. Each parcel is headed by a control byte, one bit of which indicates the presence of speech in the parcel. Otherwise, silence is assumed. The remaining control bits may be used in vocoder-specific ways to pass control information which must be synchronized to a particular byte.

The PCM capability is implemented with a single chip μ -255 law PCM Codec (Intel 2910). Bandlimiting filters are provided for the input and output. Silence detection is provided by a threshold detector with a 100 ms hold time after a threshold is exceeded.

The voice processor control card provides a general interface for a separate vocoder or speech processor mounted either internally or externally. The digital and analog signals required for this interface are routed to a back panel connector to service external speech processors and are wired to one of the remaining internal card slots to accommodate a suitably packaged single card speech digitizer module. The external connection has been used to transmit LPC speech generated by a Lincoln Digital Signal Processor (LDSP).

Hardware

Figure 3 shows a packet voice terminal prototype. The terminal is constructed on 7" x 7" wirewrap cards in a standard rack-width cabinet. The basic basket has room for six cards with expansion option to nine. The access controller consists

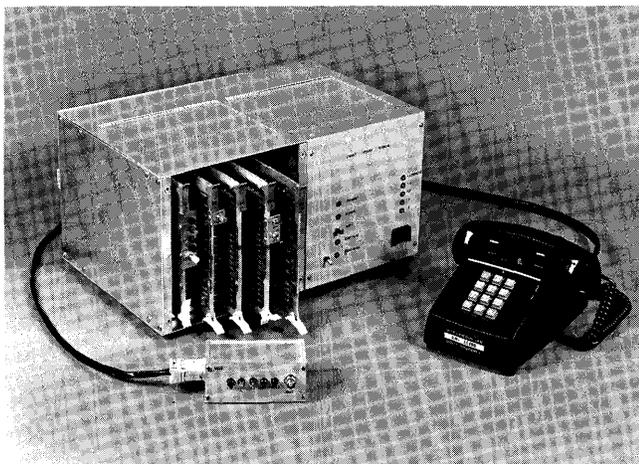


Figure 3—Prototype packet voice terminal

of two cards. The protocol processor and its extension memory comprise two more cards. A fifth card contains the PCM Codec and the voice processor switching mechanism. The sixth slot is wired to accommodate any of the single card vocoders which are described in the following sections.

NARROWBAND SPEECH PROCESSORS

Three single card speech processors are currently being designed for use in the packet voice terminal. Two of these are different 2400 bit/sec channel vocoder realizations. The third is an embedded waveform coder designed to explore the issues that arise when a variable rate feature is available to the network.

Sampled Data Analog Channel Vocoder

One of the narrowband speech processor options in the process of development is a filter bank-based system, modeled after the UK JSRU channel vocoder¹¹, which features two highly sophisticated custom NMOS sampled data analog LSI devices.¹² The spectrum analyzer is realized as a single chip and relies mainly on the charge coupled device approach for implementing the 19 band pass filters required. The filter bank synthesizer is contained on a second chip which uses solely the switched capacitor signal processing approach. These devices are under development by the Texas Instruments Central Research Laboratory.¹³

An overall block diagram of the vocoder is depicted in Figure 4. It is a full duplex system featuring four transmission rate options selectable at power-on. The essential features are a simple analog conditioning system, custom NMOS LSI devices for analysis and synthesis, and two microprocessor complexes for performing pitch extraction and controller functions. Connections are provided for both handset and tape inputs, and a parallel I/O port is provided for the terminal interface.

On the transmit side, analog inputs are coupled to the CCD analyzer through a 5th order elliptic pre-sampling filter and a

switch-selectable preemphasizer. The CCD analyzer implements a sampled data bandpass filter bank after Sproull and Cohen¹⁰ at a 10 kHz sampling rate and outputs log PCM encoded spectral envelope estimates upon demand from the controller. The analog signal is tapped prior to the preemphasizer and connected to the pitch conditioner. The conditioner drives the pitch extraction microprocessor whose output is reported on each frame boundary to the controller. The pitch and voicing information so derived are incorporated into the output data buffer prior to transmission.

At the receiving end inputs from the protocol processor are unpacked and decoded by the controller. Reconstructions of the spectral envelope estimates as well as excitation information are then passed to the synthesizer device in digital form. The synthesizer receives a full update each frame time and converts the digital data to analog form for processing by the synthesis filter bank. Synthetic sampled-data speech thus produced is then passed to a simple second order low pass filter and finally to a switch switchable deemphasizer with five roll-off choices.

The spectral analysis chip (Fig. 5) implements a sampled data version of the classic JSRU channel vocoder filter bank at a 10 kHz rate. The analyzer is comprised of 19 bandpass filters, half wave rectifiers, and low pass filters which provide a smoothed estimate of the spectral energy in each frequency band. The bandpass filters are realized as 100 tap FIR structures implemented using the CCD split gap electrode technique.

The low pass filters are of the third order Butterworth type with a 35 Hz cutoff and are realized using the switched capacitor technique. The real pole is used as a decimation stage permitting a 10:1 sample rate reduction prior to the complex pole thereby tending to decrease sensitivity to coefficient inaccuracies.

At frame boundaries, a trigger pulse issued by the controller initiates a read-out sequence wherein the 19 low pass filter outputs are simultaneously sampled and sequentially encoded via a 5-bit logarithmic A/D converter. The converter, at 1.5 dB per step, affords 48 dB of dynamic range and requires 1 msec per conversion. An interval of 19 msec is therefore necessary to report all 19 encoded spectral envelope estimates sampled at the frame boundary.

The synthesizer subsystem approximates the vocal tract spectral envelope by summing together in anti-phase a set of 19 suitably weighted filter responses (Figure 6). The filter

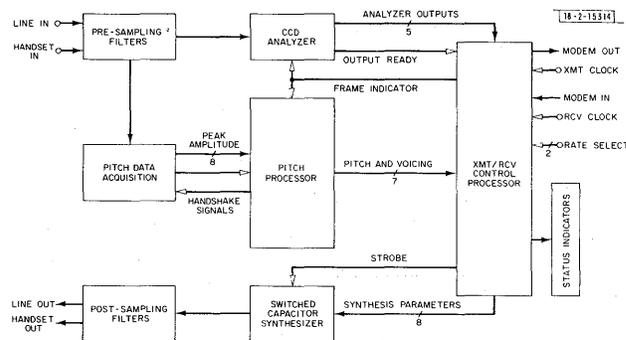


Figure 4—NMOS vocoder architecture

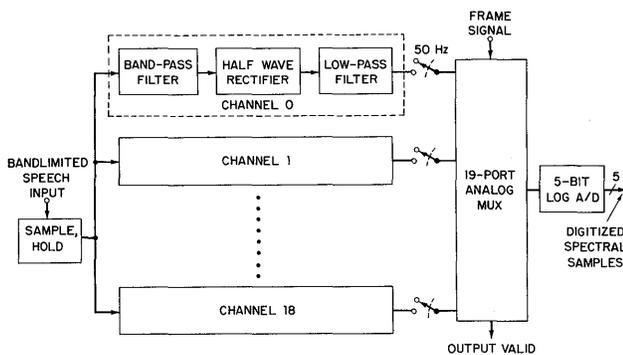


Figure 5—Spectrum analyzer

bank is of the JSRU vocoder type which is characterized by a set of simple second order resonators. The resonator center frequencies correspond with those of the analyzer filters on a one-to-one basis, but the bandwidths are fixed at either 45 or 65 Hz to approximate the shape of a formant. Due to the relatively high Q_s involved, CCD FIR realizations were impractical, and a recursive, switched capacitor approach operating at a 40 kHz rate was used instead.

The filter bank excitation assumes the form of either a fixed amplitude impulse train during voiced frames or a noise-like sequence comprised of random polarity impulses during unvoiced frames. The period of the voiced excitation is determined by the transmitted pitch word, that of the noise is fixed at 1 msec.

The filter bank gains are provided by the controller in a 5-bit log PCM format. The gains are sequentially converted to analog voltages and buffered via an exponentiating D/A and analog delay line. At frame boundaries the 19 gain parameters are passed simultaneously to a set of identical 35 Hz Butterworth low pass interpolating filters whose outputs drive the gain modulators.

The pitch/voicing subsystem is of the Gold type¹⁴ with a modified final decision algorithm.¹⁵ The analog preprocessing section consists of a 600 Hz 3rd order Butterworth low pass filter, a peak detection circuit, and an 8-bit A/D converter to capture waveform extrema. Waveform extrema and timing information are communicated to an 8085A microprocessor where the pitch detection algorithm is implemented. Pitch period data are resolved in 100 μ sec increments and are log-encoded to 6 bits. Pitch period and voicing data are reported to the control processor once per 20 msec.

The controller is realized using a second 8085A and performs the functions of maintaining overall timing, coding, and formatting the transmit data, communicating with the protocol processor, decoding received information, and driving the synthesizer device. Combinations of one-bit and two-bit PCM coding techniques are applied to the 19 log spectral data to achieve the several desired bit rates (1200, 2400, 3600, 4800 bps).

The entire system is mounted on a single 7" \times 7", 72-socket wirewrap board and occupies one slot in the PVT mainframe. The assembly comprises 28 commercial devices, 15 discrete component carriers, and 2 custom NMOS sampled data devices. Power dissipation is approximately 5W.

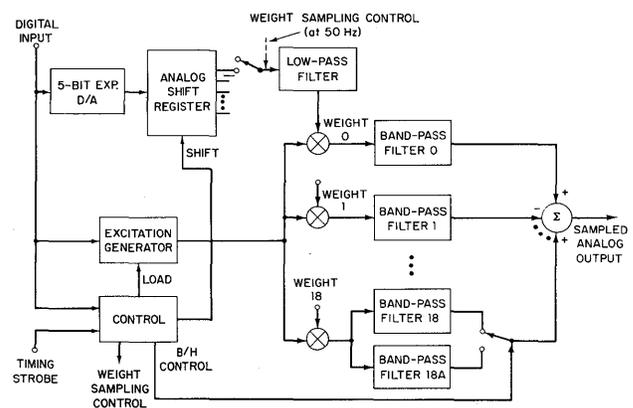


Figure 6—Spectrum synthesizer

Digital Channel Vocoder Implementation

The architecture has been developed for a digital implementation of a JSRU type filter bank vocoder that interfaces to the packet voice terminal protocol processor. A package count of about 13 devices is projected for the design composed entirely of commercially available or soon-to-be-available integrated circuits. The analysis/synthesis tasks are implemented with a distributed signal processing architecture using five Nippon Electric Company (N.E.C.) μ PD7720 DSP chips. This is a single chip, signal processing oriented microcomputer which could be used to implement a variety of speech digitization systems, including linear predictive-type coders. Communication among the N.E.C. processors is achieved via a standard 8-bit microcomputer bus coordinated by an Intel 8051 microcontroller. A compact analog subsystem based on the Hitachi HD44212 single chip Codec-with-filters has also been designed, which in acoustically benign applications results in vocoder performance equivalent to that obtained with laboratory quality audio conditioning.

The channel vocoder architecture is shown in Figure 7. The analog input speech is processed through the analog-to-digital conversion (ADC) subsystem which produces a serial data stream coupled to a 3-chip analyzer array. The analyzer signal processing task is distributed such that one N.E.C. chip computes the weights for the lower 10 spectral channels while a second computes the upper 9 spectral weights. A third N.E.C. device implements the Gold pitch detector providing frame-wise pitch estimates and voicing decisions. Each analysis frame, the resulting coded 19 spectral weights, and two pitch detector outputs are transferred over an 8-bit (Intel 8080 type) microcomputer bus under the control of an Intel 8051 microprocessor system to the PVT external vocoder port. In a similar fashion, the control processor receives coded synthesis parameters each frame from the protocol processor and transmits them to an N.E.C. chip synthesizer array. One μ PD7720 synthesizes the lower 10 spectral channels of the output speech while a second chip synthesizes the remaining upper 9 spectral channels. By making use of the 7720's input and output ports, the upper and lower spectral portions are digitally summed and transferred to the digital-to-analog converter (DAC).

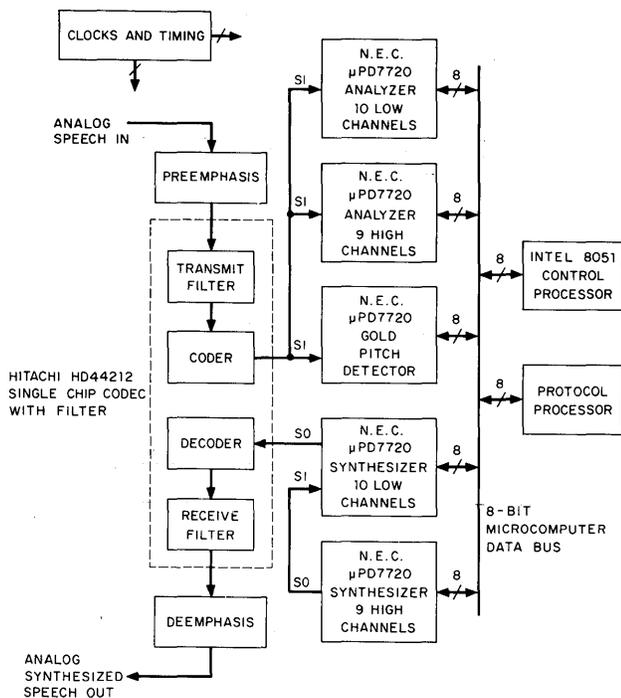


Figure 7—Digital channel vocoder

The vocoder will be packaged on a single 7" x 7" board, will comprise 7 LSI and 6 MSI devices, and will dissipate about 7W.

Embedded Waveform Coder

A compact embedded waveform coder is being developed to be interfaced to the Packet Voice Terminal and used for variable-rate speech experiments on the wideband channel.¹⁶ The coder is designed to be interoperable at 16 kbps with present standard CVSD encoders used in various military communication systems, but when higher channel capacity is available, produces a speech quality comparable to PCM at 64 kbps. The basic encoder-decoder (Fig. 8) consists of a CVSD encoder at 16 kHz with filtered input $x(t)$, data stream $b(n)$, and integrated loop signal $\hat{x}(t)$. The backbone encoder is augmented with the open loop encoding of the difference signal $\Delta = x(t) - \hat{x}(t)$ at an 8 kHz sample rate. This extra information serves as an additive correction to the CVSD approximation to $x(t)$. At the receiver-decoder the CVSD data stream, $b(n)$, is used to recreate the output signal $\hat{x}(t)$. If no more data are available (i.e., the receiver rate is 16 kbps), the final output is simply the CVSD signal $\hat{x}(t)$ low pass filtered to produce $s(t)$.

Each 20 ms four separate groups of data words are output from the encoder to the protocol processor. The highest priority group consists of simply the CVSD bits packed into byte form. The next highest priority group is comprised of the most significant two bits of the six-bit Δ word packed four to a byte, the next group of the middle two bits, and so on. If only the two highest priority packets are received, a decoder can produce speech at a 32 kbps enhanced CVSD rate. If the next

lowest priority packet is received, 48 kbps speech will be output. Finally all four received packets will support a 64 kbps output speech signal for that 20 ms period.

A 7" x 7" single board hardware realization of the enhanced CVSD device based on current commercial PCM, CVSD, and 8085 microprocessor devices comprises 32 chips and is near completion. The input and output low pass filtering is performed by a single chip containing a pair of switched capacitor filters designed for PCM Codec use. Two separate CVSD chips are used, each of which implements the CVSD encoding loop including the two smoothing filters. In one case the chip is used for encoding, in the other for decoding. Finally the Δ low pass filtering, sampling, and logarithmic quantizing is performed by one half of a commercial μ -255 law Codec chip. The digital-to-analog conversion of the Δ signal in the decoder is performed by the other half of the Codec chip. Timing circuitry and interface logic is implemented with several MSI TTL packages. Power dissipation is projected at 8W.

SUMMARY AND CONCLUSIONS

A modular packet voice terminal design has been presented based on a unique functional partitioning concept capable of supporting a variety of voice digitizers in an equal variety of communication network contexts. Each subsystem is microprocessor-controlled, which offers further flexibility potential through microcode alterations. Additionally, several specific voice digitizer subsystem designs have been discussed spanning the data rate spectrum from 2.4 to 64 kbps and in one case featuring a multiple rate capability through an embedded coding approach. It is expected that these voice terminals will provide a very powerful experimental tool for exploring issues relating to integrated voice/data communications over advanced, wideband transmission media.

REFERENCES

1. Casner, S. L., Mader, E. R., and Cole, E. R., "Some Measurements of ARPANET Packet Voice Transmission," 1978 NTC Convention Record, pp. 12.2.1-5.

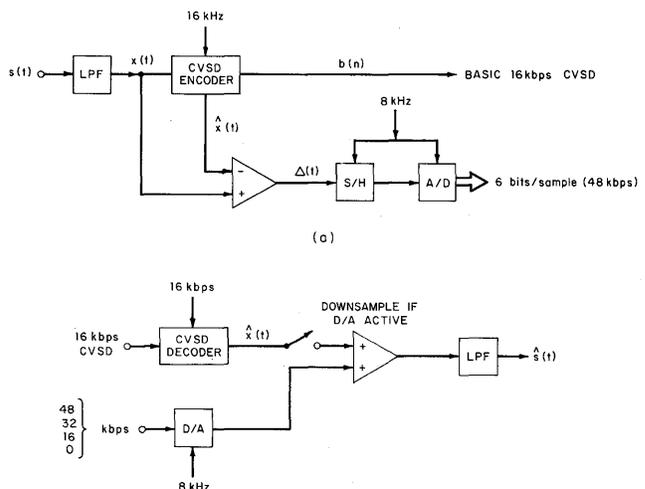


Figure 8—Embedded waveform coder

2. Gold, B., "Digital Speech Networks," *Proc. IEEE*, 65, November 1977, pp. 1636-58.
3. Chu, W. W., et al., "Experimental Results on the Packet Satellite Network," 1979 NTC Convention Record, pp. 45.5.1-12.
4. Kahn, R. E., "The Introduction of Packet Satellite Communication," 1979 NTC Convention Record, pp. 45.1.1-6.
5. Gitman, I. and Frank, H. "Economic Analysis of Integrated Voice and Data Networks," *Proc. IEEE*, 66, November 1978, pp. 1549-70.
6. Bially, T., Gold, B., and Seneff, S., "A Technique for Adaptive Voice Flow Control in Integrated Packet Networks," *IEEE Trans. Comm.*, Vol. COM-28, pp. 325-333, March 1980.
7. Johnson, D. H. and O'Leary, G. C., "A Local Access Network for Packetized Digital Voice Communications," National Telecommunication Conference, pp. 13.4.1-5 (1979).
8. O'Leary, G. C., "Local Access Facilities for Packet Voice," *Proc. Fifth International Conference on Computer Communication*, pp. 281-286 (1980).
9. Johnson, D. H. and O'Leary, G. C., "A Local Access Network for Packetized Digital Voice Communication," *IEEE Trans. Comm. Syst.* (to be published).
10. Sproull, R. F. and Cohen, D., "High Level Protocols," *Proc. IEEE*, 66, November 1978, pp. 1371-86.
11. Holmes, J. N., "The JSRU Channel Vocoder," *IEE Proceedings on Communication, Radar, and Signal Processing*, (127) February 1980, pp. 53-60.
12. Blankenship, P., "Custom NMOS LSI Channel Vocoder," *Record of SPIE Tech. Symposium East, SPIE Vol. 180*, pp. 25-32, April 1979.
13. Hewes, C. R., et al., "A CCD/NMOS Channel Vocoder," *Record of the IEE International Specialist Seminar on "Case Studies in Advanced Signal Processing"*, 18 September 1979, Peebles, Scotland, pp. 134-139.
14. Gold, B., "A Computer Program for Pitch Extraction," *JASA*, 34, 9, 16, 1962.
15. Kingsbury, N. and Amos, W., private communication, Marconi Space and Defense Systems, October 1978.
16. Tierney, J. T., and Malpass, M. L., "Enhanced CVSD—An Embedded Speech Coder for 64-16 kbps," *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, March 1981, Atlanta, GA.

Engineering computer network (ECN): A hardwired network of UNIX* computer systems**

by KAI HWANG, BENJAMIN W. WAH, and FAYÉ A. BRIGGS

Purdue University
West Lafayette, Indiana

ABSTRACT

This paper reports the design and operational experiences of a packet-switched local computer network developed at Purdue University. Hardwired communication links (1 megabaud) are used to interconnect seven UNIX computer systems (two PD11/70, one VAX 11/780, and 4 PDP11/45). Over 20 microprocessors and 210 timesharing CRT terminals are connected to the seven hosts. Instead of using the UUCP protocols for dial-up UNIX networks, several protocol programs are locally developed to make possible the hardwired UNIX networking. The network provides the capabilities of virtual terminal access, remote process execution, file transfer, load balancing, and user programmed network I/O. Only at the lowest protocol level is the DDCMP of DECNET used. The network is expandable and provides appreciable bandwidth with moderate cost and low system overhead. Described in this paper are the network architecture, system components, protocol hierarchy, local UNIX extension, load balancing methods, and performance evaluation of the Purdue ECN network.

INTRODUCTION

This paper presents the implementation experience, operational lessons, and performance assessments of a moderate-cost local computer network developed at Purdue University. The distributed computing system, named Engineering Computer Network (ECN), is presently composed of seven Digital Equipment computers of various models (VAX 11/780, PDP 11/70, and PDP 11/45). All the DEC computers in ECN run with UNIX operating systems. Hardwired communication links with one megabaud rate are used to interconnect these UNIX computer systems, which are located in three adjacent buildings at the main campus of Purdue University. The ECN UNIX network differs from the dial-up UNIX network developed by Bell Laboratories¹³ in communication links and network application protocols. The dial-up UNIX network is interconnected mainly by telephone lines with rate of 300

baud or 1200 baud. The ECN has hardwired connections providing much faster communications between the network hosts. Another hardwire-connected UNIX network has been reported by Chesson.² In general, hardwired connections are required to have up to 3 megabaud for local computer networks.²⁰

Instead of using the UUCP communication protocols between UNIX systems as described in Nowitz,¹⁴ the ECN network group at Purdue University developed several high-level application programs, called *con*, *cs*, and *rx*. These programs provide the following network functions:^{3,7,18}

- *Virtual terminal access.* A user can connect the physical terminal to a pseudoterminal on any other host computer in ECN. The *con* program provides the virtual terminal protocol.
- *Remote execution environment.* The *cs* and *rx* programs provide the capability of executing a string of commands on a remote host machine with apparent load balancing and local I/O standards.
- *File transfer/remote device access.* The *cs* program, together with several specially developed network functions, provides simple file and directory transfers between hosts.
- *User programmed network I/O.* By issuing set teletype (*stty*) functions on an open file in a pool of special UNIX files, any user-written program can directly connect to any machine in ECN, disconnect, wait for connection, send signals, etc.

The ECN network differs from the DECNET in the fact that DECNET must run with DIGITAL operating systems like the RSX-11 series, DEC-10/DAS 85, and IAS systems. DECNET uses the digital network architecture (DNA) protocols: DDCMP, NSP, and DAP, as described in DECNET.⁵ ECN uses DDCMP protocol for node-to-node communications only at the physical-link level. The NSP and DAP functions of DECNET are not used in ECN. Special advantages of ECN network are distinguished by the following features:

- *Appreciable network bandwidth with moderate system cost.* Using the DMC-11 network links,⁶ the ECN has demonstrated the bandwidth of 400 Kbaud between processes residing in the same host and 250 Kbaud between processes at different hosts.

* UNIX is a trademark of Bell Laboratories.

**The research reported herein was supported in part by Department of Transportation Research Contract No. R92004 and in part by National Science Foundation Research Grant No. MCS-78-18906.

- *Relatively low system overhead in message routing.* Internal buffering and copying are minimized. The network buffer pool resides outside the kernel space. Packet of variable lengths up to 512 bytes of data can be sent with a short header of at most 12 bytes.
- *Reconfiguration flexibility for future expansion.* The interconnection structure is quite flexible to allow up to 256 DEC machines in the network. DMC11 line driving codes are shared over multiple units, and simple static routing tables are used for packet routing.

This paper is divided into three parts. Architectural development, system features, and communication links of ECN are presented in the second and third sections. Hierarchical ECN communications protocols and local UNIX software extension are described in the fourth and fifth sections. The sixth section provides some analysis and measurements on network performance. Lessons we have learned and continued R/D efforts on ECN are given in the concluding section.

NETWORK TOPOLOGY AND SYSTEM COMPONENTS

The engineering computer network (ECN) at Purdue University is a packet-switched computer network of seven Digital Equipment VAX and PDP11 minicomputers connected with 20 Intel, Southwest Technical, and Motorola microprocessors and over 200 CRT terminals. The seven DEC computers in ECN include one VAX-11/780, two PDP 11/70's, and four PDP 11/45's. These computers are interconnected by 1-megabaud digital communication links. The network presently assumes a star structure, as depicted in Figure 1. Each computer runs a separate UNIX interactive timesharing operating system (Version 7).^{1,15} The networking of these UNIX-based minicomputers is made possible with the use of coaxial cables and the Digital Equipment DMC-11 interface drivers.⁶

ECN is a packet-switching network with decentralized control. Instead of implementing the switching and routing functions in interface message processors (IMPs), as done in ARPA net, the IMP functions are distributed directly inside the host machines. The motivation for choosing this IMP-in-host architecture is to reduce the total system cost and to shorten the development period of a working subnet in a university environment. Of course, this architecture presents the shortcoming of an added switching burden for each host, which would otherwise concentrate on computation duties. The star structure with the embedded IMP functions may also pose the problem of reduced network reliability. However, when cost effectiveness has been weighed against potential disadvantages, the architecture has been sustained, in the areas of both performance and availability, since 1978.

The seven host computers in the ECN system are coded A, B, P, AARL, EEG, VE, and LISP machines in Figure 1. Basic components and functional features of these host computers are specified in Table 1. The host machine A, being at the center of the net, is directly connected to four other host machines. All seven hosts are hardware/software interconnected at the highest level. A user at a terminal connected to any of the hosts can access the remaining hosts as if his/her terminal were directly connected to those host machines. A

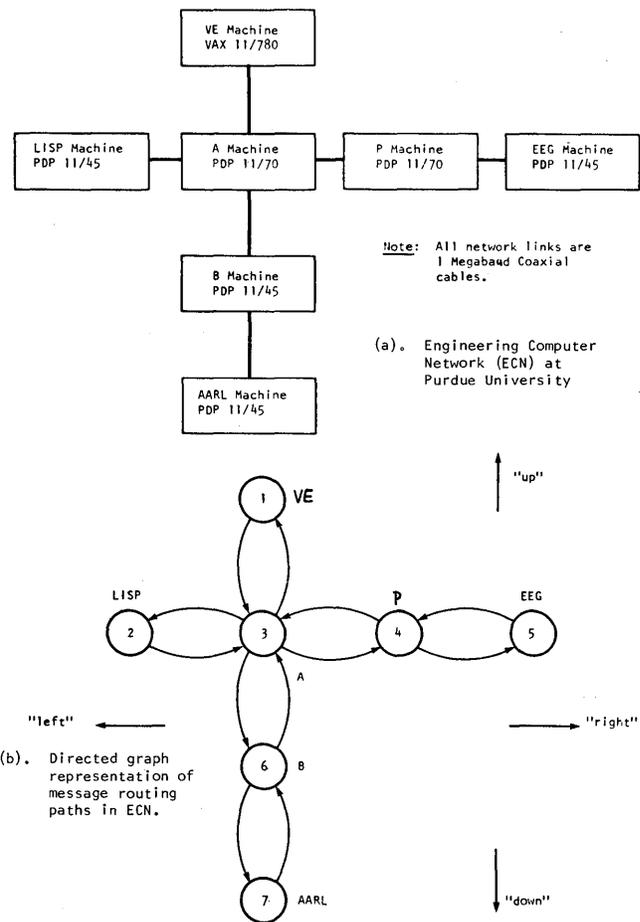


Figure 1—Architectural interconnections of the Purdue engineering computer network

user may have programs simultaneously running in several host machines transmitting data from one to another.

Topologically, the net can also be viewed as a three-level "tree" system, with Node A as the root, four hosts (B, P, VE, and LISP) at the second level, and two hosts (AARL and EEG machines) at the third level. The hosts B and P are directly connected to the AARL and EEG machines, respectively. These two "leaf" machines, at the third level, are research laboratory computers and are network hosts; but they do not serve as switching nodes. With the distributed control among the hosts, the message routing in the network is done on an interrupt basis. In other words, multiple traffic paths may exist concurrently by timesharing use of some common intermediate nodes.

Being at the center of the network, Machines A (PDP 11/70) and B (PDP 11/45) support high-speed communication links to five other PDP-11 machines and to the PUCC (Purdue University Computing Center) CDC 6500/6600 computers. The three hosts (VE, A, and B) support 136 serial data lines connected to CRT terminals, 20 microcomputers, and various data acquisition devices throughout the network, operating at rates ranging from 1.2 Kbaud to 38 Kbaud. An automated document preparation facility is also implemented in the UNIX network for entering/editing text and equation material. Besides word processing, the ECN is also connected with

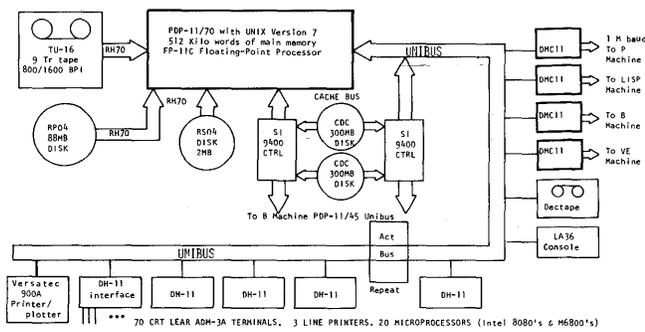


Figure 2—System components of the A machine in ECN

a graphics laboratory. The detailed configuration of the central host, Machine A, is shown in Figure 2. The DH-11s are interfaces for connecting 70 terminals, three line printers, and 20 microprocessors to the UNIBUS of A machine. The VAX 11/780 hardware includes 2 megabytes of main memory and two 267-megabyte disks. The PDP 11/45 (B machine) hardware includes 128K words of main memory, two disk drives totaling 272 megabytes of online storage, and a high-speed floating-point array processor (AP-120B). The 210 time-sharing terminals connected to seven hosts in ECN are primarily Lear-Siegler ADM-3/3A and Hewlett-Packard 2640 CRT terminals.

The UNIX (Version 7) operating system includes the high-level language C, F77, DEC's Fortran IV plus, BASIC, Macro-11, APL, PASCAL, cross-assemblers for various microcomputers, and many other software development tools.^{1,18} To the PUCG computers, the ECN serves simply as a remote job entry (RJE) station. The EE microprocessor laboratory is supported by A machine (PDP 11/70). Currently connected to the ECN are the following microcomputer systems: 8 Southwest Technical 8K systems based on Motorola 6800; one Intellec 8 Model 80 8K system, one Intel 800 MDS with 32K RAM and 16K ROM with dual-drive floppy disc, one Intel 848 MCS with 1K RAM, 10 Prompt 80/85 1K design systems based on Intel 8080, an SKD 8086 system, and one F8 32K system based on Fairchild F-8 microprocessor. All the microprocessor systems have RAM, monitors on ROM, and connections for downloading from the PDP 11/70 A machine. The A machine has cross-assemblers for both the 8080 and 6800 microprocessors.

In the near future the present star ECN will be reconfigured, with the addition of another VAX 11/780 computer (VM), into a double-loop computer network, as depicted in Figure 3. The two VAX systems (VE and VM) and the two PDP 11/70 (A and P machines) will form the main loop, serving as the backbone of the system. The four PDP 11/45 laboratory machines will form the secondary loop. The VE and the A machines will then serve as a gateway between the two loops. A proposal is being considered that would eventually extend the current ECN to an even larger network of minicomputers and microprocessors to serve nine engineering schools at Purdue University.

NETWORK LINKS AND INTERFACE LOGIC

Local communications between two computers on the UNIX network are controlled by a pair of DMC-11's,⁶ one on each

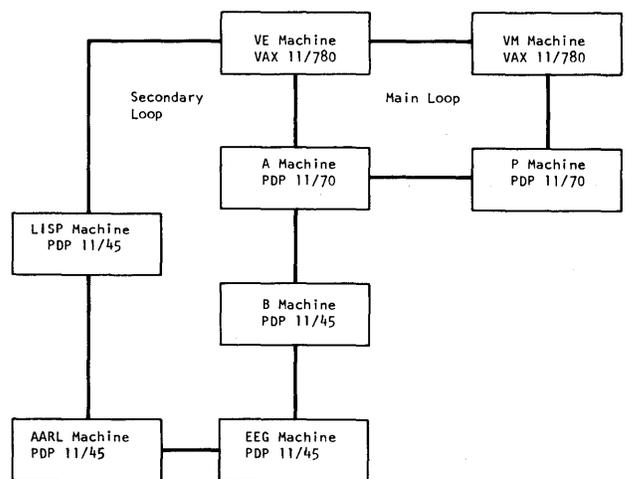


Figure 3—Double-loop configuration being considered for the next phase of ECN

computer. At present, with full-duplex and one-megabaud operations, each DMC-11 consists of a DMC11-AL microprocessor module and a DMC11-MA line unit module connected by a one-foot cable. A pair of coaxial cables (Belden 8232 double-shielded coaxial cable) are used to connect two DMC-11's. The DMC11-AL microprocessor module is a hex-sized single PC board that fits into a hex small peripheral controller (SPC) slot. It contains a 300-nsec bipolar microprocessor, a read-only-memory implementing the DDCMP protocol, local scratchpad memory, and UNIBUS interface. The DMC11-MA line unit module is also a hex-sized PC

Table 1—Architectural features in each host computer of the engineering computer network

Host	Architectural Features
A PDP11/70—UNIX Version 7	1 megabytes main memory, 355 megabytes disk memory tape drive, 70 terminals, 4 printers, printer/plotter (See Figure 2) FP-11C floating point processor, 20 microprocessors
B PDP11/45—UNIX Version 7	256 kilobytes main memory, 292 megabytes disk memory, AP-120B array processor, paper tape punch, 25 terminals
VE VAX 11/780—UNIX Version 7	2 megabytes main memory, 590 megabytes disk memory, 32 terminal lines
P PDP11/70—UNIX Version 7	1 megabytes main memory, 443 megabytes disk memory, 3 printers, 65 terminals
AARL PDP11/45—UNIX Version 6	256 kilobytes main memory, 192 megabytes auxiliary memory, 270 megabytes disk memory, image robotics I/O, 1 printer, 6 terminals
LISP PDP11/45—UNIX Version 7	256 kilobytes main memory, 316 megabytes disk memory, 2 image display systems, 10 terminals, 1 printer
EEG PDP11/45—UNIX Version 7	256 kilobytes main memory, 73 megabytes disk memory, video display, 9 terminals, analog I/O

board for use in SPC slots. It includes serial-to-parallel conversion and a built-in modem for local operation at one megabaud over coaxial cable up to 6,000 feet long. The DMC11-AL implements the DDCMP protocol in hardware; this makes efficient data communications possible. The DMC-11 is also responsible for character and message synchronization and header and message formatting. These relieve the programmer from many low-level details in data communications.

All communications between the PDP-11 and the DMC-11 are through eight bytes of control and status registers (CSRs). These registers are addressed as 76XXXY, where Y ranges from 0 to 7, and are implemented with random access memory. Four bytes of these registers are multipurpose data port registers. Their meaning is controlled by the other registers, and their use is governed by the DMC-11 microprocessor. The format and contents of the data port registers depend on the transfer type (input or output). They are loaded by the PDP-11 on input transfers and are loaded by the microprocessor on output transfers. The other four bytes of the CSRs contain commands, status information, and definition for the type of transfer. All commands, command completions, and status information pass through these registers.

The PDP-11 program is completely insulated by the DMC-11 from the communications link and the DDCMP protocol. The program initializes the DMC-11 by supplying the base address of a 64-word table in PDP-11 memory, which is called the base table. Once the base table is specified, it belongs to the DMC-11 and is readable only to PDP-11 programs. The base table is used by the DMC-11 to keep a snapshot of protocol activity for power fail recovery and defining the characteristics of the data link. Immediately after the base address is defined, the PDP-11 program performs another input transfer to define the characteristic of the link (full or half duplex). The DMC-11's will then automatically start up the DDCMP protocol and synchronize themselves in a few time intervals.

From this point on the PDP-11 program can request and use the multipurpose CSRs to provide the bus address and byte count of messages to be transmitted or buffers to be filled on reception. Transmit commands will be reposted as completed when successfully acknowledged. Receive commands will be reported as completed when an entire message has been successfully received in correct sequence. Successful command completions will interrupt the PDP-11 processor, if enabled. The PDP-11 program may queue up to seven buffers for transmission and seven empty buffers for reception by supplying buffers to the microprocessor faster than it returns them. This allows the transmission and reception of messages to be pipelined. Data integrity may be lost when more than seven buffers are queued.

The DMC-11 is designed with a number of features for reliable operation and ease of maintenance. During normal operations, the DMC-11 keeps counts of communications and transmissions. These counts are recorded in PDP-11 memory. Occasional retransmissions are handled by the DMC-11. The microprocessor informs the PDP-11 program of unusual or error conditions involving the communications channel, remote end of the link, DMC-11 hardware, or PDP-11 program. For reliable data transmission at one megabaud, buffer size is limited to 512 bytes.

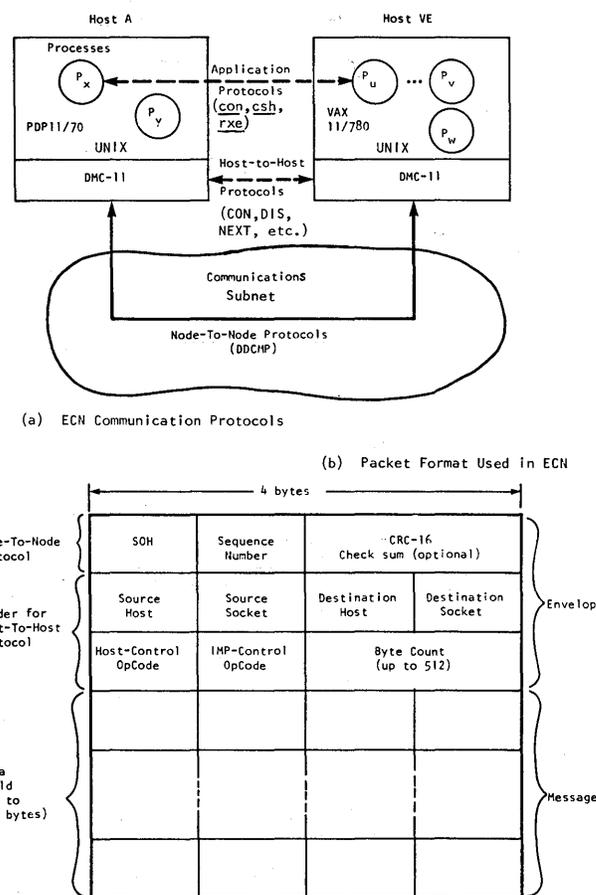


Figure 4—Communication protocols and packet format used in ECN

Some codes were written at Purdue for sequence numbers and general consistency checking and the issuance of idle packets that are used to detect the activation of the line. Although not serious, a number of minor bugs have been found in our DMC-11 hardware. On rare occasions, the DMC-11 will lose a buffer with no error status. Strange traps have occurred on one CPU when the CPU at the other end is halted. Defective devices on the UNIBUS which keep the bus too long cause problems. The sequence and timing of commands to first-time initialize the DMC-11 were found to be critical. Internal UNIX errors that lock out console error messages cause the DMC-11 to malfunction. The DMC-11 also spuriously issues RDYO interrupts (with RDYO bit clear) when under very heavy load. All these errors have been corrected, and the DMC-11s are now operating satisfactorily.

COMMUNICATIONS HIERARCHY AND SYSTEM PROTOCOLS

A hierarchy of communication protocols has been developed at Purdue University to allow resource sharing between host DEC computers and terminals in ECN and to perform various network functions as listed in the introduction. Processes within host DEC computers communicate with processes either in other host computers or in terminal handlers by means

TABLE 2—Augmented function library for the protocol programs used in ECN

Mnemonics	resulting system call	Functions Performed	Error Control
<i>fd = mxfile()</i>		Finds, opens and returns a free network file <i>fd</i> from the pool "/dev/mn/n"	
<i>mxwait (fd,socket)</i>	<i>stty (fd, {3,socket,0})</i>	Waits for connection to local socket number <i>socket</i> from any host in the network; reruns when connection arrives.	
<i>mxcon (fd,host,socket)</i>	<i>stty (fd, {1,host, socket})</i>	Connects local network file <i>fd</i> to foreign <i>host</i> , foreign <i>socket</i> ; returns if a process at the foreign host has issued <i>mxwaitl</i>) and connection is complete.	path does not exist, time out.
<i>mxscon (fd,host,socket)</i>		Similar to <i>mxcon</i> except <i>host</i> is a string of characters.	<i>host</i> does not exist in table
<i>mxdis (fd)</i>	<i>stty (fd, {2,0,0})</i>	Disconnects file <i>fd</i> ; occurs automatically on a close (<i>fd</i>).	
<i>read (fd,buffer, count)</i> <i>write (fd,buffer,count)</i>		Reads and writes buffers; reads a minimum of whatever is written in the buffers or 512 bytes and does not wait for the buffer to fill.	
<i>mxsig(fd,signal)</i>	<i>stty (fd, {6,signal,0})</i>	Sends the process or process group (see <i>mxgrp</i>) at the other end of connection, a UNIX signal number <i>signal</i> see (11)	
<i>mxgrp (fd)</i>	<i>stty (fd, {5,0,0})</i>	Places the current process and all its children by forks in the same unique <i>process group</i> .	
<i>mxeof (fd)</i>	<i>stty (fd, {7,0,0})</i>	Write an end of file (EOF) onto the connected file <i>fd</i> ; all <i>reads</i> at the other end receives a zero byte count (EOF); all <i>writes</i> at this end are ignored.	
<i>mxserve (socket)</i>		Answers requests for service on <i>socket</i> by the <i>con</i> and <i>ch</i> servers (described in the fifth section).	

of several specially developed programs. In this section we present a functional characterization of the ECN protocol hierarchy. Detailed software constructs of network application programs will be described in the next section.

Each host in ECN is identified by a one-byte host number. Processes residing in each host are uniquely identified by a socket number. A connection between a process in the local host and a process in a foreign host is specified by four one-byte numbers: source host, source socket, destination host, and destination socket. This naming convention will allow multiple connections to the same local host/socket pair, analogous to the telephone PBX service where multiple calls can be directed to the same unique phone number. A subset of the socket numbers at each host is reserved for connection services by system processes. Variable-length packets are used in ECN to allow hosts to communicate with each other. The packet format used in ECN is illustrated in Fig. 4b. The first four bytes in the host-to-host header form the connection number. The next two bytes are integer opcodes used for host-host and IMP-host controls, to be described below. The byte-count indicates the number of bytes being transferred in the data field, which may contain a maximum number of 512

bytes. The choice of this maximum size of 512 bytes matches the capacity of a typical disk block.

ECN uses a multilevel protocol similar to that implemented in the ARPA net. The major difference lies in the way packet-switching functions are being implemented. ARPA net uses a separate IMP as a switching processor, whereas in ECN the IMP functions are implemented by the hosts with the aid of the interface microprocessor DMC-11. The ECN communication protocols consist of three layers, as illustrated in Figure 4a. These protocols are implemented in C programming language, augmented with a user-callable function library for performing various network functions. Most of these functions reformat the arguments into the appropriate *stty* call on the open network file descriptor, supplied as the first argument. A brief listing of these functions and the resulting system calls are given in Table 2.

In the kernel space the network software in each machine is split into two parts. The *mx* device driver/*deve/mx/x* appears as special files to UNIX. *Open*, *close*, *read*, *write*, and *stty* calls on *mx* files pass control to the *mx* driver, which generates packets containing host-to-host protocol and passes them to the IMP process for delivery. The IMP process re-

ceives buffers (packets) from the local and neighboring hosts. The IMP examines the destination address on each arriving packet and looks up the host number in its routing table, which maps host numbers to external link numbers. The packet is then enqueued for output via the appropriate line driver.

Node-To-Node Protocol

In the lowest level of line control, the interface microprocessor DMC11 implements the same DDCMP (digital data communication message protocol) protocol used in DEC-NET.⁵ A common type of IMP-to-IMP envelope is prefixed to the host-to-host packet header, as described in Figure 4b. This envelope contains an SOH (start of header), a sequence number, and an optional check-sum. The DDCMP protocol detects channel errors using CRC-16 (16-bit cyclic redundancy check). Errors are corrected by automatic retransmissions. Sequence numbers in the envelope insure that messages are delivered in proper order without omissions or duplications.

Host-To-Host Protocols

In the middle level is the protocol for packet exchanges between the hosts. The packet header contains two function control opcode fields. Described below are opcode mnemonics used in these fields and the corresponding packet control functions to be performed.³ The host-to-host operations performed include the CON (connect), DIS (disconnect), NEXT (ready for next packet), SIG (signal interrupt), and RST (broadcast reset). The IMP-control field when nonzero indicates an IMP-to-host or host-to-IMP control opcode. A *dead* code indicates a dead host. All packets sent to a dead host should be bounced back to its source host or destroyed when both source and destination hosts are temporarily disconnected.

The CON operation requests that the connected name in the first four bytes be established. Connection is established when a pair of these is exchanged, one in each direction. If receiving host has a process with matching *mxwait* (*fd*, *socket*) pending, the matching CON is sent. If not, the CON is queued and picked up later by a *mxwait*. After connection, a CON with the same connection name results in timeout. The DIS function breaks the connection named in the first four bytes. Disconnection is complete when a pair of DIS is exchanged. The NEXT is sent by the consumer of the data packet and indicates that the data has been transferred from kernel into user space and is ready for the next data packet. The SIG sends an interrupt signal number in the first data byte to the receiving process at the other end of the connection. The RST causes the source host to inform the destination host to reset all known connections between the two.

Application-Level Protocols

In the highest level are the interprocess communication protocols. So far, three application programs have been written at Purdue University to facilitate the UNIX networking.

The *con* program allows a user to connect his physical terminal to a pseudoterminal on any other host machine. This virtual terminal protocol provides local/remote echoing by the use of *stty/gtty* functions to be described in the next section. The *cs*h program (for *connected shell*) is used to control remote process execution. It takes the host name and a sequence of commands as its arguments. The commands are executed on the specific host computer with standard I/O redirected to the local host. The *con* and *cs*h are also used under programmed network I/O, and file transfer/remote device accesses. The *rx*e program performs a load-balancing algorithm and sends jobs to the network machine with the least load average.

APPLICATION PROTOCOLS AND UNIX EXTENSION

In this section we describe the three application protocols developed at Purdue: *con* (connect virtual terminal), *cs*h (connected shell), and *rx*e (remote execution environment).

Virtual terminal program (*con*)

Con is an extended shell command that takes a terminal connected to the local host to act like a terminal connected directly to a remote host. The synopsis of the command is *con hostname*. When this command is entered into the shell with a valid host name, login messages such as password prompt are communicated from the remote host to the local host. From this time on, the local terminal acts like a terminal connected directly to the remote host. The base level shell exits when a final control-D is typed and the connection is broken. *Con* is also designed so that the actions of *escape* (hold terminal output) and the *rubout* (interrupt) key are immediate and not "squishy" because of network buffering.

The sequence of actions performed on the local host when a valid *con* command is entered from the terminal is as follows: A free network file is obtained by using the function *mxfile* () (see Table 2) at the local host. The function *mxscon* () (see Table 2) is called to connect this local network file to the remote host on Socket 1. When the connection is established, the *con* at the local host is split into two parts: One reads from the terminal (*fd0*) and writes to the network file; the other reads from the network file and writes to the screen (*fd1*). As described in the read/write commands in Table 2, reading can be done without waiting for the entire buffer to be written. At the remote host, the *con* server *S-con* is responsible for establishing the connection when a connect arrives on Socket 1. *S-con* forks once to generate a child process. The child *S-con* then finds a free pseudoterminal and forks into two parts. One part is reading from the net and writing to a pseudoterminal while the other part is reading from the same pseudoterminal and writing to the net. A pseudoterminal at the remote host consists of two sides that are named */dev/ttyx* and */dev/ptyx*. Anything written on */dev/ptyx* looks as if it has been typed in at */dev/ttyx*, while everything printed out at */dev/ttyx* can be read at */dev/ptyx*.

Certain escape and command character sequences, such as an *stty* command, when issued on */dev/ttyx*, are first trans-

lated into a command sequence before it is read by /dev/ptyx. This command sequence is then sent or other operations are performed. The format of such a command consists of an escape byte called IAC (interpret as command), followed by a command code byte, possibly followed by data for that command. The commands currently implemented include ST (set teletype), GT (get teletype), IN (interrupt signal), QU (quit signal), EF (end of file), and DM (data mark). The *S-con* is only responsible for data transfers and never interprets the commands.

Remote process protocol (*cs**h*)

Csh is an extended shell that runs a shell on a remote host, with its standard I/O the same as *cs**h*'s standard I/O. The synopsis of the command is.

```
csh  hostname  [-l user password]  "commands"
```

The quotes are not needed if special characters for the shell (such as |, ^, etc.) do not exist in the commands. If the *-l* option is omitted, the commands are run under *userid = user*, *dir = /usr/user* on the remote machine. Interrupt, quit, or hangup signals on the local host will send a hangup to the remote process.

The use of the *cs**h* command can be illustrated by the following examples. Suppose the local host is the A machine,

```
nroff filename | csh p opr
      processes the file on the A machine and prints it on the
      P machine.
csh p -l username password "cat > file 2" < file 1
      transfers file 1 on the local machine to file 2 on the
      remote machine.
```

Other capabilities of *cs**h* include transferring a directory of files.

The sequence of actions performed on the local host when a valid *cs**h* command is entered is as follows: The local *cs**h* connects to Socket 2 on the remote host by using the functions *mxfile* () and *mxscon* () (see Table 2) and writes three lines containing name, password, and command, each terminated by "\n," in a single write. When the connection is established, the *cs**h* at the local host is split into two parts: one reads from standard input and writes to the net while the other reads from the net and writes on standard output. When the half that is reading from standard input gets an EOF, it writes an EOF to the net and exits. The other half that is reading from the net will exit when the command exits at the remote host and sends an EOF to the local host. At this time the local *cs**h* exits. At the remote host the *cs**h* server (*S-csh*) listens for a connection on Socket 2. When one arrives, the connection is established and a child is forked to handle it. The child *S-csh* opens file descriptors 0, 1 and 2 (standard I/O and error) as net files and reads three items: name, password, and command line. If the name is non-null, "/etc/l-csh l-csh name" is invoked and does the lengthy job of looking up and verifying the password (still in the kernel net buffer) and executing a shell with the command line. The command performs its I/O

from the net. If the name is null, the command is run under user name *user*. All the children of this process are placed and executed in a separate process group. To speed things up, the shell is not called if no special characters exist in the command line. When the command eventually exits at the remote host, an EOF is sent to the local host to terminate the *cs**h*.

Remote Execution Environment (*rx**e*)

Rxe is a scheduling routine developed to run a selected set of commands on the most idle machine available in an (almost) transparent manner. These commands are generally CPU-bound programs that require a relatively small amount of file transfers. Therefore it would be cost-effective to execute the job in a remote host. The commands currently implemented include the compilers for c(*cc.*) and FORTRAN (*f4p.*, *fortran.*, *f77.*), microprocessor cross assemblers (*mas80.*, *mot68.*) and word-processing programs (*nroff.*, *troff.*). The period at the end of the command is used to distinguish jobs to be run in *rx**e* against jobs to be run on the local host. The synopsis of *rx**e* is

```
command  [-V]  {-H include-file}  arguments...
```

When one of the above commands is executed, *rx**e* first pre-processes the command line arguments. The . is stripped off from the command. Any argument that does not start with a - is assumed to be a file that will be transferred with the command to a remote host if the command is executed there. The *-V* flag causes a verbose listing of *rx**e*'s operations to be printed (the machine used and the files transferred). The *-H* include-file causes include file to be copied to the remote host with the command. The *-H* include-file can be repeated if several files are to be included. The *-H* option forces include-files to be transferred together with the command. Since the command may be executed on a remote machine, files included but not transferred would not be found at the remote host. Some examples of the use of this command are as follows:

```
cc. f1.c f2.c f3.0 f4.0 -H vars.h
      executes the C compilation command cc f1.c f2.c f3.0
      f4.0 with an include file vars.h on a remote machine
nroff. paper | opr
      runs the word processing program "nroff paper" on the
      most idle machine and prints it at the local host.
```

To effectively select a machine that is "the most idle," the machines must be characterized to indicate the degree of idleness. This is represented by a single number, called a load average, that is maintained in each network machine's kernel. The load average is a number that can characterize the load at a computer. Therefore, computers with higher load averages are more heavily loaded. Load average of the current machine is defined as the approximate factor of increase for the physical time it would take a given process to run on the current machine over the physical time needed for the same process to run on a completely idle PDP-11/70. It is calculated from several factors, including number of running processes, num-

ber of background processes, number of disk transfers, amount of swapping, amount of interrupts, and a site-dependent constant. The site factor is used to characterize machines with different architectures and speeds. It was developed experimentally by running compilers on all the network machines and takes into account disks, the network, memory speed, and other system dependencies as they apply to running compilers. Currently, the PDP 11/70's have a site factor of 1. The B machine, which is a PDP 11/45 with a cache, has a site factor of 1.5; and the AARL machine, which is a PDP 11/45 without a cache, has a site factor of 2.5.

The computation of the load average takes into account only a finite number of characterizing parameters and makes assumptions about things like the average mixes of CPU/IO-bound jobs, the number of child processes a process forks, the amount of memory used, etc. It is only an approximate characterization of the machines. Very few results can be reported now regarding improvement in response time; but in general, it is much faster to run the command on a less idle machine than on the local host. Experimentation is still needed in the future to further improve the performance.

The sequence of actions performed in the local and remote hosts are described here. First, *rx*e preprocesses the command by stripping off the . Next, *rx*e connects successfully to each network machine on Network Socket 3. In each network machine, the *rx*e server (*S-rx*e) is waiting (*mxwait* ()) for a network connect to Socket 3. When *S-rx*e receives this connect, it sends a two-byte load average (from /dev/kt) to the originating host. The host *rx*e picks the computer with the minimum load average and sends a 40-byte "idline" with host-name, uid, command, and *mxscon* () to this machine's Socket 4. If the host does not want the command to be processed at this machine, a disconnect is sent to disconnect Socket 3, and *S-rx*e goes back to wait for another connect on Socket 3. Two network channels are used here to avoid a race condition.

For the machine that receives the 40-byte "idline," a *mxwait* () is executed to wait for a connect to Socket 4 (*mxscon* ()) from the originating host. This wait is timed out in case a connect is not received in 15 seconds. When the line is connected, *S-rx*e forks a child *S-rx*e to become the new *S-rx*e server, which goes back to wait for a new connect on Socket 3. The parent *S-rx*e sets up a scratch disk directory, waits and receives the source files from the host on Socket 3, and copies them to the scratch directory. The argument files follows in a similar manner. On receiving all the files, the parent forks again, with the child processes executing the command and argument files received with file descriptors *fd* 0, 1, 2 connected to the net (like *csh*). When the command terminates, *S-rx*e closes Socket 3, and any resulting files created are sent to the originating host over Socket 4. Finally the parent removes the scratch directory and exits. The above descriptions have only touched the basics of the design. Interested readers should refer to the program listings of *con*, *csh*,^{1,3} and *rx*e⁷ for details.

PERFORMANCE ANALYSIS AND MEASUREMENTS

In evaluating the performance of the ECN, we focus our attention on estimating the *mean response time* of a job issued

from a node of the network. It is assumed that each job can be processed on any node of the network. The objective of the model is to compare the performance of two scheduling strategies for dispatching jobs to nodes of the network. Currently, only a few commands, such as *cc* (compilation) and *nroff* (text formatting) are implemented on the system for load-balancing purposes. Since the ECN performs load balancing for a small class of jobs, the results obtained below would be optimistic on the whole, since some nodes would still have high workloads and hence encounter high response times.

Figure 1b shows a directed graph representation of the ECN. For communication between any two adjacent nodes (1 hop), the maximum throughput experienced by a single user is about 250 Kbaud. Although a three-hop communication requires an intermediate node, it was also found from measurements on the system that the processing performed by the intermediate node is negligible and does not contribute significantly to the workload at that node. This is expected, since the only processing required by the intermediate machine is to transmit the packet from the input DMA to the output DMA device. These measurements permit us to assume that the intermediate processors cause negligible delays in forwarding the bypassing packets. Several snapshot throughput measures on the current ECN were obtained, as shown below:

Path	Hops	Throughput in Kbaud
P-P	0	384
A-B	1	273
P-B	2	180
P-AARL	3	136

With time sharing use of the communication links by multiple users, the above throughput per single user can be increased to approach the maximum rate of 1 megabaud.

Most performance evaluation of computer networks considers only the behavior of the communication channels and not the behavior of the model processors.⁹ Delay at the channels contributes most to the total delay. Others have developed analytical models based on the destination probability of a job from a source node.¹⁶ The model developed here evaluates scheduling strategies for jobs based on the workload characterization of both the channels and the node processors.

It is assumed that a job is formatted as a message that consists of a command identifier and a list of arguments. The command message is dispatched to the communication interface (CI), where it is queued to await transmission over a physical link to the CI of another node. The message delays encountered vary in going from a source to a destination as the workload on the intermediate links changes. The destination processor executes the process specified by the command identifier using the list of arguments. A result message generated at this node is routed back to the initiating node processor. It is assumed that the command and result message lengths are independent and exponentially distributed random variables. The overall response time of the job would depend on the channel delays and the workload at the destination processor.

Let N represent the set of nodes in the graphical repre-

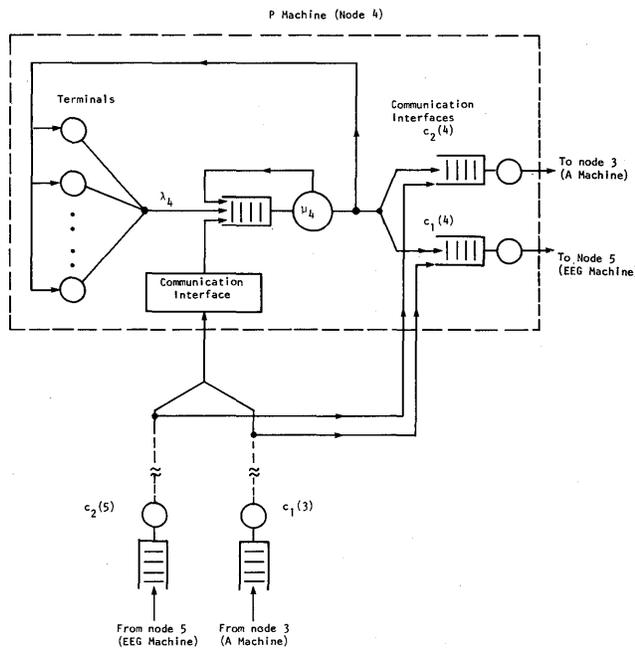


Figure 5—Queuing network model of Node 4 (P machine)

sentation of the ECN shown in Figure 1b. Hence $N = \{1, 2, \dots, 7\}$. The communication device, $c_m(i)$, routes a message from a node i to its immediate neighbor node j ; if $i \neq 3, m = 1$, if node j is the “right” or “down” neighbor of node i ; and $m = 2$, if node j is the “left” or “up” neighbor of node i , as labeled on Figure 1b. Figure 5 shows a queuing network of Node 4 (P machine) and its associated communication devices.

The intermediate nodes of a message path can be found by using an $n \times n$ routing table, R ($n = 7$, for the ECN). The routing model creates a static nonadaptive logical path from source i to destination j . Specifically, $R(i, j)$ contains the index ($k = 1, 2, \dots, 7$) of the next node (or “hop”) on the logical path from i to j . Hence, given the source and destination nodes as i and j respectively, the sequence of intermediate nodes visited by a command message describes the forward path ($i \rightarrow j$) iteratively as

$$I(i, j) = \{i, i_{k+1}, i_{k+2}, \dots, j\}$$

where $i_{k+1} = R(i, j)$, $i_{k+2} = R(i_{k+1}, j) \dots$, and so on until $R(i_p, j) = j$ for $i_p \in I(i, j)$. Similarly, the return path ($j \rightarrow i$) can be obtained as $I(j, i)$. In order to evaluate the performance network, the effects of two scheduling strategies were studied for the ECN. The first strategy S_1 sends a job from a node i to node j for processing, if node j has the minimum estimated response time at time t ($lrt_j(t)$) of a job processed at Node j , using a processor sharing model. Hence,

$$lrt_j(t) = \frac{\bar{x}_j}{1 - \rho_j(t)}$$

where \bar{x}_j is the mean service time of the processor at node j and $\rho_j(t)$ is the measured processor utilization at time t and is defined as the fraction of time the processor was busy during the interval $[0, t]$. Hence the first strategy can be specified as

follows: dispatch job that arrives at time t from node i to a node j , where j is the processor node with a $\min_{k \in N} \{lrt_k(t)\}$. The system provides a status report of the network in which the load average and utilization of each processor are updated periodically. This information can be used for scheduling purposes.

The first scheduling strategy does not consider the overhead of message transmission from a source node to a destination and the return path. In the second strategy, S_2 , we define an estimated response time of a job at time t dispatched from node i and to be processed at node j . The estimated response time at time t is given by

$$\bar{W}_{ij}(t) = \sum_{k \in I(i, j)} \bar{T}_{c_m(k)}(t) + lrt_j(t) + \sum_{k \in I(j, i)} \bar{T}_{c_m(k)}(t)$$

for $i \neq j$. When $i = j$, $\bar{W}_{ii}(t) = lrt_i(t)$. $\bar{W}_{ij}(t)$ consists of three components, namely, the estimated delay time of the command message in the forward path, the estimated response time of the job processed remotely at Node j , and the estimated delay time of the result message in the return path, all at time t . The delay of a message in each communication channel can be modeled as an M/M/1 queuing system, as shown in Figure 5. Hence, the estimated delay time in channel $c_m(k)$ at time t is

$$\bar{T}_{c_m(k)}(t) = \frac{\bar{m}_k}{1 - \rho_{c_m(k)}(t)}$$

where m_k is the mean message length of jobs departing from node k and ($\rho_{c_m(k)}(t)$) is the measured utilization of channel $c_m(k)$ at time t and is defined as the fraction of time channel $c_m(k)$ was busy in the interval $[0, t]$.

Figure 6—Response time distribution for scheduling strategies S1 and S2

The second strategy can then be summarized as follows: dispatch job that arrives at time t from node i to a node j , where j is the processor node with $\min_{k \in N} \{\bar{W}_{ik}\}$. Figure 6 illustrates the response time distribution of the two scheduling

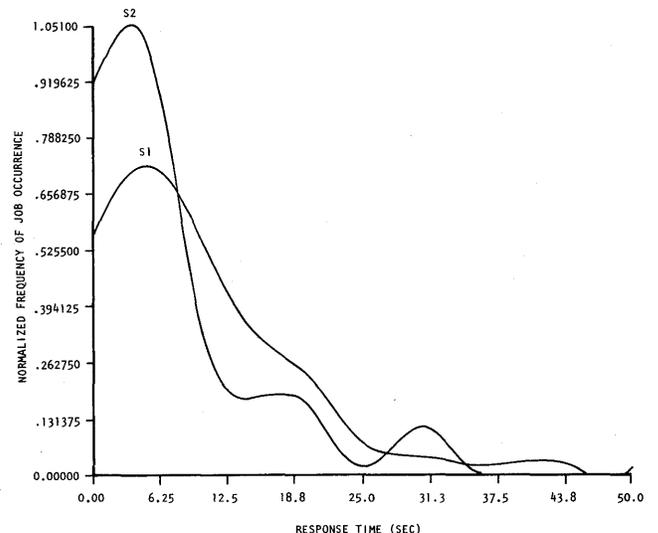


Figure 6—Response time distribution for scheduling strategies S1 and S2

strategies discussed above for a given system load. It can be seen from the distributions that the scheduling strategy taking into consideration the channel delays in dispatching the jobs has a smaller mean and standard deviation of the response time.

Two measurements were performed on the ECN to evaluate the effect of job classes on the response time under varying workload. The first job class consists of a channel-bound job in which a large file is transferred from the VE (Node 1) periodically to a "null" device at the node with the smallest load average. The second job type consists of a processor-bound job (an executable tight-loop program) that is dispatched periodically for execution to a node with the least load average. These transactions are performed under a wide variety of workloads, and a record is kept on the effect of the destination processor utilization on the response time. The measurements showed that the response time of the channel-bound job was virtually independent of the utilization of the destination processing mode. This is expected, since the channel loads were generally light, although the node utilizations varied considerably.

CONCLUSIONS

The experiences accumulated from developing the UNIX network of DEC computers at Purdue University are summarized below:

1. Hardware components and communications links of the networks are readily available from standard DEC product lines. No special hardware designs are needed to construct such a modest but effective local computer network. This off-the-shelf approach saves significant development overhead with a controlled budget.
2. No major changes of the UNIX operating system were made to establish the network functions. A small number of changes, however, required an in-depth understanding of the operation of UNIX. Only a handful of specially written system programs (*con*, *csh*, *xre*) at the highest protocol level, together with a library of host-to-host network functions (see the fourth and fifth sections) are needed to establish the virtual terminal, remote process execution, file migration, and user-programmed I/O capabilities.
3. The ECN is being reconfigured to a double-loop computer network (Figure 3). Over 210 CRT terminals and 20 microprocessors are currently connected to the seven minicomputers in ECN. The two PDP 11/70 computers (A and P machines) are also connected via 0.2 megaband lines to the Purdue Computing Center, which itself has over 250 connected terminals. Users at terminals connected to ECN can use the computing center facilities (CDC 6500/6600) in batch mode. The CDC computer users cannot use the ECN facilities from their terminals. This restriction is enforced to insure the network services to engineering users.
4. The ECN performs satisfactorily for research and teaching usage by engineering schools at Purdue. The performance analysis given in the sixth section shows that

the communication line utilization per user is only at 15% to 40% of its maximum baud rate. This means the performance of the network can be further upgraded by timesharing use of the communication links. This is definitely an area worthy of further R&D efforts.

5. The reliability of each host in the ECN net is rather high. However, whenever a host fails, all the terminals connected to it are disabled and all the data flow paths containing this failing node are broken. In this sense, the availability of the star network (Figure 1) is expected to be much lower than that of the loop network (Figure 3). Fault tolerance capabilities built into the Ohio State double-loop network¹⁰ are being considered to enhance the availability of ECN.

ACKNOWLEDGMENTS

The engineering computer network project was initiated and supervised by Professor Clarence L. Coates of the School of Electrical Engineering at Purdue University. Staff members directly involved in the network installation, development, and maintenance include William R. Simmons, Bill Croft, George H. Goble, Craig Strickland, Michael Marsh, Joe Royeis, Curt Freeland, and Peter Miller, all of whom are with the EE Digital Service group at Purdue. The local extensions of UNIX software for network operations are also attributed to the digital service group. In particular, Mr. Croft developed the library of network functions and the *con* and *csh* programs. Mr. Goble wrote the *xre* program and made some modifications to the UNIX kernel. The authors wish to express their gratitude to Professor Coates, Mr. Simmons, Mr. Goble, and other staff members for sharing their firsthand experiences, on which this paper is based. Assistance from P. Loomis, V. Hill, and V. Johnson in preparing the manuscript is also appreciated.

REFERENCES

1. Bell Lab. Technical Staff, *UNIX Time-Sharing System: UNIX Programmer's Manual* Seventh Edition, Vol. 1, Vol. 2A, 2B; January 1979.
2. Chesson, G. L. "The Network UNIX System," *Operating Systems Review*, Vol. 9, No. 5, 1975, pp. 60-66.
3. Croft, B. "UNIX Networking at Purdue," *Technical Report* (unpublished) School of Electrical Engineering, Purdue University, Lafayette, Indiana 1979.
4. Davies, D. W. et al, *Computer Networks and Their Protocols*, John Wiley & Sons, Inc., New York, 1979.
5. Digital Equipment Co., *The DECNET*, Chaps 1-3, Maynard, Mass. 1976.
6. Digital Equipment Co., *Terminal and Communications Handbook*, 1978, pp. 2-78 to 2-97.
7. Goble, George H., "RXE Program and Load Balancing in ECN," Private Communications, 1980.
8. Hwang, K., *Distributed Processing and Computer Networks*, EE660 Class Notes School of Electrical Engineering, Purdue University, Lafayette, Ind. 1980.
9. Kleinrock, L. *Queueing Systems, Vol. II, Computer Applications*, Wiley Interscience, New York 1976.
10. Liu, M. T., "Distributed Loop Computer Networks," *Advances in Computers*, Vol. 17., Academic Press, Inc., 1978, pp. 163-221.
11. McQuillan, J. M. and Cerf, V. G., *A Practical View of Computer Communications Protocols*, IEEE Computer Society, Catalog No. EHO-137-0, 1978.

-
12. Newkirk, J. and Mathews, R. *A Guide to Array Processing Under UNIX*, Peninsula Research, Palo Alto, Calif., 1978.
 13. Nowitz, D. A. and Lesk, M.E., "A Dial-Up Network of UNIX Systems," Bell Laboratories, Murray Hill, N.J. August 1978.
 14. Nowitz, D. A., "Uucp Implementation," Bell Labs., Murray Hill, N. J., Oct. 1978.
 15. Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *The Bell System Tech. Journal*, Vol. 57, No. 6, August 1978, pp. 1905-1930.
 16. Samari, N. K. and Schneider, G. M., "The Analysis of Distributed Computer Networks Using M/D/Y and M/M/I Queues," *Proc. of The First Int'l. Conf. Dist. Compt. Systems*, Oct. 1979, pp. 143-155.
 17. Schwartz, M. *Computer Communication Networks Design and Analysis*, Prentice Hall, Englewood Cliffs, N. J. 1977.
 18. Staffs of Digital Service Group, "Introduction to EE UNIX," School of Electrical Engineering, Purdue University, September 1980.
 19. Strickland, C. "EED (editor)," Supplement to *UNIX Programmer's Manual*, School of Electrical Engineering, Purdue University, Lafayette, Indiana, 1979.
 20. Thurber, K. J. and Freeman, H.A., "Architecture Considerations for Local Computer Networks," *Proc. of The First Int'l Conf. on Distributed Computing Systems*, October 1979, pp. 131-142.

A protocol for a new double-loop computer network and its implementation

by S. LEVENTIS
G. PAPADOPOULOS
S. KOUBIAS
and J. CONSTANTINIDES

*School of Engineering
University of Patras, Greece*

ABSTRACT

In this paper a new double-loop computer network is presented, as well as its hardware implementation. The proposed protocol permits simultaneous transfer of variable-length messages even between interfering segments of the network. Various concurrent transmissions can also take place with this protocol. These operations, together with a completely distributed control mechanism, make this new network capable of providing automatic traffic regulation. In addition, the reliability of the system is improved. The nodes of the network can be implemented by existing programmable LSI communication protocol controllers. This simple hardware implementation and the above capabilities make the proposed network very attractive for local networks with high traffic demands. In this paper a node realization, based on the 8085A microprocessor chips, is also presented. Every node consists basically of two SDLC Chips, one DMA controller and the Interrupt Controller.

INTRODUCTION

In recent years a variety of loop networks has been presented.¹⁻⁷ The main effort of the researchers has been directed to the improvement of the reliability and the throughput of the system. A list of the most representative loop networks is shown in Table I, where the loops are compared with respect to (1) concurrent transmission, (2) service of variable-length messages, and (3) distributed control mechanism.

The reliability of a network depends strongly on the simplicity of the nodes, the modularity of the system, and the existence of distributed control. The loop networks combine the above properties and thus have become very attractive in applications that are geographically contained. They are especially well suited to the development of distributed control, and many researchers have addressed their attention to such protocols.

The network throughput depends on the degree of use of the lines, capacity connecting the nodes, and the simultaneous

service of the network users. Full use of line capacity depends on efficient use of the line bandwidth. One cause of bandwidth waste is the headers that are added in the messages. A significant percentage of bandwidth is also wasted for stuff bits with protocols that use messages of fixed length. For these protocols the packet size must be computed by taking into consideration the leader size in such a way that the bandwidth waste is minimized. The optimum packet size is obtained by minimizing the product of the average number of packets per message and the length of the packet.¹³ That is,

$$\frac{d}{dB_p} [f(B_p) \cdot (B_p + H)] = 0$$

where B_p is the packet size and H is the header of the packet.

The above expression, however, is very restrictive, because, on one hand, it requires very good a priori knowledge of the demand statistics and, on the other hand, it does not permit an easy readaptation of the network to changes in the network loads. In general we can say that protocols that handle messages of variable length are dynamically self-adaptive for a greater range of load servicing.

Analytical and simulation results of the loops Newhall, Pierce, and DLCN¹²⁻¹⁶ show clearly that the performance of the network with respect to channel use, message delay, user response time, system throughput, etc., improves significantly for protocols that have the following characteristics:

1. They permit the transmission of variable length messages.
2. They permit simultaneous loop access to multiple users.

Although simultaneous loop access to many users is a highly desirable feature of loop networks, an unlimited increase of this capability would result in a waste of the channel bandwidth for control. In addition, the complexity of the nodes would increase considerably.

To overcome these problems Jafari and colleagues⁷ constructed a double-loop network in which one loop is used for message transmission, the other for control. The addition of a control loop considerably improves the network response.

TABLE 1

LOOP	concurrent transmis.	packets of var. leng.	distributed control
FARMER/NEWHALL	NO	YES	YES
WELLER	NO	NO	NO
FARBER	NO	YES	YES
PIERCE	YES	NO	NO
DLCN	YES	YES	YES
SPIDER	YES	NO	NO
SDLC	NO	YES	NO
JAFARI	YES	YES	NO

The drawback of this structure, however, is that one loop, the control one, is underused.

The double-loop structure we have proposed, in addition to having the flexibility of concurrent service to many users, uses both loops fully by allowing message transmission in both of them. One loop transmits messages clockwise and is called *forward*; the other transmits counterclockwise and is called *back*. The proposed double loop also gives another highly desirable property, better traffic regulation. This is so because it can achieve

1. Concurrent service on overlapping segments
2. Compensation for asymmetrical demands by means of the backward loop

In addition, the proposed structure retains all the advantages of the previous loop structure, such as

1. Distributed control
2. Transmission of variable-length messages
3. Simple nodes
4. High reliability
5. Cost and space modularity

DESCRIPTION OF THE NEW LOOP STRUCTURE AND THE PROTOCOL INTERFACE

As was mentioned already, the proposed network consists of two loops. The outside loop carries traffic clockwise (forward) and the inside loop carries traffic counterclockwise (backward). Both of these loops transfer information frames and control frames. The topology of the network is shown in Figure 1. Also depicted in this figure is a combination of concurrent loop accesses, from which it is clear that communication between interfering node pairs is permitted.¹¹

Generally every node acts as a relay with one-bit delay in the forward and backward directions. In Figure 2 the rough schematic of each node is shown. It is seen that two SDLC chips (one for each loop) under microprocessor control form the basic hardware for each node. The SDLC chip was found very convenient because it combines the relay mode, the transmit and receive functions, assembly disassembly of protocol frames, and digital PLL synchronization all in one unit.

The information packets traveling in the loops have the format of the frames of the bit-oriented SDLC communication protocol,⁹ except that the address and control fields of the

frame have different meaning here, since both of these fields are used as addresses. The address field defines the destination address of the packet, and the control field contains the source address of the packet. The actual use of these fields for all kinds of packets used in the interfacing protocol is shown in Figure 3 and described in the following paragraphs.

Two kinds of frames travel around in the forward and backward loops:

1. Control frames
2. Information frames IF← IF→

The control frames are characterized as general frames, since they are recognized by all the nodes and play an important role in the traffic regulation of the network. The control frames are distinguished in

1. Pass control frames (PCF)
2. Start frames (SF)
3. Forward subcontrol frames (FSF)
4. Backward subcontrol frames (BSF)

The PCF, SF, and FSF circulate in the forward loop, while in the backward loop only the BSF general frames circulate.

Besides the control frames, the proposed protocol makes use of two more special control characters. These are the EOP (01111111) and the IDLE (more than 15 continuous 1s). The PCF followed by an EOP character makes it possible for the next node it will go through to gain control of the loop, if it has a message to transmit, by converting the EOP character to an opening flag.

In the beginning of a cycle of operation, all nodes are in the 1-bit delay mode except for the node that is the current controller of the loop. This node sends an SF that denotes to all the other nodes down the line the identity of the controller for this current cycle of operation. Right after the SF, the controller sends a message for node *n* in the forward loop (IF_{c→n}). As soon as node *n* recognizes its address in the IF_{c→n}, it exits from the 1-bit delay mode and assumes the role of a subcontroller. In this capacity it sends an FSF message in the forward loop and a BSF message in the backward loop followed by an EOP character. At the same time it continues to receive information from the controller. The FSF and BSF frames contain in the field A2 the address of the subcontroller and in the field A1 a general address that is recognized by all the nodes of the network.

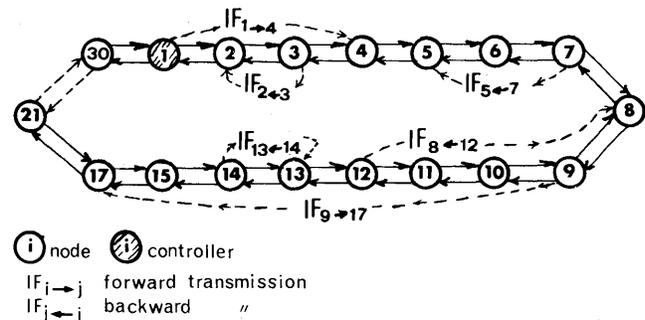


Figure 1—The new loop structure

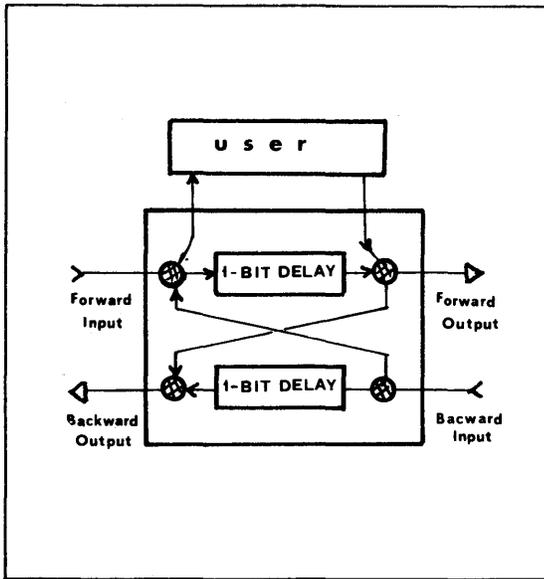


Figure 2—Node structure

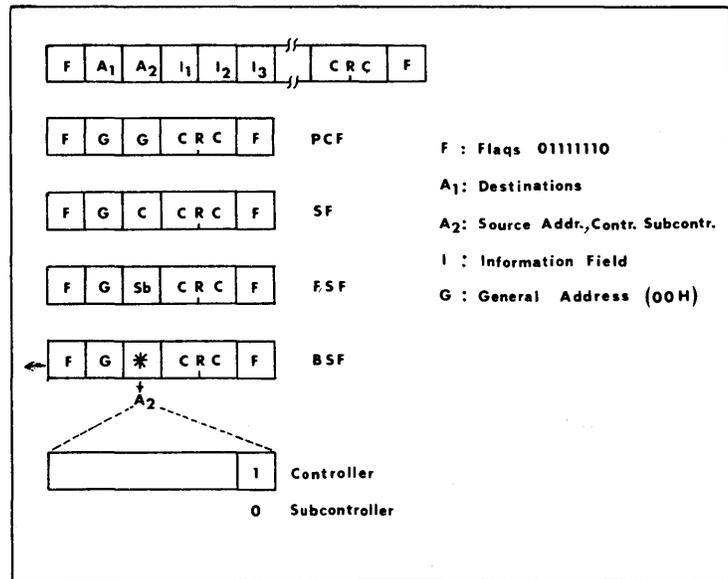


Figure 3—The frame format

The nodes between the subcontroller and the controller can transmit messages $IF \leftarrow$ in the backward loop responding to the BSF frame and the EOP character. These backward messages can go all the way up to the controller, but not beyond it.

Every node forward of the subcontroller responding to the FSF and the EOP can function in one of the following two ways:

1. It can become a second subcontroller if it has a message to be transmitted in the backward loop up to the first subcontroller.
2. It can send a forward message all the way up to the controller. In this case this node keeps the SDLC chip of the backward loop in the 1-bit delay mode. In case it has messages for both directions, it decides which one to send according to a local algorithm. All the nodes beyond the second subcontroller that receive the $IF \rightarrow$ can function in a manner similar to that described for the second subcontroller.

The current loop cycle ends when every node in the loop has responded to the processes initiated by the $IF \rightarrow$ of the main controller and has returned to the 1-bit delay mode. In the meantime the main controller, as soon as it completes the transmission of the $IF \rightarrow_n$, begins to send IDLE characters in the forward loop. Also, as soon as it receives an EOP character in the forward loop, it begins to send IDLE characters in the backward loop. These IDLE characters will return to the main controller only when all the nodes have returned to the 1-bit delay. When that happens, the controller sends a PCF frame followed by an EOP character in the forward direction and flags in the backward. Thus, control is transferred to the next node in the forward direction that has a message to transmit, and the current controller returns auto-

matically to the 1-bit delay mode. (More details of these above functions can be found in a previous work by the authors.¹¹⁾

To determine the performance of the proposed double-loop structure in comparison to other existing structures, an expression is written for the time required to execute complete loop cycle,

$$t_c = 8 \cdot \text{MAX}(I) + \text{MAX}(I^*) + I_0 + A + kj \cdot t_{GF}$$

$$A = t_{\text{SYN}} + t_{\text{SF}}$$

$$B = t_{\text{IDLE}} + t_{\text{SYN}} + t_{\text{PCF}}$$

where

- I is the length of the information field of the frame.
- I^* is the number of zero bit insertions per frame required by the SDLC protocol.
- I_0 is the number of bits required for the fields A_1 , A_2 and CRC and the flags of the frames.
- A is the time required for the synchronization of the nodes.
- B is the time required to pass control to another node.
- kj is the number of subcontrollers in the current cycle of operation.
- t_{GF} is the length of the general messages FSF or BSF or PCF or SF.

Consequently, if n is the total number of cycles required for the service of a set of demands, then the total servicing time will be

$$t_o = n \cdot \{8 \cdot \text{MAX}(I) + \text{MAX}(I^*) + t_{GF}\} + nA + (n - 1)B$$

$$+ t_{GF} \sum_{j=1}^n kj$$

Preliminary studies have show that for the case of fixed-length messages and for a dense set of demands the servicing time is improved with the proposed protocol up to a factor of

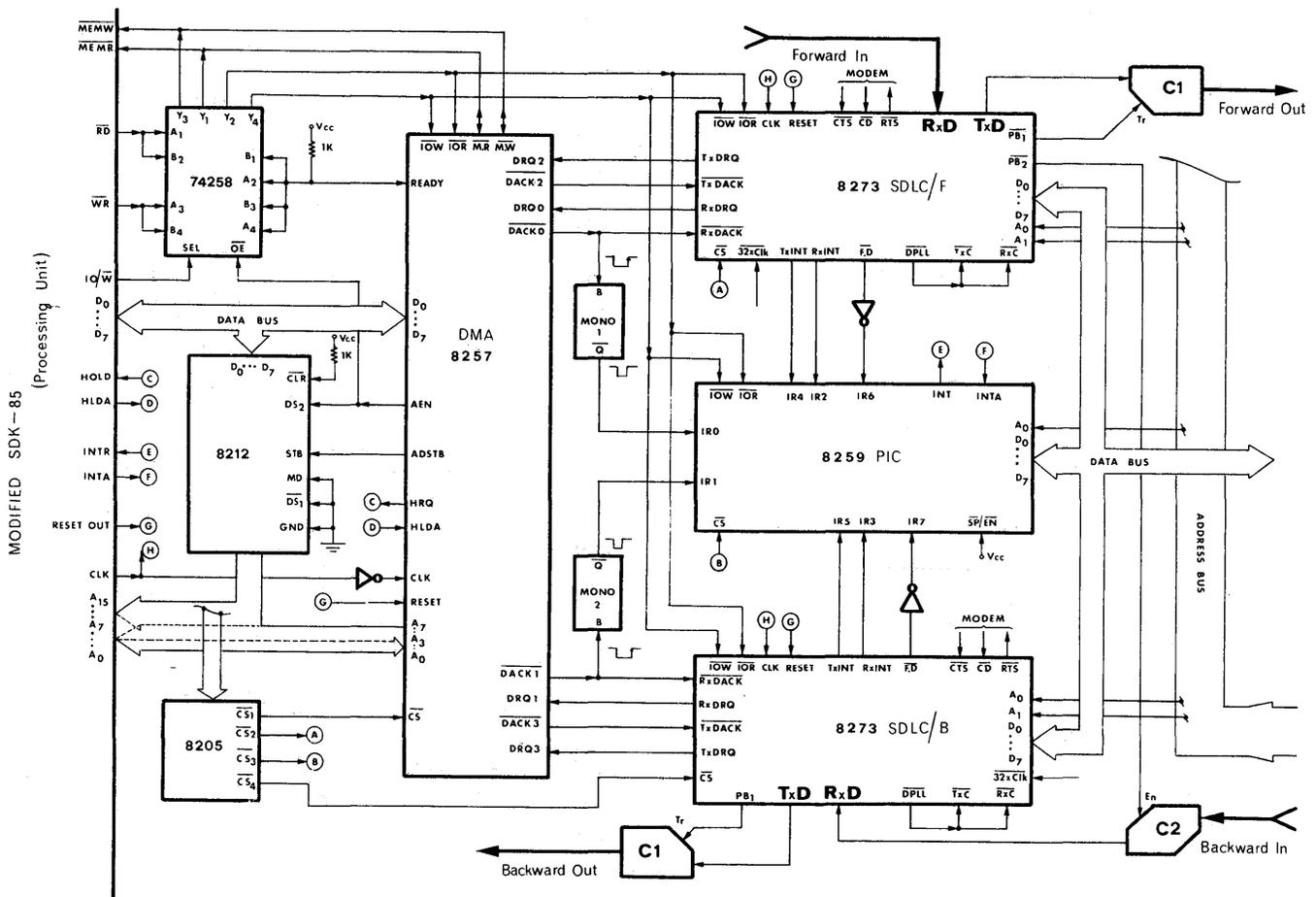


Figure 4—Node design

3 in comparison to the corresponding times for the Pierce and Newhall loops. Analytic and simulation studies of the new protocol are currently under way.

HARDWARE IMPLEMENTATION OF THE PROPOSED DOUBLE LOOP

One realization of the proposed double-loop network is presented here. As shown in Figure 2, the heart of the node is two SDLC chips. These two chips, supported by a microprocessor, carry out all the functions of the node in response to general frames. From the discussion in the preceding section we can summarize the possible functions of these nodes as (1) controller, (2) subcontroller Type 1, (3) subcontroller Type 2, (4) transmitter in the forward direction, (5) transmitter in the backward direction, (6) receiver in the forward direction, (7) receiver in the backward direction, (8) 1-bit delay forward, and (9) 1-bit delay backward. It must be emphasized that a node can be one or more of these states, since we have two loops.

Although many decisions must be taken in each node, the work that must be carried out by the microprocessor is not so heavy, because the SDLC chips perform many of the node tasks by themselves. The SDLC chips we use in this design are the INTEL 8273, which may be programmed in one of

the following modes of operation:²⁰ (1) Frame assembly-disassembly, (2) 1 bit-delay, (3) automatic DPLL synchronization of the loop, (4) NRZI coding-decoding, (5) DMA handshaking, (6) selective frame reception, (7) automatic checking, (8) frame error detections, (9) modem handling. It can be seen that one of the tasks of the microprocessor is to program the SDLC chips in the appropriate mode. This is done with a few instructions in time interval small in comparison to the maximum bit rate of the SDLC chip (64KHz).

The programming of the SDLC chips is dictated by the protocol we are proposing. Thus, the microprocessor must continuously read the fields A1 and A2 of the control frames that are detected by the SDLC chips so that it can keep a running file for the current states of the other nodes of the network. On the basis of this information the microprocessor decides whether it can send a message in the forward or backward direction and programs the SDLC chips accordingly.

For the network to operate at the maximum rate of the SDLC chips, these chips must be programmed in DMA data transfer mode. In this mode all data transfers occur directly between the SDLC chip and the memory, and the microprocessor is free for other operations. It returns to the control of the SDLC chip when it is notified by means of the signals TxINT or RxINT of this chip to examine whether the transmission or reception was successful.

Another important point in the implementation of the pro-

posed protocol is that the microprocessor must read the fields A1 and A2 as soon as they are detected by the SDLC chips so that it can determine the nature of the frames and/or their destination. To accomplish this function, the SDLC chips are programmed in the non-buffered mode. In this mode the fields A1, A2 are transferred to the memory by means of the DMA controller, as if they were information bytes, and are not buffered in the 8273 until the reception of the frame. Since the microprocessor must examine these fields as soon as they arrive and take the appropriate decisions, a logical circuit was designed that generates two interrupts for the CPU syn-
 nous with the first two RxDACK signals.

The schematic for the circuit is shown in Figure 4. It is seen that the rising edge of the DACK pulse of the forward loop at the time the DMA transfer has been completed generates another narrow pulse through a monostable that is applied to the interrupt request input IR0 of the programmable interrupt controller (PIC) 8259. A similar pulse from the backward loop is applied to the input IR1. The PIC recognizes only the first two interrupt requests IR0 (IR) and ignores the rest until it recognizes the interrupt request that is received from line RxINT/F (RxINT/B).

Although the SDLC chips support the new loop protocol, there are nevertheless some requirements for the hardware implementation of the protocol that need special attention. These requirements are critical and are summarized below:

1. The asynchronous capture of the loop by a node for BSF transmission.
- and
2. The interruption of the one-bit delay mode by a subcon-
 troller for the transmission of FSF or BSF followed by an EOP character. Thus, the problem is the imple-
 mentation of these operations without disturbing the bit stream on the loop, or, put another way, without disturbing the synchronization of the loop.

The above requirements are implemented by means of the circuit C1, which is placed between the transmission lines and the Tx D output of the SDLC chip, and circuit C2, which is placed between the transmission line and the Rx D input of SDLC chip in the backward loop, as shown in Figure 4. Circuit C1 generates an EOP character right after the transmission of an FSF or a BSF control frame without disturbing the synchronization of the nodes. The circuit diagram of C1 is shown in Figure 5a, and its operation is described by means of the timing diagrams of Figure 5b.

Circuit C2 is used only in the backward loop. Once it is activated by the PB2 port of the forward SDLC chip, it converts the flag of the backward loop into an artificial EOP character that causes the backward SDLC chip to exit from the one-bit delay node and to transmit a BSF. Once the BSF is transmitted, the backward SDLC chip automatically returns

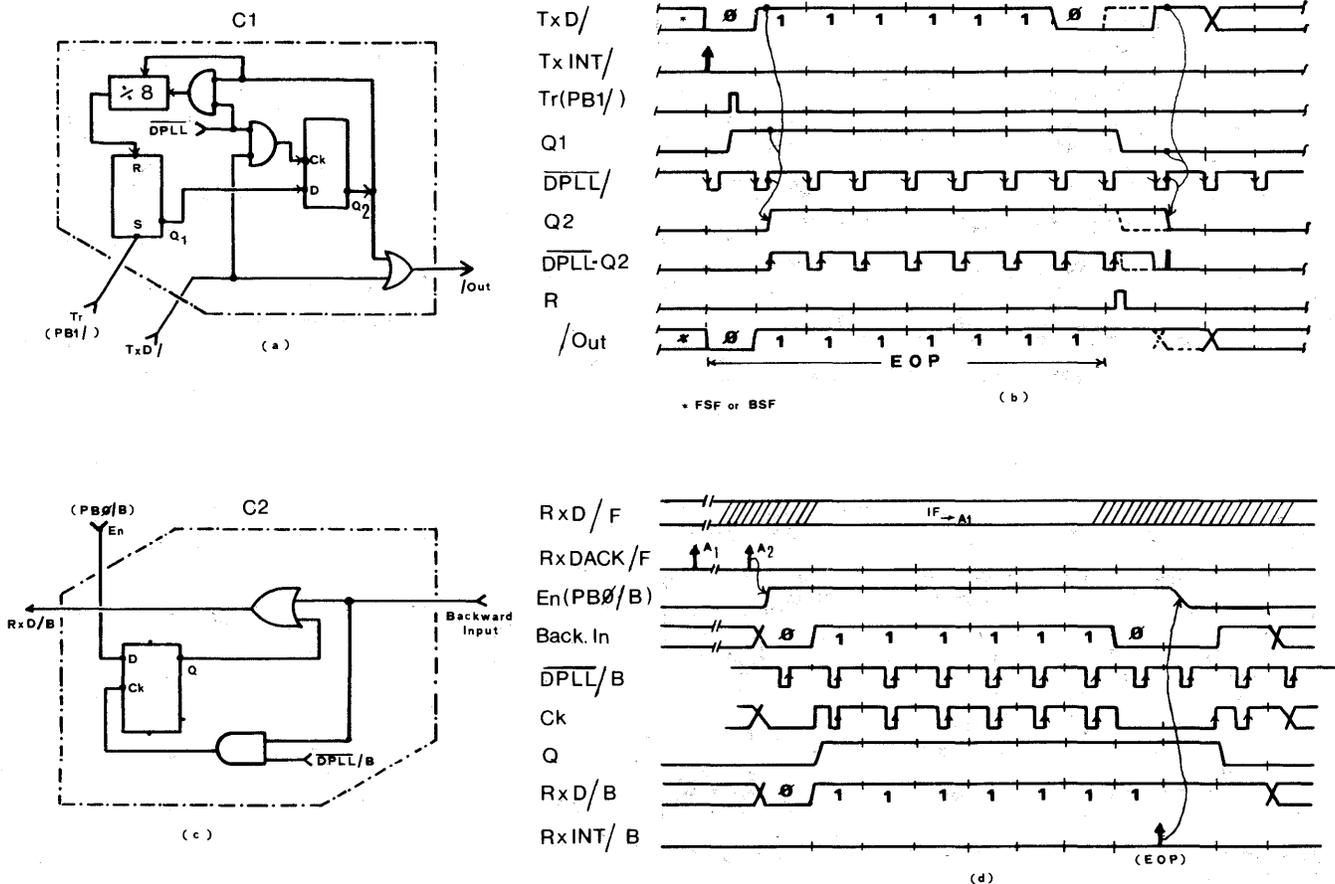


Figure 5—Description of circuits C1 and C2

to the one-bit delay mode. The circuit diagram of C2 is shown in Figure 5c, and its operation is described by means of the timing diagram of Figure 5d.

CONCLUSIONS

In this paper a new double-loop structure and its protocol have been presented. Both loops carry information and control messages. This structure combines all the desirable properties of loop networks, such as concurrent servicing of variable-length messages, distributed control mechanism, reliability, cost-space modularity, simple node structure, dynamic reconfiguration, and traffic regulation. Preliminary evaluation has shown that it improves significantly the throughput of the classical loop networks. A dynamic simulation of this structure is under way and the initial results support the conclusions of the preliminary evaluation. The complete simulation results will be presented in a subsequent paper. The design of the nodes of this loop network is also presented. The implementation is based on the SDLC chip, thus resulting in a very simple and modular node structure.

REFERENCES

1. Pierce, J.R. "Network for Block Switching of Data." *BSTJ*, 51, (1972), pp. 1133-1145.
2. Farmer, W.W., and E.E. Newhall. "An Experimental Distributed Switching System to Handle Bursty Computer Traffic." *Proc. ACM Symposium "Problems in the Optimization of Data Communications System"*, Dience Mtn., Georgia, October 1969.
3. Farber, D.J., and K. Larson. "The Structure of a Distributed Computer System—The Communication System." In *Proc. Symp. on Computer Communications, Networks and Teletraffic*. Brooklyn, New York: Polytechnic Institute of Brooklyn Press, 1972, pp. 21-27.
4. Fraser, A.G. "Spider—A Data Communication Experiment." Computing Science Technical Report No. 23, Bell Telephone Laboratories.
5. Liu, M.T., and C.C. Reames. "The design of the distributed loop computer network." *Proc. Int. Comput. Symp.*, 1 (1975), pp. 273-282.
6. Liu, M.T., and C.C. Reames. "Message Communication Protocol and Operating System design for DLCN." *Proc. Annu. Symp. Comput. Arch.* 4th, 1977, pp. 193-200.
7. Jafari, H., J. Spragins, and T. Lewis. "A New Modular Loop Architecture for Distributed Computer Systems." *Trend and Application*, 1978, Distributed Processing.
8. Jafari, H. "A New Loop Structure for Distributed Microcomputing Systems." Ph.D. dissertation, Oregon State University, 1978.
9. IBM Synchronous Data Link Control General Information, Report No. GA27-3093-1.
10. Liu, M.T., R. Pardo, D. Tsay, J.J. Wolt, B.W. Weid, and C. Chou. "System Design of the Distributed Double-Loop Computer Network." First International Confer. on Distr. Comp. Systems, Huntsville, Alabama, October 10, 1979.
11. Papadopoulos, G.D., C. Leventis, and S.A. Koubias. "A New Protocol for a Distributed Loop Communication Network." A Selection of Papers from INFO II, 2nd International Conference on Information Sciences and Systems, University of Patras, Greece. *Advances in Communication*, Vol. 1, pp. 297-304. D. Reidel Publishing Co., 1980.
12. Hayes, J.F., and D.N. Sherman. "Traffic Analysis of a Switched Data Transmission System." *BSTJ*, 50 (1971), pp. 2947-2978.
13. Anderson, R.R., J.F. Hayes, and D.N. Sherman. "Simulated Performance of a Ring-Switched Data Network." *COM-20*, No. 3 (1973), pp. 576-591.
14. Carsten, R.T., and J.M. Morton. "Simulated Static Models of Single and Multi Newhall Loops." *National Telecom. Confer.*, 3, (1978), pp. 44.1.1-44.5.7.
15. Reames, C.C., and M.T. Liu. "Design and Simulation of the Distributed Loop Computer Network (DLCN)." Third Annual Symp. on Comp. Archit., 1976.
16. Babic, G.A., M.T. Liu, and R. Pardo. "A Performance Study of the Distributed Loop Computer Network." *Proceedings of Comp. Network Symp. Gaiserbury*, 1977, pp. 66-75.
17. Kropfl, W. J. "An Experimental Data Block Switching System." *BSTJ*, 51 (1972), pp. 1147-1165.
18. Liu, M.T. "Distributed Loop Computer Networks." *Advances in Computer*, 17 (1978), Academic Press, ISBN 0-12-012117-4, pp. 163-221.
19. Leventis, S., S. Koubias, and G. Papadopoulos. "A Performance Study of a New Double-Loop Computer Network." Unpublished.
20. INTEL Peripheral Design Handbook, 1979.
21. Motorola SDLC chip.

ILLINET—A 32 Mbits/sec. local-area network*

by W.Y. CHENG, S. RAY, R. KOLSTAD, J. LUHUKAY, R. CAMPBELL, and J.W-S. LIU

University of Illinois at Urbana-Champaign
Urbana, Illinois

ABSTRACT

ILLINET is a fiber-optical ring network designed to provide wide band linkages between host computers for the purpose of facilitating file transfers at speeds near those of fast I/O devices in the hosts. Its structure is similar to the Distributed Computing System. ILLINET will eventually connect several PDP-11's, a PRIME computer and a network of microcomputers. These computers are used in a variety of real-time and batch processing applications. Currently they are already interconnected via 9600 baud lines in a star configuration to provide access to simple terminals. This paper describes the network architecture, control structure, and hardware configuration of ILLINET.

INTRODUCTION

The rapid development in VLSI technology has made host computers and terminals smaller and cheaper. In recent years it has become rather common for an organization to have several computing systems with substantial processing and memory capacity operated and maintained within the same building or in several closely located buildings. These computing systems may each serve a wide range of simple and intelligent terminals. The need to share data, programs, processing power, and I/O facilities invariably makes it necessary for the computers and terminals to be interconnected in the form of a local area network. Indeed, many local area networks have been designed and implemented. Among the well-known local area networks are Xerox ETHERNET¹, Bell SPIDER², and LSCNET³. These networks have been designed to provide low delay access via interactive terminals to host computers at relatively low cost per interconnection and with ease for network extension and reconfiguration. Since the effective link bandwidth in such a network is divided evenly among all terminals and hosts, it is often impossible to facilitate transfer of large files between host computers at high speeds required in many applications.

Many studies have shown that the performance of a local resource sharing network and distributed data base system

depends critically on the communication bandwidth between hosts.⁴ In particular an effective resource sharing environment can be achieved only when wide band data links between hosts are available to allow file transfers at speeds near those of fast I/O devices in the hosts. ILLINET is a local area network designed to accomplish this goal. Its structure is similar to the Distributed Computing System (DCS) at the University of California, Irvine.⁵ This paper describes ILLINET which has been designed and is currently being implemented in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

In section 2 the design objectives of ILLINET are discussed. These objectives impose several constraints on the network configuration and control structure. Section 3 gives an overview of ILLINET. In section 4 the link level data packet format is described together with the hardwired network access and link control protocols. Finally, the hardware architecture is presented in section 5.

DESIGN OBJECTIVES

ILLINET will eventually connect several PDP-11's, a PRIME computer, and a network of microcomputers. These computers are operated and maintained by the Department and are used in a variety of real-time and batch processing applications. Currently they are already interconnected via 9600 baud lines in a star configuration to provide access to 50-60 simple terminals. All of these computers are located within one building although ILLINET is designed to allow inter-building connections. It is envisioned that one of the nodes on ILLINET will be a PDP-11 which will serve as a gateway to the main campus computing facility.

The primary purpose for the design and implementation of ILLINET is to enhance existing computation facilities so that the resultant computer network will support effectively a variety of research activities in the areas of distributed operating systems, distributed data base systems and file servers. In order to assure that transmission links between the nodes and link-control level protocols will not be the bottleneck in inter-processor communication and data flow, it was decided that ILLINET is to be constructed using the latest cost effective technology. The transmission medium used in ILLINET is fiber optics because of its ability to support high bit rates and

* Supported in part by Grant NSF MCS79-06945.

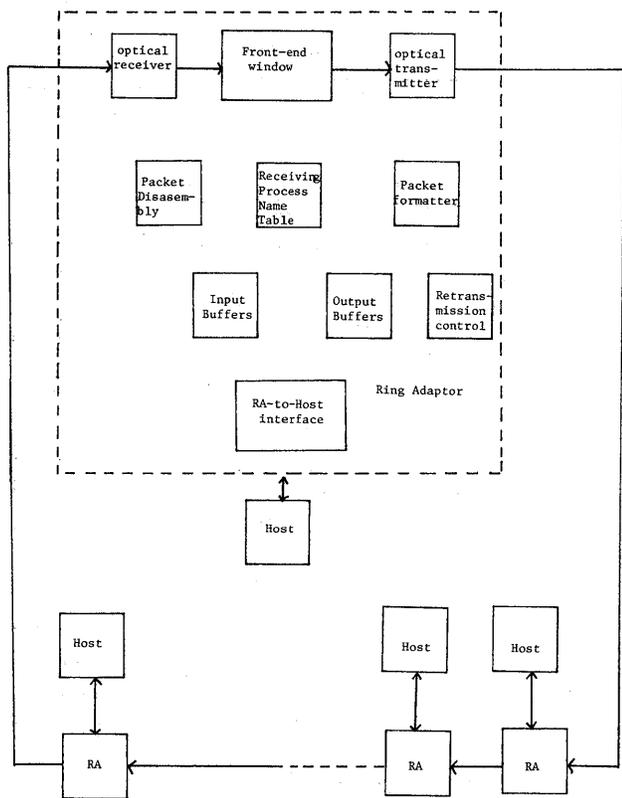


Figure 1—The configuration of ILLINET

allow reasonable interfaces. A link bandwidth of 32 Mbits per second is achieved with the use of ECL circuits. Since most of the network access and link control protocol functions are implemented in hardware, nearly all this bandwidth will be available for interprocessor communication.

The need to avoid the difficult task of providing bi-directional signal transmission and proper termination of the optical fibers dictated that ILLINET be a ring network. The packet switching discipline and distributed network control structure are used. Because of the high data link bandwidth and the relative short loop delay in ILLINET, it is not necessary that the most efficient network access control scheme be used. The version of token control scheme implemented in ILLINET is described in sections 3 and 5. It is similar to the scheme used in DCS. It will undoubtedly provide sufficiently low access delay and high network throughput.

In order to support high-level process communication in broadcast mode and to allow transparent transfer of destination process from one node to another, associative addressing is used in ILLINET. Address recognition hardware and link control protocols are both designed to support efficient broadcast communication in the network.

NETWORK OVERVIEW

The configuration of ILLINET is described in Figure 1. It contains no central controller or primary station to carry out clock synchronization and access control functions. In each of

the ring adaptors (RA) on the ring, there is a 16-bit active data path (hereafter referred to as front-end window) between the optical receiver and transmitter in the front end. More specifically, a RA functions as a repeater which retransmits the incoming data stream. The portion of the data stream appearing in the front-end window may be examined by the RA.

A host can gain access to the network via the ring adaptor attached to it. To each of the hosts on ILLINET, the network functions as a packet-switched network. To send a message, the host segments the message into network packets of a maximum size of 4K bits. Each packet is delivered individually to the RA where it is stored in one of the output buffers. The completion of the loading of the data packet into the output buffer is acknowledged by an interrupt sent by the RA to the host. The host in turn can signal the RA to commence accessing the network and transmitting the data packet. The transmission of the data packet is then carried out under the control of the RA without host intervention. Under normal operating conditions, the data packets will be delivered to the destination in the order in which they are sent from the host to the RA, and duplicate and lost packets will not occur. However, reliable sequenced delivery is not guaranteed. Mechanisms to assure reliable datagram delivery and message sequencing and reassembly are carried out by the hosts.

The RA monitors the data stream passing through its front-end window at all times. When there is a packet to be delivered, the RA removes the access control token (01111111) from the ring when the token appears at its front-end window. There is only one control token in the ring. When the RA receives the token, it is allowed to transmit one data packet. The format of the data packet is shown in Figure 2. Besides the receiving process name there are CRC check, duplicate mark and acknowledge/repeat request fields. The data packet is retransmitted until positive acknowledgments are received from all RA's serving active processes whose names match the receiving process name in the packet header. (We will return to discuss the acknowledgment and repeat request features in the next section.) The use of this stop-and-wait ARQ scheme simplifies the host-to-host synchronization. Since the bandwidth of the host-to-RA interface is significantly lower than the network bandwidth, the host-to-host throughput will not be limited by its use⁶. Furthermore, since there are two output buffers in the RA, network access for transmission from one buffer over the network can be carried out while the host loads the other output buffer. Thus, the speed of large file transfer between hosts will be limited primarily by the bandwidth of the host-to-RA interfaces.

In order to facilitate dynamic renaming and broadcast mode communication, a high-speed static RAM is provided in each RA to store the local active process name table as suggested in Mockopetris⁷. This table is updated by the host. When a

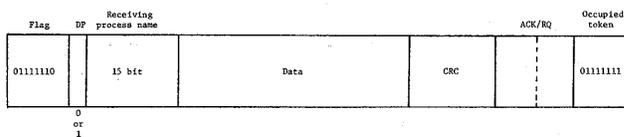


Figure 2—Link level data packet format

data packet passes the front-end window of a RA, the RA checks the receiving process name field to determine whether it matches any of the names in its process name table. The data packet is copied into one of the 16 input buffers as it is simultaneously subjected to CRC checks. The data packet is kept in the input buffer only when there is a process name match, the data packet is free of error, and there is an input buffer available for its storage. In this case, the RA interrupts the host to inform it of the reception of the data packet. As the data packet passes through its front-end window, the RA makes comment in the acknowledgment/repeat request field. Such comments serve as acknowledgments to the sending RA.

Within the ring only the data path between the optical receiver and the transmitter inside the sending RA is open. Hence, under normal operating conditions, the sending RA will remove the data packet when it returns to the optical receiver. By checking the contents of the acknowledgment/repeat request field in the returned data packet, the sending RA can decide immediately whether retransmission of the data packet is warranted. Either when the data packet transmission is completed successfully or is aborted after retransmission a maximum number of times, the sending RA releases the token and interrupts the host. By checking the status of the RA, the sending host can determine whether the transmission of the data packet is successful. If the other output buffer is nonempty and if the transmission of the previous data packet is successful, the host may signal the RA to commence network access again. On the other hand, if the delivery of the data packet fails, the host may ask the RA to attempt retransmission again or to invoke error diagnosis process. Thus, the sending RA is guaranteed the use of the data link for the delivery of both the data packet and the associated acknowledgment.

PACKET FORMAT AND LINK CONTROL PROTOCOLS

The link level data packet format is shown in Figure 2. The data field is sandwiched between the packet header and trailing control fields. The header consists of the flag, "0111110," marking the beginning of a data packet, duplicate mark (DP), and the receiving process name field. The sending process name, packet sequence number and higher-level control information are considered here as parts of the data field. The trailing control fields consist of the cyclic redundant check code (CRC), the acknowledgment/repeat request (ACK/RQ) field, and the occupied token "0111111",* marking the end of the data packet. The receiving process name and the data are supplied by the host. The other fields are generated by the sending RA.

We note that the data packet format is similar to that in HDLC. To achieve data transparency a zero is inserted following every occurrence of 5 contiguous 1's in the data stream between the flags and the occupied token as in HDLC. The

flag and the token are the only control fields containing more than 5 1's and hence can be uniquely identified at link level. Before the zero insertion the data field is $n \times 16$ bits long for some n between 0 and 255. The 16-bit CRC code specified by the generating polynomial $x^{16} + x^{12} + x^5 + 1$ is used for detecting errors in all bits between the flag and the ACK/RQ field.

A data packet is marked as a duplicate by the sending RA with its DP set to 1. A RA can check the first 16 bits (after zero deletion) following the flag to determine if the packet is intended for some local process and whether the data packet is a duplicate one. The last 8 bit field before the occupied token is the ACK/RQ field. When a data packet leaves the sending RA, its ACK/RQ field is reset to off to mean negative acknowledgment and no repeat request. As the data packet passes through its front end, each RA on the ring may acknowledge whether the data packet is properly received by marking its comment in the ACK/RQ field. A RA sets the ACK field if the receiving process name matches the name of a local process name and if the data packet is copied and stored in its input buffer ready to be delivered to the local process. The RQ field is set when there is a process name match. However, either due to error detected in the data packet or due to input buffer overflow, the data packet is not correctly copied into the input buffer. Thus, the RA may request the data packet be retransmitted.

The operation of the sending RA is described by the flow-chart in Figure 3. Before transmitting a data packet, the acknowledgment state of the sending RA and the number of retransmissions count are initially reset to zero. When the data packet is being transmitted for the first time, the duplicate mark is set to 0. As the data packet makes a round trip around the ring appropriate comments are collected in the ACK/RQ field from all RA's on the ring. By scanning the ACK field, the sending RA may determine whether the ACK field is set (meaning that some RA made a positive acknowledgment). If the ACK field is set, the acknowledgment state of the send RA is set to 1. The repeat request field is set if any RA made a repeat request. The sending RA will immediately retransmit the data packet in this case. However, this time the DP bit is set to 1 to mark the data packet as a duplicate. If, on the other hand, the RQ field is found to be off when the data packet returns to the optical receiver in the sending RA, the transmission is considered completed. We note that the acknowledgment and the repeat request fields in the returned data packet not set by any RA will be interpreted by the sending RA that the receiving process name does not match the name of any active processes on the ring. In this case, the sending RA immediately retransmits the data packet. Since this data packet has not been received by any RA, it is not marked as a duplicate.

The operation of a RA which is not transmitting is described by the state transition diagram in Figure 4. Such a RA is in one of two states, α or β . When the DP in a data packet is 0, any RA may copy the data packet and make comment in the ACK/RQ field. Once a RA receives a data packet and stores it in an input buffer, it writes a positive acknowledgment in the ACK/RQ field of the data packet and enters state β . As this data packet reaches the sending RA, its acknowledgment state will be set to 1. Hence, during sub-

* The bit pattern representing the occupied token is the same as that used to represent the control token. That a token is occupied (and, therefore, is not trapped by a RA which is waiting to obtain the token) is signified by this pattern following a matching flag.

i: transmission account
I: the number of maximum allowable transmission attempts
S: Acknowledgement State
DP: duplicate mark
ACK: Acknowledgement
RQ: Repeat Request

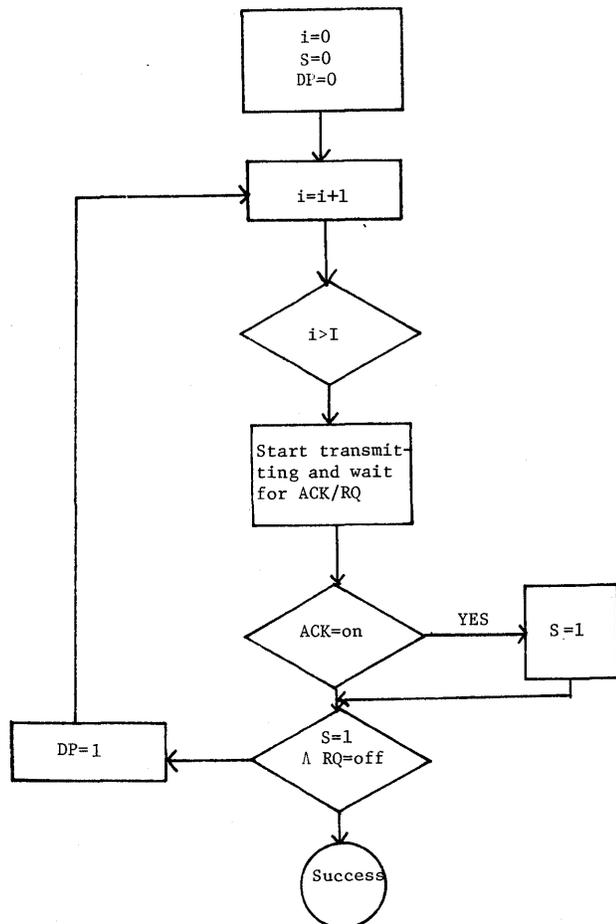


Figure 3—Operations of the sending ring adaptor

sequent retransmission of the data packet, the *DP* is set to 1. While it is in state β , the RA is inhibited to make the comment in any data packet marked as duplicate.

A RA enters state α when it makes a repeat request comment in the *RQ* field of the last data packet passing through its front-end window. When the duplicate mark in the data packet is set to 1, RA copies the data packet into its input buffer and makes positive acknowledgment in the *ACK/RQ* field only if it is in state α . Thus, reception of duplicate packets is prevented except in the relatively rare cases when noise causes messages to be garbled on more than one link in the network.

To summarize, a RA which is not transmitting will set the *ACK* field of the data packet passing through its front-end window and thus make a positive acknowledgment of its reception if (1) it is in α state, or it is in β state, but the *DP* of

the data packet is 0, (2) the receiving process name in the data packet matches some process name in the receiving RA, (3) there are free input buffers, and (4) the CRC check detected no error in the data packet. Similarly, it will make a repeat request if it is in state α , or it is in state β , the *DP* of data packet is 0, and one of the following conditions is true: (1) the CRC check found the data packet to be erroneous, or (2) there is not free input buffer and the receiving process name of the data packet matches with that of some local process.

We note that there is no need to initialize the RA to be in α or β state. Being self-synchronized, the RA should function correctly even if some RA's are in state α and some RA's are in state β at the time when the transmission of any data packet commences.

ERROR RECOVERY

In the two nodes that have been implemented to date, error recovery hardware is not included. Because of the limited knowledge in the failure characteristics of the type of networks such as ILLINET, it was decided to postpone the implementation of these hardwares. Instead, network error recovery functions are carried out by the hosts. However, time-out and interrupt circuits are included in each of the RA's for the detection of malfunctions in the RA or networks. For example, when a RA has a data packet to be sent but has waited for a long time for the control token, an interrupt is sent to the host when a preset time-out period expires to alert the host possible network malfunctions and invoke recovery procedure. Similarly, if after a RA caught the control token and transmitted a data packet but the occupied token at the end of the data packet does not return after a maximum loop delay or if the transmission lasted too long a period of time, appropriate interrupt signals are sent to the host in its diagnosis to pinpoint the cause of network malfunction. The input buffer and output buffer memory modules are completely independent. Therefore, it is possible to support echo transmission mode. In this mode, a sending RA stores the data packet transmitted from its own output buffer when the packet returns from the network. It is also possible for a host to separate the RA from the network. In this case, a data packet may be transmitted directly from the output buffer to the input buffer of the RA. Thus, individual RA's can be tested independently making isolation of malfunctioned RA a relatively easy task.

Hardware for recovery from error conditions involving the token is included in our design. Under normal operating conditions there is only one control token circulating around the ring. Failure or transient noises in both RA's or the link may cause the token to be lost or duplicated. We refer to these conditions as no token or duplicated token, respectively. Clearly, the no token condition exists when the network is turned on initially.

To explain how the no token or duplicated token conditions are to be handled in ILLINET, let us discuss the error conditions that can occur in ILLINET. As described above, all RA's monitor the data stream on the ring as it passes by their 16-bit front-end windows. Data streams arriving at the optical

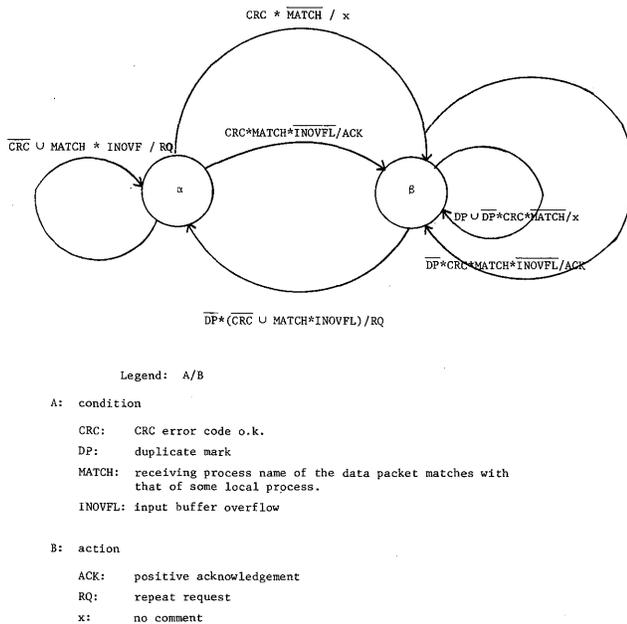


Figure 4—State diagram of a ring adaptor which is not transmitting

receiver in a RA is not relayed to the optical transmitter unless this data stream represents the access token or when it is preceded by a flag and the flag is detected by the RA. At the end of the data packet, the last remaining 16-bit of data in the front-end window are delivered to the optical transmitter for transmission only when the occupied token marking the end of the data packet is detected. Hence, any data stream with no leading flag and occupied token is blocked by the front-end of some RA. A data stream with a leading flag but no occupied token is truncated by 16-bits after passing through each RA until the data packet disappears. A data stream containing occupied token but no leading flag becomes a control token instead. Thus, the continuous circulation of random or broken data packets left on the ring due to failures in the sending RA or intermittent noise is prevented. "Garbage collection" in this case is not required.

In the sending RA, the data path between the optical receiver and transmitter is normally open during the transmission of a data packet. This data packet is removed from the ring when it returns. If for some reason the data path within the sending RA is closed when the data packet returns to the sending RA, it will be left circulating on the ring. We note that this error condition is a serious one. If the duplicate mark in the packet is not set and if the receiving process name matches the names of some active process on the ring, the input buffers in the RA serving these processes will eventually overflow since the data packet will be copied by these RA's each time it passes by their front end. In this case, a RA monitoring the network will see well-formatted data packets pass by even though the no token condition exists. Since the sending process name and packet sequence number are considered as parts of data and not monitored by the RA's, this type of no token condition can be detected either by the receiving hosts after the received data packets have been examined or by a

RA after waiting for some access token for a period of time longer than the maximum access delay on the ring. If in a N -node ring with loops delay L the maximum packet length is T seconds, and each RA is allowed to transmit k times before freeing the token, the maximum access delay is roughly $(N - 1)(k)(L + T)$. (For example, in a 6 node, 1 km ring network, the length of this period is approximately 10 msec. with $k = 16$.) Fortunately, we believe that this type of no-token condition rarely occurs in ILLINET. When it does occur, it is handled as follows: when a RA observes data stream but no control token passes by its front-end window for a period of time longer than its estimated maximum access delay, it opens the data path between the optical receiver and transmitter. Thus, it removes the "garbage" from the ring. However, the no token condition persists.

The no token condition can be detected easily in the case when there is no data stream circulating on the ring. In this case, a RA can decide that there is no token on the ring after one maximum loop delay. (In our previous example, this time is roughly $7 \mu\text{sec.}$) This type of no token condition is handled in the following manner. When a RA observes no data stream in the ring and there is no access token passing by its front end for a period longer than one maximum loop delay, it will enter a time-out period and continue to monitor the activities on the ring. If when its time-out expires and no token is observed, it will insert a token on the ring. By making the differences between the time-out periods of the different RA's equal to or longer than one loop delay, we are assured that once such a no token condition occurs, a token will be generated in a reasonably short time. Moreover, only one token will be generated in most cases.

The duplicate token detection scheme is designed for the general case when the exact loop delay is not known or may be variable. In this case, the duplicate tokens can be detected reliably at the host level. That there are more than one RA transmitting data packets at the same time can be detected by the sending host by examining the sending process name and packet sequence number in the data field of the packets arriving at the optical receiver of its serving RA. However, the need of the host intervention will undoubtedly significantly lower the network throughput. Alternately, we may require that the sending process name be placed in the first 16 bits of the data field. The sending RA can, therefore, determine whether the packet arriving at the optical receiver is the same one sent on the ring by itself. When the received data packet is found to be from another RA, the sending RA can conclude that duplicate token conditions exist. Again, by removing all data streams arriving at its receiver, a sending RA will delete all tokens from the ring.

HARDWARE STRUCTURE

The hardware structure of the RA's already implemented is described by the block diagram shown in Figure 5. To satisfy the different speed requirements of the different functional blocks of the RA at a minimum cost and complexity, it is implemented in ECL, STTL, and LSTTL. A RA consists of three PC boards, the front-end, memory module and re-transmission control logic, and RA-to-host interface.

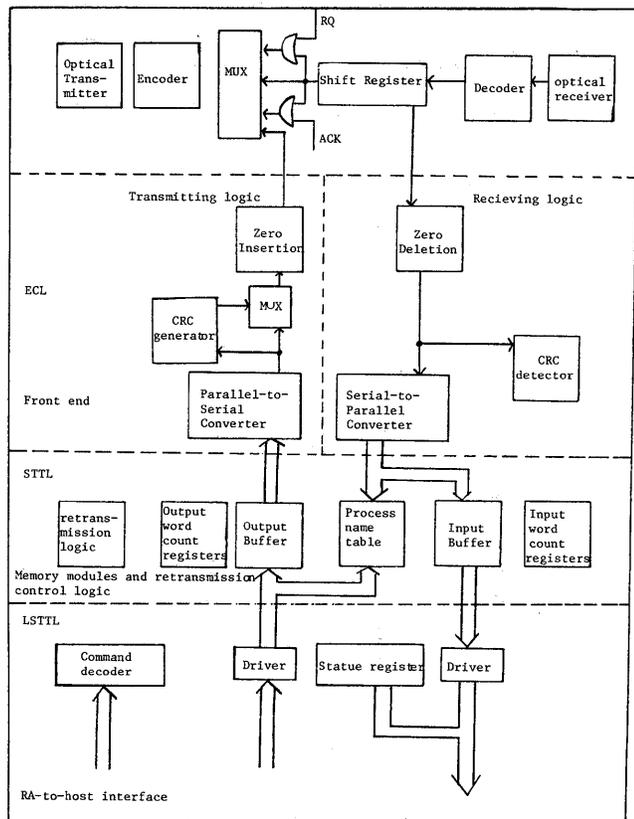


Figure 5—Ring adaptor hardware structure

The front end contains the transmitting and receiving logics. Between the optical receiver and transmitter, there is a shift register which serves as a delay buffer. The RA may hold up the incoming data stream, scan and process the contents of the various fields as they appear in the shift register. Here, appropriate comment is generated and inserted in the ACK/RQ field of the data packet. Then the last 16 bits containing the ACK/RQ field and the token are shifted out to the transmitter. The major functions of the transmitting logic are parallel-to-serial data conversion, zero insertion CRC error code generation, and data packet formatting. The major functions of the receiving logic are zero deletion, serial-to-parallel conversion and CRC check. All these operations are carried out bit-serially and are implemented in ECL logic.

Within the memory modules and retransmission logic, there are input and output buffers, process name table, and retransmission control circuits. The buffer memory are segmented into 256 16-bit word pages. Sixteen pages are used as input buffers and two pages are used as output buffers. The input and output buffers are organized as independent modules, each is capable of supporting either read or write operation at 32 Mbits/sec. Upon detection of the flag, the input buffer write operation is initiated. If at the time data is being

transferred from the input buffer to the host, this transfer operation is halted temporarily. The input buffer write operation will be terminated when the occupied token marking the end of the data packet is detected, when the receiving process name in the data packet does not match any names in the process name table, or when the duplicate mark is found to be set indicating that the data packet is already copied by the RA. Any temporarily halted memory transfer operation will then be resumed.

There are two output buffers in the output modules to allow the process of waiting for network access and data transfer from the host to be carried out concurrently. A 32 Kx1 memory module implemented with Intel 2147H3 is used to store receiving process names. (Currently, we are using only one chip containing 4 K in each RA.) The 15-bit receiving process name is used to address this RAM table. An output bit from the table being 1 indicate a match of the receiving process name with some local process name in the table. Thus, the process of checking receiving process name match can be carried out in 55 nsec. The table can be dynamically updated by the host within 500 nsec. All buffer memory operations, receiving process name checking and updating, and retransmission control are carried out at word level and are implemented in STTL logic.

Finally, the RA-to-host interface contains the command decoder, RA status registers and interfaces to and from buffer memory modules. These circuits allow the RA to appear to the host as a peripheral device and can be easily linked to the host via a DMA interface. This portion of the RA is implemented in the TTL logic.

ACKNOWLEDGMENT

The authors wish to thank Cyrus Weise, Kurt Horton, and Izumi Suwa for suggestions and help in the design and implementation of ILLINET.

REFERENCES

1. Metcalfe, R.M. and D.R. Boggs, Ethernet: distributed packet switching for local computer network, *CACM* 19, 7, July 1976, 395-404.
2. Frazer, A.G., Spider—an experimental data communication system, Proceedings International Communications Conference, 21F-1-10, *CACM*.
3. Pogran, K.T. and D.P. Reed, the MIT laboratory for computer science network, Local Area Networking NBS Special Publication 500-31, April 1978, 22-23.
4. Weber, H., D. Baum and R. Popescu-Zellin, ESA—an evolutionary system architecture for a distributed data base management system, Proceedings of Berkeley Conference on Distributed Processing, 1979.
5. Farber, D.J., A ring network, *Datamation*, February 1975, 44-46.
6. Burton, H.O. and D.O. Sullivan, Error and error control, *Proceedings of the IEEE* 60, 1972.
7. Mockopetris, P., Design consideration and implementation of ARPA LNI nametable, University of California, Dept. of Information and Computer Science, Technical Report 92, Irvine, CA, April 1978.

SOFTWARE

A survey of currently implemented Pascal extensions*

by T. N. TURBA and S. H. COSTELLO

Sperry Univac
St. Paul., Minnesota

ABSTRACT

This document presents an overview of the more common and useful Pascal extensions that have been implemented to date. It describes, in general terms, the functionality introduced by these extensions and provides a rationale for the implementation of many of them. Examples are provided to give the reader a better idea of how each extension was implemented without going into great detail or formality in describing the syntax and semantics used by each implementation.

INTRODUCTION

Extensions to a language are made primarily for ease of use and increased functionality. This document contains a list of selected extensions to the Pascal language that fall into these categories and are currently implemented. Extensions that are machine-dependent or have highly specialized uses have been largely excluded, as have features such as compiler di-

* This survey reflects information available at time of publication. No claim is made that it is exhaustive, is complete, or does not contain information that may be out of date. This survey will be revised and republished at a later date. Therefore, readers who have updated or additional information should contact the authors.

rectives and debugging aids. An extension was considered to be any feature not found in Draft 6 of the proposed ISO Pascal standard. Table I describes the Pascal implementations that were studied in the preparation of this report.

PASCAL ENTITIES AND DECLARATIONS

Identifiers

Identifiers may be extended by adding characters to the set of letters and digits that are found in an identifier. The underscore character (“_”) is commonly allowed in identifiers because it enhances the readability of identifiers by separating descriptive parts of a name. For example, *next_node_pointer* is more readable than *nextnodepointer*. The dollar sign (“\$”) is also allowed by several implementations for similar purposes.

Comments

Some Pascal implementations allow imbedded comments, so that a comment can be contained within a comment. This

TABLE I—Pascal implementations included in this survey

Implementation	Version	Date	Document number	Systems available on	Implementors
Pascal/VS TI Pascal UCSD Pascal	II.0	6/80 2/80	SH20-6168-0 946290-9701	IBM 370 TI 990 most 8 and 16 bit microcomputers	IBM Texas Instruments UCSD and Softech Microsystems
Pascal 6000	release 3	1/79		CDC 6000, Cyber 70 & 170 series Univac series 1100	ETH Zurich and University of Minnesota Naval Ocean Systems Center, San Diego
NOSC Pascal	1.2 × 1.8			Univac Series 1100	University of Wisconsin, Madison
UW-Pascal		4/79		Univac Series 1100	

permits the user to comment out a whole section of code that may itself contain comments. UCSD Pascal implements this feature by recognizing "{...}" and "(*...*)" as comments, but not "{...}" or "(*...)". Thus "(*...{...}...*)" becomes a single legal comment. Pascal/VS accepts "(*...)" and "{...}", as the standard mandates, but introduces "/*...*/" as a distinct form in order to allow imbedding of comments.

Comments that start with a dollar sign ("\$\$") are often taken to be compiler directives. An exception is Pascal/VS, which does not recognize compiler directives within comments, but requires that they be preceded with a percent sign ("%") and that they *not* appear in a comment.

Numbers

Many Pascal implementations provide a method for using numbers in a base other than ten, usually binary, octal, or hexadecimal. For example, the hexadecimal integer F65A would be expressed as #F65A in TI Pascal, and as 'F65A'X in Pascal/VS. Pascal/VS also employs the suffixes XC and XR to permit hexadecimal constants that can be used as strings or real numbers, respectively.

Order of declarations

The required order of declarations (labels, constants, types, variables, and procedures) has been relaxed in several implementations. This is usually done to insure the independence of separate source files that may be inserted into the compilation via a COPY or INCLUDE compiler directive. Rules for declaration ordering are also relaxed to allow TYPE declarations to occur before CONST declarations in order to allow a constant of a structured type.

Labels

Labels have been commonly extended in two ways. One extension is to allow a label to fall outside the standard range of 0..9999. This is often done by letting a label have any positive integer value that can be represented on the machine being used. Another extension is to permit a label to be an identifier rather than just a number, making programs with labels more readable.

Constants

Some Pascal implementations permit an expression to be used in a CONST declaration. This lets the user construct sequences of related constants. For example:

```
CONST radius = 39.75;
      pi = 3.14159;
      circumference = 2 * pi * radius;
      area = pi * radius * radius;
```

It also allows the user to create constants that could not otherwise be created (e.g., CONST bell = CHR(7)). This can be extended even further to permit constant expressions anywhere a constant is required (e.g., TYPE wobbles = ARRAY [1..2*maxlim] OF REAL).

A constant declaration can also be enhanced by adding the capability to define a constant of a structured type. This can only be done, as indicated earlier, if the order of declarations is relaxed, so that the TYPE declaration can occur before the CONST declaration. For example:

```
TYPE complex = RECORD
                    re, im: REAL
                END;
CONST threefour = complex(3.0, 4.0);
```

Types

String type

Some implementations offer an explicit predeclared string type that permits strings which can vary in length at execution time within a maximum string length set at compile time. Two examples of string declaration in UCSD Pascal are as follows:

```
VAR title: STRING; {max length of title is default—80
                   characters in UCSD Pascal}
name: STRING[20]; {maximum length of name is 20
                  characters. NOTE: in Pascal/VS,
                  STRING [20] is expressed as
                  STRING (20)}
```

Strings are implemented by associating with each string a hidden length field that keeps track of the string's length during execution. This length field is updated every time the string is assigned a value.

Variable length strings are very versatile, because strings of different lengths are totally compatible with each other. In addition, single characters within a string can be individually referenced (e.g., title [5] := "a"). When a STRING type is provided, the user is normally supplied with a good selection of built-in procedures and functions to return the length of a string, extract substrings from a string, match a pattern of characters within a string, etc. (See Table III on built-in string manipulation routines.)

Record type

The syntax for declaring record types can be modified to give the programmer more control over the way that storage is allocated and overlaid. For example, Pascal/VS permits the user to omit field names in order to leave blank fields for padding. It also provides a means for placing the tag field anywhere in the fixed part of a record instead of only at the end. In addition, it allows the user to specify a byte offset, in parentheses, after a field name to force the Pascal compiler to alter the way it aligns storage. Since these kinds of extensions are very machine-dependent, this report will not detail them further.

File types

Pascal 6000 has a SEGMENTED file type (e.g., VAR f: SEGMENTED FILE OF t1) that permits files to be segmented into logical units. An end-of-segment marker can be placed on a file with the predefined procedure PUTSEG(f). The procedure GETSEG(f,n) moves the file pointer ahead n segments. REWRITE(f,n) moves the file pointer ahead in the same fashion to prepare the file for writing.

Using similar syntax, TI Pascal has RANDOM files (e.g., VAR f: RANDOM FILE OF t1). As the name implies, random files permit the components of the file to be accessed randomly by using an index number. For example, READ(f,5,q) is used to read the fifth item in file f into variable q .

UCSD Pascal has untyped files for writing or reading whole blocks of data to or from a disc file. Untyped files are declared by simply leaving the type specification out (e.g., VAR f: FILE). An untyped file has no window variable and can only be used with the procedures BLOCKWRITE and BLOCKREAD.

UCSD Pascal also has an interactive file type, which is a text file with special properties for interactive I/O. (See section on interactive I/O.)

Fixed and decimal types

TI Pascal provides a FIXED predefined type which is a scaled binary number with its precision expressed in terms of p and q , where p is the total number of binary digits and q is the binary scale factor. It also provides a DECIMAL predefined type for applications requiring that operations be done using decimal arithmetic.

Extended precision numeric types

Several Pascal implementations have integer and real types that may optionally have greater precision. This is especially important on 16-bit machines, where it is often necessary to escape from the 16-bit integer range of $-32,768$ to $32,767$. TI Pascal supplies a LONGINT type that may contain values over 2 billion. UCSD Pascal allows the user to specify the number of digits required when declaring an integer variable (e.g., VAR x: INTEGER [8]). In a similar fashion, TI Pascal also lets the user specify the number of significant digits required when declaring a real variable (e.g., VAR x: REAL(12)).

Variables

External variables

An external variable is one that is common to two or more separately compiled modules. This feature is implemented in a variety of ways. Pascal/VS externalizes variables with a DEF declaration that employs the same syntax as the VAR declaration. These external variables may be accessed in another

module with a REF declaration that follows the same syntax. TI Pascal has a COMMON declaration that places variables in a common block that is compatible with FORTRAN's named common. An ACCESS declaration is used to specify that a routine may use a given variable in the common block.

Another way of making variables external is by placing them in an externally compiled unit that is referenced as a whole, so that each variable in that unit can be used without being declared. (See section on compilation units.)

Static variables

Static variables are not temporarily allocated space on a stack for each call to a procedure or function, but rather always occupy the same memory location during the execution of the program. Static variables are used to provide a routine with variables whose values are not lost when a return is made from the routine (i.e., their old values are still around when the routine is called again). They can, therefore, provide variables that are local to a routine but are global to all calls of that routine, including recursive calls.

In some Pascal implementations, static variables can be declared with a STATIC declaration using the same syntax as the VAR declaration. In other Pascal implementations, variables are made static only by being global. In TI Pascal, static variables are created by placing them in a COMMON declaration.

Compile-time initialization of variables

Needless use of execution time and program storage for initializing variables with assignment statements can be eliminated by a Pascal extension that allows static variables to be initialized during compilation. Several Pascal implementations provide this feature with a VALUE declaration. An example in Pascal 6000 is

```
TYPE    ducks = (mallard, bluebill, teal, woodduck);
        flock = ARRAY [1..5] OF ducks;
VAR    groucho,harpo: ducks;
        zeppo,chico: flock;
VALUE  groucho = woodduck;
        harpo = teal;
        zeppo = flock
        (teal, teal, mallard, woodduck, bluebill);
        chico = flock(5 OF mallard);
```

The VALUE declarations for *zeppo* and *chico* show how structured variables can be initialized. Pascal/VS is similar, but would require the use of “:=” instead of “=,” “mallard:5” instead of “5 OF mallard,” and would require that the variables be declared in a STATIC or DEF declaration.

Functions and procedures

Passing parameters

Pascal/VS offers a method to pass parameters by read-only reference. That is, a variable or expression may be passed to

a routine, but will be treated as if it were a constant within the called routine. Passing a parameter by read-only reference is implemented by using a CONST declaration in the formal parameter list of a routine with the same syntax of a VAR declaration (e.g., PROCEDURE myproc(CONST s: INTEGER, ...)).

An important extension to parameter passing is the ability to pass arrays with dynamic dimensions to a routine, so that the routine can handle arrays of different sizes without recoding the routine. There are several ways of implementing this feature besides the controversial conformant array schema. For example, in TI Pascal, the upper bounds of a dimension can be replaced with a question mark in the formal parameter list (e.g., FUNCTION max(vector: ARRAY [1..?] OF INTEGER): INTEGER). A built-in function (called UB) is also provided to obtain the actual upper bounds of a dimension during execution. Pascal 6000 provides a means for declaring a dynamic array type (e.g., TYPE data = ARRAY [INTEGER] OF REAL). Any array that has the same element type and number of dimensions and an index of the same base type of a dynamic type may be passed to a formal parameter list that specifies that dynamic type (e.g., VAR ages: ARRAY [1..200] OF REAL can be passed as the type *data* of the last example). This can only be done if the type in the formal parameter list is preceded by the word DYNAMIC (e.g., FUNCTION mean (VAR a: DYNAMIC data)). The built-in functions LOW and HIGH are provided to retrieve the upper and lower bounds of a dynamic array at runtime.

Accessing routines written in other languages

Many Pascal implementations provide a method for calling routines written in other languages. FORTRAN is the most common language to be made accessible from Pascal. This makes the entire FORTRAN library available to Pascal users. A FORTRAN routine is usually accessed by declaring a routine with the directive FORTRAN replacing the procedure or function body. This informs the Pascal compiler to look for the routine elsewhere and to use FORTRAN conventions when calling it. (Note: There are still some differences between FORTRAN and Pascal of which the user must be aware. For example, FORTRAN stores its arrays in column-major order, so that a [i,j] in Pascal is a (j,i) in FORTRAN.)

External routines

An external routine is a procedure or function that can be called from a separate Pascal compilation. One way this can be done is by allowing routines to be declared as being EXTERNAL or ENTRY. An ENTRY routine is one that is marked to be exported for use by other compilation units. For example, in NOSC Pascal such a routine could look like "PROCEDURE ENTRY myproc(...)" followed by the procedure body. This routine could then be imported by another compilation unit by declaring the procedure with the directive EXTERNAL replacing the procedure body in the same way that the directive FORWARD is used (e.g., PROCEDURE myproc(...); EXTERNAL). In Pascal/VS, the ENTRY direc-

tive is placed immediately after the routine heading, rather than before the routine name. Pascal 6000 defines external routines using the directive EXTERN; however, entry routines are defined by a compiler directive in a comment within the routine heading.

Another way of making routines external is by placing them in an externally compiled unit that is referenced as a whole, so that each routine in that unit can be called without being declared. (See section on compilation units.)

Compilation units

A compilation unit is a module of code that is compiled at one time. The only standard compilation unit is the program. However, in order to access routines and variables outside the main program, new types of compilation units have been introduced. For instance, Pascal/VS has external compilation units called segments. A segment module looks like a program, except that the word SEGMENT is used in place of PROGRAM and there can be no code outside procedures and functions. Variables may be exported from a segment to a program or another segment via a DEF declaration (see section on external variables), and routines may be exported by declaring them as ENTRY routines (see section on external routines). Several Pascal implementations accomplish the same effect by using dummy programs that contain only declarations and routines.

UCSD Pascal permits access to external compilation units by means of a USES declaration. When a program declares that it USES a unit, all of the declarations in the accessible (INTERFACE) part of the unit can be referenced as external global constants, types, variables, procedures, and functions without being declared. External compilation units in UCSD Pascal have two parts: an INTERFACE part and an IMPLEMENTATION part. The INTERFACE part of the unit contains declarations and routine headings that may be directly accessed by programs that use that unit. The IMPLEMENTATION part contains declarations and routine bodies that are private to the unit and are thus hidden from the program that USES the unit. This means that packages can be created that the user can access with a USES declaration without having to bother with complex declarations or implementation details. For example, a user could access an entire package of graphics routines by merely including a "USES graphicspack" declaration in a program. An example of the basic layout of a UCSD Pascal UNIT is

```
UNIT unitname;
INTERFACE
  {declarations and routine headings}
IMPLEMENTATION
  {declarations and routine bodies}
END;
```

UW-Pascal has a somewhat similar method of accessing external compilations by having units that represent environments in which programs can be compiled. The interface part of the environment is known as the ENVIRONMENT DECLARATIONS. This part of the environment is compiled by itself and then must be listed on the compiler call line when a program is compiled within that environment. The implementation

part of the environment is called the ENVIRONMENT ROUTINES and is compiled separately and then linked to the program by the system linker or collector.

EXPRESSIONS

Alternate symbols

Some Pascal implementations use special characters as alternate symbols for representing operators. For example, Pascal/VS permits “|” for OR, “→” for NOT, “&” for AND, and “→ =” for “< >”.

Boolean expressions

Exclusive or operator (XOR)

The exclusive or operator, XOR, is available in Pascal/VS. This operator simplifies some expressions since (a XOR b) is equivalent to ((a AND NOT b) OR (NOT a AND b)).

Partial evaluation of a Boolean expression

In many instances, it is not necessary to evaluate all of a Boolean expression. For example, in the expression (a OR b), b does not have to be evaluated if a is TRUE. This permits expressions such as (i > = lowerbounds AND i < = upperbounds AND a(i) IN setostuff) to be evaluated without error if the index of i is outside the range *lowerbounds* ..*upperbounds*.

Equality tests on structured variables

In UCSD Pascal, the relational operators “=” and “< >” have been extended to perform comparisons on variables of type ARRAY or type RECORD. This feature eliminates the need to compare such structures on an item-by-item basis.

Logical expressions

Pascal/VS permits bit-by-bit logical operations to be performed on integers. The following operators may be used in logical expressions:

- AND (“&”) —logical and (e.g., 3 & 5 = 1)
- OR (“|”) —logical or (e.g., 2 | 4 | 8 = 14)
- XOR (“&&”) —logical exclusive or (e.g., 3 && 5 = 6)
- NOT (“→”) —logical ones complement (e.g., →0 = -1)
- “< <” —logical shift left (e.g., 4 < < 2 = 16)

- “> >” —logical shift right (e.g., 8 > > 1 = 4)

String expressions

Pascal implementations that have a STRING type usually provide a method for concatenating strings. For example, Pascal/VS uses “||” as a string concatenation operator. Other Pascal implementations provide a function such as CONCAT to do this. (See Table III on built-in string routines.)

Type override operator

TI Pascal uses “::” as an explicit type override operator. It is used between a variable and a type identifier to form an expression of that type (e.g., varname :: INTEGER). The “::” operator does not perform a conversion; it merely allows the machine representation of the variable to be taken as a value of the type specified. This feature is used to overlay different type templates on a variable in order to use it in different ways at different times.

Use of type identifiers as functions

Pascal/VS permits scalar type identifiers to be used as functions for conversion within expressions. For example, *color*(1) will return *green* if *color* was defined as a type equal to (*red, green, blue*). This provides an inverse to the ORD function for all scalar types. (Standard Pascal provides such an inverse only for type CHAR.)

STATEMENTS

The ASSERT statement

The ASSERT statement is used to generate a runtime error if a certain condition does not hold true. For example, ASSERT a = b will generate a runtime error if a does not equal b when the statement is executed. The ASSERT statement is used to insure that if a program does not meet critical assertions it will not be executed to completion.

The assignment statement

UW-Pascal provides an alternate form of the assignment statement for cases in which a variable occurs on both sides of the “:=”. For instance, *a := a + 1* can be expressed as *a + := 1*. This can be done with the operators “+”, “-”, “*”, “/”, DIV, MOD, AND, and OR. The implementors of UW-Pascal claim that this improves both the readability and the efficiency of a program.

Pascal/VS allows structured constants (see section on constants) to be assigned to variables of structured types.

The CASE statement

Most Pascal implementations provide an OTHERWISE clause for the CASE statement in order to catch values that do not have a corresponding case constant. The OTHERWISE clause appears at the end of the CASE statement. For example:

```
CASE year OF
  jan,sep: x := 5;
  mar,jun: x := 6
  OTHERWISE x := 0
END;
```

Several Pascal implementations also extend the CASE statement to allow a range in a case label. Using this feature, the case label *jan, feb, mar, apr, may*: could be expressed as *jan..may*:

The FOR statement

TI Pascal extends the FOR statement to provide a means of traversing a set. This is done by using the IN operator in a new way. For example, FOR *j* IN *setostuff* DO... increments *j* over the values that exist in the set *setostuff*. This is equivalent to FOR *j* := *firststuff* TO *laststuff* DO IF *j* IN *setostuff* THEN... .

The WITH statement

TI Pascal allows synonyms to be defined in the WITH statement. For example, within the statement WITH *x* = *a*[*j*], *a*[*k*] DO..., the synonym *x* can be used to denote *a*[*j*]. Such synonyms have a scope that is local to the WITH statement.

Loop control statements

Several Pascal implementations have introduced alternate ways of exiting and iterating FOR, WHILE, and REPEAT loops. This is done to eliminate some of the need for using GOTO statements.

A loop can be exited by jumping to the point immediately following the end of the loop. Pascal/VS has a LEAVE statement that accomplishes this by exiting the innermost loop. UW-Pascal has a statement of the form EXIT *label* that exits the loop (or any other structured statement) that is labeled with the *label* specified (e.g., 340: FOR *j* := 1 TO 5 DO IF *a* < 5 THEN EXIT 340 ELSE...). TI Pascal has a similar statement of the form ESCAPE *label*. This works in the same fashion, except that the *label* is an identifier that is not declared in a LABEL declaration, but is implicitly declared by its use.

A loop can be iterated by jumping to the point immediately preceding the end of the loop. Pascal/VS has a CONTINUE statement that iterates the innermost loop in this fashion. UW-Pascal has a CONTINUE *label* statement that iterates the loop labeled with *label*.

Statements that exit a procedure or function

Several Pascal implementations have supplied a statement that exits from a procedure or function. This was done to offer an alternative, or replacement, for the method of exiting a routine with a GOTO statement. Pascal/VS has a RETURN statement that causes an exit from a routine when executed. UW-Pascal allows an expression to follow the word RETURN in order to return a function value from a function. UCSD Pascal offers an EXIT (*q*) statement where *q* is the name of the routine to be exited. This statement causes the routine call stack to be reset to the point where the last call of *q* was made. For example, if procedure *q* calls procedure *p*, and an EXIT(*q*) statement is encountered within *p*, then control is transferred to the point immediately following the call of *q*. Thus, EXIT(*q*) has the effect of terminating the execution of *q* and all subsequent routines called since the call of *q*.

I/O

Interactive I/O

Interactive I/O is a very important extension to the Pascal language since communication between a Pascal program and an interactive terminal would be very cumbersome without it. This is because Pascal I/O is defined in such a way that the buffer variable of a file is defined to have a value when the file is reset. Since the predefined file INPUT will normally be reset when execution begins, input will be requested before the program has a chance to print a prompting message.

This problem can be dealt with in several ways. One way is to not have the buffer variable set when a file is reset. This, however, forces the program to perform a GET or a READLN with no parameters before accessing the buffer variable. Another method, known as "Lazy I/O," delays the GET operation until the buffer variable is actually accessed. This method requires that the runtime system keep track of whether the buffer variable is current; if the variable is not current when it is accessed, the delayed GET operation must then be performed. Yet another method sets EOLN to TRUE when a file is reset, which has the apparent effect of inserting an empty line of input before the first line of the file. With this method, the end-of-line marker becomes a line separator rather than a line terminator.

Regardless of the method chosen, it must be decided which file or files the method should be applied to—all text files, only the INPUT file, or only files that the user specifies. Some of the ways this is done are as follows: Pascal 6000 requires that files to be used interactively must be followed by a slash ("/") when declared in the program heading (e.g., PROGRAM *p*(INPUT/,...)). Pascal/VS has a built-in procedure INTERACTIVE that must be used in place of RESET for files that are to be interactive. UCSD Pascal has an INTERACTIVE type that is like the type TEXT, but is acted on differently by the standard I/O procedures. INPUT and OUTPUT are predefined to be of type INTERACTIVE in UCSD Pascal.

Input extensions

TI Pascal allows the READ and READLN procedures to contain a field width specifier for formatted input. For example, READ(f,a:w) reads *w* columns and gets a value for *a* from those columns.

Several Pascal implementations permit strings to be input directly. Characters are read and inserted into the string until the string is full, or until the end of the line is reached.

Output extensions

In some implementations, the procedures WRITE and WRITELN have been extended to allow the output of hexadecimal and octal numbers. For example, WRITE(f,a:5 HEX) prints the value of *a* in hexadecimal form within a field width of 5. Likewise, OCT is used to print numbers in octal form.

Pascal/VS permits output items to be left-justified within a field by letting the field width be a negative number. For example, WRITE(f,b:-10) prints the value of *b* left-justified in a 10-character field.

UW-Pascal permits the values of user-defined enumerated types to be output.

BUILT-IN PROCEDURES AND FUNCTIONS

Pascal implementations differ widely on the availability of extra intrinsic routines, the names of these routines, and their calling parameters. A built-in routine that performs a given operation may be a procedure in one implementation and a function in another. Likewise, it may have different names and different parameters. In line with the purpose of this survey, Tables II through VII provide only a brief description of some of the various routines and do not attempt to describe their exact calling parameters or other implementation details.

Table II—Mathematical routines

Routine type	Common names	Description
REAL function	LOG	Returns the base 10 logarithm of a number.
INTEGER function	PWROFTEN	Returns the value of 10 raised to a given power.
INTEGER function	EXPO	Returns the exponent part of a real number.
Scalar function	MIN	Returns smallest value of any number of parameters of the same scalar type.
Scalar function	MAX	Returns largest value of any number of parameters of the same scalar type.
REAL function	RANDOM	Returns a random number in the range (0.0, 1.0).

Table III—String manipulation routines

Routine type	Common names	Description
INTEGER function	LENGTH	Returns the length of a string.
INTEGER function	POS, INDEX	Matches a given pattern within a string and returns the position in the string for the beginning of the matched substring.
STRING function	CONCAT	Concatenates a series of strings.
Procedure or STRING function	COPY, SUBSTR	Extracts a substring of a given length, starting at a given position in a string.
Procedure or STRING function	DELETE	Deletes a substring of a given length, starting at a given position in a string.
STRING function	TRIM	Removes trailing blanks from a string.
STRING function	LTRIM	Removes leading blanks from a string.
Procedure	TOKEN	Breaks a string into tokens consisting of identifiers, operators, or special symbols.
Procedure	DECODE	Reads from a string as if it were a TEXT file.
Procedure	ENCODE	Writes to a string as if it were a TEXT file.

TABLE IV—Storage manipulation routines

Routine type	Common names	Description
Procedure	MARK	Sets a pointer to the location of the start of memory available for allocation via the NEW procedure.
Procedure	RELEASE	Releases all memory allocated past a given location. Often used in conjunction with MARK.
INTEGER function	LOCATION, ADDRESS	Returns the machine address of a given variable or the entry point of a given routine.
INTEGER function	SIZE, SIZEOF	Returns the amount of storage (usually in bytes) required by a given type or variable.
INTEGER function	MEMAVAIL	Returns amount of memory currently available.

TABLE VI—I/O Routines

Routine Type	Common names	Description
INTEGER function	COLUMN, COLS	Returns position in the output line of the next character to be written to a given TEXT file.
Procedure	OPEN	Opens a file.
Procedure	CLOSE	Closes a file.
Procedure	EXTEND	Opens a file for output and places the file pointer to the end of the file.
Procedure	WRITEEOF	Writes an EOF marker on a TEXT file.
Procedure	SEEK	Moves the file pointer to a given record.
INTEGER function	STATUS	Returns the status of the last I/O.
Procedure	SET PROMPT	Prints a prompt for input and inhibits the carriage return and line feed.
Procedure	BLOCKREAD	Reads from a UCSD Pascal untyped file.
Procedure	BLOCKWRITE	Writes to a UCSD Pascal untyped file.
Procedure	PUTSEG	Writes an end-of-segment marker on a segmented file.
Procedure	GETSEG	Moves the file pointer forward a given number of segments.
BOOLEAN function	EOS	Returns TRUE if file pointer is at an end-of-segment marker.

TABLE V—Conversion routines

Routine type	Common names	Description
REAL function	FLOAT	Converts an integer into a real number.
STRING function	STR	Converts a number into a string, or may convert a character or a PACKED ARRAY OF CHAR into a string.
DECIMAL function	DEC	Converts a number into a number of type DECIMAL in TI Pascal.
FIXED function	FIX	Converts a number into a number of type FIXED in TI Pascal.
LONGINT function	LINT	An INT function for long integers in TI Pascal.
LONGINT function	LROUND	A ROUND function for long integers in TI Pascal.
LONGINT function	LTRUNC	A TRUNC function for long integers in TI Pascal.

TABLE VII—Miscellaneous routines

Routine type	Common names	Description
Procedure or function	DATE, TIME, DATETIME	Returns the current date and/or time.
Procedure or INTEGER function	CLOCK, TIME	Returns time in microseconds since start of run.
Procedure	HALT, STOP, ABORT	Stops program execution.
INTEGER function	CARD	Returns the cardinality of a set (number of elements).
Procedure	GOTOXY	Moves cursor to a given location on the display screen.
Scalar function	UB, HIGH, HBOUND	Returns the upper bounds of a given dimension of an array type or variable.
Scalar function	LOW, LBOUND	Returns the lower bounds of a given dimension of an array type or variable.
Scalar function	HIGHEST	Returns the highest possible value of a scalar type or variable.
Scalar function	LOWEST	Returns the lowest possible value of a scalar type or variable.
BOOLEAN function	UNDEFINED	Returns TRUE if a given variable is undefined.

A standard tool for information resource management

by MICHAEL E. MEYER
Honeywell Information Systems
McLean, Virginia

ABSTRACT

Information resource management has emerged from the 1970's as the term that accurately describes the methodology used for managing an enterprise's information resource. The term itself indicates that we have raised our level of comprehension from only data in the computer environment to the total information resource, which includes computer and non-computer entities consisting of data, the processes that use it, and their relationships. It is a synthesis of previous terms like database management, data management, etc., which have roots far back into the origins of electronic data processing. The origin, evolution, and growth of this methodology are traced for the purpose of establishing the need for standard tools in this age of multivendor enterprises. With the expansion of data independence to include nondatabase files, where programmer productivity is becoming a real issue and the approach of distributed systems and databases is imminent, a standard information resource management tool is an absolute requirement. Current efforts to develop an information resource dictionary system (IRDS) standard will be discussed with some ideas on the directions being taken.

INTRODUCTION

In the last few years, we have seen a geometric growth in the tools available for use in the data processing industry, from sophisticated hardware using chip technology, to software providing every possible functional capability. We have witnessed the emergence of database technology, which has focused increasing significance on data as an independent resource. We hear more talk every day of extending the independence of data to all file structures. These factors, coupled with increased concern over programmer productivity, conversion cost, and distributed heterogeneous systems have forced business and industry to focus on the control of the total information resource, including all data, processes, environments, users, hardware systems, etc.

Software technology, used to manipulate databases, has been moving at the leading edge of the industry for some time. At the same time, software used to locate, catalog, control, and manage the data has been lacking, particularly for use in

multivendor environments. Software for use in managing the total information resource is simply not in existence.

The requirement for increased productivity mandates the use of an information resource management system for the control of software development as well as the maintenance of software after development. In short, an information resource management system is needed for the control of the System Life Cycle. The use of an information resource management system in a multi-vendor hardware environment mandates the use of standard information resource management systems.

HISTORICAL PERSPECTIVE

We have come a long way toward controlling the data or information resource. Some of the steps we have taken were transparent and are described below.

The Early Years

Some of us remember the early days of this industry, when we were encouraged to comment freely in our assembly language programs so that the poor souls that inherited our programs could readily discern our logic (or lack of it). Those commenting techniques were sustained and enhanced with the use of "meaningful datanames" as we rose to the heights of "self-documenting" higher-level languages. If you stop and think, those comments, when made for data, were the beginnings of data management.

We became more sophisticated as time went on, developing manual lists of files that we used, and even more importantly, that we began to share. We also began to inventory the programs we used and to catalog them in libraries in some way. As the number of users of our systems began to proliferate, we began to list them with some simple attributes. Some of these lists eventually became automated, as extensions to tape library systems, or as stand-alone systems. As we cataloged files, we also found it handy to list the characteristics of the files so that sharing could be encouraged. As disk technology became available, we found that sharing of files on both media could be accomplished by providing common file definitions cataloged on a medium that could be copied and used either manually or in an automated fashion. In addition, we found

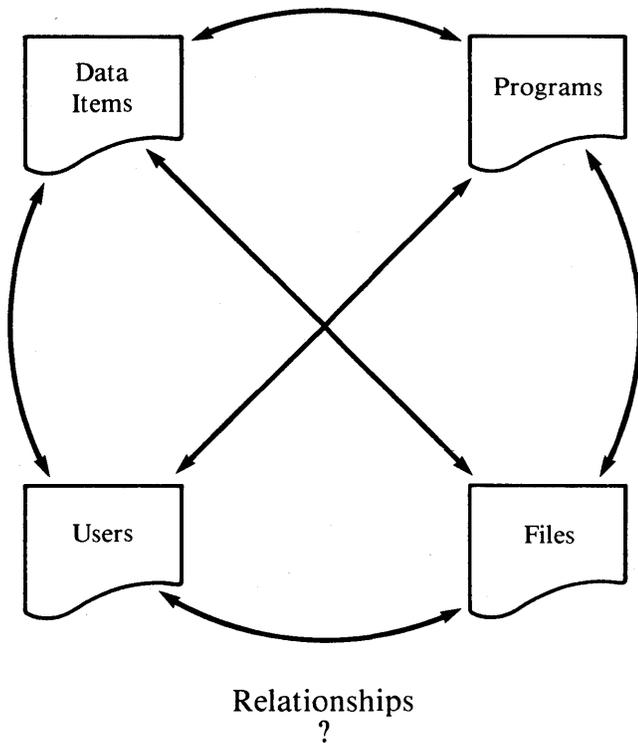


Figure 1—Unknown relationships

that comments placed in the file definitions served as documentation of the defined files. More importantly, we began to comment and document each data element of the defined file.

All of the above cataloging techniques existed in a variety of ways in the early days of data processing. They had mutual characteristics in that they all provided data management facilities of one form or another in a very rudimentary fashion. Most of the catalogs were manually maintained or, if automated, were single format record entries which could not be related automatically. All relationships between data elements, records, files, programs, etc. were strictly in the mind of the programmer. All of these individual catalog systems provided documentation of the programs, files, records, and data elements used by systems. In general, these techniques were used with small, independent systems. Little system interaction, if any, was present and it was usually provided by file interface programs which simply reformatted data into the form the requesting system could understand. Figure 1 illustrates the divorced cataloging problem in the early days.

MIS / Databases

The advent of second generation hardware systems allowed for the increase in the size and complexity of software systems. Software and associated files that had been restricted due to hardware limitations now could expand in size. Large Management Information Systems were proclaimed as the answer to the fragmented small system approach used earlier. Some of these Management Information Systems were ex-

remely complex both in their functionality and in their data relationships.

As a result of the Management Information System approach, the need for large, complex, shared files became evident. Out of this necessity and the availability of disk technology, came the "database" approach. The Database Management System (DBMS) was born at this time in order to provide commands, available to the programmer, which collected disk input/output and space management primitive functions and provided for a shared database environment. Many of these DBMSs used the network or hierarchical model, and the significance of data relationships became readily apparent. We began to be concerned with the interrelationships of data files, records, and elements. In addition, the shared approach used in the DBMS also caused implementors of MIS to become concerned with the interrelationships of programs with the database. The significance of change control became rapidly obvious. The level of comprehension of the industry had evolved to the information level.

Additionally, because of the shared environment, standard database definitions had to be created, used, and maintained. In most organizations, one or several persons were given database definition responsibility. The nature of most definitions was such that commenting was relatively easy, and aided in defining the nature of the data. This function, which began as a service to programming staffs, rapidly expanded to provide a service to users, etc. Thus was born the function of Data Base Administration.

At the same time, File Management Systems began to appear. Old favorites like MARKIV, WORK10, COGENT-II/IDS, and others were introduced at that time. These systems used large forms, were batch oriented, and provided similar command functions, as did the DBMS, for sequential and indexed sequential files. The original versions were cumbersome, but effective in providing reports for the end user.

Other technology began to appear on the data processing market at the same time which had a direct relationship to shared data and databases, and could benefit from directories. End User Facilities (EUFs) and the benefits of allowing end users to access data directly, in an interactive mode, required that specific data definitions be provided to End User Facility

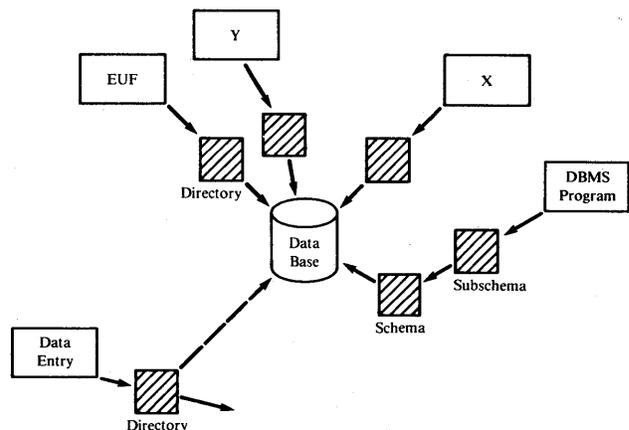


Figure 2—Multiple directories

software. In addition, Data Entry Facilities (DEFs) were introduced to allow for the direct entry of data into files and databases. These packages also required directories to define data entry criteria, such as size, range-of-values, editing characteristics, etc.

The one common requirement for all of these systems was the need to define the data and its characteristics and in some cases, the data relationships. Each system used its own data definition language (DDL) with its own peculiar syntax for defining data. Most data definition languages were COBOL oriented, and thereby provided natural commenting capability. It was necessary for the Data Base Administrator (DBA) to code a new set of DDL for each new DBMS, EUF, DEF, or FMS in use by his organization. This situation was compounded by the fact that new DBMSs, EUFs, DEFs, and FMSs were springing up like poppies. Figure 2 illustrates the multiple directory problem.

Another issue or concern in the MIS/Database environment was the need to establish ownership of shared data. Not only did the DBA have to be cognizant of the processing entities using data but also of which user owned and had responsibility for the data. In addition, accountability of the users owning the processes was necessary.

Database Standards

The federal government and private companies alike, with their increasing commitment to data processing and with particular emphasis on the new database technology, were concerned with the impact on conversion by the many DBMSs and their data definition (and manipulation) languages. As a result, an effort to standardize both the definition and manipulation languages was undertaken.

In the early 1970s, these standards efforts resulted in a major step, the introduction of the concept of "data independence," or the separation of data (and their definition) from the processes that use data. The Conference on Data Systems Languages (CODASYL) Data Base Task Group (DBTG) introduced the terms "schema" and "subschema" and defined the schema as the definition of the database as it is stored, while it defined the subschema as the programmatic view of the database. The significance of this approach was that data definitions were raised to a higher level of criticality and complexity. The Data Base Administration (DBA) activity was now a more formal and critical function in the industry. Much more emphasis was being placed on the management of meta-data or data about the data.

System designers and end users were becoming cognizant of data as a resource. It made sense to use data if they already existed somewhere, rather than redefine and recollect them. In addition, change control methodology was being applied to software development as it had been applied in engineering for some time. Configuration Management was being applied by all serious software engineering facilities and required knowing where the data were located, in addition to what they looked like, who owned them, and what processed them. These requirements necessitated the development of a "dictionary/directory" of data for both the end users and the systems personnel.

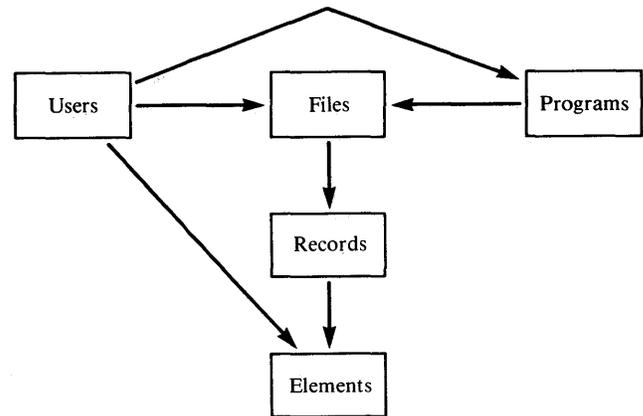


Figure 3—Embryonic dictionaries

Data Dictionary/Directories

The early 1970s saw the rise of Database Directories or Data Dictionary/Directories. These systems, in their early stages, consisted of several programs that would update and access a file, in many cases itself a database, and provide reports on data and relationships. The simple entities and relationships of the early Data Dictionary/Directories (DD/D) were limited to programs, the systems they operated within, the files they processed, and the records and data elements within those files. Figure 3 illustrates these embryonic relationships. It was soon realized that only the tip of the iceberg had been touched, and soon these had evolved into major systems encompassing other data such as user profiles, work stations, access control, and operational considerations, etc. What had originally started as being data oriented, had evolved into being information oriented!

The early and mid-1970s saw a rise in the number of Data Dictionary/Directories in the marketplace. The interest in what this type of system could provide began to focus on the total data or information available about the business enterprise.

The 1970s saw some interest in the on-line possibilities of Data Dictionary/Directories. The placement of editing characteristics in the Data Dictionary/Directory via the data definitions made it a natural for programs to access when editing data. The placement of subschemas in programs made a compiler interface attractive in that subschemas could be defined in the Data Dictionary/Directory and be provided to compilers. These types of Data Dictionary/Directories were called "active" in that they were used at compile and run time. The opportunities for the role of the Data Dictionary/Directory seemed endless.

ANSI/X3/SPARC DBMS Study Group

A major event occurred in the late 1970s which further propelled Data Dictionary/Directories into prominence. The American National Standards Institute, X3 Committee (Computers and Information Processing), Standards and Planning Requirements Committee's Study Group on Data Base Man-

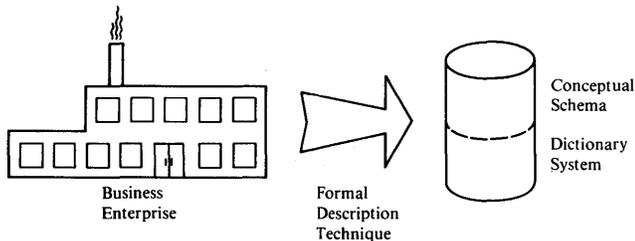


Figure 4—Business modeling

agement (ANSI/X3/SPARC DBMS-SG) issued a report which discussed the framework of database systems and is referred to as the ANSI/SPARC architecture. This architecture continued the two schema approach used earlier but referenced the schema as the “internal” schema and the subschema as the “external” or “user view” schema. Two significant references were made in the report prepared by two members of the group:

1. the Data Dictionary/Directory figures prominently in any database system and serves as a META-DATABASE and
2. prior to undertaking the development of a database system a conceptual model should be built modeling the enterprise.

The second reference was important in that for the first time a standard information model (called by ANSI/SPARC a “Conceptual Schema”) was being proposed which could serve as the basis of database (and even nondatabase or non-automated) systems. This Conceptual Schema differed from the Data Dictionary/Directory in that it was designed to serve as a model of the business enterprise whereas the Data Dictionary/Directory modeled only that which had been implemented. It seemed that the concept of information management was maturing. Figure 4 illustrates the Conceptual Schema and its development through a formal description technique.

CURRENT ACTIVITY

There are many issues at this time that surround the subject of Data Dictionary/Directories and Conceptual Schemas. Several groups are working on standards in this area.

British Working Party

In 1977, the British Computer Society Data Dictionary Systems Working Party published a report on the work it had been doing since January 1975. The significance of the report was that not everyone in the working party could agree on the role of a dictionary system. It was agreed, however, that the dictionary system played a crucial role in conversion from one DBMS to another, and that the description of data needed to be placed in this facility for generation of DDL’s for any

DBMS. The report also placed great emphasis on the dictionary as the residence of the conceptual schema. It recommended that the conceptual schema defined by ANSI/SPARC, be divided into two (2) parts, a conceptual schema and a derived schema. Much time, in the report, was spent discussing conceptual schemas; however, the report did emphasize the need for providing a tool to assess impact, provide standard code generation, multiple database version handling, utility interfaces, consistency checking, access control, mappings, and DDL generation, etc.

IRDS Technical Committee

In 1978, a new ANSI/X3/SPARC Database Systems Study Group (DBS-SG) was formed to try and take the ANSI/SPARC architecture and propose standards. One of the task groups formed within the Study Group was the Data Dictionary Task Group. This group was very active and soon realized that the scope of a Data Dictionary System was much greater than just involving Database Systems. It petitioned its parent, the DBS-SG, to form as a full-fledged technical committee to prepare a standard for Data Dictionary/Directories. After a long process, involving ratification by the DBS-SG, SPARC, and X3, the Data Dictionary Task Group was formed as X3H4, on June 11, 1980. The significance of the move underscored the commitment by industry to Information Resource Management because the petition for formation as a technical committee named the committee “Information Resource Dictionary System (IRDS)!” The IRDS Technical Committee has a goal of providing a draft standard by January 1983.

NBS Standards Effort

The federal government’s National Bureau of Standards (NBS) has also undertaken to write a Federal Information Processing Standard (FIPS) on Data Dictionary Systems which will be available by late 1982. This coincides with the IRDS Technical Committee effort.

FUTURE GOALS

In the author’s view, future standard tools for Information Resource Management must treat the following requirements.

System Life Cycle

The System Life Cycle (SLC) must be supported with any set of tools specified by any standard. These standard tools must support the conceptual modeling of the current manual or semiautomated systems that the enterprise has in place, so that a thorough understanding of existing methods is promoted. The idea of a process or procedure must be thought of in generic terms so that it can apply to either manual or automated activities. Specifications of these activities in terms of a baseline must be supported.

Data Dictionary/Directories have traditionally supported the documentation of files, programs, users, etc. Information Resource Management tools should provide for the entire scope of activities required to support the SLC. A critical part of this support should lie in "versioning" or systematic change mechanisms for data as the life cycle matures. Another activity should be the "staging" of information, such that these tools document stages of growth and change simultaneously. Figure 5 illustrates the key role an Information Resource Management tool, like an IRDS, could play in the SLC.

Configuration Control

The activity of Configuration Control has been treated in Data Dictionary/Directories, but in the author's view, in a very limited manner. Standard tools for Information Resource Management must consider a much more expanded scope if they are to be useful to an Information Resource Manager. The manager of the future is going to be concerned with geographical entities, be they databases, computers, work stations, users, etc. Standard tools must support the modeling of these entities. The advent of distribution, whether of databases or of systems, demands that type of modeling.

Future resource managers will have to consider such things as work stations, office locations, database and file locations, etc. The tools used by them will have to consider multi-vendors and a scope of impact that has never been considered before, such as loss of a work station, distributed node, etc.

Productivity

One of the most important thrusts of the 1980s will be the emphasis on improvement of productivity. Information Resource Management should have a significant contribution to make toward this goal, and tools that are developed will be forced to be standardized.

Information Resource Management tools should be able to store standard data definitions, common code for error routines, tables, and common code for table searches, etc., that can be accessed by any vendor's software. A programmer should be able to declare file and database names, common code, tables, etc. and the Information Resource Management tools should provide the definitions or code for insertion into the program. The ultimate activity should be the generation of whole programs based on specifications stored in an Information Resource Management database such as an IRDS.

The capability of the IRDS to store common definitions can allow for a reduction of effort in converting from one hardware or software vendor to another. The significant cost of converting files, programs, and databases to capitalize on cost-effective hardware can be reduced through the use of standard Information Resource Management data definitions stored in an IRDS and standard tools which can generate schemas, subschemas, EUF directories, etc., in the particular syntax required for the DBMS, EUF, etc. Significant cost reduction can be realized in that those programs using the old

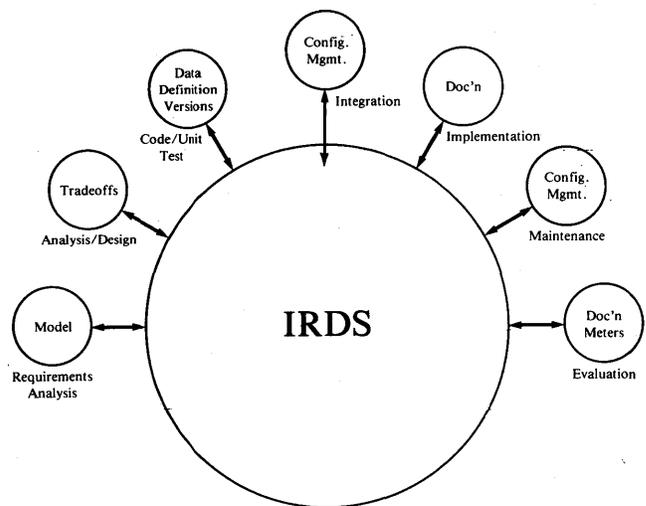


Figure 5—System life cycle and the IRDS

generation directories or schemas can simply use the newly generated ones. Conversion effort can now be directed on the program logic.

In addition to reducing conversion costs, the creation of a new set of data definitions when acquiring a new package should no longer be necessary. The vendor supplying the new package can merely supply a module which can interface with the standard IRDS database and create the directory required by the vendor's package in the syntax it understands. The need to define separate directories for EUFs, Data Entry Facilities, etc., can evaporate since only generating modules to provide that directory syntax are required. Figure 6 illustrates the use of an Information Resource Management tool like the IRDS in generating data definitions from a single source.

The concept of fully active and integrated tools should come into being in the 1980s. The necessity for standard tools should become even more obvious as we start interfacing with language processors and operating systems. Much of the editing performed in application programs can be transferred to an on-line tool such as an IRDS.

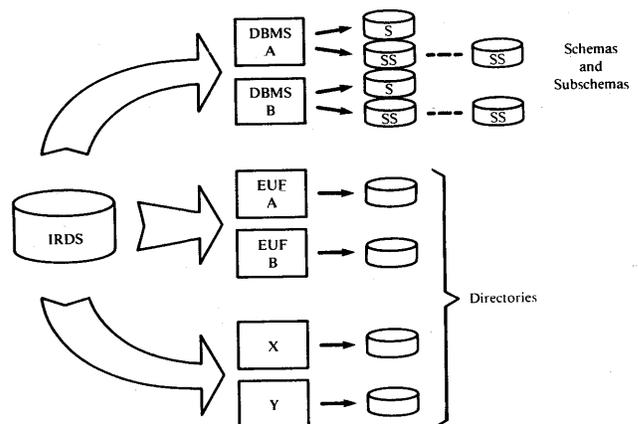


Figure 6—Single source data definitions

Flexibility/Extensibility

In the author's view, the most important requirement for Information Resource Management standard tools is flexibility. Standard tools should allow the Information Resource Managers to specify either the simplest model or the most complex model needed to accomplish the mission. These tools should allow specification of any number of entities from a menu of base entities, and allow the luxury of defining an expanded set of entities. Managers should have the ability to define attributes that are meaningful and not part of the base set of attributes for a given entity. Additionally, they should be given the flexibility to define their own coding structure, and not be encumbered by predefined coding structures created by the vendor.

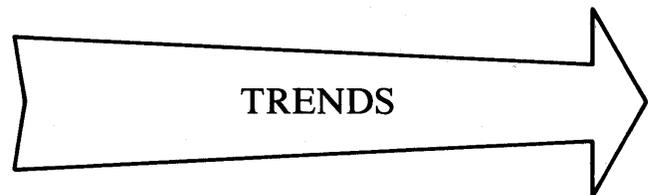
The Information Resource Manager should be given the ability to define meaningful relationships, in an ad hoc manner when necessary, and not be encumbered by sysgen requirements. Flexibility is needed to define any relationship between entities and not be encumbered by vendor predefined relationships.

The last flexibility requirement is that managers need the ability to query and report against the database used for Information Resource Management in an ad hoc manner. They should have the capability to specify reports including the contents of the reports. Canned reports should be kept to a minimum, since variations of report requirements are exponential.

SUMMARY AND CONCLUSIONS

We have come a long way from the early days, when documenting the data we used was the farthest thought from our minds. We have progressed over the years from just using this resource, to a state of the art in which attributes of the data we use are extremely important. We have elevated ourselves from the level of data orientation to the level of information orientation, and we have expanded our scope to include many more entities, and their attributes. Figure 7 illustrates the trend we have followed over the years.

Our concern for programmer productivity, conversion cost containment, and managing configurations reflects a maturing attitude in the data processing community. Standard tools to provide control of the information resource can reduce significantly the costs of conversion and configuration change errors. The work that is under way in ANSI/X3/H4 as well as the National Bureau of Standards will soon bear fruit and will start us up the long, slow path towards true Information Resource Management.



Little concern over data	Some concern over data	Concern over data in database	Major concern over data source; conversion, and productivity
Each programmer defines physical and logical files	Some central definition; "copy" files	Central definition of data; duplication exists between databases	Central definition; one source of information

Figure 7—Trends

REFERENCES

1. "ANSI/X3/SPARC DBMS Framework Report," Report of the ANSI/X3/SPARC Data Base Study Group, 1977.
2. Bachman, Charles W. and Daya, Manilal, "The Role Concept in Data Models," 1979.
3. Chen, P. P-S., "The Entity-Relationship Model—Toward a Unified View of Data," ACM Transactions on Database Systems, March 1976.
4. "CODASYL Data Base Task Group, April 1971 Report," ACM, New York, 1971.
5. "Data Base Directions—The Next Steps," NBS Special Publication 451, U.S. Department of Commerce, September 1976.
6. Dolotta, T.A., et al., *Data Processing in 1980-85—A Study of Potential Limitations to Progress*, New York, John Wiley, 1976.
7. Gall, R. Michael, "The Role of Data Base Management in Information Resource Management," *Proceedings 2nd Annual Database Symposium*, May 1980.
8. Hillery, Norris and Trigger, Weston, "A Systems Engineering Facility," *Proceedings NBS Symposium*, 1978.
9. "Information Resource Management—The New Challenge," A Special Report by the Diebold Group, *Infosystems*, June 1979.
10. *Information Technology—Some Critical Implications for Decision Makers*, The Conference Board, Inc., New York, 1972.
11. Martin, James, *Principles of Data-Base Management*, Englewood Cliffs, New Jersey, Prentice-Hall, 1976.
12. Monia, Joan, "Uses of the Data Dictionary," from working papers of the Data Dictionary Task Group (ANSI/X3/SPARC DBS-SR), July 1979.
13. "Proposal for X3 Standards Project on data dictionary systems," Proposal to DBS-SG, SPARC, and X3 for formation of X3/H4 Technical Committee, 1980.

SAGA: A system to automate the management of software production

by R. H. CAMPBELL and P. G. RICHARDS

*Department of Computer Science, University of Illinois
Urbana-Champaign*

ABSTRACT

SAGA is a software development system designed to integrate software production tools and techniques into a flexible management system through the use of special attributed grammars to represent management schemes. Both the software life cycle and project components are described by formal grammars. This formalization will aid understanding of management techniques for complex projects and encourage the automation of repetitive and tedious managerial tasks. Project direction and management is monitored in SAGA from inception through completion and allows identification and scheduling of critical events along with integration of project specification, design, implementation, certification, and maintenance. The SAGA project is expected to have a particular impact on quality software production for reliable computer applications.

INTRODUCTION

The process of software design, production, and maintenance follows a pattern of activity often referred to as the software lifecycle. The management of the lifecycle is critical to the success of the eventual software product¹. The response to this critical management problem has been the evolution of software engineering. Software engineering has improved the production of high-quality software through the use of new tools². Methodologies to improve design, programming, and maintenance have been proposed. The effectiveness of these tools depends on the proper management of information they produce. The SAGA project proposes a method of integrating many of these software production tools and techniques into a flexible, formal management system for software development.

Goals

The goal of the SAGA project is a formal management methodology and system that will enhance production of complex, reliable, certifiable software. Complex software is typified by a long development period, interaction between several

developers, complex module interfaces, and requirements for auditing, certification, and quality. We view software management as the recognition of valid sequences of (perhaps concurrent) activities in a software lifecycle. Certification of a software product corresponds to the recognition of a valid history of software development, verification, and validation. A satisfactory system for development of such software requires:

- Recognition of valid sequences of activities in software life cycle.
- Consistent application of a well defined management policy throughout the life of the project.
- The ability to ascertain development status of the software project, and retrieval of project information.
- Automation of repetitive and tedious management tasks (e.g., auditing, version control).
- Automated identification of intermodule dependencies between and within project phases.

Desirable attributes of this kind of software development system include:

- Machine processable specification, design, and implementation languages.
- Centralized and coordinated storage and processing of all project information (e.g., requirements, specifications, designs, data, source and object code, testing information, documentation, design decisions).
- Appropriate communication and documentation tools (e.g., interuser communication, intraproject communication, communication between users and management system).
- Ability to integrate already developed tools (e.g., automated program verifiers, test data generators, optimizers, performance analyzers).
- Checking of the consistency of intermodule dependencies.

Other Systems

Many systems have been developed to aid software development. Among them are CADES³, Bell Laboratories,

Unix/Programmer's Work Bench⁴, and Gandalf⁵. Each of these systems satisfies some of the criteria above.

CADES

The CADES (Computer Aided Development and Evaluation) system was developed at International Computers Limited as an operating system development aid. It is composed of a database, a language interface called SDL, and a formalism for transformations of problems called *structural modeling*. Structural modeling specifies the relations between data and the objects that manipulate the data, and supports refinement of both the relationships and the objects until an implementation is realized. The database is used to store these relationships and maintain auditing history on the refinements that have been applied. CADES provides facilities for project organization, version control, interfaces to compilers/linkers for automatic invocation after module modification, and facilities for including other tools for further analysis of the project database.

PWB

The Programmer's Work Bench is an adaptation of the UNIX operating system to the needs of large software development projects. PWB provides an efficient programming environment that is separated from the system on which the programs are to be executed. It provides additional tools for software development, including a Source Code Control System⁶ (SCCS) and remote job entry software. Unix provides facilities for editing and file storage, document preparation, and user communications. SCCS provides version control and auditing of modules of source code, documentation, or test data.

Gandalf

Gandalf is an interactive software development system for the ADA programming language. A syntax-oriented editor permits entry of programs. The language INTERCOL and software development control facility of Gandalf is described by Tichy⁷. INTERCOL is one of a class of languages known as Module Interconnection Languages. INTERCOL represents the structure of systems by describing module interfaces. Interface consistency is maintained by type checking between modules and notifying appropriate developers when inconsistencies are found. INTERCOL also has the ability to describe multiple versions of software using a concept of "families" of modules and systems.

Analysis of Other Systems

No system presently solves the problem of managing software development in a complete and satisfactory manner. The isolated collection of tools in PWB requires the programmer to remember important procedures and to use the tools cor-

rectly. Structure-based systems such as INTERCOL and CADES attempt to restrict the software development process to ensure that intermodule interfaces are correct and consistent. Neither system integrates the restrictions with the target source code nor extends automated management to all phases of the project. Our proposal combines the various software development system approaches into a flexible and effective system.

THE SAGA SYSTEM

Approach

The SAGA system provides an integrated approach to the management of the software production process by combining various existing techniques for recognizing, representing, and analyzing formal specifications. Its primary components are:

- Formal representation of management policy by *management grammars* (LALR[1] attributed BNF grammars which use events in the software life cycle as terminal symbols).
- Primitives for specifying module and system structures, and events that occur in the lifecycle.
- A central database with provisions for storage of all project related information.
- An inter-project library for sharing code, data, and development procedures.
- Formal specification and constraint of database manipulation(s) via *development grammars*. The formal specification allows such features as automatic recompilation and auditing.
- Formal representations of and uniform interfaces between specification, design, and implementation languages which permits mechanical consistency checking within and between phases in the software life cycle.
- User oriented communication facilities that include not only the ability to "talk" and "mail" between users but also archival notesfile facilities to record policy decisions and allow discussions to take place in writing in a machine readable form.

The management and development grammars specify the sequence of acceptable events in the software project from its inception through its completion. Events can be generated by programmer interaction or by the partial parsing of a sentence of a grammar. The management grammar represents policy and its terminal symbols are events generated by the programmer or by the system. As sentences of the language specified by the management grammar are parsed, different management primitives are invoked. These primitives can start subtasks controlled by other management grammars, declare events to higher level tasks, or invoke specific software tools controlled by development grammars. The development grammars are used to control access to specific tools such as compilers and editors. This hierarchical management system can be used to configure complex and concurrent project development schemes. Tracing of the parsing can be done to any granularity, thus allowing auditing to any level.

The database manipulation routines have, as integral components, bookkeeping routines which audit intermodule references and ensure consistency through the project. This level of bookkeeping is required to simplify recognition of unconsidered specifications or uncoded designs and issue requests for their completion.

Information in the SAGA system should be represented in a machine processable form, that is, high level language. This is to allow identification of the events that are considered important to the management policy. In addition, it is expected that investigations into automated analysis of project phases such as validating a design for consistency with its specification will require that specifications and design be represented in a high level language. We believe the SAGA database could be a useful tool for the future development of such automated analysis. Management and development grammars based on the syntax of the specification, design, and implementation high-level languages allow SAGA to control project development to the individual statement level if required.

Applications

The management schemes employed in a SAGA development system are intended to enhance human engineering aspects of software production. For software development of applications which must be very reliable, the management schemes can impose a precise and rigid development discipline. Alternatively, it might provide an environment for rapid development of scientific or research programs that do not require such rigid control. The system can be used to enhance the productivity of the system developer by providing on-line project information, coordinating efforts and module sharing among teams, prompting for completion of standardized documents, cross referencing between phases of the project, and providing status reports of the project.

Since the management policies for the project are explicitly stated (by the management grammars and the attributes on the project languages) and the policies are enforced by the software tools themselves, validation of software produced under the system is much simpler. It is possible to log every operation taken during the development in order to satisfy strict auditing procedures. Formal structuring of the development process may allow more rigid validation assumptions to be made (by either automatic or manual theorem provers) about specifications and coding of modules.

The SAGA system is designed to direct programmer activity without imposing excessive restraint. It is expected that managers will recognize the need for balance between programmer control and freedom. SAGA provides an excellent vehicle for experimentation in various management policies and can be used to analyze the effects of those policies.

Example

Below are some grammar fragments for a hypothetical SAGA system that control updates to project source code. A single management specifies version and release policy. A

development grammar controls updates to source modules. The grammar is represented using the usual BNF meta-symbols {} to denote repetition. [x] indicates semantic action "x" is to be performed (described in the narrative below). Terminal symbols which are events are represented as "EVENT".

The management grammar

```

<new version> ::= <initialize modify> <modify module>
                <release>
<initialize modify> ::= "NEW_VERSION" [1]
<modify module> ::= {< new code>
                    <validation & verification>} +
<new code> ::= {"CODE" [2]} +
              "CODE_COMPLETE" [3]
<validation & verification> ::= "VERIFY" [4]
                              "VERIFY__COMPLETE"
<release> ::= "RELEASE" [5]

```

The development grammar

```

<code> ::= {{<module work>} +
            <check consistency>} * "DONE" [6]
<module work> ::= "EDIT" [7]
<check consistency> ::= "CHECK" [8]

```

The events used by this example are described below:

- **NEW VERSION**—The project manager wishes to authorize a change to a source module, and declares this event.
- **CODE**—The project manager indicates changes may proceed by declaring this event.
- **CODE COMPLETE**—This event is requested by the project manager when all requests for new coding have been made.
- **VERIFY**—Project manager indicates that the modification is approved and testing should start by declaring this event.
- **RELEASE**—Project manager can release the tested module as a new version by declaring this event.
- **DONE**—A programmer declares this event when all modifications to the module are complete.
- **EDIT**—Raised by invoking an editor on the source module.
- **CHECK**—Programmer uses a compiler or analyzer on the source module.

Under a SAGA system using these grammars, a possible sequence of events is described below:

1. Someone requests a change to a source module. The project manager indicates that a change is to occur by declaring the "NEW VERSION" event. Management primitives invoked at [1] request a reason and description of the change from the project manager, which will be stored in the new version's documentation.

2. The project manager declares one or more "CODE" events. Primitives invoked at [2] request the name of the module and the programmer assigned to make the change. A temporary copy of the module is created. The programmer is authorized to use the development grammar to access the temporary copy. The system uses the development grammar independently and asynchronously of the management grammar. After the "CODE COMPLETE" event, the management grammar primitive at [3] waits for all coding subtasks to complete.

3. The programmer invokes the editor, which declares "EDIT". Primitives at [7] check his authorization and allow the editor to proceed.

4. After editing, the programmer uses a compiler to check the source module for errors. This invokes primitives at [8] that make sure no undefined subroutines are referenced.

5. After the programmer is satisfied that the changes are correct, he declares the "DONE" event, which causes primitives at [6] to terminate the subtask using the development grammar.

6. The project manager is notified that the modules are changed, and is allowed to declare event "VERIFY" to start verification of the module. Primitives at [4] start another development grammar for verification. The project manager waits until the "VERIFY COMPLETE" event is declared by the verification subtask.

7. Verification is completed and "VERIFY COMPLETE" is declared. The manager is notified that the module is ready, and declares the "RELEASE" event. Primitives at [5] make the temporary copy into a new release in the database catalog, notifying the appropriate users that the new release is complete.

PROTOTYPE

A prototype SAGA system is being implemented in Pascal. A construction tool integrates grammars for development and management into the SAGA management system and provides an interactive interface to the user for text and program editing and project work. The construction tool is based on an LALR parser generating system and skeleton interactive editors. Semantics attached to the management and development grammars invoke management primitives (coded as Pascal routines) to manipulate the project database. An example development system for Pascal will be designed at the conclusion of this prototype implementation.

CONCLUSION

The formalization of management in software development projects will improve understanding of the project lifecycle and strengthen the validity of software certification. The SAGA system provides an approach to the automatic generation of software development systems and the eventual formalization of management schemes. Management of the development process can be applied to all interactions and information in the project from the moment of entry to the computer through its lifecycle.

Although considerable research and development is required to realize a production version of SAGA, the prototype specification suggests that such systems can be constructed and that management schemes for the production of software can be described using augmented grammars. A prototype SAGA system is currently being constructed. We welcome comments, suggestions, and examples of management schemes that have been applied to actual software production projects.

ACKNOWLEDGMENTS

This research was partially supported by NASA Grant NSG 1471 at the University of Illinois, and by NAS 1-14472 while the authors were in residence at ICASE, NASA Langley Research Center. The authors would like to acknowledge the helpful comments of Martin McKendry and John Knight.

REFERENCES

1. Brooks, F.P., *The Mythical Man Month*, Addison Wesley, Reading, MA., 1975.
2. Jensen, Randall and Charles Tonies, *Software Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.
3. Pratten, G.D., "The CADES Software Development System", Internal Document, International Computers Ltd, Kidsgrove, Stoke-on-Trent, England, 1978.
4. Dolotta, T.A., R.C. Haight, and J.R. Mashey, "The Programmer's Workbench", Bell System Technical Journal, Vol. 56, No. 6, July-August 1978, pp. 2177-2200.
5. "The Gandalf Project", Presentation at the Software Tools Workshop, Pin-gree Park, Colorado, May 1979.
6. Rochkind, M.J., "The Source Code Control System", IEEE Transactions on Software Engineering, SE-1, December 1975, pp. 364-370.
7. Tichy, Walter F., "Software Development Control Based on System Structure Description", Carnegie-Mellon University Department of Computer Science Technical Report CMU-CS-80-120, January 1980.

The development facility approach to improved software development

by DAVID W. JOHNSON

The Upjohn Company
Kalamazoo, Michigan

ABSTRACT

The use and benefits of a software development facility (SDF) within an organization are reviewed. A generalized SDF framework is presented which suggests the need for a common SDF "kernel" which an organization may tailor and supplement to meet its specific application development needs. The development of a SDF prototype within a particular software development environment is described. Finally, some experiences with the use of the prototype are discussed.

INTRODUCTION

The term *software crisis* has been used to describe the serious problems currently facing much of the data processing community. During the past several years hardware technology has shown rapid advancement. Each advancement brings with it a reduction in the unit cost of the hardware in addition to providing the user with improved capabilities, such as speed, memory size, and storage capacity. As unit costs decrease and capabilities improve, organizations become more and more eager to take advantage of the technology. New and more complicated applications become desirable, and improvements to existing applications become mandatory if the organization is to remain competitive. All this places increased demands on the software development and maintenance group within the organization. Unfortunately for these groups, the technology used in developing and maintaining software has not kept pace with the rapid advances in hardware technology. The advancements that have been made are often met with resistance because of their complexity or the training required for the staff to take advantage of them. Because of this, software development and maintenance costs have taken a continually increasing share of the data processing budget of an organization, thus precipitating the crisis.

The current literature contains numerous approaches aimed at solving or reducing the software crisis. Many of these concentrate on the development of tools that help to make the software developer more productive. Examples of this from hardware and software vendors include ICCF (Interactive Computing and Control Facility) and ADF (Application Development Facility) from IBM, USER-11 from North County

Computer Services for use on DEC PDP-11's, and Hewlett-Packard's IDS (Integrated Display System). Examples of systems developed in house include CSMAGIC at Data General and the Interactive Program Generator at Exxon.^{1,2}

The purpose of this paper is to describe one of the more promising efforts directed toward a solution to the software crisis, the software development facility (SDF) approach. The fundamental way in which this approach differs from previous approaches is that it involves the integration of software development tools into an easy-to-use system relieving the software developers of much unnecessary or redundant effort. Information required by the system is maintained in a software database. In the early stages of software development the database contains primarily design information. As refinements are made, the database expands to contain information about the makeup of the various software objects that make up the actual system being developed. During this expansion process the SDF insures that the necessary information is collected and that the stored information remains in a logically consistent state. Once a complete description of the software system has been achieved, the information is used to aid in the construction and testing of the software. After the development is complete, the SDF role continues by controlling the software execution and giving support to the maintenance of the system. Figure 1 shows a high-level SDF overview.

The benefits of such a facility are numerous. They include improvements in the software design process, since most manual drawing, charting, and redrawing is eliminated and the computer is used for completeness checking and design analysis; the elimination of needless human effort in using the various software development tools by providing an easy-to-use integrated environment to the users; and improvements in the maintenance process, since complete documentation and performance data are maintained by the facility.

There have been several in-house attempts to integrate at least a portion of the development activities. Two examples of this are The Programmers Workbench at Bell Labs³ and CPDS, Chevrons Program Development System¹.

The remainder of this paper describes a generalized framework for an SDF within an organization, gives an implementation example currently in use in the Research Division of The Upjohn Company, and cites productivity data obtained from its use on several projects.

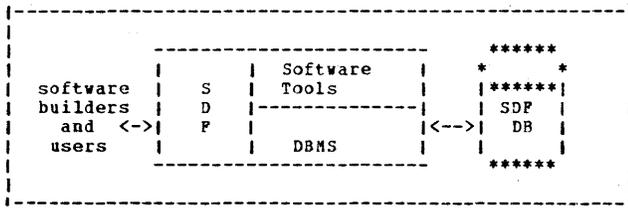


Figure 1—An abstract SDF overview

A GENERALIZED SDF FRAMEWORK

We present here a generalized framework for an SDF and discuss how an organization may use the concepts to develop an SDF for their own unique circumstances. Because of the great diversity of environments in which software is developed, one cannot claim complete generality in any approach. Therefore this discussion assumes a state-of-the-art environment providing a time-sharing interface to the software developers, along with a flexible operating system command language.

Figure 2 presents an architectural overview of the SDF framework. Six basic subsystems are identified, along with four classes of users and two primary data stores. The plan is that an organization be presented with a basic SDF kernel, which it must then expand upon and tailor to suit its own particular needs. This is the task of the person or group labeled SDF administration. The first function the SDF administrator must perform is to establish the database environment. Since the SDF kernel contains its own database subsystem, this may not require additional effort. On the other hand, if the organization desires to use its own DBMS in conjunction with the SDF, database access modules will need to be written to support the remaining kernel code. Once this is accomplished, the SDF administrator must generate a meta definition for his organization. For this he uses the meta definition subsystem, which allows him to define the types of software objects of interest, the types of relationships between software objects, and the transformations (tools) that will operate on the software objects. For example, he may define an object of type MODULE, specify the attributes about modules to be stored in the database, define a relationship of type CALLS between objects of type MODULE, and finally define a transformation COMPILER to be performed on objects of type MODULE. Since this definition process could be a bit time-consuming, the SDF kernel will contain a standard set of object types, relationship types, and transformations, which the SDF administrator may modify as he desires. Once the meta definition is complete, the SDF becomes available for use. The work of the SDF administrator is not complete, however. As SDF usage increases, changes and additions to the meta definition will probably be required. In addition, specialized tools meeting the specific application needs of the organization must be developed and integrated into the SDF.

The development subsystem provides the interface for the primary users of the SDF, the software developers. Functions are provided to support the entire software development process, from design through implementation. The design function allows the system architects, program designers, and

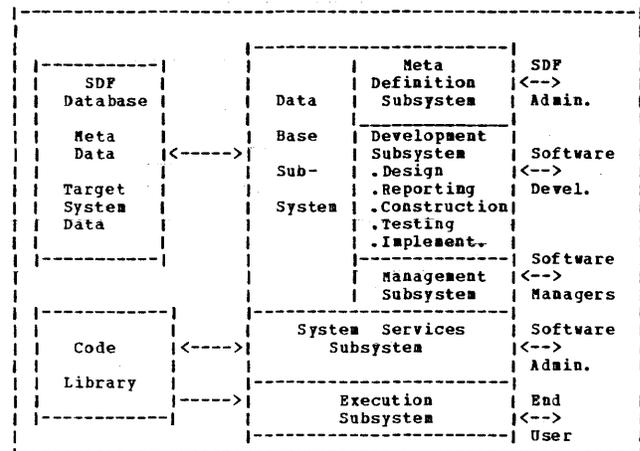


Figure 2—SDF architectural overview

database designers to specify their designs to the SDF and to have them checked for accuracy and consistency and stored in the design database. The SDF kernel provides the mechanism for expressing the design in terms of the meta definition, discussed earlier. Facilities are also provided for maintaining the design data, e.g., updates and deletions. The reporting function then allows the designers and others to examine the design in terms of a series of reports and analyses that may be used for design improvement and communication. An example of such a report would be the CALLING sequence for a particular MODULE.

The construction function is used by the programmers in developing code for the various software objects defined by the designers. Whenever possible, code for an object is generated either partially or entirely. Examples of partial code generation include beginning and ending statements, declarations, and module calls. Such code must be further enhanced by the programmer using the system editor. Code to be completely generated requires a fairly high degree of specialization, with a considerable amount of intelligence in the code generator. Examples of complete code generation include database definitions and module code for specific application types. All code for software objects is managed by the SDF, which has facilities for production and test versions as well as for maintaining a historical record of changes to the code. Tools used in the construction process, such as compilers and loaders, are integrated into the SDF framework by the SDF administrator so as to take advantage of all available information and eliminate any unnecessary effort on the developers' part.

Once a software object has been coded, the SDF testing function is used to aid in its testing. Static testing tools can be integrated into the SDF in a fashion similar to that used for construction. Dynamic testing tools will require that test data and results be available from the database. Where necessary, module stubs and drivers are automatically generated. For modules that require complicated inputs and outputs a series of simulators will need to be developed by the SDF administrator to meet the organization's needs.

The management subsystem provides data about the development progress of the various software objects making up a system. Each time a development or testing function is per-

formed an entry is made in the management portion of the database. Software managers may input data such as estimated completion dates for various objects and receive reports on the progress of the system. Ad hoc inquiries are also available when information about a particular object is desired.

The system services subsystem provides two primary functions: code library management and transformation control. Code is maintained on a system basis with test and production versions. Most of the tools integrated into the SDF will be executed under the control of the system services subsystem. Batch execution will be performed in parallel with other development functions. Code generated by tools will automatically be stored in the appropriate library.

The operation subsystem forms the interface between the end user of the target system and the SDF. As such it has two primary responsibilities, first to regulate access to the target system and second to monitor its operation. The monitoring activities should include the gathering of performance data for both processing and data objects and the recording of target system status for the diagnosing of error conditions.

The database subsystem is an integral part of the SDF. It controls all access to the database and allows concurrent use by the various SDF users. Most organizations interested in the use of an SDF will already be using a database management system (DBMS). The characteristics of this system will influence the types of objects the organization defines and the types of transformations that will be developed in the development subsystem. Since the organization has undoubtedly developed considerable expertise in using their DBMS, it seems reasonable that the same DBMS should serve as the storage mechanism for the SDF. This would seem especially reasonable if query and report writing capabilities were available with the DBMS.

AN IMPLEMENTATION EXAMPLE

This section of the paper describes a prototype SDF currently in operation within the Research Division of The Upjohn Company. The system, called USDF, is written in PL/I and uses the VM/CMS operating system. The database system used is System R, a relational DBMS⁴. Data in the SDF database is stored in the form of tables (relations). Two basic groups of tables exist, the meta tables and the target system tables. Data in the meta tables specify the types of software objects of importance. For example, in addition to object types like MODULE and PROGRAM we see VM object types such as VMACHINE (virtual machine) and System R object types like TABLE and COLUMN.

In addition to object types, the meta tables contain data about the attribute types and values to be maintained about objects. An example of an attribute type for object type MODULE would be LANGUAGE, for which a definite list of values (PL/I, FORTRAN, BAL) could be specified. The final meta construct is of a relationship type. A relationship indicates that two object types are related in this way: one of the object types is considered the source and the other the target. An example relationship type whose source object type is MODULE and whose target object type is TABLE

Table Name	Column Name	Column Description
OBJTYPE	OTNAME	Name of object type
	DEPEND	Is this a dependent otype
	CLASS	Name of object type class
	DESC	Description of object type
ATTRTYPE	ATNAME	Name of attribute type
	OTNAME	Name of object type
	ATYPE	Domain set of attribute type
	LENGTH	Max. length of attr. value
	NONVAL	Max. number values allowed
	USE	Is attr. required
	DESC	Description of attr. type
ATTRVAL	ATNAME	Name of attribute type
	OTNAME	Name of object type
	CODE	Code for attr. value
	VALUE	Attr. value for code
RELTYPE	SOTNAME	Name of source object type
	RTNAME	Name of relationship type
	TOTNAME	Name of target object type
	CRTNAME	Name of complementary rtype
	DEPEND	Is this a containment rtype
	CONN	Connectivity of rtype
	DESC	Description of rtype

Figure 3—Description of meta tables

might be USES TABLE. The columns of the meta tables are entered by using the facilities of the meta definition subsystem. Information in the meta tables is summarized in Figure 3. This subsystem allows for the initial loading of the meta definition and provides facilities for subsequent changes. The current meta tables contain about 20 different object types, 30 attribute types, and 50 relationship types.

The second type of tables in the SDF database are the target system tables. These tables store information about the actual objects making up the system. The tables are of a general nature. One table stores information about objects, three tables store information about attributes for objects (integer, character, and real), and one table stores information about the relationships defined. The columns for these tables are summarized in Figure 4.

Table Name	Column Name	Column Description
OBJECT	OTNAME	Name of object type
	SYSTEM	Name of development system
	ONAME	Name of object
	CRDATE	Creation date of object
INTATTR	SYSTEM	Name of development system
	ONAME	Name of object
	OTNAME	Name of object type
	ATNAME	Name of attribute type
	DONAME	Dependent object name
CHARATTR	VAL	Value of integer attribute
	SYSTEM	Name of development system
	ONAME	Name of object
	OTNAME	Name of object type
	DONAME	Dependent object name
	CODE	Code for attr. value
REALATTR	VAL	Attr. value for code
	SYSTEM	Name of development system
	ONAME	Name of object
	OTNAME	Name of object type
	ATNAME	Name of attribute type
REL	DONAME	Dependent object name
	VAL	Value of integer attribute
	SOSYSTEM	Name of source object system
	SOTNAME	Name of source object type
	SONAME	Name of source object
	RTNAME	Name of relationship type
	TOSYSTEM	Name of target system
	TOTYPE	Name of target object type
	TONAME	Name of target object

Figure 4—Description of target system tables

```

-----
| PLEASE SELECT DESIRED DEVELOPMENT FUNCTION
| design
| PLEASE ENTER DESIGN FUNCTION TO BE PERFORMED
| help
| THE FOLLOWING COMMANDS MAY BE ISSUED AT THE DESIGN LEVEL:
| DEFINE - DEFINE A NEW OBJECT OR NEW INFORMATION ABOUT
| AN EXISTING OBJECT.
| DROP - DROP AN OBJECT AND ALL ITS INFORMATION.
| CHANGE - CHANGE INFORMATION ABOUT AN OBJECT.
| DELETE - DELETE INFORMATION ABOUT AN OBJECT.
| RETURN - RETURN TO DEVELOPMENT LEVEL.
|
| PLEASE ENTER DESIGN FUNCTION TO BE PERFORMED
| define
| ENTER INPUT FILENAME(S).  DEFAULT IS TERMINAL.
|
| PLEASE ENTER OBJECT-TYPE TO BE DEFINED
| help
| THE FOLLOWING OBJECT TYPES MAY BE DEFINED:
| EXEC
| EXECPARM
| PROGRAM
| MODULE
| MODPARM
| DATABASE
| TABLE
| COLUMN
| VMACHINE
| VDASD
| SYSTEM
|
| PLEASE ENTER OBJECT-TYPE TO BE DEFINED
| module
| PLEASE ENTER NAME OF MODULE
| abc
| MODULE HCMS ABC : ADDED
-----

```

Figure 5—Using the design function (novice user)

Information is entered into the target system tables via the design function of the construction subsystem. The design function is driven by the meta definition described earlier. The software designer describes his design by using a relational language similar to PSL⁵. Language statements are grouped into sections, each of which describes information about a particular software object. Statements may be entered in batch or interactively. For a novice interactive the user system prompts for all portions of the statements and provides extensive HELP facilities if needed. For the more experienced user the prompts may be eliminated by entering the entire statement. Figure 5 shows an example design scenario for a novice user of the design function. The same design information could be given by an experienced user, as shown in Figure 6.

As the software designers enter information via the design function, they also use the facilities of the reporting function to look at, analyze, and summarize the design. The user has access to the information via a high-level query language and via a series of canned parameterized reports. Figure 7 shows an example scenario requesting a calling structure report.

Once the design of a software object is complete, the construction function is used to develop its code. Currently code may be generated for objects of type table and module. Table code is in the form of SQL create table statements. Module code generation is done on a specific application-type basis. All code generated is PL/I or PL/I with embedded SQL statements. Complete code generation is performed for modules whose only function is to perform database interaction. Partial code generation is performed for modules that perform full screen I/O. To do this the analyst defines a screen layout of USDF and identifies the attributes of its items. A complete module to test the screen is then generated, and further processing enhancements are made by the programmer. When

```

-----
| PLEASE SELECT DESIRED DEVELOPMENT FUNCTION
| design
| PLEASE ENTER DESIGN FUNCTION TO BE PERFORMED
| define module abc
| attr language PL/I
| rel calls xyz
-----

```

Figure 6—Using the design function (experienced user)

necessary, the code is completed by the programmer by using the ENHANCE function, which turns control over to the system editor. After the code is completed, the COMPILE function is used. This function automatically selects the proper set of transformations to be executed to produce the required object code. For example, all System R modules are run through a preprocessor before going to the PL/I compiler. All transformations that do not require interactive input from the user are performed in batch under the control of the system services subsystem. The program performing the construction function sends a message to the virtual card reader of a special control machine, which then autologs one of several available task machines to perform the function. The original source code is obtained from the originating user via a DASD link. Messages are sent to the user about the progress of the operation. During this time the user may be doing other construction functions, such as ENHANCE. When the operation is complete, the results are distributed to two locations. All useful code from the operation is stored in the appropriate code library. Any error messages and return codes are sent to the originating user's virtual card reader when they may be examined by using the CHECK function. Figure 8 shows the architecture of the system services subsystem.

The current implementation of USDF has not addressed several of the areas discussed in the section "A Generalized SDF Framework." The testing and implementation functions remain to be developed, along with enhancements to the management subsystem, which currently maintains a record of all development functions performed on software objects. USDF has, however, been used in its current form for the development of several software systems, including its own development, with encouraging results.

The largest project to use USDF to date has been the development of a fairly sophisticated animal management system.

```

-----
| PLEASE SELECT DESIRED DEVELOPMENT FUNCTION
| report
| PLEASE ENTER TYPE OF REPORT DESIRED.
| stru
| ENTER OUTPUT FILENAME(S) FOR STRU.  DEFAULT IS TERMINAL.
| ENTER LEVEL 1 OBJECT NAME.
| gdf
| ENTER RELATIONSHIP NAME.
| calls
| STRUCTURE REPORT FOR RELATIONSHIP CALLS
| SYSTEM NAME = HCMS
| 1 GDF
| 2 SA6
| 2 GETID
| 3 CCID
| 2 BEGTRAN
| 2 ENDTRAN
| 2 SYERROR
| 3 RESTRAN
| 2 SYS.CLEAR
-----

```

Figure 7—Example structure report

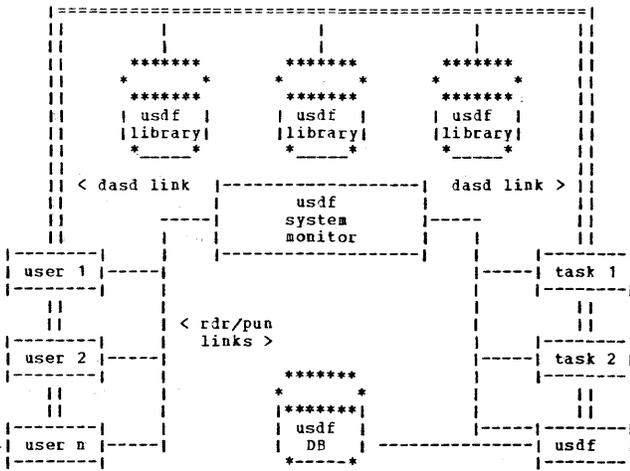


Figure 8—System services subsystem architecture

Data about the animals is maintained in a series of System R tables. Reports directing the activities of the animal caretakers are produced daily. As the activities are performed, data of interest are recorded on the report and later entered into the database by the caretakers, using formatted 3270 screens. The system was developed by two individuals (analyst and programmer) using the current methodology of structured design, structured programming, and walk-throughs. The system consists of approximately 15K lines of PL/I code distributed into some 100 modules. The design information consists of data about 300 software objects and their attributes with nearly 2000 relationships defined between the objects. The USDF code generation facilities were used extensively for database interaction and screen processing modules. An analysis of the code for the entire system shows that approximately 45% of the code was generated by USDF. This together with the disciplined development approach has resulted in a minimal number of postimplementation changes (five in the first four months). Mean time to make a change was 15 minutes using the facilities of USDF. Development data from the project were compared to several other in-company projects of similar scope. These are summarized in Figure 9. The projects ranged from 15K to 25K lines of code. Person-hours per line of code ranged from .14 for the project using USDF to .39 for the average of the non-USDF projects. It is also interesting to note that the use

	Animal Project using USDF	Project X	Project Y
Lines of code	15,000	19,000	25,000
Man hours per line of code	.14	.42	.36
370/148 CPU sec. per line of code	13	-	19

Figure 9—Development data comparisons

of USDF did not take more computer resources (13 sec./line) than a non-USDF project (19 sec./line). This may be due to a situation where the USDF overhead is compensated for by a reduction in compilation and testing time requirements.

CONCLUSION

Upjohn's overall experience with the use of USDF has been favorable. Programmers are able to begin using the system with only a cursory overview. They are especially pleased to learn that documentation is a natural by-product of using USDF and that they will not be required to produce additional paperwork. We have found it an easy task to define new object, attribute, and relationship types as needed. Further, since USDF is defined within itself, the tasks of making enhancements, adding new tools and new features, and correcting bugs have not been a problem. Future plans for USDF include enhancements to the testing and management areas, the addition of a database design tool, and its use on a larger project. It is expected that as usage of the facility increases, future systems will be better able to take advantage of past efforts and thus show additional productivity improvements.

REFERENCES

1. "Proceedings of a Conference on Application Development Systems." DATA BASE ACM (Winter-Spring 1980).
2. "Programming Work-Stations." *EDP Analyzer* (October 1979).
3. Dolotta and Mashey. "An Introduction to the Programmers Workbench." Second International Conference on Software Engineering, October 1976.
4. Chamberlin, D. *System R: Relational Approach to Database Management*. San Jose: IBM Research, 1978.
5. Teichroew, D.T. *PSL Language Reference Manual*. The University of Michigan, The ISDOS Project, March 1980.

CARL—Experience of an application using clusters

by E. LEVINSON, L. S. LEVY, and J. B. SALISBURY

Bell Laboratories
Whippany, New Jersey

ABSTRACT

A medium size application program (~45,000 lines of source code) was developed using a modular design and Liskov's clusters as the main structuring data mechanism. We discuss the implications of this approach to the overall software development, including the supporting abstractions and software tools needed.

DEVELOPMENT CONTEXT

The CARL programming project was undertaken in a development context where the following constraints applied since the program being developed was part of a larger set of programs. The following constraints applied: First, FORTRAN IV was required to be the official source code and moreover the source code had to be portable to a number of different computers. Second, an existing library of graphic subroutines and character manipulating subroutines were already available within the larger program set. In view of these constraints the development was undertaken jointly using the following machine/ operating system combinations: PDP-11, Honeywell 6000/ GCOS, IBM 370/TSO, IBM 370 /CMS.

Programs were written initially either on the PDP-11 under the UNIX* system⁵ or on the Honeywell time-sharing system. Under the UNIX system all of the UNIX tools were available for software development. Additionally, communication facilities were available to transport programs between these two machines. Source code was in RATFOR, a FORTRAN pre-processor language, facilitating structured code. The IBM environments were used primarily for testing portability once the programs had run in the Honeywell environment.

THE APPLICATION

The application for which this program was developed is the computer production of administrative route layouts (ARLs). ARLs are used by engineering personnel to plan the most effective arrangement of the *outside plant*, i.e., the cable network connecting the central office switching equipment to end

users. Essentially, ARLs summarize the information required by planners and present the information in a form which more directly relates the outside facilities to specific end use locations. An illustrative ARL is shown in Figure 1.

The information presented in the ARLs is obtained from plant location records (plats). An example plat, corresponding to the ARL of Figure 1, is shown in Figure 2. A plat is a map showing the location of vaults (manholes), the cables connecting these vaults, the distances between the vaults, and some information on the types of equipment and interconnections.

The transformation of plats to ARLs is effected by the Computerized Administrative Route Layout Program (CARL). Input to the program is from worksheets summarizing, in list form, the relevant data from the plats. These data are checked for consistency and connectivity to ensure that each cable pair has a *unique* path connecting it to the central office. Additional computations are to determine

1. for any vault, the minimum and maximum lengths of the connections back to the central office;
2. if the network is considered as a directed acyclic graph (dag), what is the direction to be associated with a cable running between a pair of vaults—called the *flow*;
3. information such as change of gauge in a cable, splicings between cables, and the presence of other equipment as shown on the plat.

The CARL program, operating on the data in the worksheets obtained from the plats, transforms these data, performs the auxiliary computations summarized above, and generates a sequence of output commands to a computer output microfilm (COM) device which generates a 35 mm negative of the ARL.

To appreciate the nature of this transformation, let us examine one of the cable vaults in some detail. Figure 3 is an enlarged drawing of a portion of Figure 2. To the left of vault 4 in Figure 3 you can see three sheaths (physical cables) entering the vault. The top one contains 2,100 pairs which are labeled 1, 1-2100 (cable 1, pairs 1 to 2100). Inside the vault 100 pairs, the ones numbered 2001 to 2100, are spliced to the sheath that goes out along DANKO DRIVE. (If you were driving along this street you would see these 100 pairs as a small black cable strung along utility poles. (There are 300

* UNIX is a trademark of Bell Laboratories.

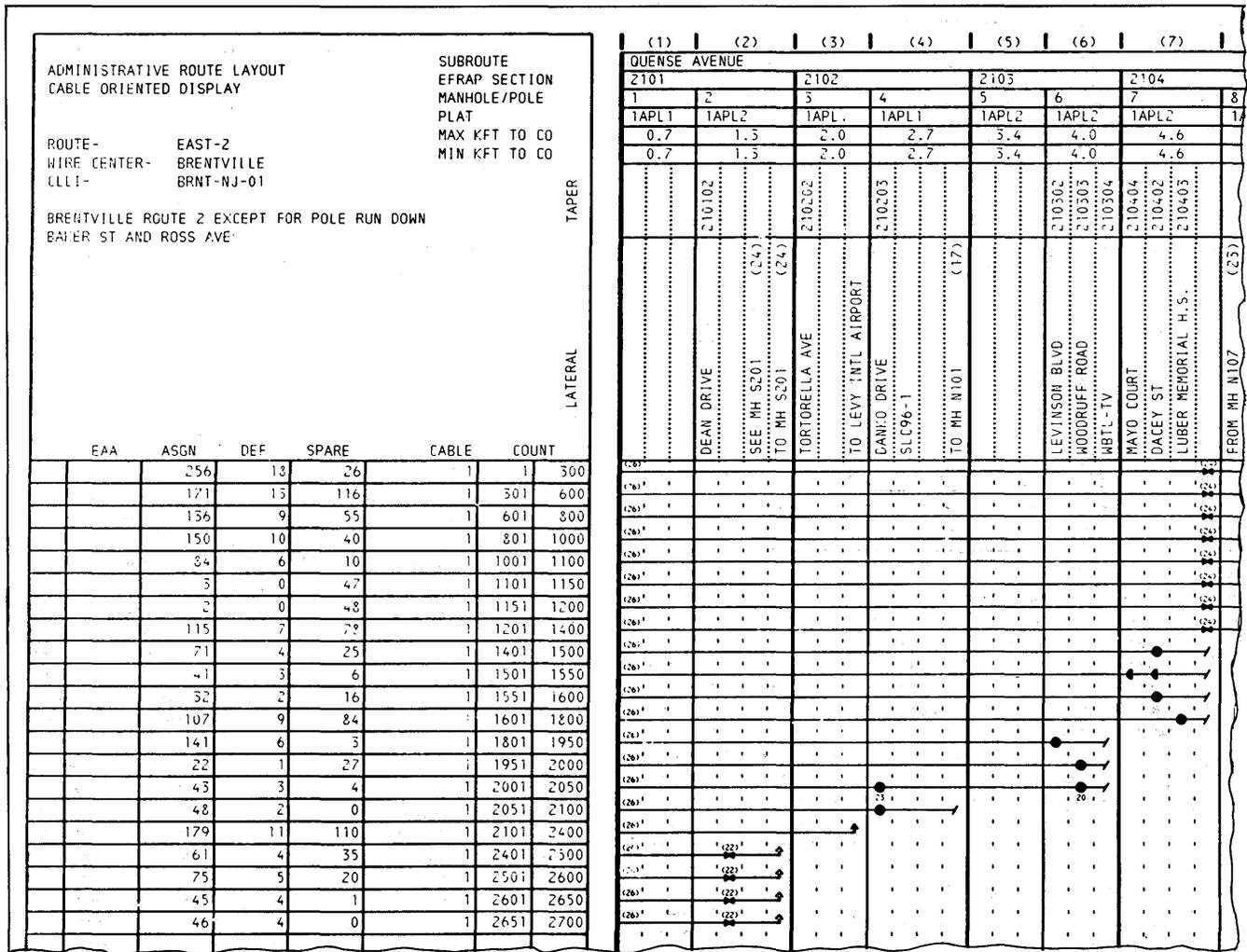


Figure 1—Example administrative route layout

pairs in the cable, the 100 numbered 1,2001- 2100 and 150 pairs numbered 2,1651-1800.) The cable would appear to end (start) at the corner of Danko Drive and the cross street. You would see a vault cover somewhere near this intersection. There is probably a telephone company truck parked right next to it.)

These 100 pairs, 2001-2100, appear in the route layout (Figure 1) as two lines. There is a big dot on these lines in the column labeled DANKO DRIVE. This is under the heading numbered (4) at the top. The other subcolumns represent other details of equipment located at this vault.

The vaults are ordered linearly, as specified by the user. When pairs continue on to non-contiguous vaults a symbol and labeled column are automatically inserted. This occurs to pairs 1 to 1,500 of cable 2 which leave the top of vault 2 and go into vault N101 while the linear display continues with vault 3 (Figure 1).

In Figure 1 the horizontal rows are ordered by cable name and pair. CARL can also order them by displayed line length. This ordering is used by the Telephone Engineer to identify and group together those pairs which serve specific areas. Another ordering is available too. This one is based on the

administrative area designation shown in the leftmost column (ALLOC AREA).

ARCHITECTURE

The CARL architecture had to meet three objectives. The system was to be segmented so that several people could build it, working simultaneously. We wanted to produce results—actual drawings—as early in the development as practicable. It was important, so that we could pass the maintenance on to others, that simple application processing could be expressed by simple, clear code.

Modularization

Segmenting the system implies modularization.² The critical question is “Modularization with respect to what?” For modularizing the data, we chose to use the input worksheet organization and the output data areas as the basis*. The data

* The similarity of this approach to the Jackson design methodology is apparent.

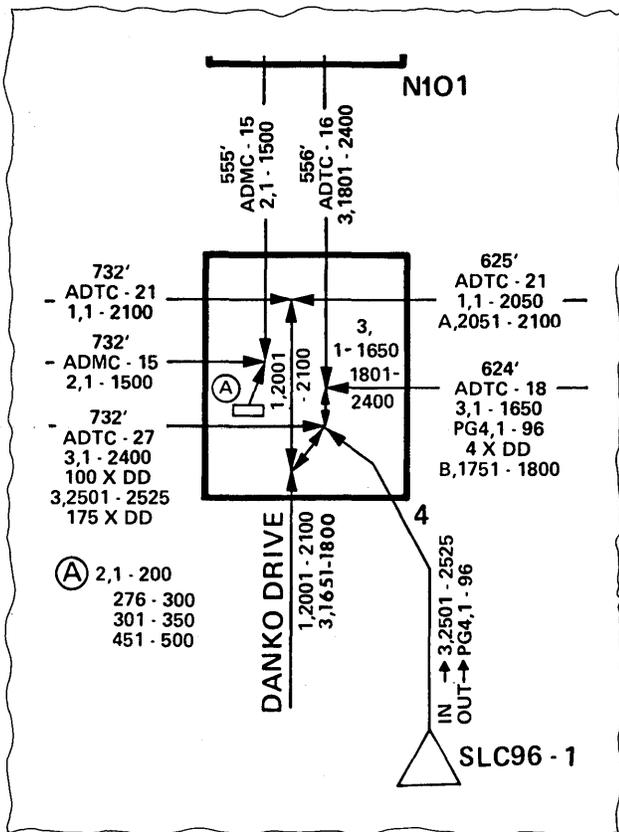


Figure 3—Section of plat for vault 4

This table, SR, contains three fields: position, name, and `sef_list`. Each field is followed by its type, length, and prose description. We used four types: *integer*, *character*, *index* to another table, and *logical*. Field lengths were given in words; for character strings the maximum length was also given. (Character strings were stored left adjusted and padded with blanks.)

The field domains were enumerated or otherwise specified whenever necessary. The domain specifications were symbolic and no attempt was made to reduce them or the field names to constants. This reduction was performed by the RATFOR³ preprocessor. Constants appearing on input worksheet were used. This provided a direct, documented link between the input data and the internal data structures.

Each table cluster also included a dump routine. This was used during development to verify correct program operation. After deployment, these dumps were used to debug the application code. Figure 6 shows a sample dump.

References to the tables appear throughout the application code. These are mostly function and subroutine calls with field names and values as arguments. To produce clear, readable code required names that were descriptive, not merely mnemonic. To insure consistency and to aid the reader we used the following conventions.

1. All field names (attributes) were written in lower case and prefixed by the containing table name in upper case (e.g. `LK_low_pair`)

5 **CARL Subroute Worksheet**
 Include all subroutes - use as many sheets as needed

Route _____
 Wire Center _____
 Page _____ of _____
 Prepared by _____

Subroute Name	Subroute Number	Name (20 characters maximum)
SUBN		
Subroute Section	EFRAP sections in the subroute, moving away from the Central Office (4 digits each, separated by commas)	
SUBS		
SUBS		
Subroute CO end	All EFRA sections, outside the subroute, of its central office end (4 digits each, separated by commas)	
SUBC	Enter 'CO' if subroute is connected directly to CO	
SUBC		
Subroute Field end	All EFRA sections, outside the subroute, of its field (or far) end (4 digits each, separated by commas)	
SUBF		

Figure 4—CARL subroute worksheet

2. Subroutines performing operations were named `<operation>_<table name>` (e.g. `set_LK`)
3. Functions retrieving field values had the form `<type>value_<field name>` where type was I (*int*) or L (*logical*) (e.g. `Ivalue_LK_low_pair`)
4. Subroutines retrieving these values had the form `value_<table name>(<field name>, ...)` (e.g. `value_LK(LK_low_pair, ...)`)
5. Some function names were chosen with an eye towards how they would appear in code. The membership attribute was written `member_<table name>` so that an *if* statement, where it most often appears, would read naturally—*if* (`member_LK(key, ...)`)... Similarly, the iteration control forms for *for* (`i = first_LK(...)`; `cond`; `i = nxt_LK(...)`).

Table Cluster Design

Each table cluster was implemented as a special case of a generalized table. This permitted us to address two problems without affecting the cluster's code. We wanted to protect ourselves from the underlying generalized implementation. (For simplicity, we chose a linear linked list.) If this proved too slow it would be changed. This way the change would be localized. In addition, we wanted to avoid fixing the sizes of each table. To do this within the confines of FORTRAN we implemented the generalized table with a paging scheme. This also permitted us to control the amount of memory the table cluster used.

Under this architecture the application code becomes a series of transformations: from the input string into tables, from one set of tables to another, and from tables into a string of graphics command.

IMPLEMENTATION

Application Code

The success of this modularization depends on separating each table's implementation from the application code. This was done by implementing each table as an independent CLU-like cluster.¹ All operations on, and access to, table elements and fields were via the table cluster. The application code needed only the element's index and the field's name table to retrieve it. A cluster's subroutine and function names and the RATFOR INCLUDE file names required to use them were created using a very restricted syntax:

```
<subr-name > :: = <operation > _ <table-name >
[_ <field-name > ]
<INCL-file > :: =
<table-name > defs
/*name translations */
| <table-name > dcl
/* type declarations */
```

Liskov¹ identifies four types of externally known procedures in a cluster: operations, attributes (field names), iterators, and exception handlers. Our clusters contains the first three. Exception handlers were not built for two reasons. First, FORTRAN's exception handling mechanism is primitive or non-existent and a mechanism which works in one operating system is not likely work in another. Second, the type of exceptions likely to occur (invalid index, table overflow, an attempt to reset a key field, and an attempt to create a duplicate key) permitted only one course of action—STOP. For the last two the cluster wrote an appropriate message and halted. The table overflow problem was side-stepped with a virtual storage allocation scheme. The first exception was ignored, without serious consequences.

Separate implementation of each routine for each cluster would be a tedious and error prone task. Instead an abstract table data-type was defined. This datatype dealt only with records (table elements) and their keys. All references to the table dependent fields were eliminated. Knowledge of how the records were linked to one another was contained within the cluster which implemented the abstract table. In our im-

plementation the tables were stored as simple linked lists. A table index was the virtual address of the first word in the record.

We used a virtual storage scheme for the abstract tables because there were no reasonable limits for the table sizes. Furthermore, with the virtual storage scheme we would not have to manage the individual table limits. The table overflow problem disappears from the clusters and reappears in the storage manager. Here it becomes easy to alter the limits by changing the size of secondary storage used to store swapped out pages.

This design successfully hid the details of a table's implementation from the application code. In fact, the implementation details were hidden at different levels.

Level	Items known at this level
Application	Field, attribute, operator, iterator names and indices
Clusters	Location of a field in a record and the key fields
Table handler	Record size, keyfield location, record linkage mechanism, and virtual addressing
Virtual storage	Location of each word in virtual storage and the virtual storage size

All of this careful structuring effort described above has only one purpose: The application code only should contain application related objects and application related processing; all the rest must be hidden. The following example shows application code which could generate a table describing the drawing's heading—the ordered list of cable vaults. The input data comes from several worksheet and is hierarchically organized. The Subroute worksheet orders the subroutes (highest level) and the elements within the Subroute. This data is contained in the SR (SubRoute) and SEF (Subroute EFRap) tables. The cable vaults within each EFRAP section are ordered on another worksheet and are contained in the EVA (Efrap VAult) table. These tables are scanned with nested loops. The innermost loop adds the current vault to the list for the drawing's heading, the OV (Ordered Vault) table. The vaults are numbered and this number is the key for the OV elements.

```
nOV = 1
for
  (iSR = first_SR(eoSr); !eoSr; iSR = nxt_SR(iSR,
  eoSr)
  {# for-all subroutes
  iSEFp = Ivalue_SR_sef_list(iSR)
  for (iSEF = first_SEF(iSEFp,eoSEF); !eoSEF;
  iSEF = nxt_SEF(iSEF,eoSEF))
  {# the intermediate
  iEF = Ivalue_SEF_ef(iSEF)
  iEVAp = Ivalue_EF_eva_list(iEF)
  for (iEVA = first_EVA(iEVAp,eoEVA);
  !eoEVA; iEVA =
  nxt_EVA(iEVA,eoEVA))
```

Table name	Key length	Description
SR	1	The master list of ordered subroutes for the route database. These data are derived from the SUB-ROUTE worksheets.

Field name	Type	Length	Description
position	int	1	The position of the subroute as it is to be shown on the route layout output. Position 1 is the leftmost position in the display region of the route layout.
name	char	(5,20)	The subroute name.
sef_list	index(sef)	1	Pointer to the SEF sub-list of ordered EFRAP sections associated with the subroute.

Figure 5—Sample Data Dictionary Entry

```
dump of SR
867> 1 PRENTISS 1155
875> 2 ROBERT E. LEE 1164
883> 3 WEST END 1185
891> 4 PARIS 1194
899> 5 HARRISON 1212
```

Figure 6—Dump of SR Table

```

      {# for-all vaults in this EFRAP section
        iVA = Ivalue_EVA_va(iEVA)
        call append_OV(nOV,
          ONE_WORD, iOV)
        nOV = nOV + 1
        ...code...
      } # (EVA)
    } # (SEF)
  } # (SR)

```

This sample code demonstrates that the benefit of information hiding and abstract data types can be achieved in FORTRAN based production programs.

Table Clusters

Information exchange between table clusters and T routines. The generalized table handling routines, called T routines, provide services to the customized cluster routines. The T services are

- get-T
- put-T
- init-T
- search-T
- insert-T
- first-T
- next-T

The details of these T services need not concern us for the moment, nor the particular parameters used in the service. A service can be either a FUNCTION subprogram or a SUBROUTINE subprogram (in FORTRAN terminology).

The essential point is that T services are general table handlers which work on pages in virtual memory, and the table parameters are passed to the T handler (possibly by pointing to a header in which the table parameters are defined). Therefore, any new clusters required by an applica-

```

SUBROUTINE create_XX(segment)
INCLUDE XXenv
INTEGER segment
INTEGER the_key_length, the_element_length
LOGICAL found
  XX_modified = .FALSE.
  XX_ndx = NIL_index
  DO i = 1, 3
    CALL move(' ',1,XX_bks(i),1,4)
    the_element_length = 3
    the_key_length = 8
    CALL DIRget('XX ', segment, found, XX_bus)
    IF(!found)
      CALL init_T(XX_bus(1),'XX ',
        the_element_length,the_key_length,
        T_structure_simple, segment)
  RETURN
END

```

Figure 7—A typical cluster routine

```

SUBROUTINE create_tablename(segment)
INCLUDE tablenameenv
INTEGER segment
INTEGER the_key_length, the_element_length
LOGICAL found
  tablename_modified = .FALSE.
  tablename_ndx = NIL_index
  DO i = 1, bufferlength
    CALL move(' ',1,tablename_bks(i),1,4)
    the_element_length = bufferlength
    the_key_length = charkeylength
    CALL DIRget('tablename ', segment, found,
      tablename_bus)
    IF(!found)
      CALL init_T(tablename_bus(1),'tablename ',
        the_element_length,the_key_length,
        T_structure_simple, segment)
  RETURN
END

```

Figure 8—A typical template

tion can obtain a T service by supplying the appropriate parameter.

What T services do not (and cannot) do is deal with the contents of any table element.

What cluster routines look like

Figure 7 shows a typical cluster routine to initialize the XX table. This routine is particular instantiation of a more general template shown in Figure 8.

The template in Figure 8 is customized to the routine of Figure 7 in various ways:

- XX is substituted for the tablename.
- The values of some parameters of the XX table are substituted for generic parameter names *bufferlength* and *charkeylength*;
- Subprograms at the T level are used, with appropriate parameters.

In more complex cluster routine generation, the routine generated depends on both the type and size of the fields of the elements in a table. The data associated with a given table which describe the field parameters are stored in a data dictionary.

THE CLUSTER GENERATOR

The basic principle on which the cluster generator works is the substitution of specific values for parameter names in the templates. The specific values for all parameters are stored, directly or implicitly, in a data dictionary. A typical entry in the data dictionary is that of the XX table shown in Figure 9.

The global structure of the cluster generator is:

```

for each table in the dictionary
  for each routine required in that table's cluster

```

```

if the routine depends only on properties of the
table
  {substitute table parameters and add the
  routine to the table cluster}
if there is a separate routine for each field
  {for each field in that table substitute table
  and field parameters and add the routine to
  the table cluster}
if there is only one routine but its construction is
field dependent
  {calculate the field dependent portion and
  add it to the template;
  substitute table parameters in the
  modified template and add it to the
  table cluster}

```

The cluster generator makes use of *sed*, the UNIX stream editor, which does the parameter substitution for the templates of the cluster routines. The cluster generator also adds needed definitions and declarations to auxiliary RATFOR files.

Experience with the Cluster Generator

The cluster generator has been used in CARL program development to produce all of the 36 clusters and associated declaration and COMMON files. Currently run on a PDP 11/70 with the UNIX operating system, the cluster generator produces 36 modules, one module for each cluster and its associated files, in approximately four hours. (The cluster generator can run unattended in off hours.)

It is clear that the major objectives of the cluster generator described above have been realized. Extensive testing has provided a high degree of confidence in the correctness of the clusters, which are a significant portion of the overall source code. Many of the tables required in the program were not defined until relatively late, and the cluster generator made the development of the associated software very simple. Finally, in the process of tuning the system the set of templates was modified and significant performance improvement obtained.

An additional advantage of the cluster generator that was not anticipated is the benefit of having "standardized" software. The RATFOR environment on which the programs were initially tested included a CASE statement, but other

RATFOR environments to which the program had to be transported did not include the CASE statement and it had to be replaced by an equivalent *computed GOTO*. It was a relatively simple matter to pipe the output of the cluster generator to a filter which replaced the CASE statements by their equivalents.

EXPERIENCE

The initial version of the completed program, consisting of ~45,000 lines of RATFOR source code was developed by five programmers in approximately 30 programmer/months for an effective productivity rate of ~1,500 lines/person/month.

When the initial version was substantially functionally complete, performance tuning was begun. The performance tuning was facilitated by the layered abstractions used in the implementation. The application code was kept intact but the supporting cluster routines were modified in several ways. First, the most heavily used clusters were rewritten to avoid the paging overhead imposed by the virtual memory and general table management routines. Second, buffering was added to the remaining clusters to minimize these same effects in general. Third, the way that empty tables were handled was modified. The net effect of these three changes was to increase performance by a factor of 4. Additional gains of another 25% through the use of an optimizing compiler have been estimated based on trial compilations of portions of the code.

It is generally claimed that the value of abstraction in programming is that it hides implementation details, allowing a more efficient implementation to be easily substituted if needed. This result was realized by the combined modularization and data abstraction design. A good example of this is the buffering used for table elements. In the initial implementation, an element being updated would be brought into a buffer where several fields could be updated, and then the element would be written out to the virtual memory. With this explicit buffering scheme, an unnecessary write might occur if the next element needed is the one just written. (In compilers, this type of event is eliminated by a peephole optimization or equivalent. However, in this application there is no way to know whether the element should be written out until the next element is selected.)

The buffering scheme was changed as an optimization after the program had been tested, so that any element to be examined is brought into the buffer. There are individual buffers for each table. Elements are not written from the buffer unless they are modified, and they are not written until the buffer is required for a subsequent retrieval.

The change in the buffering scheme was validated by regression testing, and yielded a measurable improvement in performance for negligible programming effort. (Only the templates of a few cluster routines had to be revised—the code generator produced the updated code for the 35 clusters.)

The set of development tools was a nice complement to the underlying abstract design. UNIX shell⁶ programs were used for both the cluster code generation and the error routine formatting as described above. The time sharing system on which the code was initially tested provided useful symbolic

```

tablename = XX
  routines = add create append set value Ivalue Lvalue member
  first nxt fetch
  type = s
  keylength in words = 2
    fieldname = va_1
      datatype = x(VA)
    fieldname = va_2
      datatype = x(VA)
  fieldname = pairs
  datatype = i()

```

Figure 9—Data dictionary entry for the XX table

tracebacks, and symbolic dumps of the tables were written to allow inspection of intermediate results. These two items—the calling sequence and the top-level data structures—were generally sufficient to diagnose any problems. We have described above the multiple system context that was used for program development. Although we were aware from the beginning of the project that portability of the program was a constraint, we had some portability problems—source code that worked on one system but not another.

One method of attacking these problems was to run the source code through the PFORT verifier. The PFORT verifier is a two-pass program. After the first pass, it lists any FORTRAN constructs in the program that are not acceptable to all the compilers that PFORT knows about. The second pass checks the correspondence between subprogram definition and usage. Although there are compatibility problems that can be detected only on the second pass, it is costly to run and suitable for running primarily when the program has been completed. Thus the second pass was not appropriate for program development. Furthermore, PFORT deals only with FORTRAN incompatibilities. In fact, we were transporting RATFOR programs and found several inconsistencies between different versions of RATFOR. (Example, the RATFOR preprocessor that we initially used had a CASE statement; a RATFOR preprocessor on another system did not have a CASE statement.) A second problem that we encountered using multiple systems was the existence of different versions of the program under development on different systems at the same time. This problem arose because we did not initially have a source code control system. In retrospect, we should probably have been more careful with the multiple versions.

The layered approach through the use of clusters was one of the main factors in the success of the project. The integrity of the data structures as insured by this means was virtually 100%. In only one case that we know of did an error occur, and that was due to a faulty implementation of the cluster mechanism rather than “application” code. Thus insofar as the application code was concerned one could guarantee that

if a set of input lists was used to generate an output list, the input lists would remain intact, and further that only the specifically addressed fields of the output list would be affected. The examination of the state of the lists was often sufficient to establish the nature of any program errors.

A somewhat surprising revelation rather late in the development was that part of the specification called for the solution of a problem that defied efforts to find a satisfactory program solution and which on closer examination turned out to be NP hard.⁹ Recall that one desired computation is the minimum and maximum length paths from any vault back to the central office. Abstractly this problem can be stated as follows: Given a digraph, G , with a positive weight associated with each edge, find the maximum and minimum length acyclic paths between a distinguished vertex and every other vertex. The minimum length path computation is easy. But it is not hard to show that the maximum length problem is directly reducible to the Hamilton path problem. Needless to say, we adopted a somewhat ad hoc revision of the specification.

REFERENCES

1. Liskov, B., Snyder, A., Atkinson, R. and Chafert, C. “Abstraction Mechanisms in CLU” CACM August 1977, p. 564-576.
2. Parnas, D. “On the Criteria to be used in Decomposing Systems into Modules” CACM December, 1972, p. 1053-1058.
3. Kernighan, B.W. and Plauger, P.J. *Software Tools* Addison-Wesley 1976.
4. Kernighan, B.W. and Ritchie, D.M. *The C programming language* Prentice Hall 1978.
5. Ritchie, D.M. and Thompson, K. “The UNIX Time-Sharing System” CACM July 1974 p. 364-375.
6. Bourne, S.R. “The UNIX shell” BSTJ July-August 1978, p. 1971-1990.
7. Kernighan, B.W. Lesk, M.E., and Ossanna, J.F. Jr. “Document Preparation” BSTJ July August 1978, p. 2115-2136.
8. Ryder, B.G. “The PFORT verifier” *Software-Practice and Experience* 1974, p. 359-377.
9. Karp, R.M. “Reducibility among combinatorial problems” in *Complexity of Computer Computations* R.E. Miller and J.W. Thatcher, eds., Plenum Press 1972.
10. Jackson, M. *Principles of Program Design* Academic Press, 1975

The software configuration management database

by EDGAR H. SIBLEY

University of Maryland
College Park, Maryland*

P. GERARD SCALLAN

Stanford University
Palo Alto, California

and

ERIC K. CLEMONS

The Wharton School, University of Pennsylvania
Philadelphia, Pennsylvania

ABSTRACT

Software Configuration Management (SCM) is becoming well known in certain sectors of the community (e.g., in Large Scale Data Processing for the U.S. Government) but remains virtually unknown elsewhere. The need for aids to the large scale information systems development process is very well known and documented. This paper deals with SCM as one of the tools for better systems development. It shows an integration of some previous tools into a system that could be implemented on a database management system (DBMS) as an extension to the data dictionary directory (DDD). An architecture for such an SCM system is suggested.

I. INTRODUCTION

Configuration Management was first developed in the mid-sixties as a means of recording, controlling, and systematically modifying the hardware architecture of an information system. The U.S. Government figures largely in the early literature^{1,2,3} and a first book on the topic appeared in 1971.⁴ Yet Software Configuration Management (SCM) was, and still remains, somewhat less understood. The term is normally used to describe a particular set of methods that aid in software specification, design, implementation, and maintenance.

There are many manual and automated aids that have been developed for improving the information systems development process,^{5,6,7} but these tend to be "free standing" and not integrated through the life cycle: They try to solve a problem,

but often do not provide a general context. Moreover, they often ignore the modern concept of database management as part of the process. SCM is a possible context, providing overall management of the process.

In this paper we first motivate, then present, and finally extend the concept of a software configuration management system. We review the traditional software system life cycle, stressing the opportunities for maintaining control over the scheduling, sequencing, and approval of operations, and for maintaining history of system status, tests completed, and approvals received at various checkpoints. We then provide an overview of the data maintained in an SCM system. Finally, we extend the concept of an SCM to an active system, one that not only maintains histories but enforces controls. While this portion—Section IV—is the most important part of the paper, we did not feel that it was possible to present this material without first presenting the background and review of the novel features of SCM that are contained in the first three sections.

A. The Need for SCM

In much of today's small software systems development, problems abound: the system documentation is almost nonexistent, and management of change is *ad hoc* at best. Because software "fixes" or fast updates are needed during the latter stages of the life cycle, the latest version (i.e., the currently running system) often incorporates unrecorded and sometimes even unapproved changes. These changes may be to either procedure definition, database or data structure definition, or both. Naturally, maintenance becomes a shambles. Even in larger systems, the process is often too *ad hoc* in nature.⁸

*The author's current address is Alpha Omega Group, Inc., World Building, Silver Spring, Maryland.

The earliest uses of SCM techniques appear in the various structured design ideas and use of top down approaches. The concept of good *flow of control* occurs in programming work-bench systems and modern operating systems,^{9,10} but even these do not usually provide a means to ensure that the control is linked to the software management process.

B. The System Life Cycles

Since there are many definitions of the phases of the software life cycle, we provide our terminology in Figure 1. We show here two processes in parallel: the software and data life cycles. There are, indeed, many ways of helping or improving the systems process: a 1974 review of procedure specification methods shows that there are a multiplicity of overlapping techniques—some automated and others manual—that cover the process.

The process known as software configuration management incorporated many methods, techniques, and disciplines from software engineering. More significantly, it incorporates them in a framework that permits effective management control of whichever techniques are to be used.

II. THE SOFTWARE CONFIGURATION MANAGEMENT PROCESS

A. SCM Definitions

The *Configuration* consists of the documentation of requirements specification (user needs), functional specifications, and physical design of programs, procedures and data that exist at any time. Thus, the configuration varies with time as specification, design, and implementation change.

The *Baseline* is an existing approved specification of the system. Thus, the configuration will evolve into the baseline as the system moves towards acceptance. But, because the definition of needs changes, the baseline will also change, though at a slower rate than the configuration. The baseline and the configuration are both controlled under SCM. The baseline represents an approved goal or target for the implementation, while the configuration represents its current status.

Change Control is effected through a series of documents (forms) and managerially enforced procedures. It ensures a controlled environment for changes to both the configuration and the baseline. This process entails:

1. Suggestion by user or implementor of the need for a change, made in a special format, with an approval list
2. Review, by a design review board (DRB) of each request
3. Incorporation into the system documentation (whether approved or not—to record the history)
4. Alteration of the configuration or baseline, if the change is approved

The reviews are conducted by either users or the DRB. The reviews will be held regularly as changes are suggested, and at

- | | |
|---|---|
| 0. PERCEPTION OF NEED | |
| 1. requirements specification | |
| 1.A. ANALYSIS OF REQUIREMENTS | |
| 1.B. OBTAINING USER AGREEMENT | |
| 2. DESIGN OF PROCEDURES | 2. DESIGN OF DATA STRUCTURES |
| 2.A. FORMULATION OF ARCHITECTURE | 2.A. DATABASE DESIGN |
| 2.B. SPECIFICATION OF MODULES | 2.B. CODING DATABASE SCHEMA |
| 3. IMPLEMENTATION OF ENTIRE MODULE | 3. INITIAL DATABASE IMPLEMENTATION |
| 3.A. WRITING MODULES (CODING, JCL, DDS, ETC.) | 3.A. WRITING LOAD PROGRAMS |
| 3.B. DOCUMENTATION | 3.B. DOCUMENTATION |
| 3.C. SUBMISSION FOR APPROVAL | 3.C. SUBMISSION FOR APPROVAL |
| 4. TESTING PROCEDURES | 4. TESTING DATABASE |
| 4.A. PERFORMING TESTS | 4.A. LOADING INITIAL DATABASE |
| 4.B. PERFORMING REGRESSION TESTING
(AS NEEDED) | 4.B. WRITING DATABASE TEST
PROGRAMS (MAY COINCIDE
WITH PROCEDURE TESTING) |
| 4.C. OBTAINING APPROVALS | 4.C. PERFORM TESTING |
| | 4.D. OBTAIN APPROVALS |
| 5. INSTALLATION | |
| 5.A. BUILDING LIBRARY FILES AND FULL DATABASES | |
| 5.B. OBTAINING APPROVAL/AUTHORIZATION | |
| 6. MAINTENANCE | |
| 6.A. PROCESSING CHANGE REQUESTS | |
| 6.B. CHANGING DESIGN OF PROCEDURES AND DATABASES | |
| 6.C. CHANGING IMPLEMENTATION | |

Figure 1—Phases of the system life cycle. This life cycle comprises two processes in parallel: the software and data life cycles.

discrete points in the life cycle, e.g., at final designs, installation, etc. One important review will take place when both acceptance test procedures and data are first specified. This overall process is shown diagrammatically in Figure 2.

B. SCM Components—Information Required

The principal items of information required for SCM are:

1. Major initial user requirements for each software product, indexed as necessary
2. Changes (whether approved, rejected or under consideration) to the user needs, with a history of authorizations and complete documentation on rejections
3. Documentation of the system design process, including design of user manuals, programs, procedures (job control, etc.) and data structures
4. Proposals for alterations to the design of database or procedure, recorded in a specified format and with a record of the events leading up to its adoption or rejection
5. The programs (including source and object code), for all versions that have been approved or are in the course of obtaining approval
6. Procedures for running these programs, such as compilation, load and execution command strings and instructions to the operator and ultimate user
7. Test databases for all modules and systems of modules that exist for such versions

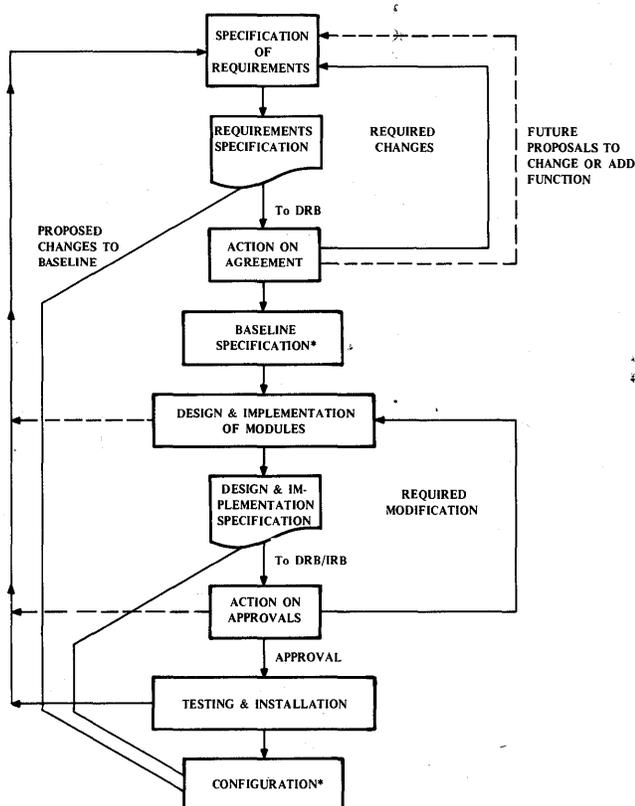
8. All schema and file definitions, user parameters (authorization, etc.) and other items that normally exist in a well implemented Data Dictionary/Directory
9. Descriptions of those other features needed for correctly running the system with high integrity (instructions for database dumps, rollback, checkpoint procedures, etc.)
10. The environment used by all the versions of the system (e.g., operating system and compiler version identification, hardware configuration)
11. A record of the tests applied to each version and their current status

As such, it will be seen that there is some reason for implementing an SCM system (SCMS) on a database management system (DBMS): the need for many entities with complex relationships, the need for excellent reporting and query features, etc., are all satisfied by the features of a DBMS. Moreover, a DBMS normally provides good security and integrity.

III. A SOFTWARE CONFIGURATION MANAGEMENT SYSTEM

Software Configuration Management should be used throughout the system life cycle of any major information system; it acts as a unifying discipline that attempts to:

1. Assure the consistency of the components for procedure and data (requirements, design, implementation, and



*It is intended that both the baseline specification and the configuration have components in a format available to humans, as well as their machine-readable components.

Figure 2—The overall SCM process

- documentation) at each point in the life of the system
2. Allow adequate status reporting and management control during the development process
3. Assure that all the implications of proposed changes are visible prior to their adoption and that proper test procedures follow their implementation in the software prior to their adoption
4. Record the development of the system, both initially and through subsequent modifications, thus maintaining an audit trail and ensuring that people are held responsible for their actions

A. Specifying Requirements

The first phase of the large system software life cycle is the Requirements Specification, which is usually the responsibility of a group of systems analysts acting in close cooperation with the ultimate system users.

An SCMS demands that the requirements be structured hierarchically and that an *Author* be associated with each section. The Author is the individual or group leader of a team responsible for writing, modifying, and maintaining that section, and ultimately for assuring that the system design fulfills the requirement.

Normally, requirements are not static: They will be modified in response to changes in user needs and unforeseen difficulties or conflicts in meeting the requirements during design and implementation phases. Changes in the Requirements Specification should be made only with correct authorization from a *Requirements Reviewer* following circulation of a *Requirements Change Draft* to those possibly affected by the change to the baseline. It is clear that several historical versions or baselines of the Requirements Specification will exist. This is done partly to support subsequent audit of the evolution of the baseline specification. Additionally, this is done so that the requirement specification corresponding to the existing configuration will remain available despite evolution of this specification into a later baseline.

The functions of the Author and Requirements Reviewer may be performed by teams, committee-type groups, individual managers, or some combination of these, depending on the appropriate balance between flexibility and representation of all relevant interests.

B. Designing the System

At the center of the development process stands the design phase—the transformation of the user-oriented Requirements Specification into program, database, and file description, documentation outlines, and testing criteria that serve as the guide to the system implementation. Here, we define the design phase to embrace the entire group of system components, including:

1. Programs and their documentation
2. Data-structures and relevant dictionary entries
3. Operating procedures—both JCL and operator instructions

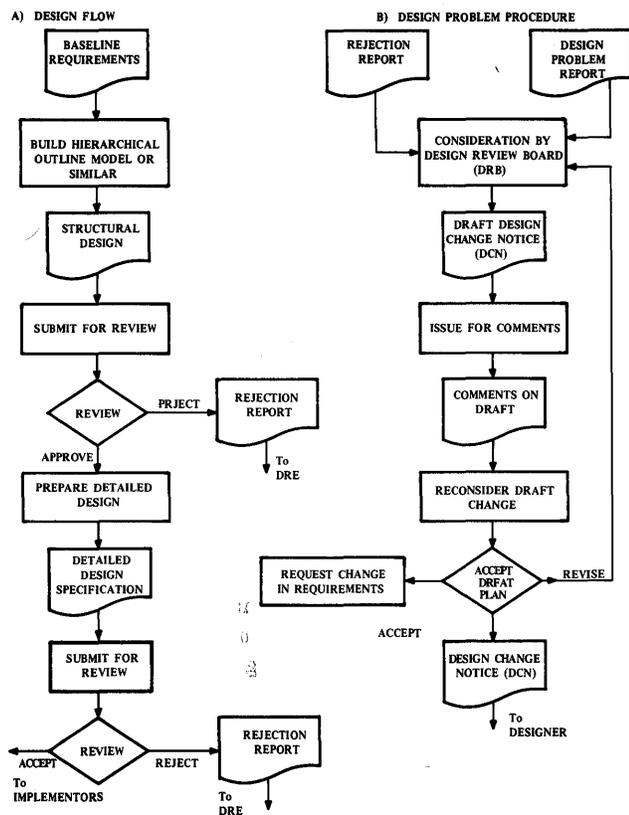


Figure 3—Design phase of an SCMS

4. Testing data and procedures
5. Documentation for the guidance of the user and for maintenance
6. The operating system and ancillary software

The System Design Phase consists of two distinct parts. The first is the construction of a hierarchy of design modules using top-down techniques. This specifies the architecture of the system by describing the functions to be performed by each module and by the modules with which it interfaces. Each module also references units of the Requirements Specification that it is intended to fulfill. Thus, the completion of Stage I of the design provides a skeleton on which the system is hung and sets up the principal mechanism through which the possible effects of modifications may later be traced.

Stage II of the Design Phase consists of the elaboration of the modules and data structures established in Stage I, to produce a detailed specification of the implementation consistent with the Requirements Specification and Stage I architecture. At the lower levels, these modules are the basic working documents of the implementors. They may be written in a formally defined syntax such as a Problem Definition Language, but this is not necessary; it is a management option.

A module* originates when it is specified by a Designer operating at a superior level in the hierarchy. It is assigned to a Designer for Stage I specifications; a Designer may be an

individual or a group of analysts. Upon completion of Stage I Design, the module is entered into the SCMS. It should now be reviewed for acceptance by:

1. The Designer of its superior in the hierarchy or its calling program if a network
2. The Designer of each module with which it interfaces, to ensure compatibility
3. The data administration group
4. The Author of each Requirement unit that it fulfills

The module may also be subject to other reviews, for example, to ensure adherence to documentation standards. Each Reviewer is required to issue an Approval or a Rejection Notice (with reasons) within a stated period. If only approvals are obtained, this version of the module is accepted, otherwise it is referred back to a Design Supervisor who arbitrates conflicts and who may require module redesign.

Changes to a previously accepted version of a design module may arise because of altered requirements or because of later problems in the implementation. In the latter case, the problem is recorded in a *Design Problem Report*. This is considered by a Design Review Board or Design Supervisor, which decides which units are affected and causes a Draft Change Notice to be issued. It is circulated and comments are solicited. The Design Supervisor may have the draft revised and repeat the process or may issue the draft as a *Design Change Notice*. The Designers of the affected modules are then required to prepare new versions, which are subject to the same acceptance procedures as the originals. If the data structure design does not comply with that of the data administrator, its redesign will be requested. Changes in the Requirements Specification are treated in the same fashion. This process is illustrated in Figure 3.

Upon approval of a Stage I module version, the module is referred to a Stage II Designer, who may or may not be the same as in Stage I. The process is repeated, except that the Stage II design is also subject to review by the Designer of the corresponding Stage I module version.

During the design of a new system, the data administrator must coordinate the structures required by the various modules and produce the database design and schema to satisfy their needs. When modules are being added to an existing system, the data administrator must assure that needs of the new module are met, without compromising the needs of the existing system.

C. Implementation and Testing

A design module may contain specifications for one or more implementation modules. Upon acceptance of the design module (in a particular version), each implementation module is assigned to an *Implementor*—an individual or team responsible for programming or technical writing. The Implementor writes a version of the *implementation module*, which is a collection of software elements and/or units of documentation closely related and best tested as a unit.

A parallel effort by the data administration staff involves

* In the material that follows, we shall use the term *module* to mean program or load module, comprising both executable procedure and data specification.

writing database load or conversion routines, preparing database documentation, and providing of test data.

Upon completion of an implementation module version, it is submitted for testing and review. Each version of a module must be subject to complete testing, in accordance with the criteria laid down by management, as well as the specific criteria determined by the Designer of the implementation module specification. First, the module is tested on a stand-alone basis: Do its source modules compile correctly? Are any illegal references made to other modules? Does it pass the implementor test procedures? Next, it is tested in conjunction with those other modules that use it. Such test procedures are an important part of the design and implementation process, and some implementation modules are primarily used for testing others. Implementation modules are also reviewed in much the same manner as design modules; i.e., by the implementors of interfacing modules and the designers of the general specification. The review may take the form of a structured walk-through or some less formal sign-off procedure, depending on the management and system analyst policy.

Acceptance of an implementation module occurs only when all the necessary tests have been successfully completed and all approvals obtained. Failure of a test, or denial of approval, results in a *rejection notice*, which refers the module back to the implementor, the data administrator, or both, for revision and further testing. The creation of a new version of a design module requires that each dependent implementation module be revised: This process occurs when a new baseline has been specified, and naturally the process of moving to a new configuration that reflects this change in requirements is time consuming. In fact, for major software projects, there may be several versions of the baseline and many implementation versions of the configuration all coexisting.

Implementation problems arising subsequent to acceptance are considered by an Implementation Review Board (which may contain the same personnel as the Design Review Board). They may refer the problem to a systems analyst if the problem solution appears obscure. After consideration, they may issue an Implementation Change Notice, requiring alteration to the modules in some specific way, or they may return the problem to the Design Review Board through a Design Problem Report. The process is illustrated in Figure 4.

IV. AN ARCHITECTURE FOR AN ACTIVE SCMS

An automated software configuration management system may be thought of as either active or passive. A *passive* SCM system aids the manual process of configuration management by storing data on baselines and configurations, on approvals or rejections and other aspects of the development process, and by permitting subsequent query or audit. An *active* SCM system does this and more: It provides some element of control by enforcing at least some management policies, and thus makes avoidance or contravening of management decisions more difficult. To take a single, simple example: an active software configuration management system might make it impossible to replace a library copy of a module without demonstration that all recorded tests had been successfully run and

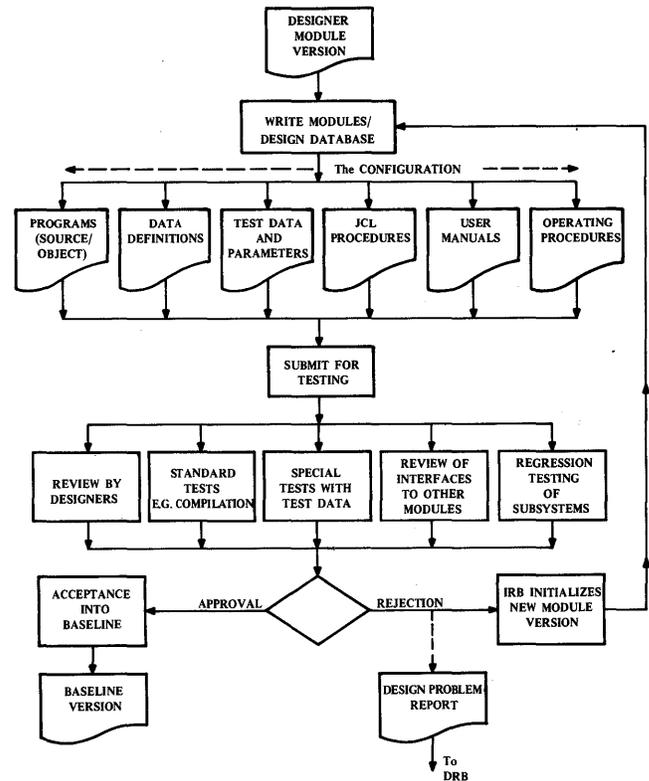
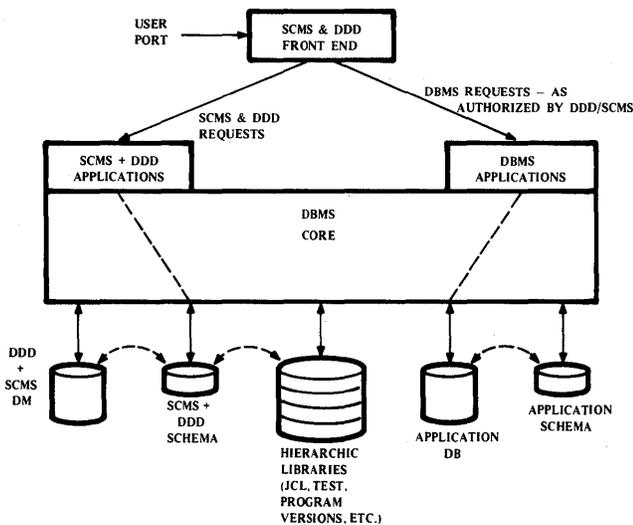


Figure 4—Implementation and testing of a module

all necessary approvals received. Clearly, the cost of a SCM system will depend on its capabilities. Costs will be greater for an active rather than a passive system; similarly, for an active system, costs will increase as the number of functions performed.

We note the strong symbiotic relationship between the SCMS and a database management system with a data dictionary/directory (DBMS with a DDD). In fact, we believe that in the future effective information resource management systems will contain aspects of SCM and DDD, and that it will be difficult to separate the two or distinguish between their functions.

Currently, DDD systems are known alternatively as data dictionaries, data directories, or information resource management systems.¹² There are, at present, two approaches to their implementation. The first involves a stand-alone product, a data dictionary system that does not require a database management system for its own operation, although of course its operation extends to the management of data in applications databases. The second involves a data dictionary system closely linked to a database management system, using this DBMS to support its own operations. There are reasons that justify each approach. Users of stand-alone dictionary systems are less likely to be locked into their selection of database vendors by their choice of a dictionary system. However, implementation of a dictionary linked to a DBMS enables the dictionary to benefit from many of the strengths of the DBMS. Among these are programming language and query language interfaces for examining the dictionary, security and access control, and integrity controls.



NOTES:

1. All DBMS Requests are controlled (validated, etc.) through SCMS & DDD.
2. All updates to application systems are controlled through SCMS and recorded in DDD prior to writing/testing through Libraries.

Figure 5—An active SCMS architecture

In addition to the distinction between DDD systems that are stand-alone and those that use a DBMS, we make a distinction between those that have a passive interface to the data and those with an active interface. A DDD with a passive interface stores information on data—access rights of individuals, acceptable values or transitions between values, who is responsible for particular data items—and permits query of this information. Thus a passive system stores data policy and permits inquiry about this policy but makes no attempt to enforce this policy. A DDD with an active interface goes further, prohibiting invalid database accesses or updates.

Thus, we have seen that a data dictionary system can be either stand-alone or can employ a DBMS for its own use, and can either have an active or passive interface to the applications programs and data. In some active systems employing a DBMS the DBMS appears dominant—it is the DBMS's conceptual schema that stores policy and the DBMS's database control system that enforces it. In others, the data dictionary system appears dominant. We feel that as more integrated and advanced systems are developed this distinction will become less important. Commercial systems with data policy enforcement are now becoming available.¹³

We have been actively studying the design of an active software configuration management system, one that enforces management policy and controls upon the life cycles of a large system development effort. This active SCMS employs a data dictionary/directory system, implemented using a database management system, and processing active data maintenance interfaces. This architecture is illustrated in Figure 5; it can be seen as an extension to the data dictionary or as a piece of general purpose software that replaces it and is roughly twice as complex and costly. We favor this architecture and the active approach to software configuration management because, despite our faith in humanity, we have little faith in its individuals, programmers or otherwise, to police themselves.

In developing the SCMS there are several important points that must be made relative to the DBMS and OS interfaces:

1. If the SCMS is to store all useful information about the programs, testing, and parameters associated with previous versions of the system, then it must have special database libraries that are very large and may be unusual in format by normal DBMS standards (i.e., they may contain code, JCL statements, link editing information, and even operating system versions)
2. If the SCMS is to play an active role in testing and controlling the interfaces for the different hierarchies of libraries and in setting up the run parameters of previous configurations for retesting, or other operations, then there will probably need to be some operating system modifications, although these may only be at the DBMS-OS interface, and may not be very extensive

As an example of the operation for an SCMS, consider the request of a user to test a piece of software and to promote it to system status if approved. We assume that the programmer has already tested this module, and that the systems testing engineers have provided their tests for both the module and for the system containing the module. If the new module is a change to an old module due to improvements needed to the entire system, the new tests may be identical to the old ones, or they may be augmented.

The procedure at a high level is then:

1. Request to test received by SCMS
2. Request to validate user from SCMS to DBMS to SCMS application interface
3. SCMS requests module testing alone. This involves retrieval of linking and editing programs, with call modules, the JCL stream for the test, and the specific module test data.
4. Module is promoted to next level library (i.e., it was in the programmer's library, it now goes to the separate module library).
5. The SCMS requests testing of the module in its various uses; if, for example, the module is a low level subroutine, it may be called by several different and separate modules. Each test is made by selecting the relevant modules from the SCMS applications database and link editing them, calling the JCL and test data and comparing the results with the correct values.
6. The SCMS promotes the module to the active library, and all link editing for various uses is modified to incorporate that version, after which the previous module is deleted from the active library. However, the previous version will still be available in the SCMS-DB in case it is required after some undetected bug brings down the new system.

Those who have carefully reviewed the inner workings of a DDD will see many similarities. Indeed, the DDD often contains many items that the SCMS is expected to contain, e.g., the participating persons, with identification attributes, the programs and databases, with their components and interactions. Thus, the active SCMS is a special extension, though

very significant, of the DDD. Moreover, it can be considered an application program running on the DBMS. Note that under those circumstances the SCMS is also exerting substantial control over the DBMS.

V. CONCLUSION

A preliminary design has been made of an active SCMS, as specified in the previous section. This includes a first round design of the schema for the SCMS, assuming that the whole will be implemented on a CODASYL-type DBMS. The use of a system that contains an active DDD interface has been found to reduce the effort. The need to provide multiple level libraries for software and test data does not appear to be too difficult, although this does require some modification to the calling of standard operating system functions (i.e., intercepting calls). Once again, we see that the standard operating system and DBMS interfaces in their present forms are not well architected for modern software production.

The production of quality software has been one of the major goals of software vendors and large application software producers. It is probably true that no one area has given more misery to modern organizational management. The area known as Software Engineering is still in its infancy; it is hoped that as the area matures and develops it will substantially reduce the difficulty of software production.

We propose here a framework for better software production, and in this our goals are somewhat the same as the Programmers' Workbench projects that have been proposed and implemented. We feel that the problems of software production are more fundamental than simply providing a better working environment for programmers and analysts, though we believe that tools to improve the working environment will of course be of some value. We believe that tools to provide better management control, to provide auditability of evolution of both specification and implementation, and to assure that management policy is followed will be of even

greater value. These tools impose some control over programmers and analysts, rather than inspire them to still greater flights of fancy.

An active software configuration management system is intended to provide an environment that supports as much control as management believes necessary. The manager who believes in an open shop with no holds barred may allow the controls to be loose or non-existent. We feel that such managers should be allowed to exist (but not long as managers); we believe that most managers would relish the ability to know about and control the process of software development.

REFERENCES

1. Department of Defense, "Configuration Control—Engineering Changes, Deviations, and Waivers," #MIL-STD-480, October 1968.
2. Department of Defense, "DOD Automated Data Systems Documentation Standards," Instruction #7935.1, September 1977.
3. General Accounting Office, "Contracting for Computer Software Development—Serious Problems Require Management Attention to Avoid Wasting Additional Millions," FGMSD-80-4, November 1979.
4. Czerwinski, F. L. and T. T. Samaras, "Fundamentals of Configuration Management." John Wiley & Sons, 1971.
5. Sibley, E. H., "A System Specification Language," Infotech State of the Art Report #19, Commercial Language Systems, 1974, pp. 475-503.
6. Ross, D. T. and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition," IEEE Transactions of Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 6-15.
7. Bell, T. E., D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer Aided Software Requirements Engineering," IEEE Transaction on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-60.
8. Brooks, Jr., F. P., "The Mythical Man-Month. Essays on Software Engineering." Addison-Wesley Publishing Co., 1975.
9. Morrison, J. P. "Data Stream Linkage Mechanism," IBM Systems Journal, Vol. 17, No. 4, 1978, pp. 383.
10. Washey, J. R., PMB/UNIX Shell Tutorial, September 1977.
11. Teichrow, D., "Improvements in the System Life Cycle," Information Processing 74, IFIP Congress Proceedings, pp. 972-978.
12. Lefkovits, H. C. "Data Dictionary Systems," QED Information Sciences, Inc., p. 450.
13. Nijssen, G. M., Private conversations, August 1980.

Euclid—A language for compiling quality software

by DAVID B. WORTMAN, RICHARD C. HOLT, and JAMES R. CORDY

University of Toronto
Toronto, Ontario, Canada

and

DAVID R. CROWE and IAN H. GRIGGS

I.P. Sharp Associates Ltd.
Toronto, Ontario, Canada

ABSTRACT

This paper discusses the design and implementation of a production-quality compiler for the programming language Euclid. Euclid is a Pascal-based system implementation language that has features to aid in the production of well-modularized, verifiable system software. The emphasis in this paper is on the insights that were gained in programming language design and implementation as a result of implementing this compiler.

INTRODUCTION

The programming language Euclid^{1,2} is a system implementation language that was designed for writing *verifiable* system software. Euclid extends Pascal to provide language features necessary for the construction of system software (e.g., compilers, operating systems, and message switching systems).

The authors of this paper have been involved in the design and implementation of a compiler for Euclid.^{3,4} This compiler is now operational on large PDP-11 computers under the UnixTM time-sharing system. This paper describes our experience in the design and implementation of a compiler for Euclid. Euclid contains many features that touch the limits of the state-of-the-art in programming language implementation. We comment on the difficulties that we encountered implementing these features. We describe the organization of our compiler and discuss the quality of object code that we were able to produce. We hope that these observations will be of benefit to future programming language designers and implementors.

A BRIEF HISTORY OF EUCLID

The design of the programming language Euclid was originally commissioned by the Defense Advanced Research Projects

Agency of the U.S. Department of Defense. It was intended as the implementation language for a provably secure operating system. Euclid was designed by a committee consisting of Drs. B.W. Lampson and J.G. Mitchell from Xerox Palo Alto Research Center, Prof. J.J. Horning from the University of Toronto, Prof. G.J. Popek from UCLA and Dr. R.L. London from the USC Information Sciences Institute. Prof. J.V. Guttag from USC also provided considerable assistance in the later stages of the language design. The first report defining Euclid appeared in the February 1977 issue of SIGPLAN Notices.¹ Popek *et al.*⁵ discuss the design of the language. London *et al.*¹² present a set of proof rules for Euclid.

The Euclid compiler described in this paper was begun in mid 1977 as a joint project of the Computer Systems Research Group at the University of Toronto and the Special Systems Division of I.P. Sharp Associates Ltd. The development of the compiler was funded by the U.S. Department of Defense and the Canadian Department of National Defence. The design and implementation of this compiler required 6-8 man years of effort spread over 2 1/2 calendar years.

Development of the compiler proceeded in stages both to accommodate our need to bootstrap the compiler on an existing system and to give us time to study and understand the complexities of the language. We began by defining a restricted subset of Euclid called Small Euclid. This was the smallest subset of Euclid that was both large enough to be useful for writing a compiler and small enough so that it could be trivially transliterated into an existing programming language such as C.⁶ A transliterator was written that transformed Small Euclid programs into the programming language C. We then used the existing C compiler to produce executable code. This artifice made it easy to bootstrap the early stages of the compiler. More importantly, it made us start programming in Euclid from the beginning thus forcing us to learn the language thoroughly. This early immersion in Euclid helped us avoid unanticipated problems later in the project. The Small Euclid transliterator became operational in December 1977.

*UnixTM is a trademark and servicemark of Bell Telephone Laboratories, Inc.

Once the transliterator was working, we set out to make ourselves independent of the C compiler. We specified a larger subset of Euclid called Middle Euclid and wrote a translator that transformed Middle Euclid programs into PDP-11 assembly language. This translator was written in Small Euclid. The Middle Euclid translator became operational in June 1978. At this point the transliterator could be discarded. The remainder of our project was devoted to increasing the subset of Euclid that we could compile by implementing more language features, especially those features that were needed for verification. The subset compiled by the current compiler is called Toronto Euclid.

EUCLID LANGUAGE FEATURES

This section describes some of the language features that distinguish Euclid from Pascal and from other system implementation languages.

Explicit Control Over Name Visibility

Most programming languages in the Algol/Pascal family of languages use the scope rules of Algol-60, i.e., an identifier is visible in the block in which it is declared and in all contained blocks. The visibility rule in Euclid is much more restrictive. An identifier is visible in the scope in which it is declared. If it is to be visible in contained scopes then it must be explicitly imported into those scopes via an *import* declaration. Identifiers declared in modules (see below) are visible outside of the module if and only if they are explicitly exported from the module by an *export* clause. The qualification *pervasive* attached to the declaration of a constant, type or routine (procedure or function) causes the object so qualified to be automatically imported into all contained scopes.

Explicit Control over Access

In most programming languages the ability to reference a variable implies the ability to assign to it. The designers of Euclid gave the programmer more explicit control over where variables can be modified. Control over access is supplied at points where a variable is being imported or exported. If the programmer makes no explicit declaration then an imported or exported variable has the attribute *readonly*; it may be read, but not modified. In order for a variable to be modifiable it must explicitly be given the attribute *var*.

Generalized Types

Euclid includes a number of generalizations on the Pascal concept of type.

Type equivalence

Pascal uses a very simple and very restrictive rule for determining when two types are equivalent; two types are equiv-

alent if and only if they are derived (directly or indirectly) from the same type definition. This rule for type equivalence is called *name equivalence*. Euclid uses a much more general *structural equivalence* rule to determine when two types are the same. Briefly, two types are equivalent in Euclid if they have the same structure and if corresponding values in the two type definitions are equal. See the Euclid Report² for a more detailed description of this rule.

Type safety

Euclid is, with one carefully controlled exception, completely type safe. A Euclid compiler can always determine the correct type of every variable and constant. Because Euclid is type safe, a compiler for Euclid can do a very stringent check of the entire program for correct use of variables and constants. We feel that this very strong type checking helps eliminate many common programming errors.⁴

Parameterized types

Pascal allows the definition of types in programs. In Euclid this concept was generalized by allowing type definitions to have parameters. A *parameterized type* acts like a template that defines a family of types. Each time that a parameterized type is used to create a specific type (an *instance type*) actual parameters are supplied corresponding to the formal parameters declared for the type. These actual parameters are substituted into the template to create the instance type. See Holt and Wortman⁷ for a description of implementation techniques for parameterized types.

Non-manifest types

In Pascal all types are static, i.e. all the characteristics of the type are known at compile-time. In Euclid, type definitions can contain values that are *non-manifest*, i.e., the values are not known until the scope containing the type definition is entered during program execution. Different executions of the same scope may create different definitions for the same type.

Nested types

In Euclid type definitions can be lexically nested. For example, definitions for scalar and aggregate data types can occur within the definition of a module type. Inner type definitions can use identifiers that are defined in outer type definitions. The nesting of types, especially module types required new implementation techniques that have been discussed in more detail elsewhere.⁷

Opaque types

To support the concepts of data abstraction and information hiding, Euclid introduced the concept of an *opaque type*. All

types exported from modules are opaque in the sense that the internal representation of the type cannot be determined by a client of the module. Specifically, opaque types are never equivalent to any other type even though they would be so under the Euclid type equivalence rule. Type opacity is necessary if modules are to be treated as plug-replaceable units of programs.

Modules

The module mechanism in Euclid serves two purposes. It allows abstract data types to be constructed and it provides the facilities required to do information hiding, as suggested by Parnas.⁸ An example of a Euclid module (taken from the storage allocation pass of our compiler) is given in Figure 1. This module implements a stack of integer counters. The implementation of the stack is hidden within the module. A client of the module has access only to the routines and variables that are exported from the module. Note the use of *const*, *readonly*, and *var* to control access to variables both within and outside of the module. For efficiency reasons, the top entry in the stack is implemented as a scalar variable. The information hiding provided by the module mechanism makes this implementation detail invisible to clients of the module. In fact, this module can be replaced with a simpler module that uses an array element for all of the counters and implements *top* as a function that returns the value of the top item in the stack. Such a replacement would be entirely transparent to clients of the module.

Aliasing

Euclid prohibits the aliasing of variables, i.e., the situation where two distinct identifiers refer to the same storage location. This allows program verifiers to assume that an identifier refers to a single distinct object. Most mechanical verifiers need to be able to make this assumption. We found that some very obscure programming errors were detected by the compiler as a result of this enforcement of the non-aliasing rules. For further details see Wortman and Cordy.⁴

Machine Dependent Features

Since Euclid is to be used for writing system software, the language designers included features for specifying machine dependent aspects of programs. Euclid has a special kind of record called a *machine dependent record*. Fields in such a record are given explicit offsets, and, optionally, explicit bit positions within a unit of storage. Euclid also allows the programmer to declare variables at absolute addresses. Routines whose bodies are *code* blocks allow use of assembly language inserts in a way that is consistent with the rest of the language.

Euclid requires that a module using machine dependent features be itself labelled machine dependent in order to document the internal machine dependency.

```

var countStack:
  module

    { Count Stack Semantic Mechanism }
    imports (Error);
    exports (readonly top, { top value in count stack }
            IncrementTop, DecrementTop, Push, Pop, SetTop);

    pervasive const countStackSize := 25;
    var csp : 0 .. countStackSize := 0 { count stack index }
    var count : array 0 .. countStackSize of SignedInt;
    var top : SignedInt; { top value in count stack }

    { for reasons of efficiency, the top value in the count
      stack is held in the variable top, rather than in the
      array count. However to simplify the logic, csp is
      treated as if count(csp) were used even though it is
      not. count(0) is an unused dummy.
    }

    procedure Push(const newCount : SignedInt) =
      imports (var csp, var count, var top, Error);
      begin
        if csp = countStackSize then
          Error;
        end if;
        count(csp) := top; { ok even if csp = 0 }
        csp := csp + 1;
        top := newCount;
      end Push;

    procedure Pop =
      imports (var cap, var top, readonly count, Error);
      begin
        if csp = 0 then
          Error;
        end if;
        csp := csp - 1;
        top := count(csp); { ok even if csp = 0 }
      end Pop;

    procedure IncrementTop =
      imports (var top, readonly csp);
      pre (0 < csp);
      begin
        top := top + 1;
      end IncrementTop;

    procedure DecrementTop =
      imports (var top, readonly csp);
      pre (0 < csp);
      begin
        top := top - 1;
      end DecrementTop;

    procedure SetTop(const newValue : SignedInt) =
      imports (var top, readonly csp);
      pre (0 < csp);
      begin
        top := newValue;
      end SetTop;

  end module { countStack };

```

Figure 1—Example of a Euclid module

Legality Assertions

Legality assertions in Euclid are compiler-generated Boolean expressions that describe necessary and sufficient conditions for the execution of a given statement without violation of the semantic constraints of Euclid. For example, legality assertions might be generated to guarantee that an array subscript is in range or that the evaluation of an arithmetic expression produces the mathematically correct result (e.g. no arithmetic overflow occurred). Legality assertions are important because they make all of the possible semantic errors in a program visible to the programmer and more importantly, to the program verifier. Wortman⁹ discusses the issues involved in implementing legality assertions in Euclid.

HARD PROBLEMS IN THE IMPLEMENTATION OF EUCLID

This section discusses several of the difficult problems that we encountered during the implementation of Euclid.

Generality

As discussed above, the design of Euclid generalized on features in other programming languages in many ways. This generality forced us in many cases to devise new implementation techniques to cope with these more general features.

Feature Interaction

In the process of implementing our compiler, we found several cases where unanticipated interactions among language features lead to intractable implementation problems. The Euclid committee worked very hard with us to resolve these problems. A few of the problems we encountered are sketched briefly below.

Type equivalence vs. everything

The very general rule for type equivalence had a number of unanticipated consequences. For example, it constrained the way that storage could be allocated for non-manifest types because these types might be equivalent to completely manifest types. The type equivalence rule also made it difficult to define the semantics of assignments and of parameter passing.

Initialization in types vs. generality

It is possible as a part of a Euclid type definition (e.g. a record type) to specify that certain variables in the type have initial values. Because Euclid also allows the programmer to declare data structures containing such types, a Euclid compiler has to be able to generate arbitrarily complex code to initialize components of these data structures. For example, if

a programmer declared an array of records where some of the fields in the record had to be initialized then the compiler would have to generate a loop to initialize all of the elements of the array.

Nested and multi-use modules

The ability to nest module definitions and the ability to create multiple instances of the same module type created many problems in defining the semantics of modules and in producing an acceptably efficient implementation. These problems are discussed in more detail by Holt and Wortman.⁷

Variant records vs. parameterized types

In order to create a type-safe variant record mechanism, the designers of Euclid required that the tag field of the variant record be specified as a formal parameter of a parameterized type. This was an unfortunate linking of two otherwise distinct language features. It led to a number of difficulties in specifying the semantics of variant records, especially in cases where variant record declarations were nested.

Lack of implementation restrictions

There were several instances in which the original definition of Euclid¹ lacked semantic restrictions which would allow an efficient implementation. For example, a variable could be declared with a type that specified a subrange of integers, for example, *e1* .. *e2*. Unfortunately, *e1* and *e2* could be non-manifest constants so that the actual subrange would not be determined until the program was executed. The problem that arose was that the subrange could result in a variable that was either a signed or an unsigned integer; different machine instructions would be required in these two cases. This problem was solved by requiring that non-manifest subranges result in signed integer variables so that the compiler could know which kind of code to generate.

COMPILER ORGANIZATION

The design of the compiler was heavily influenced by the following considerations. The Euclid language was newly designed and rather complex. The compiler had to be small enough to run on a minicomputer and yet be able to handle large programs. The compiler had to be quite reliable. It had to be relatively portable, so that without too much effort it could be run on another computer or could generate code for another computer.

New Implementation Techniques

A number of new compilation techniques were used in developing the compiler. The new model we developed for rep-

resenting types and modules⁷ was used to design the symbol and type tables and the run-time addressing of type descriptors and module variables.

Another new technique was the use of a notation called S/SL to formally specify the syntax of interpass streams.¹⁰ Each of the major passes of the compiler beyond the Scanner is driven by an S/SL program that parses its input stream (produced by the previous pass) and emits an output stream for the succeeding pass. (Each S/SL program is translated to a table of integers that is interpreted by a procedure written in Euclid.)

Passes of the Compiler

The compiler consists of six major passes plus three minor passes. These were designed to fit in limited memory and to capture functionally coherent tasks within compilation. The major passes are: Scanner, Parser, Builder, Conformance, Allocator and Coder. Each pass is run sequentially as a separate task. The passes communicate via disk-resident symbol and type tables and a serial I-code stream (see below).

The Scanner divides the source Euclid program into tokens, evaluates numeric literals and replaces identifiers by unique numbers. The text of identifiers is stored in a disk resident "name table" for use by later passes. The Parser performs the traditional task of validating syntactic structure.

The Builder creates disk resident symbol and type tables based on declarations in the source program; it also enforces Euclid's import/export rules as well as the rules of access to values. The Conformance pass does type checking and constant folding, together with a number of tasks required specifically by the Euclid language. It checks for dynamic aliasing of variables. It inserts "legality specifiers" into its output stream. These legality specifiers are special tokens used by an optional pass, the Assertion Lister, to create legality assertions.

The Scanner, Parser, Builder and Conformance passes are (almost) completely target machine independent. The final two major passes, the Allocator and Coder, are by their nature machine dependent. We were surprised to discover after their construction that a large portion of these two passes is in fact also machine-independent. The Allocator determines displacements and logical bases for all variables and it allocates scalar variables to machine registers as an optimization. The Coder allocates temporaries and emits assembly language; it does extensive local optimization.

The three minor passes of the compiler are the Namer (inserts names of external references into the generated assembly language), the Error Lister, and the Assertion Lister (optionally creates legality assertions from the output stream of Conformance using the name table).

Use of I-Code

One interesting technique used in the compiler was a standard stream among compiler passes. The Parser originally produces this stream, which we will call I-code. I-code is essentially the complete syntax-checked Euclid program en-

coded as a sequence of tokens, with operators moved into postfix positions. An S/SL program was written to read an I-code stream and to reproduce the same stream as output. This S/SL program was then replicated and served as the skeleton of each of the successive major compiler passes. The passes were constructed by modifying this skeleton, primarily by adding calls to supporting routines written in Euclid. As a rule, a pass relayed information to succeeding passes via the symbol/type table rather than by modifying the I-code stream. As a result all the major compiler passes are similar in structure and hence are relatively easy to understand.

PRODUCTION QUALITY COMPILER

This section describes some of the "production quality" features that we have incorporated into our Euclid Compiler.

Thorough and Rigorous Enforcement of Semantics

Type checking

The compiler checks that types match where appropriate. Both the structure of and values (such as subrange bounds) in the type definitions are examined. In instances when the compiler cannot determine whether values match, it generates legality assertions asserting that they do.

Visibility checking

The compiler enforces the information hiding mechanisms of Euclid. This includes checking that identifiers are imported and exported when required. Also, the compiler checks that variables are not modified in scopes in which only read access is permitted.

Alias and overlap checking

The compiler enforces the Euclid language rules against a variable being accessible in a scope by more than one name, and against two variables referring to overlapping storage. These checks include, for example, checking that variable actual parameters in a routine call do not overlap each other or any variables imported into the routine. In instances when the compiler cannot determine whether variables overlap, it generates legality assertions asserting that they do not. (This occurs, for example, when $a(i)$ and $a(j)$ are passed as variables in the same routine call; the appropriate assertion is $i \text{ not} = j$).

Run time checking

Euclid allows the programmer to specify that assertions in a scope are to be checked at run time. The compiler optionally generates code to check programmer-specified assertions. At present, it does not generate code to check legality assertions. However, additional checking code is generated to ensure that

a program's behaviour can be deduced from its source code. These checks ensure that array subscripts are within their proper range, and that case tag values select one of the case alternatives.

Code Quality

The Toronto Euclid compiler produces code comparing favourably in quality with code emitted by compilers for less structured system implementation languages. For the compiler itself, the code generated was smaller and faster than the code generated by the C compiler for the transliterated version. Using smaller examples, Euclid programs were compared to carefully hand-tuned C programs which exploited C idioms that the transliterator could not employ. The two compilers emitted code of very similar size and speed. However, the Euclid source programs were significantly easier to understand and debug.

Most optimizations performed by the compiler are aimed at producing "locally" good code for individual expressions or statements. Code quality is greatly improved by dealing with common programming constructs as special cases. For example, the statement " $i := i + 1$ " is translated to a single PDP-11 INC instruction. Other high-frequency operations that are optimized include setting a variable to zero, multiplying a variable by a small constant, moving one record variable to another, and routine calls.

Expressions involving compile-time manifest values are folded by the compiler. The Conformance pass computes the values of expressions that can be calculated at compile time, and substitutes these values into the internal representation of the program for use by later passes. The Coder pass then performs further folding, both of expressions and of entire statements. For example, when the code emitter determines that the condition in an *if* statement has a manifest value, it discards either the *then* or the *else* part of the statement as appropriate. This folding of *if* statements provides a simple, yet effective, form of conditional compilation.

Optimizations performed by the compiler encourage the Euclid programmer to use language features to construct well-modularized programs, by eliminating or reducing the costs of using these features. For example, uses of the module mechanism are optimized by the compiler. Using information created by the Builder pass, the Coder pass eliminates use of a display mechanism for most modules. Thus partitioning a program into modules generally costs nothing in the efficiency of the emitted code.

Since well-modularized software typically contains a large number of small routines (procedures and functions), the compiler pays particular attention to minimizing the overhead of routine calling. Standard routine prologue and epilogue sequences, such as those used by C, must (for example) save and restore all registers which might possibly be affected by a routine of arbitrary complexity: for a simple routine, this epilogue and prologue may take as much time to execute as the actual computation performed by the routine. The Euclid code emitter generates minimal prologue and epilogue code for each routine, tailored to fit the number of parameters, number of local variables, etc. of the routine. For many rou-

tines, the prologue can be eliminated entirely and the epilogue can be reduced to a single instruction.

CONCLUSIONS

This paper has described our experiences in designing and implementing a compiler for the programming language Euclid. Although we encountered a number of problems due to the ambitious design of Euclid, we were able to produce a compiler for Euclid that generates quite efficient code. The design of Euclid allowed the compiler to make a number of checks for semantic correctness that could not be done in other languages. Our experience has been that the extensive checking performed by the Euclid compiler has been a considerable help in allowing us to produce software efficiently and effectively.⁴

Compiler Status

The Euclid compiler described in this paper has been operational since December 1979. The compiler runs on DEC PDP-11 models 44, 45, 50, 55 and 70 under the UnixTM time-sharing system. The compiler generates PDP-11 code that is almost entirely model independent. It is being used by universities for research and by industrial firms for production software. Toronto Euclid has been used in a project at the Computer Systems Research Group to write a small UnixTM-like operating system called TUNIS. Holt and Cordy have designed an extension to Euclid called Concurrent Euclid.¹¹ This language is a subset of Toronto Euclid that has been extended with concurrency features, notably monitors. Concurrent Euclid is presently implemented for the Motorola MC68000 microprocessor. Implementations for the Motorola 6809 microprocessor and the DEC PDP-11 are under way. I.P. Sharp Associates Ltd. has recently started work on a Euclid compiler for the Digital Equipment Corp. VAX-11/780 computer.

ACKNOWLEDGMENTS

Development of the compiler described in this paper was funded by the Defense Advanced Projects Research Agency of the U.S. Department of Defense and by the Chief, Research and Development, of the Canadian Department of National Defence.

The Euclid Committee^{1,2} and Prof. J. V. Guttag provided an immense amount of assistance and encouragement during the development of the compiler.

Preparation of this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Lampson B.W., J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek, Report on the Programming Language Euclid, ACM Sigplan Notices, v. 12, n. 2, February 1977
2. Lampson B.W., J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek,

- Report on the Programming Language Euclid, Xerox Palo Alto Research Center Technical Report, (to appear 1981)
3. Holt R.C., D.B. Wortman, J.R. Cordy, and D.R. Crowe, The Euclid Language: A Progress Report, *Proceedings of the ACM National Conference*, December 1978
 4. Wortman D.B. and J.R. Cordy, Early Experiences with Euclid, *Proceedings of the 5th International Conference on Software Engineering*, March 1981
 5. Popek G.J., J.J. Horning, B.W. Lampson, J.G. Mitchell and R.L. London, Notes on the Design of Euclid, *Proceedings of the ACM Conference on Language Design for Reliable Software*, ACM Sigplan Notices, v.12, n.3, March 1977
 6. Kernighan B.W., and D.M. Ritchie, *The C Programming Language*, Prentice-Hall Inc. Englewood Cliffs N.J., 1978
 7. Holt R.C. and D.B. Wortman, A Model for Implementing Euclid Modules and Type Templates, *Proceedings of the ACM Sigplan Symposium on Compiler Construction*, Aug. 1979, pp. 8-12
 8. Parnas D.L., Information Distribution Aspects of Design Methodology, *Proceedings of IFIP Congress 71*, North Holland Pub., 1971, pp. 339-344
 9. Wortman D.B., On Legality Assertions in Euclid, *IEEE Transactions on Software Engineering*, v. SE-5, n.4, July 1979, pp. 359-367
 10. Cordy J.R., R.C. Holt and D.B. Wortman, S/SL: Syntax/Semantic Language—Introduction and Specification, Technical Report CSRG-118, Computer Systems Research Group, University of Toronto, Sept. 1980
 11. Cordy J.R. and R.C. Holt, Specification of Concurrent Euclid, Technical Report CSRG-115, Computer Systems Research Group, University of Toronto, July 1980
 12. London R.L., J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell and G.J. Popek, Proof Rules for the Programming Language Euclid, *Acta Informatica*, v.10, 1978, pp. 1-26

The design and implementation of a new UNIX kernel*

by CHARLES CROWLEY

University of New Mexico
Albuquerque, New Mexico

ABSTRACT

A project to produce a kernel-based, message-passing version of UNIX is described. The system is designed to be (1) useful in teaching operating systems, (2) easily changeable, (3) easily portable, and (4) a vehicle for studying the message-passing approach to operating systems design. The system calls normally handled by the UNIX kernel are handled by system processes, each of which operates in its own address space. Interprocess communication and process environment management is done by a small kernel. The design and implementation of the system are described. Message passing as a system design method is evaluated in this context and compared with the procedure call orientation of standard UNIX. The message-based design proved successful in creating a modular and understandable system.

OVERVIEW OF THE SYSTEM

This paper will present the design and implementation of a kernel-based, message-passing operating system based on the UNIX operating system. The system emulates exactly all of the system calls provided by the standard UNIX kernel. The Level 6 UNIX kernel handles interrupts and traps, and provides process environments, process dispatching, process communication (via signals and pipes), process swapping, an IO system, and the file system.

The system described herein will be called NUKE (New Unix Kernel). NUKE is a kernel-based, message-passing emulation of the UNIX kernel. NUKE consists of (A) a kernel that provides process environments, first-level interrupt handling, process dispatching, and process communication via messages and (B) several system processes that implement the UNIX system calls. System processes are implemented exactly the same as regular UNIX processes. Figure 1 shows the structure of NUKE. The following processes are included in NUKE. (1) The process manager handles process traps and the process related system calls. (2) The memory manager allocates and frees memory, reallocates and moves stacks, and provides address translation services. (3) The memory scheduler process handles swapping. (4) The clock process handles

clock interrupts and all timing services for the system both internally and through system calls. (5) The file system process handles the file and I/O system calls. (6) The device driver processes (e.g., disk drivers, tape driver, tty driver) handle devices and interfaces to device controllers.

User processes make normal UNIX system calls which trap to the kernel which in turn converts them to messages and directs them to the system process that handles that system call. In the course of handling system calls the system processes will make kernel calls requesting services of the kernel and send messages to other system processes (via the send kernel call) requesting services from them. The arrows in Figure 1 indicate message paths between processes.

The goals for this project were to investigate message passing and to develop a system suitable for teaching. The system is based on UNIX since it is a popular system already used in many university teaching programs. It has a two-level (kernel, user) structure that made it easy to replace the kernel and create a new system with minimum effort. The UNIX kernel is strongly procedure-oriented and the conversion promised to shed some light on the differences between procedure and message-based systems. We planned to create an exact emulation of the UNIX kernel which was as simple as possible with a small kernel and few kernel calls. We wanted a system that was measurable, improvable, portable, and that would provide a basis for further operating system research.

SYSTEM STRUCTURE

The Kernel

The *kernel* of NUKE provides the process environment. It handles system calls, interrupts, and user traps and converts them to messages sent to the appropriate system process. There are seven kernel calls that can be made by system processes.

1. send (a message to another process)
2. receive (a message from another process)
3. connect to interrupt (request to handle an interrupt)
4. connect to system call (request to handle a system call)
5. return from system call (restart a system caller process)

* This research was done under the support of NSF Grant MCS76-22941.

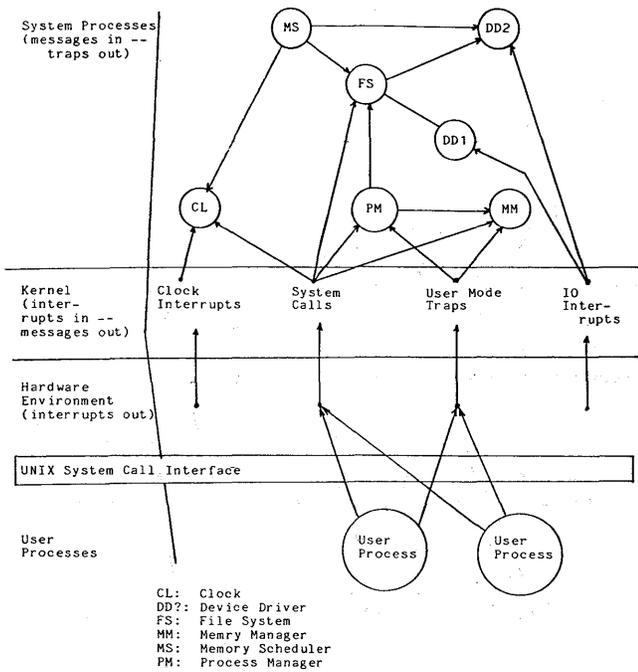


Figure 1—NUKE system structure

6. map (map part of the caller's address space into another address space)
7. trace (call the interactive debugger)

The send and receive kernel calls do interprocess message passing. It is possible to receive from a specific process or from any process. The connect kernel calls are used to initialize the system and tell the kernel which system calls and interrupts the calling system process wishes to handle.

The map system call is another means of interprocess communication. Standard UNIX is heavily based on the shared memory concept. Since a lot of code in NUKE was taken from the standard UNIX system, it was necessary to handle this and the interprocess movement of large blocks of data efficiently. The idea of mapping was developed for this purpose. A process can request that some part of its address space be mapped into (1) the physical address space of the computer, (2) the logical address space of another process, (3) the user structure of another process, and (4) the proc array in the kernel.

Figure 2 shows how mappings are used in the system. A user process makes an 'open file' system call giving a character string file name as an argument. The file system process maps into the caller's user structure to pick up, for example, the caller's current directory, and it also maps into the file name in the caller's logical address space. In the directory searching, the file system process needs to read disk blocks. This is done by the disk device driver, which maps the buffer in the file system's logical address space and reads directly into that.

The mapping concept is simple, efficient to implement, and provides efficient interprocess communication, since words are transferred by the hardware with no more overhead than accesses to the local process memory (once the mapping is established). It is a good compromise between safety, efficiency, and flexibility of implementation. In a normal shared

memory system, a block of memory is shared all the time and can be used at will. In NUKE, memory can be shared, but only through an explicit kernel call to set up the mapping. This means the kernel can insure that the mapping is safe. The kernel can implement the mapping in a number of ways depending on what forms of communication are available.

System Processes

Each system process is a normal UNIX process that operates in its own address space and is isolated from other processes except for the use of explicit interprocess communication. Each process receives system call messages from user processes (via the kernel) and performs the user service. System processes can send messages to other system processes to request services.

Figure 3 shows part of the main routine of the process manager system process. This is the code exactly as it runs in the system except that most of the cases in the large case statement have been omitted. The cases remaining are typical.

The variable *p* (line 24) will be an address mapped into the proc array in the kernel's address space. The initialize procedure called on line 33 will map into the kernel's proc array, connect to the 14 system calls the process manager handles, and connect to the user traps (bus error, illegal instruction, trace, and floating point exceptions) the system manager handles. Lines 34-91 are an unending loop which is the main processing loop of the process manager. Each iteration of the loop will receive a message and service it. The receive on line 43 requests a message from any process. Lines 44-90 are a

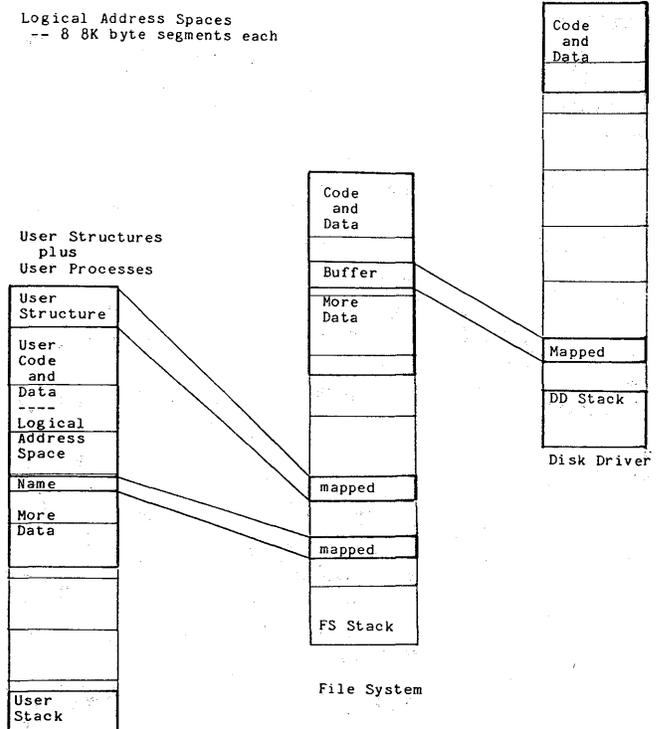


Figure 2—Mapping in NUKE

```

1 #
2 /*          pm.c          */
3
4 #include "../includes/error_codes.h"
5 #include "../includes/interrupts.h"
6 #include "../includes/messages.h"
7 #include "../includes/proc.h"
8 #include "../includes/seg_regs.h"
9 #include "../includes/signals.h"
10 #include "../includes/systemcalls.h"
11 #include "../includes/user.h"
12
13 #define SW_HIGH 03
14 #define SW_LOW 0177570
15
16 struct      /* to get at bytes of a word */
17 {
18     char    lobyte;
19     char    hibyte;
20 };
21
22 pm()
23 {
24     extern struct proc *p;
25     int msg[8], x, signal_number, pr_increment;
26     int *sw;
27     register int pid;
28     register struct proc *q;
29     register struct user *u;
30
31     trace ("PM initialize", 0, 0, 040);
32     /* connect to system calls and interrupts */
33     initialize();
34     while (1)
35     {
36         /*
37          * receive a message from any sending process.
38          * This message will be a system call message
39          * except for the traps,
40          * so pid will be the process id of the process
41          * making the system call
42          */
43         pid = receive(ANYPROCESS, msg);
44         switch(msg[0]) /* msg[0] is the message type */
45         {
46             case CSW:          /* get console switches */
47                 /* map to the switch register in high physical memory space */
48                 sw = map (PHYSICAL, SR5, SW_HIGH, SW_LOW, 2);
49                 x = *sw;
50                 sys_ret (pid, Ret_r0, x, 0);
51                 map (UN_MAP, SR5, 0, 0, 0);
52                 break;
53
54             case EXEC:         /* change the program running in a process */
55                 exec(pid, msg[1], msg[2]);
56                 /* (caller, name, arglist) */
57                 break;
58
59             case EXIT:         /* stop process */
60                 /* msg[1] = exit status, exit.c expects
61                  * it to be in the high byte
62                  */
63                 exit(pid, (msg[1]&0377)<<8 );
64                 /* (caller, status) */

```

Figure 3—Process manager

(continued on next page)

```

63             break;
64
65     case FORK:      /* create a duplicate process */
66         fork(pid);
67         break;
68
69     case GETGID: /* get group id */
70         u = map(U_STRUCT, SR5, pid, 0, 0);
71         x.lobyte = u->u_rgid; /* real group id */
72         x.hibyte = u->u_gid;
73             /* effective group id */
74         sys_ret(pid, Ret_r0, x, 0);
75         map(UN_MAP, SR5, 0, 0, 0);
76         break;
77
78     case GETPID: /* get process id */
79         sys_ret (pid, Ret_r0, pid, 0);
80         break;
81
82     case INT_SIGNAL:
83         /* kill (from another system process) */
84         /* msg[1] = receiver
85            (who to send the signal to) */
86         /* msg[2] = signal number */
87         kill(pid, msg[1], msg[2]);
88         break;
89
90     default:
91         trace("panic in PM: bad mag #",
92             msg[0], 0, -1);
93         break;
94 }
95 }

```

Figure 3 (continued)

case statement that does the appropriate processing depending on the message type. Lines 46–52 process the CSW (read console switch register) system call. First the switch register in the I/O space is mapped using the map kernel call (line 48). The kernel call returns a value which is the address to use to access the mapped memory. Line 50 is the kernel call which indicates that the system call processing is completed. The process with process identifier 'pid' will be made ready (dispatchable) and the returned value from the system call will be the value in variable *x*. The returned value is placed in the caller's saved register 0 before it is dispatched again. Note that the apparent sender of the system call message ('pid') is the system caller, not the kernel. Line 51 is the unmapping kernel call (a variant of map).

The exec, exit, and fork system calls are shown on lines 54–67. Their processing is lengthy and is done in procedures. The processing of the exit and fork system calls (lines 58–67) is also done in procedures.

The getgid system call processing (lines 69–75) maps into the system caller's user structure (the swappable part of the process descriptor) in order to pick up the group id. The real and effective group id are picked up and packed into a word in lines 71 and 72 and returned to the caller by the system return on line 73. The get process identifier system call (lines 77–79) is similar. Case INT_SIGNAL on lines 81–85 is the

internal form of the kill system call and uses the same procedure.

All the system processes have a similar structure: a main loop that accepts messages and processes them one at a time. Sometimes the message-request processing cannot be completed immediately (for example, a terminal read request). In these cases the system process records the pending request in a table and provides for it to be restarted later.

Breaking UNIX Up Into System Processes

The standard UNIX kernel is a large program in a single address space, contains many procedures that call each other frequently, and uses shared memory extensively to communicate between processes. We spent a lot of time deciding how to break the UNIX kernel up into system processes to make the system modular and relatively easy to implement. The test of the way we broke down the system is in how modular it came out and how much interprocess communication was necessary. In the next few paragraphs we will look at what memory sharing and internal messages were necessary to make the system work.

Basically, the system processes operate on their own with little communication with other system processes. The main

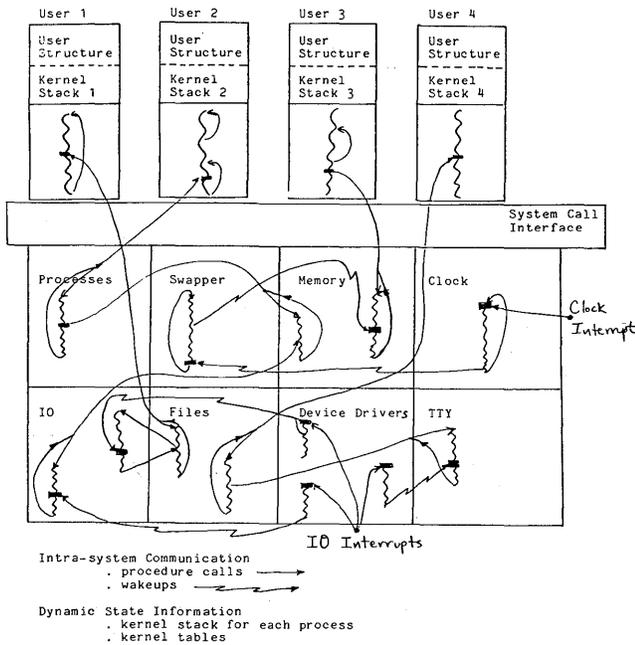


Figure 4—UNIX flow of control

exceptions are in cases involving certain actions with processes when they start, fork, or exit. These actions involve system-wide consequences since each process has resources of several kinds.

The only memory shared between system processes is the proc array and IO buffers. In the case of the proc array there are no critical sections since each process handles a part of the proc array and there are no conflicts. The IO buffers are necessary since the device drivers are separate processes.

Overall, the separation has been good, with each process logically separate from the others and very little messy interaction. If anything, there are too few system processes. We could divide up the process manager and the file system processes into logically separate parts that would interact very little or over narrow, neatly defined interfaces.

EVALUATION OF THE SYSTEM

Message Passing

The message-passing design of the system has been very successful in making a modular system. The system is more modular and easier to understand than standard UNIX. The standard UNIX kernel is a highly interconnected program in a single address space in which the parts of the system communicate with procedure calls and shared memory. Synchronization is done with a sleep/wakeup mechanism. Figure 4 shows some typical flows of control and interprocess synchronizing in UNIX.

In NUKE, the system processes are all independent. User processes never execute system code. Their system calls cause them to block and a message is sent to the appropriate system process. The thread of control of a single (user or system)

process never leaves the immediate code of that process. If service from another part of the system is needed, a message is sent to the other system process and it later sends a reply message indicating completion of the requested service. Figure 5 shows the intercommunication between system processes in NUKE.

The main advantage that comes from a message-oriented system is an increase in modularity in the system. This modularity occurs at the physical (run-time) level as well as the logical (compile-time) level. As we noted above, at run time the locus of control of each process is localized to its own code. This is the perfect environment for a tightly coupled multiprocessor system. The procedure call design technique implies large address spaces and also implies nonlocal loci of control. Such a system would not be nearly as amenable to implementation on a multiprocessor. Message passing also promotes a logical modularity because interprocess communication is more difficult and expensive than in a procedure call oriented system, although any organization used in a message-passing system also could be used in a procedure-calling system.

Synchronization in NUKE is needed only during the message-passing primitives. This is protected in NUKE by a high processor priority but it could be done easily with spin locks in a multiprocessor system. The actual message transfer times are small enough that we can afford to mask out interrupts while they are going on or compete for access to message queues with busy waiting techniques. Each system process operates on its own virtual processor and does not have to consider other processes. There are some cases in NUKE in which memory is shared. These cases are not a problem, however, since the synchronizing is done by messages and by separation of function. For example, since only one process

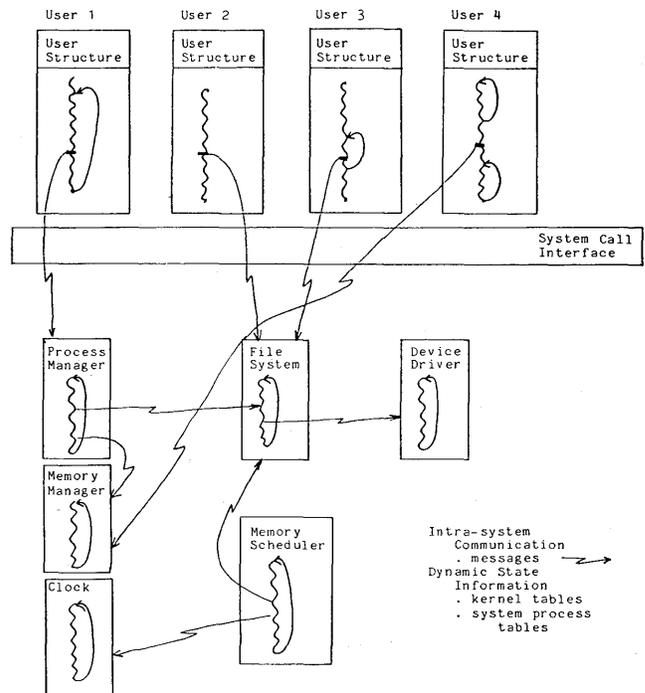


Figure 5—NUKE flow of control

creates new processes (the process manager), there are no critical section problems with the empty proc array slots. In standard UNIX there might be several processes creating a new process (that is, in the middle of a 'fork' system call) simultaneously. Hence it was necessary to lock the proc array in several places. This is never necessary in NUKE because each function is provided by only one system process and each system proceed proceeds serially.

Synchronization takes care of itself naturally in NUKE but in a way that is easy to understand and whose correctness is nearly self-evident. All the synchronization is localized to the kernel and to the interrupt and message-passing sections in particular. It can be easily understood and verified since it is all in one place and only one synchronizing technique is used. The synchronization in standard UNIX is clever but difficult to understand.

Overhead is also localized in message-based systems. If the critical message and context switching operations are put into microcode, significant speedups are possible. It is not so clear what to optimize in a procedure-based system.

Conversion From Procedure Calling to Message Passing

The design of NUKE was influenced by the need to use as much of the existing UNIX code as possible and to get a system up and running as soon as possible. The mapping feature worked particularly well in making up for the fact that system processes did not directly share memory. The system processes need common access to a few areas of memory. Once these were mapped we could use the existing UNIX code without change. Procedure calls were changed to paired message send and receives.

The major problem in the conversion was to decide how to wait for system events such as I/O completions. In standard UNIX, waiting is done by blocking the procedure in the middle of the system code and waiting for a wakeup. We could not use this technique, since our system processes could not block. We handled the problem by setting up waiting tables in each process that had to wait for an event before it could finish processing a request. A mechanism was put in to insure that these requests would be restarted when the appropriate event occurred (usually the receipt of a certain type of message by the process). We had to add another message type whose processing consisted of finding the proper entry in the wait table and restarting its processing.

The Duality of Procedure Calling and Message Passing

Lauer and Needham (1978) present the thesis that there is a duality between message-passing systems and procedure-calling systems such that they can be easily mapped into one another and their performance characteristics should be the same. This conflicts with some of the experiences and conclusions we have presented here.

It is true that there is a strong duality between the two types of systems but this is not at all surprising since they were considering idealized and generalized models of the two types of systems and it seems to be true that if you look at almost

any two things at a sufficient level of generalization they become very similar. For example, we have argued in previous work that compilers and operating systems were really doing the same job (Crowley 1979). The duality does exist in these idealized models but one must be careful in drawing conclusions from these models and applying them to real systems since implementation details can often make large differences in system performance. Lauer and Needham realized and noted this in their paper. Their intention was to start a discussion of the issues and to try to discover what general conclusions can be drawn from the models and applied to real systems.

They note the duality between a procedure call and a message followed by an immediate wait for a reply and remark that these can be implemented with the same overhead. This is not generally true since a simple procedure call to code in the same address space can be implemented with one machine instruction on most machines. There is no need to set up the mapping for the other procedure (since it is on the same address space) or to execute two kernel calls with their attendant overhead. Both of these things must be done in a message-passing system and this could involve the execution of hundreds of machine instructions. It is exactly here that the overhead of a message-passing system is found. Lauer and Needham also show that a FORK followed later by a JOIN are dual to a message sent and a reply waited on later in the code. This is a true analogy and their arguments about the cost of these two dual operations hold up since each will require kernel calls and context switching. UNIX never uses this extended form of FORK/JOIN in its kernel code so all of its procedure calls are of the very fast synchronous variety. For this reason, NUKE is much slower than UNIX.

Lauer and Needham also claim that the processing code can remain unchanged in changing from one type of system to the other and that the code will execute in the same amount of time in both systems. This claim ignores the case in which the procedures share global data. If the system uses global data (usually system tables encoding the state of the system resources) both the calling procedure and the called procedure can access these data efficiently. In a message-passing system there are no global data so tables that must be shared by several processes have to be passed or explicitly mapped into. It is only the mapping facility in NUKE that makes it come close to the efficiency of standard UNIX. Even so NUKE still requires the overhead of a kernel call to set up the mapping and another kernel call to undo the mapping.

One can make a good case for the idea that such global data are bad programming practice and lead to errors, but global data are heavily used in standard UNIX. Even Lauer and Needham in their paper talk at length about global data in procedure-calling systems and the attendant locks and synchronization that they make necessary.

Size and Performance

The system is larger and slower than the standard UNIX kernel. The UNIX kernel has about 9,000 lines of code, while NUKE has about 11,000 lines (20% larger). UNIX is about

10% assembly language (1,000 lines) while NUKE is about 2% assembly language (200 lines).

In memory space NUKE is about 50% larger than UNIX. There are several reasons for NUKE being larger than standard UNIX. First the system processes each need a user structure, a stack area, and basic structure code; none of which are needed for standard UNIX system processes (since they are really piggy-backed on user processes). The system processes have a certain amount of overhead in mapping, sending and receiving messages, and in their processing loops. Finally about half of the kernel is overhead to support message passing which is not needed in standard UNIX.

The execution time of the system is seven times slower than UNIX at the raw level (for `getpid`, which is a system call that is nearly all overhead). For normal user interactions, the system is 1.5 to 3 times slower than the standard UNIX kernel. This extra overhead comes from the extra context switching required in a message-passing system and the kernel call processing time required to set up the maps, send messages, etc.

The system is slow but its performance is not unexpected since message passing implies more overhead. We thought the benefits of message passing would be worth the extra overhead. We expect the VAX version of NUKE to compare more favorably with VAX UNIX than it does with PDP11 Level 6 UNIX since the richer instruction set is more favorable to message passing with primitive operations for queue handling and context switching. Also the character string searching instructions can be used to create a very fast scheduler.

CONCLUSIONS AND FUTURE RESEARCH PLANS

Our overall evaluation is that message passing is a good way to structure operating systems. The main advantage is that operating systems so structured are highly modular, easy to understand and modify, and easier to get working correctly. The main disadvantage of message passing is that it is definitely slower but this could be ameliorated by microcoding critical operations and using multiple processors both of which are easier to do in message-based systems.

The next phase of the project is to port the whole system over to the VAX11/780. We expect this to be straightforward since only a few parts of the system need to be changed. After

that we will optimize the system on the VAX by taking advantage of its larger instruction set. We also plan to use user writable control store on the VAX to speed up critical operations. A kernel-based, message-passing operating system is a natural for this since the overhead is concentrated in a few places, that is, in context switching and the other kernel functions.

We have mentioned that the process-oriented system had more localized threads of control and could be more easily adapted to a multiprocessor system. We would like to try to port the system to a multiprocessor to see what advantages could be gained. The idea would be to pick a system with multiple microprocessors in a tightly coupled system with some common memory and some local memory (e.g., the Intel Multibus). The system and user processes could float among the processors and the kernel would be implemented on each processor. Some of the context switching would be eliminated and this would allow true parallelism which should speed up the system. The goal of such a system would be to provide a high-performance computer system at a low cost by using inexpensive microprocessors. If the speed improvement could be shown it would be easy to drastically reduce the cost of the system by custom design of the hardware.

ACKNOWLEDGMENT

This research was inspired by discussions with Forest Baskett, Mike Malcolm, and Gary Sager on message passing and by the designs of the DEMOS and THOTH operating systems. Steve Meyers and Tom Obenauf assisted in the design and coding of the system. Jerry Basford was invaluable in the debugging of the system and the writing of several system processes.

REFERENCES

1. Crowley, C., "Parallel Developments in Programming Languages and Operating Systems," *Computer Languages*, Vol. 4 No. 2, 1979, pp. 71-82.
2. Lauer, H. C. and Needham, R. M., "On the Duality of Operating System Structure," *Proceedings Second International Symposium on Operating Systems*, IRIA, October, 1978. (Reprinted in *Operating Systems Review*, Vol. 13, No. 2, 1979, pp. 3-19.)

A security policy for a profile-oriented operating system

by CHARLES R. YOUNG

Sperry Univac
Blue Bell, Pennsylvania

ABSTRACT

A security policy for a profile-oriented operating system is described that is adopted from state-of-the-art security properties designed to meet even the strict security requirements of the Department of Defense. The policy is built around user, execution, and program profiles that serve as repositories for security related information. The security terms access category, access list, security level, clearance level, subject, object, discretionary and nondiscretionary security, and profile are defined. The six security rules that form the foundation of system security are described. Their interactions are detailed and examples are given. The six rules are: discretionary security condition, simple security condition, *-property (star property), tranquility principle, nonaccessibility of deleted objects, and rewriting of newly created objects.

INTRODUCTION

The topics of security and privacy are under increasing public scrutiny. People are more aware than ever of the quantity of information kept on them and are increasingly concerned about its proliferation. Privacy, to the extent it has a technical meaning, is a legal and social term concerning what rights, if any, an individual has over information concerning him, and what obligations are placed on other parties having access to or responsibility for such information. Security, on the other hand, is a term used to refer to the problems involved in ensuring that unauthorized people do not have access to some piece of information, no matter what the reason for controlling access, and so it is mostly involved with technical issues.

In discussing either topic, especially security, an important concept is the security policy. The security policy controlling a given piece or class of information is the set of rules governing who is to have access to what information under what conditions. A policy may be formal or informal, legally enforceable or merely administratively enforceable. A policy may have discretionary (need-to-know) and nondiscretionary (minimum security level requirements) components to it. Under the U.S. Department of Defense policy for safeguarding sensitive information, it is forbidden to give classified information to a person not having a high enough clearance for it.

(The holder of the information cannot exercise any discretion in the matter—hence the term nondiscretionary.) Having the proper clearance, however, is not sufficient reason for obtaining access—the possessor of classified information always has the right to refuse access even though the requestor's clearance is *prima facie* evidence that he is sufficiently trustworthy; in fact, a possessor is obligated to exercise that right if the requestor does not present adequate justification.

A key criterion for a workable and reliable security policy is that it not be too dependent on the overall wisdom and background of the people involved in enforcing it. The policy itself should minimize the extent of human errors of judgment or administration that can defeat the purpose of the policy. For this reason, formally established policies mark information (classify it) and give labels to people (give them clearances) with fairly simple rules about what sets of labels are required to permit access to information bearing a given set of markings. The major ingredients in the nondiscretionary aspects of security policies are a notion of a small number of levels of sensitivity of information and a much larger number of categories of information. Being granted access to information of given sensitivity implies in principle the right of access to information of lower sensitivity, but being given access to a given category of information (e.g., payroll) does not necessarily say anything about access to information of a different category (personal health records, manufacturing schedules). Note that discretionary controls may still deny access; a person may have the right to say exactly who can see his health record even though all the medical staff are "cleared" for it.⁶¹ These areas of interest (payroll, personal health records, manufacturing schedules) are formally referred to as access categories.

This paper describes a security policy for an operating system oriented toward profiles. Profiles contain non-security-related information about programs, users, and their execution environments. They also contain security-relevant information such as the objects (e.g., files) a user or program may access and with what privileges (e.g., read or write). This paper is a detailed technical presentation that combines, translates, and reorients many published articles, papers, and documents into a form usable in a profile-oriented system.

The policy is divided into two sub-policies, discretionary and nondiscretionary. The nondiscretionary policy is further divided via security levels into clearance level and access cat-

egory sub-policies. If any one of these is in effect, it applies to all subjects and all objects; for example, if discretionary security is in effect, then every access to an object by a subject must first pass discretionary access controls. Discretionary security, clearance levels, and access categories may be combined to configure any of eight enforceable policies, from no security to a configuration where all are present. The security policy is enforced by positive system controls that ensure user and system software obey the six rules summarized in the section "Rules Summary."

Terms

Familiarity with most computer related security terms is assumed; for clarity, however, there follows a list of terms whose definitions are unusually complex or whose meanings are altered in this context.

Access Category Set—An access category is one of the classes to which a subject or object is assigned; NATO, Medical, Financial, and Nuclear are examples. An access category set is a grouping of zero or more access categories; {NATO} and {NATO, Nuclear} are examples. An access category set is related to non-discretionary security and is a component of a security level. See \supset (contains as a subset) in the section "Symbols" for their properties.

Access List—An access list is an enumeration of objects and the privileges associated with individual objects in the list. Related to discretionary security, access lists are contained in user profiles, program profiles, and execution profiles.

Clearance Level—Related to non-discretionary security, a clearance level is a classification of either a subject's integrity or the value of an object's data. In the military vernacular, an object's clearance level is also known as a classification level. A clearance level is a component of a security level. Typical examples are: Top Secret, Secret, Classified, and Unclassified. See \geq (greater than or equal to) regarding their properties.

Data Privilege—A data privilege is an identification of how an object's data may be used. Read, write, and add are examples for file objects.

Discretionary Security—Discretionary security permits an owner to select other subjects that are permitted access to an object, constrained by nondiscretionary security. Represented by the access lists contained in the program, user, and execution profiles, discretionary security forms a portion of the total security policy.

Dominant—See the section "Symbols," ∞ .

Execution Profile—With respect to security-related information, an execution profile is an object that contains the access list of all objects accessible under the execution profile, and the object and data privileges associated with each object; these privileges are categorically identical for all users associating with a given object in a specific execution profile. Its access list may not contain any nested profiles as objects (see "Object Hierarchies") except for the PHI profile. An execution profile can be thought of as a group for ease-of-use purposes, since users having common operating environments can be grouped together by using the same execution profile.

Several users may be concurrently attached to an execution profile, each making use of objects contained in its access list. Conversely, a user may be attached only to one execution profile at a time, with the PHI profile being the only exception. Attaching to an execution profile effectively extends the number of objects accessible by the subject beyond those normally available under his user profile.

Nondiscretionary Security—Nondiscretionary security permits no choice over who is permitted to access an object. It forms a portion of the total security policy, with security level requirements governing accesses.

Object—An object is an identifiable system resource or entity—software-created entities such as instances of data files, programs, and libraries and hardware resources such as individual disks and processors. There are two classes of objects, primary and subordinate.

Object Privilege—An object privilege is an identification of how an object may be used. Move, rename, destroy, save, and restore are examples for file objects.

Object Type—An object type is a grouping of objects of similar class, such as all MIRAM (Multiple Indexed Random Access Method) files or all segments.

Object-Type Manager—An object-type manager is an instance of a class of programs that create and manage specific types of objects; examples are the MIRAM-file manager and segment manager. An object-type manager differs from "regular" program in that the manager has the ability to construct objects (e.g., MIRAM files) out of subordinate objects (e.g., data segments) and keep (indirectly, in the access list of the manager's program profile) all the information about the subordinate objects accessible only to itself.

Ownership—Ownership of an object allows discretionary controls over that object, such as who accesses it and with what privileges. The owner may restrict his own privileges; e.g., he may restrict file access to read-only and on desired occasions change it to read/write. An owner may never discretionarily delegate to another subject more privileges than he currently possesses.

Password—A password is a protected secret word or string of symbols that is known only to the user and that authenticates the user. The password is required for interactive terminal log on and for running batch programs. In some instances, such as when the security officer logs on, additional passwords or password-like information may be required.

PHI Profile—The PHI profile, so named because phi (ϕ) is the greek letter used in mathematics to represent the null set, is an execution profile whose security level's access category set is the null set. Thus, objects named in the access list of the PHI execution profile are nondiscretionarily restricted only by the clearance level relationship of object and subject. When the PHI profile is listed as an object in the access list of the program, user, or execution profile to which a subject is attached, he is automatically attached to it, even while attached to another execution profile.

Primary Object—A primary object is an object that is user visible and controllable; he receives ownership and all object and data privileges of any primary object for which he requests creation. This is the only type of object with which a user need be concerned. Ownership of primary objects is indicated in the user profile.

Privilege—A privilege is an identification of how an object or its data may be used. See also data privilege and object privilege.

Process—A process is a thread of control and is represented and controlled in the system by a process control block (PrCB). It is created when the user logs on or when a batch run is activated. The PrCB represents, and acts on behalf of, the user and/or program; thus, it is considered a surrogate subject. Although a process never owns an object, it requests them for its own use on behalf of the subject. When activated, the process is given the security level selected by the user from those available to him.

Program—A program is a collection of procedures containing data and executable code that acts as both a subject and an object. A program profile is associated with each program; it contains the access list of all objects that a program, when acting as a subject, may directly access. There are several kinds of programs, two of which concern security: object-type manager programs and “regular” programs, which are all non-object-type manager programs. The user can write programs of both types; object-type managers are so designated because of their special properties. A grouping of programs may be protected the same way a single program is.

Program Profile—With respect to security-related information, a program profile is an object that contains the access list of all objects accessible directly by the program and the object and data privileges associated with each object. Whenever a program is in control of a process, the program profile’s access list, in addition to the user and execution profiles’ lists, can be used for verifying proper access of objects. There is a one-for-one correspondence between program and program profile. Its access list may not contain other profiles as objects, except for the PHI profile. A program profile’s purpose is twofold: it allows a user to explicitly delegate to a program privileges to access other objects directly during its execution (see “Program Execution”); and it provides a mechanism for object-type managers to access their subordinate objects. In the latter case, both the subordinate object and the higher level object of which it is part are noted.

Security Level—A security level, roughly speaking, is a measure of a subject’s trustworthiness and denotes the maximum security operating environment; for objects, it represents its data’s value and, in part, the operating environment in which it was created. Related to nondiscretionary security, a security level has two components: a clearance level and an access category set; this is denoted by $SL \equiv (CL, \{AC\})$. See \equiv (is identical to) and α (dominates) in the section “Symbols” regarding properties of security levels.

Subject—The user, known to the system by his user ID, is a subject of the system; only he may own objects and exercise all the privileges associated with objects, including discretionary controls. A program is also a subject, but may own objects; programs may have privileges associated with objects and certain types of programs, namely object-type managers, may have exclusive control over those privileges. A process represents the user and program and is a subject in its own right, acting on behalf of the user or program by executing commands and other programs and requesting objects for use. A subject becomes an object for purposes of interuser and interprocess communication.

Subordinate Object—A subordinate object is an object that is user-invisible; it can be thought of as a building block used to construct user-visible (primary) objects or other subordinate objects. Subordinate objects are controllable by and visible to an object-type manager; although the manager does not “own” these objects in the generic sense, it does possess all object and data privileges for the subordinate objects it requests. The object-type manager can never transfer or delegate any privileges of a subordinate object it uses. For security purposes, subordinate objects are marked as “owned” in the program profile’s access list of the object-type manager that requested its creation; this is a restrictive ownership limited only to object management and true ownership of the subordinate object (for accounting purposes, etc.) can be traced to the user who owns the primary object of which the subordinate object is part. A subordinate object is also marked in the program profile with the name of its higher-level object.

User—A user is an individual who interfaces directly (via a terminal) or indirectly (via a batch run) with the system and is identified by his user ID and authenticated by his password. The user is restricted by the security level with which he runs. He is a subject while performing actions and an object for interuser communication purposes.

User ID—A user ID is a name, word, or string of symbols used to identify each user to the system and to other users or systems; it is a user’s “name.” It is nonsecret and globally available.

User Profile—With respect to security-related information, a user profile is an object that contains the access list of all objects, including those owned, accessible to the user who owns the user profile, and the object and data privileges associated with each object. The user is always attached to his user profile, whereas attachment to an execution profile may be transitory; thus, the user always has his user profile’s objects available. There is a one-for-one correspondence between user ID and user profile. Note that an execution profile is an object; thus, the user profile states all execution profiles with which the user may associate. The user profile also contains other security-related information, such as default and maximum security levels and passwords.

Symbols

The following is an unordered list of symbols, the security properties they represent, and examples of those properties.

- \supset —Read “contains as a subset,” this symbol is used in comparing the contents of two access category sets: $\{AC1\} \supset \{AC2\}$ if and only if all the members of AC2 set are contained in AC1 set. Note that the set with no members (called the empty set or null set) is contained as a subset of every set. Valid examples are

$$\begin{aligned} \{NATO, NUCLEAR\} &\supset \{NATO\}, \\ \{NATO\} &\supset \{NATO\}, \\ \{NATO\} &\supset \phi \text{ (the null set)}, \end{aligned}$$

and

$$\{NATO, NUCLEAR\} \supset \{NUCLEAR, NATO\}.$$

Invalid examples are

$$\{\text{Medical}\} \supset \{\text{Medical, Financial}\},$$

$$\phi \supset \{\text{Medical}\},$$

and

$$\{\text{Medical}\} \supset \{\text{Financial}\}.$$

- \geq —Read “greater than or equal to,” this symbol is used in comparing two clearance levels: $CL1 \geq CL2$ if and only if $CL1$ encompasses at least the clearance levels of $CL2$. That is, \geq is an ordering of all clearance levels such that a subject having clearance level value 6 can potentially access all objects whose value is 6 or less, but cannot view objects of 7 or greater. In Department of Defense terms, a person possessing the Secret clearance level may view an object of Secret, Classified, or Unclassified, since $\text{Secret} \geq (\text{Secret, Classified, Unclassified})$.

Valid examples are

$$\text{Top Secret} \geq \text{Top Secret}$$

and

$$\text{Top Secret} \geq \text{Classified}.$$

Invalid examples are

$$\text{Secret} \geq \text{Top Secret}$$

and

$$\text{Unclassified} \geq \text{Classified}.$$

- \times —Read “dominates,” this symbol orders all security levels, as follows: Given two security levels, $SL1 \equiv (CL1, \{AC1\})$ and $SL2 \equiv (CL2, \{AC2\})$, then $SL1 \times SL2$ if and only if $CL1 \geq CL2$ and $\{AC1\} \supset \{AC2\}$. Valid examples are

$$(\text{Secret}, \{\text{Medical, Financial}\}) \times$$

$$(\text{Secret}, \{\text{Medical, Financial}\}),$$

$$(\text{Secret}, \{\text{Medical}\}) \times (\text{Classified}, \{\text{Medical}\}),$$

$$(\text{Secret}, \{\text{Medical, Financial, NATO}\}) \times$$

$$(\text{Secret}, \{\text{Financial}\}),$$

and

$$(\text{Secret}, \{\text{Medical, Financial}\}) \times (\text{Unclassified}, \phi).$$

Invalid examples are

$$(\text{Secret}, \{\text{Medical}\}) \times (\text{Top Secret}, \{\text{Medical}\}),$$

$$(\text{Secret}, \{\text{Medical}\}) \times (\text{Secret}, \{\text{Medical, Financial}\}),$$

and

$$(\text{Unclassified}, \phi) \times (\text{Top Secret}, \{\text{NATO}\}).$$

- \equiv —Read “is identical to,” this symbol says that where one item is named, another could as easily be placed there; for example, in the previous examples, where SL is written, $(CL, \{AC\})$ could just as easily be written.

Basic Assumptions

This section names assumptions basic to the understanding of this paper.

- For full security, every object and every subject in the system has associated with it a security level.
- To the security policy, there is no difference between batch and interactive modes except in initial system access.

Rules Summary

The following is a condensation of the 6 security rules as interpreted for a profile-oriented operating system:

1. Discretionary Security Condition—Subjects are constrained to access only objects listed in the subject’s user profile, program profile, an attached execution profile, or, if named as an object in his program, user or attached execution profile, the PHI profile.
2. Simple Security Condition—Only if the security level of the process dominates the object’s may the object be cleared for access by the subject.
3. *-Property—A subject can modify an object OB_1 in a manner dependent on data in object OB_2 only if the security level of OB_1 dominates OB_2 ’s.
4. Tranquility Principle—A subject cannot change the security level of any object.
5. Nonaccessibility of Deleted Objects—An object cannot be accessed if it is in the deleted state.
6. Rewriting of Newly Created Objects—A newly created object is given an initial state independent of the state of all other objects, including those now existing and those no longer existing.

SECURITY POLICY

The following sections describe the internal security policy and its rules and give some scattered examples. The information presented is oriented toward the interactive user. Batch users are treated internally the same as interactive users; only logon is different.

Logon

The user logs on by supplying his user ID and password(s). Each user has defined in his user profile his default and maximum security levels; also listed as objects are all execution profiles he may access, including a default which may be none or the PHI profile. At logon the user can optionally change his default security level and default execution profile. The selected security level must be dominated by his maximum security level and the selected execution profile must be from among those listed in his user profile.

Process Creation

When the logon has been validated, the user is granted access to the system. A process (PrCB and associated controlling structures) is created on behalf of the user. The PrCB and all other structures are given the security level selected by the user for this session. This cannot be changed for the life of the process. All sub-processes and dynamically spawned control structures are also assigned the selected security level which can never be changed. Although a process never owns an object, it does request objects that are directly or indirectly requested by the user or program.

The process, from a security viewpoint, is now ready to act on behalf of a user or program. Thus, user and process have been melded to form an active subject.

Program Execution

A program runs under control of a process. Thus, it receives the objects currently cleared for use by the process (see "Object Requests and Request Validation Sequence"). Further requests for objects may be made either directly by the program or at the user's request. To verify legitimate object requests, the subject's security level and access list privileges are checked; the checks are described later. They provide the capability to provide that some objects requested may only be requested by the process while a certain program is running, i.e., the user cannot be cleared for access to the object without the given program in control. Access to an object so requested would be immediately disallowed at program termination. This capability is a logical consequence of a program's program profile and the access privileges in its access list.

As object-type managers (programs with special abilities—see below) gain and yield control of the process, the totality of objects that may be accessed by the process changes dynamically. This is also a consequence of the program profile.

Primary Object Creation

A user who has the capability can create primary objects. In this case the user is named object owner and is given total control over and access to the object and its data. Initially, therefore, creator = owner, but this may not always be the case if ownership is transferred. Once created, the creator name is kept for historic purposes only; this field could aid in tracking malicious system users. Indication of ownership of primary objects is kept in the user profile of the user who owns the object.

These newly created objects are given initial states independent of the states of any previous use of the object names. (This is one of the six security rules, rule 6: rewriting of newly created objects.)

At creation time an object is automatically assigned the subject's current security level, which is the default, or the level stated on the create command, which must be dominated by the subject's current security level.

There is a further restriction placed on allowable security levels for dynamically created objects: the level of a dynamic-

ally created object (i.e., one created by a program on behalf of the user) must dominate the level of every object assigned with read access to the process. This requirement, similar to the *-property discussed below, prevents the program from conveying information in a covert fashion by creating a series of files of, say, secret classification while reading top secret information; another colluding process with a secret clearance level could interpret the presence or absence of files as binary coded information representing the top secret information, which it is, of course, not permitted to view.

Object-Type Managers

The operating system is necessarily oriented around objects. Further, for every object type there is an object-type manager that is the one and only manager responsible for creation and management of that object type; for a given object, the manager cannot be changed. Its unique feature is that it can create higher level objects from subordinate object types. The user is also given the ability to write his own (abstract) object-type managers.

Subordinate Object Creation

A subordinate object is created when an object-type manager requires an object to construct another object of "higher" type. A subordinate object may itself be composed of other subordinate objects. For example, a MIRAM file M1 might be constructed out of several SAT (System Access Technique) files, which in turn may be constructed out of segments, and so on. In this example a user's create request causes the MIRAM object-type manager to be invoked, directing it to create a primary object of type MIRAM named M1. When security is informed of this creation, an entry is placed in the object-type manager invoker's profile, namely the user profile. The MIRAM manager, invisibly to the user, invokes the SAT object-type manager, requesting a subordinate object of (obviously) type SAT, which is declared to be part of M1. When SAT informs security of the SAT file creation, an entry is placed in the program profile of the SAT manager invoker, MIRAM, stating its subordinate nature and that it is part of M1. This procedure continues throughout the hierarchy of objects constituting M1.

A subordinate object is given the same security level as the next higher object type.

Object Requests and Request Validation Sequence

The use of an object for security purposes is logically a multistage operation. The object must first be requested by the process for its use. As the immediately following sections indicate, rule 5, rule 2, and rule 1 govern the validation of requests. Once validated, the subject is cleared for access to the object. Note that if the subject is not cleared, he cannot access the object's contents. If the object requested is a subordinate, then the next higher object in the hierarchy (indicated with the subordinate in the object-type manager's

program profile) must have been previously and validly requested; if not, then the subordinate object request cannot be validated.

At access time (e.g., during open for a file, when intended access modes are declared), the requests for non-write access (e.g., read, execute) are evaluated and granted or denied based on privileges in the access lists of the program profile if a program (or object-type manager) is executing, the execution profile if the subject is attached to one, or the user profile. The order for examining these profiles to validate a request is as follows: the object-type manager invoker's program profile (be it a "regular" program or another manager), the execution profile, and finally the user profile. The pursuit stops when an object-entry is found; thus, the program profile's access list entries effectively override the execution profile's entries, which in turn override the user profile's entries. However, if a subject desires a concatenation of all privileges, he need only specify that desire to the system. Note that there are no privileges associated directly with the actual object. And note also that as one program invokes another, the privileges in the preceding program profile are disallowed.

An example of the utility of the override and concatenate principle is the case in which a project manager has read/write privileges to a project's master file in his user profile, but read-only privilege in the project's execution profile. He might attach to the execution profile, in which case he would have read-only access to the file and thus be treated just like other project members; he might not attach to the profile and thus have read/write privileges allowing him to update the master file; or he might be attached to the execution profile and find it necessary to update the master file while still attached—in this case he merely specifies his desire to concatenate privileges to the system and then performs the update.

For write access to the object, the above rules apply but writing is also governed by rule 3 (see "Object Hierarchies").

Nonaccessibility of Deleted Objects

A subject cannot be cleared to access an object if the object is in the deleted state (rule 5).

Simple Security Condition

A subject may be cleared for an object only if his current security level dominates the object's security level (rule 2).

Discretionary Security Condition

The discretionary security condition states that a subject is (further) constrained to access only those objects appearing in any of his valid access lists (rule 1). This means the subject must have the object listed in his user profile, in the execution profile to which he is attached (if any), in an attached program profile if a program is executing, or in some combination thereof.

**-Property*

The subject is restricted during write accesses: if the subject has read access to object 1, then to alter object 2, security level 2 must dominate security level 1 (rule 3). That is, information cannot be "declassified."

Object Hierarchies

A structure is imposed on all primary and subordinate object relationships; a parent-child association must be maintained that allows only directed, rooted trees and isolated points. This particular structure is desired in order to take advantage of the implicit control conventions of, and the wealth of experience with, logical data objects structured in this way. The construct used is called a hierarchy; a hierarchy specifies the progeny of each object so that structures of the type mentioned are the only possibilities.¹² Thus, objects cannot be inferior to two different objects, and a ring of objects is forbidden. Also, in traversing a path from root to leaf, the security levels must not decrease.

As an *invalid* example, consider a MIRAM file object and a subordinate SAT file object holding the data of the MIRAM file. Assume the SAT file is classified (Secret, {Medical}). Also assume the MIRAM file is classified (Top Secret, {Medical}). Access to the MIRAM file's data must proceed first to the MIRAM file and then to the SAT file itself. If a user has a security level the same as that of the SAT file, then he could never access the contents of the SAT file and therefore the MIRAM data, since the security levels going from root (MIRAM file) to leaf (SAT file) decrease.

It is important to note that a hierarchy does not preclude networks in the database management sense; that is, two logical files could be mapped onto one physical file. From a security point of view, however, there are merely three distinct user-accessible objects to protect; security is not aware of the network relationship among these.

Hierarchies also impose a restriction on access lists in execution profiles: they may not contain any nested profiles as objects except for the PHI profile, since a collection of profiles could form a ring, where EP1→EP2→EP3→EP1.

Object Authority Delegation

Typically the owner of a primary object holds all data and object privileges over that object. If he has the capability, the owner may delegate these privileges to others; these privileges can be specified as propagatable or nonpropagatable. No user may delegate more privileges than he currently holds.

For subordinate objects, a slightly different situation exists. The object-type manager holds in its program profile all privileges for a subordinate object required for making higher level objects. The manager can almost be considered its owner for that reason; however, the manager cannot exert discretionary control over the object. In addition, the user who owns the primary object is the indirect owner of the subordinate object and exerts indirect discretionary control over the subordinate object by delegating visibility to the primary object.

Public Objects

There is a system-defined execution profile, named the PHI profile, whose access category set is null. Since the null set is contained as a subset of all sets, and since the PHI profile, by convention, is a valid object for any user, execution, or program profile, the PHI profile is potentially accessible by all valid system users.

PHI execution profile attachment need never be explicitly requested by the subject. Subjects who have the PHI profile listed in a valid access list are automatically given access (restricted by nondiscretionary security) to the objects in the PHI profile's access list. Although a user may be attached to only one execution profile at a time, this maxim is waived for the PHI profile; the subject is continually attached, regardless of attachment to another execution profile.

For an object to be placed in the access list of the PHI profile, its access category set must be null (since the PHI profile's set is; see the next section). PHI profile objects are, therefore, considered public. For example, a subject who has PHI profile access needs only, as a minimum, a secret clearance level to access a secret object listed in the PHI profile's access list.

Execution Profile Access List Constraints

Since an execution profile is itself an object, security levels of objects listed in the execution profile must be dominated by the execution profile's security level. Thus, for example, only Medical or null access category set objects may be listed in the profile whose set is {Medical}. If this were not the case, then a circumstance would present itself where a subject would be on the access list (by virtue of attaching to the execution profile) of objects he is not permitted by nondiscretionary security to access.

Program Profile Access List Constraints

As with the execution profile, the program profile is an object whose access list also contains other objects. Their security levels must be dominated by the level of the program profile, which in turn must equal that of the program.

Tranquility Principle

Once created, an object's security level may not be changed, except as noted in the section "Formularies"; the clearance level may not be increased or decreased and the access category set may not have members added or removed. This applies for the life of the object (rule 4).

Object Declassification

On occasion there legitimately occur situations in which data must be downgraded or in which an object's security level must be changed. These actions, though, constitute direct

violations of security, specifically the *-property and tranquility principle; thus, they may only be handled by the security administrator, who is granted exceptional abilities. That is, a user must contact the security administrator (possibly outside the system) and provide him with information convincing him of the legitimacy of this particular request. When satisfied, the administrator makes the desired change. This capability can be extended only by the administrator to his designated assistants, of which there are a user-defined number.

Formularies

Since the security administrator, treated as just another user but with special abilities, must operate under the security policy as described, there is as currently specified no way of accomplishing the change just described. Thus, the concept of "trusted program" is born.

A trusted program is one that, when executing, is not constrained by the *-property. There are two kinds of trusted programs:

- Those exempted from the *-property because they have been independently shown not to violate the *-property; and
- those exempted specifically to downgrade data and thereby to violate the *-property.

The first kind of trusted program is necessary to support normal operating system functions, such as a program that saves (and restores) all disk files of varying security levels to tape; although it reads and writes files of differing security levels, it would not violate the *-property if it is shown that the read/write mechanism for one file is logically disjoint from the read/write of the next file.

The second type of *-property violator, known as a formulary, is exempted from the *-property for the express purpose of performing downgrading. This is accomplished under the explicit direction of the security administrator by the formulary writing the information with the higher classification into an object of relatively lower classification.

OBJECT CLASSIFICATION SUMMARY

The following provides a breakdown of the three basic discretionary visibilities of objects. All other object visibilities are either special cases or combinations of these.

- **Private Object**—A private object is one where the owner has sole access to the object, constrained only by self-imposed restrictions and nondiscretionary security. Object and data privileges are indicated solely in his program or user profile. All subordinate objects fall into this category.
- **Program/User Profile Private Object**—User-profile private objects and program-profile private objects are those whose access is limited to subjects having the object explicitly listed in their user or program profiles, respectively; access to these objects is, of course, further constrained by nondiscretionary security. Object and

data privileges are maximally limited to attributes in the user or program profiles.

- Execution Profile Private Object—When a user is attached to an execution profile, the subject has access to all objects specified in it. Object accessibility is constrained by nondiscretionary security and object and data accessibility are further constrained by privileges present in the profile. "Public" files are a special case of this classification.

CONCLUSION

The rules forming a security policy designed to provide multi-level security have been described. The policy is built around user, execution, and program profiles, which serve as repositories for access lists. The access lists indicate which primary and subordinate objects the subject, including object-type managers, may access and with what privileges. Although most literature on security policies (e.g., Bell and La Padula's work)^{8, 9, 10} associates the access list with the object rather than the subject, the two approaches can be shown to be mathematically equivalent. But from a user's point of view, ease of use is enhanced when a profile (i.e., subject) access list is used instead of an object access list: execution profiles provide the capability for groups of users to share common objects, and program profiles provide the capability for object-type managers to possess privileges independent of and unbeknown to the user. Thus, although security, because of its necessarily restrictive nature, seems inversely proportional to ease-of-use, a profile-oriented security policy presents a pleasant alternative to the mathematical models available today.

REFERENCES

1. Ames, S.R., Jr., and J. K. Millen. "Interface Verification for a Security Kernel." *Infotech State of the Art Report—System Reliability and Integrity* (Volume 2: Invited Papers), 1978, pp. 1-21.
2. Ames, Stanley R., Jr. "User Interface Multilevel Security Issues in a Transaction-Oriented Data Base Management System." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 120-124.
3. Anderson, James P. "Computer Security Requirements: An Investigation of Computer Security Costs." ESD-TR-77-24, James P. Anderson Company, Fort Washington, Pennsylvania, January 1976.
4. Anderson, James P. "Computer Security Technology Planning Study." ESD-TR-73-51, Volumes I and II, James P. Anderson and Company, Fort Washington, Pennsylvania, October 1972.
5. Anderson, James P. "Multics Evaluation." ESD-TR-73-276, James P. Anderson and Company, Fort Washington, Pennsylvania, October 1973.
6. Bell, D. E., and E. L. Burke. "A Software Validation Technique for Certification: The Methodology." ESD-TR-75-54, The Mitre Corporation, Bedford, Massachusetts, April 1975.
7. Bell, D. E., et. al. "Secure On-line Processing Technology—Final Report." ESD-TR-74-186, The Mitre Corporation, Bedford, Massachusetts, August 1974.
8. Bell, D. E., and L. J. La Padula. "Secure Computer Systems: Mathematical Foundations." ESD-TR-73-278, Volume I, The Mitre Corporation, Bedford, Massachusetts, November 1973.
9. Bell, D. E., and L. J. La Padula. "Secure Computer Systems: A Mathematical Model." ESD-TR-73-278, Volume II, The Mitre Corporation, Bedford, Massachusetts, November 1973.
10. Bell, D. E., and L. J. La Padula. "Secure Computer Systems: A Refinement of the Mathematical Model." ESD-TR-73-278, Volume III, The Mitre Corporation, Bedford, Massachusetts, April 1974.
11. Bell, D. E., and L. J. La Padula. "Secure Computer Systems: Mathematical Foundations and Model." M74-244, The Mitre Corporation, Bedford, Massachusetts, October 1974.
12. Bell, D. E., and L. J. La Padula. "Secure Computer Systems: Unified Exposition and Multics Interpretation." ESD-TR-75-306, The Mitre Corporation, Bedford, Massachusetts, March 1976.
13. Berson, T. A., and G. L. Barksdale, Jr. "KSOS—Development Methodology for a Secure Operating System." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 365-371.
14. Berstis, Viktors. "Security and Protection of Data in the IBM System/38." *Seventh Annual Symposium on Computer Architecture*, May 1980, pp. 245-252.
15. Biba, K. J. "Integrity Considerations for Secure Computer Systems." ESD-TR-76-372, The Mitre Corporation, Bedford, Massachusetts, April 1977.
16. Blakley, G. R. "Safeguarding Cryptographic Keys." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 313-317.
17. Boebert, W. Earl, Charles H. Bonneau, and John J. Carnall. "Secure Computing." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 49-63.
18. Branstan, Dennis K. "Privacy and Protection in Operating Systems." *Tutorial on Computer Security and Integrity*, 1977, pp. V-18—V-21.
19. Broadman, I. S. "Protection Techniques in Data Processing Systems to Meet User Data Security Needs." *Tutorial on Computer Security and Integrity*, 1977, pp. V-3—V-7.
20. Browne, Peter S., and Dennis K. Branstad. "Computer Security Tutorial Notes." *Tutorial on Computer Security and Integrity*, 1977, pp. 2-1—2-9.
21. Carroll, John M. *Computer Security*, Los Angeles: Security World Publishing Co., 1977.
22. Cash, James, Andrew B. Whinston, and William D. Haseman. "Security for the GPLAN System." *Information Systems* (Volume 2, Number 2), 1976, pp. 41-48.
23. Cotton, Ira W., and Paul Meissner. "Approaches to Controlling Personal Access to Computer Terminals." *Tutorial on Computer Security and Integrity*, 1977, pp. VI-42—VI-49.
24. Culpepper, L. M. "The Feasibility of a Method of Processing Encrypted Data." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 92-95.
25. DeMillo, Richard A., Richard J. Lipton, and Alan J. Perlis. "Social Processes and Proofs of Theorems and Programs." *Communications of the ACM* (Volume 22, Number 5), May 1979, pp. 271-280.
26. Denning, Dorothy E. "A Lattice Model of Secure Information Flow." *Communications of the ACM* (Volume 19, Number 5), May 1976, pp. 236-243.
27. Denning, Dorothy E., and Peter J. Denning. "Certification of Programs for Secure Information Flow." *Communications of the ACM* (Volume 20, Number 7), July 1977, pp. 504-513.
28. Denning, Dorothy E., Peter J. Denning, and Mayer D. Schwartz. "The Tracker: A Threat to Statistical Database Security." *ACM Transactions on Database Systems* (Volume 4, Number 1), March 1979, pp. 76-96.
29. Department of Defense. *Industrial Security Manual for Safeguarding Classified Information*. DOD 5220.22-M, U.S. Government Printing Office, Washington, D. C., April 1974.
30. Department of the Air Force. *Security—Information Security Program*. AFR 205-1, Headquarters U.S. Air Force, Washington, D.C., June 1976.
31. Dobkin, David, Anita K. Jones, and Richard J. Lipton. "Secure Databases: Protection Against User Influence." *ACM Transactions on Database Systems* (Volume 4, Number 1), March 1979, pp. 97-106.
32. Downey, Peter J. "MULTICS Security Evaluation: Password and File Encryption Techniques." ESD-TR-74-193, Volume III, Deputy for Command and Management Systems, Hanscom Air Force Base, Massachusetts, June 1977.
33. Evans, Arthur, Jr., William Krantrowitz, and Edwin Weiss. "A User Authentication Scheme Not Requiring Secrecy in the Computer." *Communications of the ACM* (Volume 17, Number 8), August 1974, pp. 437-442.
34. Federal Information Processing Standards Task Group 15: Computer Systems Security. "Glossary of Terminology for Computer Systems Security." September 1975.
35. Feiertag, Richard J. "A Formal Technique for Designing Secure Communications Systems." *NTC 78 Conference Record* (Volume 3), December 1978, pp. 36.2.1—36.2.5.
36. Feiertag, R. J., K. N. Levitt, and L. Robinson. "Proving Multilevel Security"

- urity of a System Design." *Proceedings of the Sixth ACM Symposium on Operating Systems Principles* (Volume 11, Number 5), November 1977, pp. 57-65.
37. Feiertag, Richard J., and Peter G. Neumann. "The Foundations of a Provably Secure Operating System (PSOS)." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 329-334.
 38. Fisk, A. J. "The Security Officers' View of Computer Security." *Proceedings of the 1977 Carnahan Conference on Crime Countermeasures*, April 1977, pp. 113-120.
 39. Ford Aerospace and Communications Corporation. "Secure Mini-computer Operating System (KSOS). Computer Program Development Specifications (Type B-5), Department of Defense Kernelized Secure Operating System." WDL-TR7811, Ford Aerospace and Communications Corporation, Palo Alto, California, March 1978.
 40. Franking, Neal A., et. al. "Providing Data Integrity and Security through Software Interfaces." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 102-105.
 41. Gligor, Virgil D. "Review and Revocation of Access Privileges Distributed through Capabilities." *IEEE Transactions on Software Engineering* (Volume SE-5, Number 6), November 1979, pp. 575-611.
 42. Gold, B. D., et. al. "A Security Retrofit of VM/370." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 335-344.
 43. Gold, B. D., et. al. "VM/370 Security Retrofit Program." *Proceedings of the National ACM Conference*, October 1977, pp. 411-418.
 44. Gudes, E., F. A. Stahl, and H. S. Koch. "A Model for Data Base Security." *Infotech State of the Art Report—System Reliability and Integrity* (Volume 2: Invited Papers), 1978, pp. 141-156.
 45. Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman. "Protection in Operating Systems." *Communications of the ACM* (Volume 19, Number 8), August 1976, pp. 461-471.
 46. Heinrich, Frank. "Computer Science and Technology: The Network Security Center: A System Level Approach to Computer Network Security." NBS Special Publication 500-21 (Volume 2), January 1978.
 47. Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Communications of the ACM* (Volume 17, Number 10), October 1974, pp. 549-557.
 48. Hsiao, David K., Douglas S. Kerr, and Stuart E. Madnick. *Computer Security*. New York: Academic Press, 1979.
 49. Hsiao, David K., Douglas S. Kerr, and Stuart E. Madnick. "Privacy and Security of Data Communications and Data Bases." *Proceedings on Very Large Data Bases; Fourth International Conference on Very Large Data Bases*, September 1978, pp. 55-67.
 50. IBM. *An Executive's Guide to Data Security—A Translation from an IBM Svenska AB Publication*. G320-5647, IBM Corporation, White Plains, New York, October 1975.
 51. IBM. *Data Security—Threats and Deficiencies in Computer Operations—A Report on a Completed Study—A Translation from an IBM Svenska AB Publication*. G320-5646, IBM Corporation, White Plains, New York, October 1975.
 52. IBM. "IBM System/38 Control Program Facility Programmer's Guide." SC21-7730-2, IBM Corporation, Rochester, Minnesota, May 1980.
 53. IBM. "IBM System/38 Control Program Facility Reference Manual—Control Language." SC21-7731-2, IBM Corporation, Rochester, Minnesota, May 1980.
 54. IBM. "OS/VS2 MVS Resource Access Control Facility (RACF) Command Language Reference." SC28-0733-2, IBM Corporation, Poughkeepsie, New York, June 1978.
 55. IBM. "OS/VS2 MVS Resource Access Control Facility (RACF) General Information Manual." GC28-0722-4, IBM Corporation, Poughkeepsie, New York, April 1978.
 56. Jones, Anita K., and Richard J. Lipton. "The Enforcement of Security Policies for Computation." *Journal of Computer and System Sciences* (Volume 17, Number 1), August 1978, pp. 35-55.
 57. Keedy, J. L. "On Structuring Operating Systems with Monitors." *The Australian Computer Journal* (Volume 10, Number 1), February 1978, pp. 23-27.
 58. Kiebertz, Richard B., and Abraham Silberschatz. "Capability Managers." *IEEE Transactions on Software Engineering* (Volume SE-4, Number 6), November 1978, pp. 467-477.
 59. Kohler, Barrie. "Factors Influencing the Requirement for Security Products." *Sperry Univac Spring Technical Symposium*, May 1978, pp. 1-1-1-1-1-2.
 60. Kurtzberg, J. M. "Online Dynamic Testing of Security and Integrity of Operating Systems." *IBM Technical Disclosure Bulletin* (Volume 17, Number 5), October 1974, pp. 1508-1512.
 61. Lee, T. M. P., and Robert E. Murphy. "Computer Security: Where and Whither." *Sperry Univac Fall Technical Symposium*, October 1978, pp. 1-1-1-1-1-8.
 62. Lee, Theodore M. P., et. al. "Processors, Operating Systems, and Nearby Peripherals—A Consensus Report." *Secure Operating System Technology Papers for the Seminar on the DOD Computer Security Initiative Program*, National Bureau of Standards, Gaithersburg, Maryland, July 1979, pp. 8-2-8-28.
 63. Lempel, Abraham. "Cryptology in Transition: A Survey." SCRC-RP-78-43, Sperry Research Center, Sudbury, Massachusetts, September 1978.
 64. Linde, Richard R. "Operating System Penetration." *Proceedings of AFIPS 1975 National Computer Conference* (Volume 44), 1975, pp. 361-368.
 65. Linden, Theodore A. "Operating System Structures to Support Security and Reliable Software." *ACM Computing Surveys* (Volume 8, Number 4), December 1976, pp. 409-445.
 66. Lipner, Steven B. (Session Chairman). "A Panel Session—Security Kernels." *AFIPS Conference Proceedings National Computer Conference* (Volume 43), 1974, pp. 973-980.
 67. Lipton, R. J., and L. Snyder. "A Linear Time Algorithm for Deciding Subject Security." *Journal of the Association for Computing Machinery* (Volume 24, Number 3), July 1977, pp. 455-464.
 68. Lohse, Ed. "Implementation & Use of the Data Encryption Standard within the Data Communications Environment." *Computer Science and Technology: Computer Security and the Data Encryption Standard*, NBS Special Publication 500-27, February 1977, pp. 84-93.
 69. McCauley, E. J., and P. J. Drongowski. "KSOS—The Design of a Secure Operating System." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 345-353.
 70. Michelman, Eric H. "The Design and Operation of Public-Key Cryptosystems." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 305-311.
 71. Millen, Jonathan K. "Formal Specifications for Security." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 115-119.
 72. Millen, Jonathan K. "Security Kernel Validation in Practice." *Communications of the ACM* (Volume 19, Number 5), May 1976, pp. 243-250.
 73. Minsky, Naftaly. "Intentional Resolution of Privacy Protection in Database Systems." *Communications of the ACM* (Volume 19, Number 3), March 1976, pp. 148-159.
 74. Morris, Robert, and Ken Thompson. "Password Security: A Case History." *Communications of the ACM* (Volume 22, Number 11), November 1979, pp. 594-597.
 75. Muftic, Sead, and Ming T. Liu. "The Design of a Secure Computer System." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 64-70.
 76. NBS. *Proceedings of the Second Seminar on the DOD Computer Security Initiative Program*. Gaithersburg, Maryland: National Bureau of Standards, January 1980.
 77. NBS. "Specifications for the Data Encryption Standard." Federal Information Processing Standards Publication 46, Department of Commerce, National Bureau of Standards, January 1977.
 78. Nelson, Jim. "New Requirements for Cryptosystems—A Tutorial." *Sperry Univac Fall Technical Symposium*, October 1979, pp. 6-2-1-6-2-8.
 79. Nelson, Ruth, and Joseph Jarzembowski. "Multilevel Security: An Overview and New Directions." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 41-48.
 80. Neumann, Peter G. "Computer System Security Evaluation." *Proceedings of AFIPS 1978 National Computer Conference* (Volume 47), 1978, pp. 1087-1095.
 81. Neumann, Peter G., et. al. "A Provably Secure Operating System: The System, Its Applications, and Proofs" (Final Report). SRI Project 4332, Stanford Research Institute, Menlo Park, California, February 1977.
 82. Nibaldi, G. H. "Proposed Technical Evaluation Criteria for Trusted Computer Systems." M79-225, The Mitre Corporation, Bedford, Massachusetts, October 1979.
 83. Nibaldi, G. H. "Specification of a Trusted Computing Base." M79-228, The Mitre Corporation, Bedford, Massachusetts, November 1979.
 84. Padlipsky, M. A., K. J. Biba, and R. B. Neely. "KSOS—Computer Network Applications." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 373-381.

85. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM* (Volume 15, Number 12), December 1972, pp. 1053-1058.
86. Parnas, David L. "The Use of Precise Specifications in the Development of Software." *Information Processing 77* (Volume 7), August 1977, pp. 861-867.
87. Peters, Bernard. "Security Considerations in a Multiprogrammed Computer System." *AFIPS Conference Proceedings—SJCC* (Volume 30), April 1967, pp. 283-286.
88. Pirkola, Gary C., and John W. Sanguinetti. "The Protection of Information in a General Purpose Time-Sharing Environment." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 106-114.
89. Popek, Gerald J., and David A. Farber. "A Model for Verification of Data Security in Operating Systems." *Communications of the ACM* (Volume 21, Number 9), September 1978, pp. 737-749.
90. Popek, Gerald J., and Charles S. Kline. "A Verifiable Protection System." *ACM Sigplan Notices* (Volume 10, Number 6), June 1975, pp. 294-304.
91. Popek, Gerald J., and Charles S. Kline. "Issues in Kernel Design." *AFIPS Conference Proceedings National Computer Conference* (Volume 47), June 1978, pp. 1079-1086.
92. Popek, Gerald J., and Charles S. Kline. "Verifiable Secure Operating System Software." *AFIPS Conference Proceedings National Computer Conference* (Volume 43), 1974, pp. 145-151.
93. Popek, Gerald J., et. al. "UCLA Secure Unix." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 355-364.
94. Purdy, George B. "A High Security Log-in Procedure." *Communications of the ACM* (Volume 17, Number 8), August 1974, pp. 442-445.
95. Ragland, Larry C. *A Verified Program Verifier*. Ph.D. Dissertation, The University of Texas at Austin, Computer Science, 1973.
96. Reymont Associates. "Detecting and Preventing Misuse of Data Processing Systems." *Reymont Reports—Data and Direction for Business & Industry*, Rye, New York, 1978.
97. Rhode, R. "Secure Multilevel Virtual Computer Systems." ESD-TR-74-370, The Mitre Corporation, Bedford, Massachusetts, February 1975.
98. Robinson, Lawrence, and Oliver Roubine. "SPECIAL—A SPECification and Assertion Language." Stanford Research Institute, Menlo Park, California, January 1977.
99. Robinson, Lawrence, et. al. "A Formal Methodology for the Design of Operating System Software." *Current Trends in Programming Methodology—Software Specification and Design* (Volume 1), 1977, pp. 61-110.
100. Robinson, Lawrence, and Karl N. Levitt. "Proof Techniques for Hierarchically Structured Programs." *Communications of the ACM* (Volume 20, Number 4), April 1977, pp. 271-283.
101. Roubine, Oliver. "The Design and Use of Specification Languages." Stanford Research Institute, Menlo Park, California, October 1976.
102. Roubine, Oliver, and Lawrence Robinson. "SPECIAL Reference Manual" (3rd Edition). Stanford Research Institute, Menlo Park, California, January 1977.
103. Saal, Harry J., and Israel Gat. "A Hardware Architecture for Controlling Information Flow." *Proceedings of the Fifth Annual Symposium on Computer Architecture*, April 1978, pp. 73-77.
104. Saltzer, Jerome H., and Michael D. Schroeder. "The Protection of Information in Computer Systems." *Proceedings of the IEEE* (Volume 63, Number 9), September 1975, pp. 1278-1308.
105. Schaefer, Marvin, et. al. "Program Confinement in KVM/370." *Proceedings of the National ACM Conference*, October 1977, pp. 404-410.
106. Schiller, W. L. "Design and Specification of a Multics Security Kernel." ESD-TR-77-259, The Mitre Corporation, Bedford, Massachusetts, November 1977.
107. Schiller, W. L. "Design of a Security Kernel for the PDP-11/45." ESD-TR-73-294, The Mitre Corporation, Bedford, Massachusetts, December 1973.
108. Schiller, W. L. "The Design and Specification of a Security Kernel for the PDP-11/45." ESD-TR-75-69, The Mitre Corporation, Bedford, Massachusetts, May 1975.
109. Schroeder, Michael D. "Engineering a Security Kernel for Multics." *ACM Operating Systems Review* (Volume 9, Number 5), November 1975, pp. 25-32.
110. Schroeder, Michael D., David D. Clark, and Jerome H. Saltzer. "The Multics Kernel Design Project." *Proceedings of the Sixth ACM Symposium on Operating System Principles* (Volume 11, Number 5), November 1977, pp. 43-56.
111. Shamir, Adi. "How to Share a Secret." *Communications of the ACM* (Volume 22, Number 11), November 1979, pp. 612-613.
112. Shankar, K. S., and C. S. Chandrasekaran. "The Impact of Security on Network Requirements." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 96-100.
113. Sipple, Ralph E. "Hardware Insurance for Software: Protecting Software from Software Using Software." *Sperry Univac Fall Technical Symposium*, October 1979, pp. 1-2-1-1-2-5.
114. Smid, Miles E. "Computer Science & Technology: A Key Notarization System for Computer Networks." NBS Special Publication 500-54, October 1979.
115. SPERRY UNIVAC. "Crypto System Key Management, Generation and Authentication." SAO1733, Sperry Univac, Blue Bell, Pennsylvania, June 1979.
116. SPERRY UNIVAC. *Security CPSD. B-42512 (Update D2)*, Sperry Univac, Blue Bell, Pennsylvania, 1977.
117. SPERRY UNIVAC. *Series 1100 Security. B-42505*, Sperry Univac, Blue Bell, Pennsylvania, 1979.
118. Stahl, Fred, Ehud Gudes, and Harvey Koch. "The Coordination of Cryptographic and Traditional Access Control Techniques for Protection in Computer Systems." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 86-91.
119. Stork, D. F. "Downgrading in a Secure Multilevel Computer System: The Formulary Concept." ESD-TR-75-62, The Mitre Corporation, Bedford, Massachusetts, May 1975.
120. Sugarman, Robert. "On Foiling Computer Crime." *IEEE Spectrum* (Volume 16, Number 7), July 1979, pp. 31-41.
121. Sykes, David J. "The Management of Encryption Keys." *Computer Science and Technology: Computer Security and the Data Encryption Standard*, NBS Special Publication 500-27, February 1977, pp. 46-53.
122. Tasker, P.:S., and D. E. Bell. "Design and Certification Approach: Secure Communications Processor." ESD-TR-73-129, The Mitre Corporation, Bedford, Massachusetts, June 1973.
123. Turn, Rein. "Privacy and Security in Transnational Data Processing Systems." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 283-291.
124. Turn, Rein, and H. E. Petersen. "Security of Computerized Information Systems." The Rand Corporation, Santa Monica, California, July 1970.
125. U. S. Government. "Purchase Description No. RD1.094, Research and Development of Provably Secure Operating System Design and Specification." March 1979.
126. Walker, Bruce J., Richard A. Kemmerer, and Gerald J. Popek. "Specification and Verification of the UCLA Unix Security Kernel." *Communications of the ACM* (Volume 23, Number 2), February 1980, pp. 118-131.
127. Walker, Stephen T. "Department of Defense Computer Security Initiative." *National Telecommunications Conference 1978 Conference Proceedings* (Volume 3), December 1978, pp. 36.1.1-36.1.2.
128. Walter, K. G., et. al. "Structured Specification of a Security Kernel." *ACM Sigplan Notices* (Volume 10, Number 6), June 1975, pp. 285-293.
129. Ware, Willis H. (Editor). "Security Controls for Computer Systems, Report of Defense Science Board Task Force on Computer Security." OSC msp 831, an.s 42
130. Weissman, C. "Security Controls in the ADEPT-50 Timesharing System." *AFIPS Conference Proceedings—FJCC* (Volume 35), November 1969, pp. 119-133.
131. Westin, Alan F. "The Impact of Computers on Privacy." *Datamation* (Volume 25, Number 14), December 1979, pp. 190-194.
132. Williams, John M. "Selected System Concepts for Encryption." *NTC 78 Conference Record* (Volume 3), December 1978, pp. 26.3.1-26.3.5.
133. Wood, Helen M. "Computer Science & Technology: The Use of Passwords for Controlled Access to Computer Resources." NBS Special Publication 500-9, May 1977.
134. Wood, Helen M. "On-line Password Techniques." *Symposium Proceedings Trends and Applications 1977: Computer Security and Integrity*, May 1977, pp. 27-31.
135. Woodward, John P. L. "Applications for Multilevel Secure Operating Systems." *AFIPS Conference Proceedings 1979 National Computer Conference* (Volume 48), June 1979, pp. 318-328.
136. Young, Charles R. "Security Policy." *Sperry Univac Spring Technical Symposium*, May 1980, pp. 3-5-1-3-5-10.

Distributed task force scheduling in multi-microcomputer networks*

by ANDRÉ M. VAN TILBORG
Calspan Advanced Technology Center
Buffalo, New York

and

LARRY D. WITTIE
State University of New York at Buffalo
Buffalo, New York

ABSTRACT

Efficient task scheduling techniques are needed for microcomputer networks to be used as general purpose computers. The Wave Scheduling technique, developed for the MICRONET network computer, co-schedules groups of related tasks onto available network nodes. Scheduling managers are distributed over a logical control hierarchy. They subdivide requests for groups of free worker nodes and send waves of requests towards the leaves of the control hierarchy, where all workers are located. Because requests from different managers compete for workers, a manager may have to try a few times to schedule a task force. Each task force manager actually requests slightly more workers than it really needs. It computes a request size which minimizes expected scheduling overhead, as measured by total idle time in worker nodes. Using a Markov queueing model, it is shown that Wave Scheduling in a network of microcomputers is almost as efficient as centralized scheduling.

INTRODUCTION

A promising way to utilize the emerging VLSI technology is to construct network computers. Network computers are MIMD (Multiple-Instruction-stream, Multiple-Data-stream) computers.¹ Each is built as a network of autonomous computers, linked by a high bandwidth communications system and a common operating system to form a single computing system. Network computers are designed to allow parallel solution of problems in such diverse areas as numerical mathematics, simulation, and artificial intelligence. They also offer high fault-tolerance and almost unbounded modular extensibility. There are several such experimental machines in existence or under development today, including MICRONET at

SUNY/Buffalo,² Cm* at Carnegie-Mellon University,³ X-Tree at UC-Berkeley,⁴ MuNet at MIT,⁵ and Arachne at Wisconsin-Madison.⁶

The physical construction of network computers is, of course, a challenging task. However, software organization is the key to effectiveness in parallel machines.⁷ Producing distributed operating systems for network computers is still a challenging research problem because many of the concepts devised for uniprocessor operating systems are not extensible to networks, especially since most networks do not have globally shared memory.

One problem in designing distributed operating systems is the assignment of network nodes to application program tasks. To achieve parallel execution, individual tasks of a multi-task parallel program must be "co-scheduled".⁸ Conventional techniques for task scheduling in both uni- and multi-processors assume that there is a single memory space in which a unique system scheduler can keep up-to-date tables of resource assignments. Most network computers are loosely coupled, i.e. they have no common memory. Cm* is a tightly-coupled network with shared memory. Even in Cm*, memory contention causes severe delays when as few as five percent of memory references are to shared storage areas.⁹ For a very large network, it is impractical to store all scheduling data in one location. To reduce the frequency of references to shared memory, operating systems for large tightly-coupled network computers will probably use scheduling techniques similar to those for loosely-coupled networks.

To solve user problems in parallel, programming language compilers for network computers must generate separate simultaneously executable task modules. Such collections of related tasks are known as task forces.¹⁰ The task force scheduling problem (also called task assignment) is to co-schedule tasks onto available network nodes. To be useful in large networks, the scheduling technique must be de-centralized and should adapt efficiently to different network interconnection topologies. Several techniques for scheduling tasks in network computers have been suggested. Among

*This work was supported by National Science Foundation grant MCS78-03166. Construction of the MICRONET network computer has been funded by NSF equipment grants MCS77-09213 and MCS80-06925.

these techniques are contract bid scheduling,¹¹ diffusion scheduling,⁵ and Wave Scheduling.¹² Wave Scheduling is used by the MICROS operating system¹³ to co-schedule task forces in MICRONET. Wave Scheduling is intended to be applicable to networks of thousands of nodes. The primary objectives of this paper are to review the Wave Scheduling procedure, to develop a model of efficiency for Wave Scheduling, and to compare Wave Scheduling with contract bid and diffusion scheduling.

NETWORK ORGANIZATION FOR WAVE SCHEDULING

Before discussing the details of Wave Scheduling, it will be helpful to explain the model of network computation which Wave Scheduling assumes. First, it is assumed that the task processors of the host network are homogeneous and that, aside from peripheral connections and advantages of physical location, any processor is as capable of executing a task as any other processor. While this homogeneity assumption is restrictive, it is not debilitating: network computers such as MICRONET, X-Tree, MuNet, Arachne, and Cm* all very nearly satisfy the constraint. The purpose of this assumption is to eliminate special cases. One consequence will be that, in a network with heterogeneous nodes, task forces may not execute as efficiently as they might if a more detailed matching of tasks to processors were performed.

To remove unnecessary complexity from the description of Wave Scheduling, every node is assumed capable of executing exactly one unit-size user task at any time. Actually, this assumption could be relaxed somewhat to require only that tasks be unit-size and that each node be capable of executing an integral number of such tasks concurrently. It would not be difficult to extend the following analysis to nodes which execute several tasks concurrently.

It is also assumed that the scheduling procedure does not have advance knowledge of the task force arrival process, individual task resource requirements, or the volume of communication within a task force. This assumption makes Wave Scheduling a realizable technique for general purpose computing environments.

The most important assumption in Wave Scheduling is that the host network's operating system supports the hierarchical high-level operating system schema described in several recent papers.^{13,14,15} Regardless of the physical interconnection topology in MICRONET, the MICROS operating system initializes a tree-like control superstructure at bootstrap time. The hierarchical structure consists of so-called 'worker' nodes at the leaves of the control tree and 'manager' nodes at higher levels. Each manager has on the order of five to twenty immediate subnodes which send it summarized status information. Depending on the size of the host network, there may be many or few levels of managers. To reduce susceptibility to single-fault failures, the control tree reduces to a group of three to ten "supermanagers" rather than one master node at the root. Hierarchical control structures with efficient tree paths between nodes can be established automatically in arbitrarily connected network computers.¹⁴ Such a control schema greatly simplifies operating system development because it

hides details of the interconnection topology except at the lowest communication link level.

DESCRIPTION OF WAVE SCHEDULING

Assume that a task force of size S , which needs S nodes to execute, enters a queue of ready task forces at an arbitrary network node. Task forces may appear at any level of the control hierarchy, at managers as well as worker nodes. All managers in the hierarchy are empowered to try to schedule task forces which are no larger than some dynamically changing fraction of the number of worker nodes in the subtree of which the manager is the root. If a task force enters the network at a level which cannot schedule a task force of size as large as S , the task force descriptor is passed up the tree until a suitable manager is reached. Likewise, the descriptor of a task force which enters at too high a level in the hierarchy is passed down until it reaches the level which minimally satisfies its size. In either case, a manager at the appropriate level becomes the Task Force Master (TFM) for the task force. Task forces too large to be handled even by the entire network are rejected when they reach the top level.

Each TFM keeps track of the number of non-busy workers in the subtree below it. Worker counts are regularly updated by sending status summaries from lower levels of the subtree. Because communication is not instantaneous and because many TFMs are competing for the same worker nodes, each TFM may have slightly inaccurate status information. Reasonably accurate status information can probably be maintained with the rate of arrival of updates at least two times the rate of task force arrivals to a particular level.

Task Force Masters are responsible for reserving enough nodes for the task forces which they control. Competition for workers occurs both within and between managers. The TFM for a task force which needs S nodes gauges the activity of the network in its subtree and tries to reserve $R \geq S$ workers to get S . R is chosen to minimize wasted worker time in the network by balancing the processing power wasted by actually reserving too many ($> S$) workers with that wasted by reserving so few ($< S$) that another scheduling attempt is required. In a separate paper¹² it has been shown that a useful approximation for R is

$$R = \frac{S * k}{1 - u}$$

where utilization u is the instantaneous fraction of busy worker nodes in the subtree of a TFM and k is a value in the range 1.1 to 1.3, which slightly depends on u . A TFM tries to schedule a task force of size S whenever at least R workers in its subtree are believed to be available, regardless of how recently a previous attempt for that task force failed. Since the utilization of workers in a subtree is not constant, the values of R may well differ in consecutive scheduling passes for the same task force.

The request for R workers is divided among the submanagers of the TFM and proceeds down the tree as a wave of subrequests. To give large task forces a better chance of being scheduled, each manager services a request from its

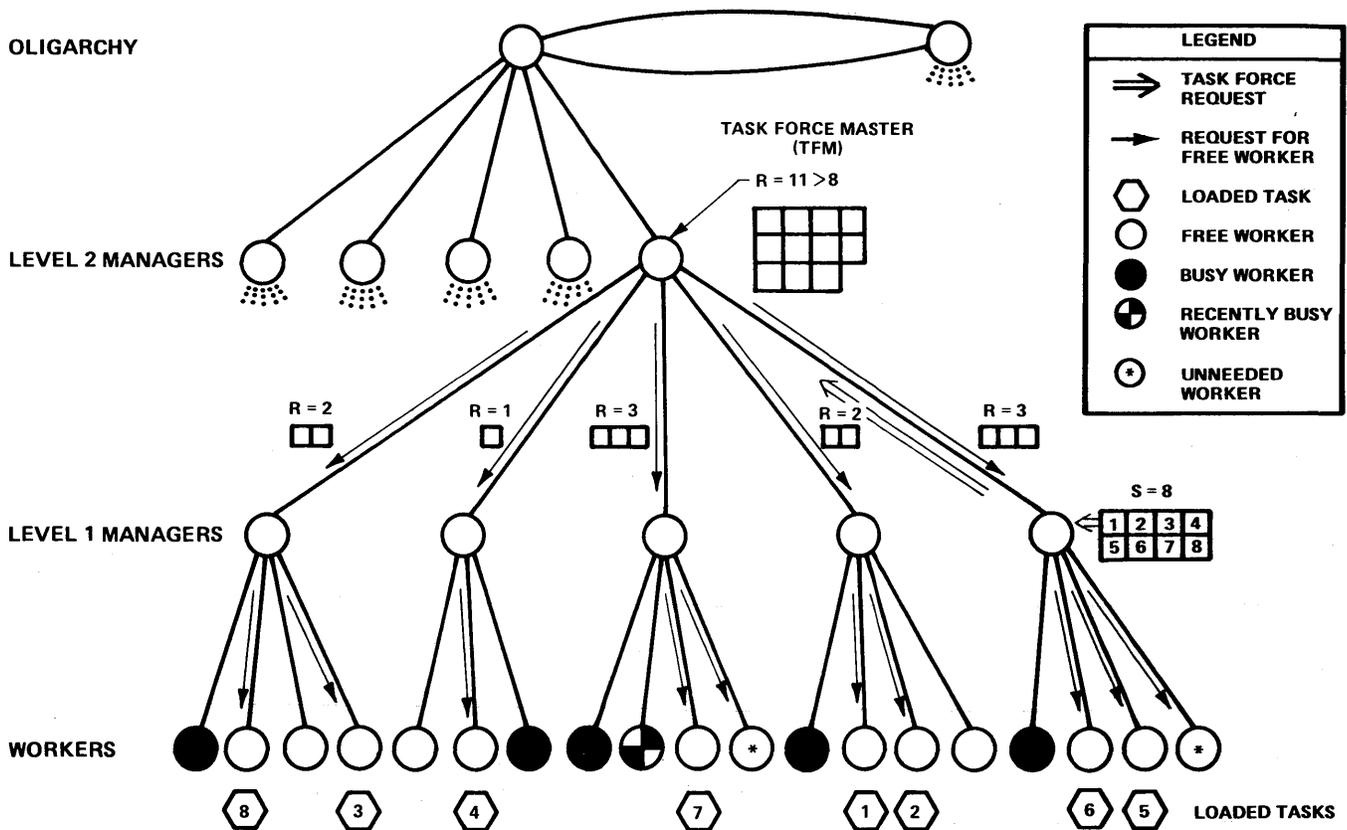


Figure 1—A task force of size $S=8$ tasks enters the network at the rightmost level-1 manager. The request is passed one level upward and a wave of subrequests for $R=11$ workers is sent towards the leaves of the tree. A total of ten workers are actually reserved. Eight of these workers are loaded with task modules (marked by hexagons) while the other two (marked by *) are released. The worker marked as “recently busy” was believed to be available by the TFM because the status summaries had not yet been updated.

manager before any local requests. Each subrequest is repeatedly divided until it reaches the lowest (level-1) managers in the control hierarchy. Managers at that level store accurate information regarding the status of (level-0) worker nodes under their direct scheduling control. The wave of subrequests to reserve R nodes for a size S task force is shown in Figure 1.

Figure 1 shows a task force consisting of $S = 8$ tasks entering the network at the rightmost level-1 manager. The task force is too large to be scheduled by that manager, so it is passed one level upward. From there a request for $R = 11$ worker nodes travels as a wave of subrequests downwards through the subtree of the TFM. A total of ten worker nodes are reserved by the wave. Only two workers were actually idle in the center cluster, but the TFM thought there were three idle workers. The tasks of the task force are assigned to the nodes indicated by hexagons. The unneeded workers marked with an asterisk (*) are released by the TFM.

When a subrequest reaches a level-1 manager, that manager reserves as many as possible of the requested number of workers. (Of course, a level-1 manager which is itself the TFM need not reserve any more than S nodes.) The level-1 manager tells its level-2 manager how many workers have been reserved. Managers at each level wait for responses to all of

their subrequests for a particular task force before passing the sum upwards. Eventually the original requesting TFM is told how many workers have been reserved.

To prevent deadlock and cope with hardware failures, each level of the hierarchy observes timeout rules. For example, a submanager reports the number of worker nodes reserved for its manager after a fixed timeout interval, regardless of whether all its subrequests have been answered.

If the number of workers which are actually reserved for a requesting TFM is less than S , the scheduling pass is considered a failure. The TFM sends commands releasing all of the reserved workers. To avoid performance penalties, it should not let timeouts release workers. The unscheduled task force returns to the queue of task forces ready to be scheduled for execution by the local node. This procedure can go on until the task force is scheduled or until a limiting number of failures is reached. After too many failures, the task force is passed up one level to a new TFM to increase the likelihood of reserving enough workers. The values of request size R which are computed by each TFM are large enough that only a few failures may occur before a task force is successfully scheduled. For example, when the host network is running at 50% of saturation, there are an average of two scheduling

attempts before a task force is successfully scheduled. Thus, single-node scheduling overhead can be kept low.

If the requesting TFM is told that more than enough workers were reserved, it broadcasts the identifier of the task force's host node to all of the reserved workers. Each worker requests n executable task module directly from the host node. The host node distributes tasks until the entire task force has been loaded by worker nodes. Some workers may not receive a task module because their requests arrive at the host node after all task modules have been distributed. Once all the task modules have been loaded, the host node informs the TFM of the location of the initial root task and unneeded workers are released by the TFM. The TFM can then start execution of the task force.

SCHEDULING EFFICIENCY

The efficiency of Wave Scheduling in microcomputer networks compares favorably with that of an idealized central scheduler. A useful measure of efficiency is the average total time a task force spends in the host network. This value is the sum of queuing and service times.

Let W represent the number of worker nodes in the host network. Assume all workers execute tasks at the same rate. For simplicity, assume that task force arrivals can be modeled as a Poisson process with parameter Λ . Thus, the interarrival time probability distribution function for task forces is

$$A(t) = 1 - e^{-\Lambda t}$$

and their probability density function is

$$a(t) = \Lambda e^{-\Lambda t}$$

Assume that the i th task to reach the network requires x_i seconds of CPU time to execute and that an average task force contains \bar{S} tasks. For convenience, define the task arrival rate $\lambda :: = \bar{S} * \Lambda$. For stability, assume that the network is in steady-state equilibrium, i.e., there exists a single node service rate μ such that

$$\frac{1}{\mu} = \lim_{i \rightarrow \infty} x_i = \bar{x}$$

and the expected number of busy nodes is less than the number of workers, i.e.

$$\frac{\lambda}{\mu} < W$$

Efficiency of Idealized Centralized Scheduler

Suppose that an ideal central scheduler knows the state of all worker nodes and is used to schedule all arriving task forces. As a first approximation, the W workers appear as a single server which is W times as fast as an individual network node. Thus, for the purposes of computing average total system time, the single scheduler case can be modeled by an $M/M/1$ queue¹⁶ where the traffic intensity equals

$$\rho = \frac{\lambda}{W\mu}$$

The average time spent in the system by a task force when a central scheduler is used, T_C , can then be found using Little's Result¹⁷

$$\bar{N} = \Lambda T_C = \frac{\lambda}{\bar{S}} * T_C$$

where \bar{N} is the average number of customers in the system. For an $M/M/1$ queue,

$$\bar{N} = \frac{\rho}{1 - \rho}$$

Thus,

$$\begin{aligned} T_C &= \bar{N} / \Lambda \\ &= \left[\frac{\rho}{1 - \rho} \right] (\bar{S} / \lambda) \\ &= \bar{S} \frac{1}{1 - \frac{\lambda}{W\mu}} \\ &= \frac{\bar{S}}{W\mu - \lambda} \end{aligned} \quad (1)$$

The excess service capacity in the system is $(W * \mu - \lambda)$ tasks/sec. Thus the average total time, T_C , to service a new force of \bar{S} tasks is given by Eq. (1).

Efficiency of Wave Scheduling

It is now necessary to estimate T_w , the average total time a task spends in the network when Wave Scheduling is used. For simplicity, assume that there are M task force masters, each of which tries to schedule a stream of task forces of average size \bar{S} arriving as a Poisson process with parameter Λ/M . In other words, the single task force stream which was used in the single scheduler case is divided into the equivalent form of M substreams with exponential interarrival times, each M times as long. The i th task still brings with it x_i units of work. However, because of the interference between competing task force masters, Wave Scheduling adds scheduling overhead to each task force. This overhead grows with increasing network utilization. For purposes of comparison with the single scheduler case, assume that average network utilization is in fact $\lambda/(W * \mu)$. From earlier results,¹² the average amount of extra scheduling work per node in a task force is found to be

$$X_{sched} = \left[\frac{R_{opt} \left(1 - \frac{\lambda}{W\mu} \right)}{\sum_{j=\bar{S}}^{R_{opt}} \binom{R_{opt}}{j} \left(1 - \frac{\lambda}{W\mu} \right)^j \left(\frac{\lambda}{W\mu} \right)^{R_{opt}-j}} - \bar{S} \right] * \bar{c} / \bar{S} \quad (2)$$

where \bar{c} is the average node reservation cost for each task of an average size \bar{S} task force and R_{opt} is the value of R which minimizes wasted worker time. Thus, the average single-task service time in Wave Scheduling is

$$\bar{x}_w = \bar{x} + X_{sched}$$

T_w can now be computed by looking at just one of the M task force masters because interference from other TFMs can be included as part of the work which an average task force

brings to the network. Letting average steady state service rate μ_w be

$$\mu_w ::= 1/\bar{x}_w$$

and (work) traffic intensity for Wave Scheduling be

$$\rho_w = \frac{\lambda/M}{W\mu_w/M}$$

Little's Result again can be used to compute the average total time in the host network for a task force when Wave Scheduling is used:

$$\begin{aligned} T_w &= \frac{\bar{N}/M}{\Lambda/M} \\ &= \bar{S} \left[\frac{\rho_w}{1-\rho_w} \right] (1/\lambda) \\ &= \frac{\bar{S}}{W\mu_w - \lambda} \end{aligned} \quad (3)$$

Comparison of Wave Scheduling to Central Scheduling

From Eqs. (1), (2), and (3), the relative efficiency of Wave Scheduling with respect to centralized scheduling is thus

$$\begin{aligned} Erel &= T_c/T_w \\ &= \left(\frac{\bar{S}}{W\mu - \lambda} \right) / \left(\frac{\bar{S}}{W\mu_w - \lambda} \right) \\ &= \frac{W\mu_w - \lambda}{W\mu - \lambda} \end{aligned} \quad (4)$$

Since

$$\begin{aligned} 1/\mu_w &= \bar{x} + Xsched \\ &= 1/\mu + Xsched \\ &= \frac{1 + \mu * Xsched}{\mu} \end{aligned}$$

the relationship of μ_w to μ is

$$\mu_w = \frac{\mu}{\mu * Xsched + 1} \quad (5)$$

Combining Eqs. (4) and (5) yields

$$Erel = \frac{W\mu}{\mu * Xsched + 1} - \lambda$$

Since $Xsched$ is always greater than or equal to zero, μ_w is never greater than μ . If $Xsched$ were actually equal to zero, i.e., reserving worker nodes was cost-free, Eq. (6) would yield a constant 1, indicating that Wave Scheduling and central scheduling were equally efficient. In practical situations, $Xsched$ is greater than zero, of course.

Using Eq. (6), it is not difficult to compute the relative efficiency of Wave Scheduling for a given network. For example, in a network with $W = 1000$ workers, average task force size $\bar{S} = 10$, task force arrival rate $\Lambda = 50$ task forces/sec,

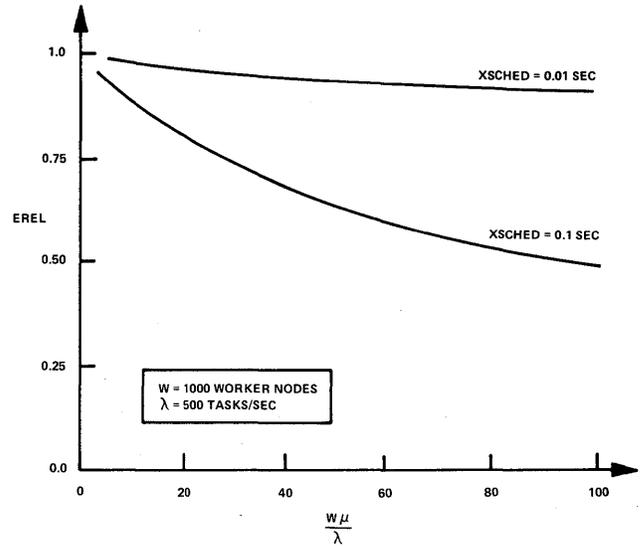


Figure 2—The relative efficiency $Erel$ of Wave Scheduling with respect to central scheduling is shown for typical scheduling overhead $Xsched$ and excess processing capacity.

single-node service rate $\mu = 1$ task/sec, and single-node scheduling overhead $Xsched = .1$ sec, the relative efficiency $Erel = 82\%$. This means that the expected total system time in the central scheduler case is 82% of the expected time when Wave Scheduling is used. In the same network, reducing scheduling overhead to $Xsched = .01$ sec increases the efficiency of Wave Scheduling to 98%. Since the number of scheduling attempts which an average task force undergoes is small, and in practical networks would lead to scheduling overheads much less than .1 sec/worker, Wave Scheduling is almost as efficient as centralized scheduling. In other words, the high degree of fault-tolerance and load balancing in Wave Scheduling can be achieved without great sacrifice in run-time efficiency.

To convey some intuition as to the nature of the results for realistic combinations of system parameters, Figure 2 plots $Erel$ against various values of network capacity $W * \mu$ and a constant task arrival rate λ . It is evident that the efficiency of Wave Scheduling declines as the cost of worker node reservations increases and also as the excess capacity ($W * \mu / \lambda$) of the network increases. Both of these effects are to be expected. The first occurs because the cost of worker reservations does not appear in the single-scheduler model at all. Therefore, as that cost increases in the Wave Scheduling model, it is bound to decrease the relative efficiency of Wave Scheduling.

The second effect, a decrease in relative efficiency as network excess capacity increases, may not be quite as intuitive. As excess capacity increases, the benefit to the local schedulers in Wave Scheduling is not as large as it is to the central scheduler. As a result, the decrease in system time for task forces scheduled by Wave Scheduling is not as large as the decrease experienced by task forces in the single-scheduler case. The result is a relative decline in the efficiency of Wave Scheduling, although the absolute time in system has actually decreased for both models. This effect has been encountered in other situations and is discussed briefly by Schwartz.¹⁸

COMPARISON WITH OTHER SCHEDULING TECHNIQUES

Several other task force scheduling techniques besides Wave Scheduling have been proposed. The objective of this section is to describe some of the more important ones and to compare them with Wave Scheduling.

Contract Bid Scheduling

Contract bid scheduling was devised as a technique to distribute tasks among the nodes of a distributed sensor network.¹¹ The collection of nodes is called a contract net and the execution of a task is dealt with as a contract between two nodes. Each node in the net takes on one of two roles related to the execution of an individual task: manager or contractor. A manager is responsible for monitoring the execution of a task and processing the results of its execution. A contractor is responsible for the execution of the task. These roles are taken on dynamically by nodes during the course of problem solving.

Contract negotiation is initiated in essentially two different ways. The normal method is for a node that generates a task to advertise its existence with a task announcement message. Such announcements can be broadcast generally, broadcast in a limited region, or sent point-to-point, depending on the amount of information which the prospective manager has about other nodes in the net. Other nodes "listen" for task announcements and, when they are idle, submit bids on those for which they are suited. The manager evaluates the bids and awards contracts to the most suitable nodes.

The normal method may not work for one of two reasons:

1. There are no available nodes.
2. No nodes have the necessary data to execute the task.

To deal with these situations the normal bid protocol is modified to produce a second slightly more complex scheme in which currently active bidder nodes indicate whether they could perform a contract if they were not busy.

Diffusion Scheduling

Diffusion scheduling is characterized by the simple strategy of transferring tasks from areas with high concentrations to areas with lower concentrations. It is claimed⁵ that any concentration of workload at a particular point will gradually flatten and spread out with time, much as impurities diffuse through a crystal lattice. There are several task force scheduling techniques which can be classified as diffusion techniques. A diffusion strategy is the scheduling mechanism for events and objects in MuNet, where the name originated. A slightly different scheme which claims to establish balanced local "broadcast regions" is suggested for the CHoPP multi-microprocessor.¹⁹ The assignment scheme used by Arachne⁶ might also be considered a form of diffusion scheduling. Tasks in Arachne are assigned to network nodes by means of a fixed, cyclic priority mechanism. If a task is passed to a node which

is already busy, the task is sent on to a pre-determined successor node which in turn can pass the task on again.

Fixed Assignment Scheduling

Probably the simplest form of task force scheduling, in the sense that it is most easily implemented, is fixed manual assignment of tasks to processors. Although it could be argued that such techniques are not of much lasting interest, both of the operating systems developed for Cm* currently use fixed assignment scheduling.^{8,10} A programmer is responsible for distributing the tasks of his task force onto some subset of the processing elements. He is assisted in doing so by a language named TASK which makes it possible to build task forces and to reference nodes of the multi-microprocessor.

Comparison of Techniques

Wave Scheduling differs from the other task force scheduling techniques briefly described above in several ways. First, Wave Scheduling is the only technique which uses a network control structure to assist in scheduling task forces. The control hierarchy for Wave Scheduling provides a means for assigning task forces to well-qualified network managers. Since task forces are not assigned to network nodes in advance by a programmer, but rather by TFMs which are dynamically selected at run-time, the reliability of Wave Scheduling is better than in fixed manual assignment where nodes which a programmer selects may be disabled before a task force is executed.

A second way in which Wave Scheduling differs from other proposed task force scheduling techniques is that it includes a mechanism for avoiding static deadlock. Because the scheduling passes made by the Wave Scheduling procedure 'roll-back' if they are unsuccessful, network nodes are never reserved indefinitely. On the other hand, in diffusion scheduling, static deadlock can occur because competing task forces are not made aware that each holds resources which the others need. Fixed assignment scheduling can lead to deadlock too, if separate programmers try to use the same network nodes simultaneously. Contract bid scheduling also does not include any provisions for detecting nor avoiding deadlocks.

Finally, Wave Scheduling is a scheduling technique extensible to networks of thousands of computers. Increasing the size of the host network has no effect on the basic scheduling procedure. In fixed and cyclic diffusion scheduling, changes in network topology are catastrophic because they can make it impossible to schedule some task forces even when many nodes are actually available. As the number of network nodes (and consequently the number of network users) increases, fixed assignment schemes become unwieldy and inefficient.

CONCLUSION

For microcomputer networks to be useful as general purpose computers, efficient, automatic, de-centralized task scheduling techniques must be devised. In the MICRONET net-

work computer, Wave Scheduling is used by the MICROS distributed operating system to schedule task forces. Wave Scheduling assigns network nodes to user tasks by using a hierarchical control schema as a foundation. Scheduling is probabilistic in the sense that the distributed schedulers are not guaranteed that their requests to schedule task forces will succeed. When a scheduling attempt fails, the descriptor for the intended task force either is passed up to the next higher level of control or undergoes another attempt at its local manager, depending on how many failures have occurred already. In large network computers, Wave Scheduling is very efficient and compares favorably with contract bid and diffusion scheduling.

REFERENCES

1. Flynn, M.J. 'Some Computer Organizations and Their Effectiveness', *IEEE Trans. on Computers*, C-21 (1972), 9, pp. 948-960.
2. Wittie, L.D. 'MICRONET: A Reconfigurable Network for Distributed Systems Research', *Simulation*, Nov. 1978, pp. 145-153.
3. Swan, R.J. et al. 'Cm*—A modular, multimicroprocessor', *AFIPS Proceedings of the National Computer Conference*, (Vol. 46), 1977, pp. 637-644.
4. Despain, A.M., and D.A. Patterson. 'X-Tree: A Tree Structured Multiprocessor Computer Architecture', *Proc. Fifth Ann. Symp. on Computer Arch.*, 1978, pp. 144-151.
5. Halstead, R.H., and S.A. Ward. 'The MuNet: A Scalable Decentralized Architecture for Parallel Computation', *Proc. Seventh Ann. Symp. on Comp. Arch.*, 1980, pp. 139-145.
6. Solomon, M.H., and R.A. Finkel. 'The ROSCOE Distributed Operating System', *Proc. Seventh Symp. on Op. Sys. Prin.*, 1979, pp. 108-114.
7. Siewiorek, D.P. 'Process Coordination in Multimicroprocessor Systems', In R.W. Hartenstein and R. Zaks (Eds.), *Microarchitecture of Computer Systems.*, Amsterdam: North-Holland, 1975, pp. 1-8.
8. Ousterhout, J.K. et al. 'Medusa: An Experiment in Distributed Operating System Structure', *Communications of the Association for Computing Machinery* 23, (1980), 2, pp. 92-104.
9. Swan, R.J. *The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: Cm**, Ph.D. Th., Carnegie-Mellon University, Aug. 1978.
10. Jones, A.K. et al. 'StarOS, a Multiprocessor Operating System for the Support of Task Forces', *Proc. Seventh Symp. on Op. Sys. Prin.*, 1979, pp. 117-127.
11. Smith, R.G. *A Framework for Problem Solving in a Distributed Processing Environment*, Ph.D. Th., Stanford University, 1979.
12. van Tilborg, A.M. and L.D. Wittie. 'Wave Scheduling: Distributed Allocation of Task Forces in Network Computers', *Proceedings of 2nd Int. Conf. on Dist. Comp. Sys.*, Paris (1981).
13. Wittie, L.D. and A.M. van Tilborg. 'MICROS, A Distributed Operating System for MICRONET, A Re-configurable Network Computer', *IEEE Trans. on Comp.*, C-29 (1980), 12, pp. 1153-44.
14. van Tilborg, A.M. and L.D. Wittie. 'High-Level Operating System Formation in Network Computers', *Proc. 1980 Int. Conf. on Parallel Proc.*, Aug. 1980, pp. 131-132.
15. van Tilborg, A.M. and L.D. Wittie. 'A Concurrent Pascal Operating System for a Network Computer', *Proc. IEEE CompSAC '80*, 1980, pp. 757-763.
16. Kleinrock, L. *Queueing Systems Volume I: Theory*, New York: Wiley-Interscience, 1975, pp. 94-99.
17. *Ibid.*, p. 17.
18. Schwartz, M. *Computer-Communication Network Design and Analysis*, Englewood Cliffs: Prentice-Hall, 1977.
19. Sullivan, H. et al. 'A Large Scale, Homogeneous Fully Distributed Parallel Machine, II', *Proc. Fourth Ann. Symp. on Comp. Arch.*, 1977, pp. 118-124.

The assignment of computational tasks among processors in a distributed system

by CAMILLE C. PRICE
Southern Methodist University
Dallas, Texas

ABSTRACT

The flexibility afforded by multiprocessor systems opens the question of how to assign computer program modules among functionally similar processors in a distributed computer network. In the model under consideration, the modules of a program are to be assigned among processors in such a way as to minimize interprocessor communication while taking advantage of affinities of certain modules to particular processors. The problem is formalized as a zero-one quadratic programming problem, and a solution is sought through an iterative technique that performs a series of transformations on an assignment matrix. Convergence to a locally optimum assignment is guaranteed, and an easily testable condition is given for which this local optimum is also a global optimum. An illustration of this algorithm is provided, results of performance experiments are reported, and suggestions are made for further study.

INTRODUCTION

A distributed computer network is considered to be a set of programmable processors interconnected to some extent by communication links.¹⁴ Recent technological advances, such as the economical fabrication of processors and the development of broadband communication facilities, have contributed to the feasibility of distributed computing systems; and the trend toward large shared database systems promises an increased popularity for the use of distributed networks. It is important that cost-effective methods be developed for these systems to control the allocation of computing resources among the jobs introduced into the network.

Scheduling theory deals with the general problem of allocating limited resources among multiple tasks when choices exist in the allocation process.⁷ The policies governing the apportionment of the resources are called scheduling rules or scheduling algorithms. Scheduling problems have demanded a great deal of attention since the development of digital computer systems, because scheduling algorithms are needed to assign a set of jobs to computer resources which are used in executing or servicing the jobs.⁶

The nature of job scheduling in a computer system depends on the functional similarity of the processing nodes and on the degree of communication available between processors. If the network consists of functionally different processors, then job scheduling is simple since each job would be designed for, and therefore assigned to, one particular specialized processor.

In a network of functionally similar processing nodes, it may be possible to assign the parts of a program freely among the processors; but in a practical sense, the communication links in a distributed network constitute inherent bottlenecks and therefore constrain the assignment of computational tasks. When high penalties are imposed for communication, the practical solution is to minimize the amount of communication between processors by assigning related tasks to the same processor. However, if the processors in the network were fully connected by high capacity data links, many feasible alternative assignments of computational tasks to processors would exist and should be evaluated by the job scheduler. In such cases, interprocessor communication would no longer be regarded as a serious constraint, but rather as a means of improving the overall efficiency of the system. The Cm* multimicroprocessor system²³ provides an example of precisely the kind of distributed system that will be considered in this paper. In the Cm* system, computational tasks, called *utilities*, may in general be executed by any microprocessor in the system.

The problem to be examined here is that of assigning computational tasks among processors in a distributed computer network having functionally similar nodes, but in which certain nodes have an advantage over others for particular jobs. The assignment is to be made in such a way as to take advantage of particular efficiencies of some processors for certain jobs while minimizing the costs of communication between jobs that are assigned to different processors.

In the next section, the problem is stated formally and formulated as a zero-one quadratic programming problem. The following section contains a description of an algorithm that can be applied to this scheduling problem. Conditions are given under which the local optimum achieved by this algorithm is also a global optimum. Results of performance experiments are reported. The final section contains a brief summary of work that has been done on this problem and gives suggestions for further study.

BASIC ASSUMPTIONS AND FORMULATION OF THE PROBLEM

The programs being executed within the distributed computer system are assumed to be partitioned into functional modules (containing executable code and/or data) which, in general, may reside on any processor in the system. There is no parallelism or multitasking of module execution within a program. Each processor may be multiprogrammed, and divide its time among several programs, but concurrent execution of the modules in one program is not considered. Thus the programs to be discussed here are serial programs, for which execution can shift from one processor to another.

Although the processors in the distributed system under discussion are functionally similar, they need not be identical. In fact, certain processors may have particular efficiencies for executing particular program modules. For example, some processors may have high speed arithmetic capabilities, access to a needed database, a large high-speed memory, access to certain peripheral devices, or other facilities associated with them that make them particularly well-suited for executing specific program modules.

The network is considered to be a fully-connected one, i.e., there is a direct communication link between every pair of processing nodes. It is also assumed that the communication paths between all processors are similar, that is, that the cost of sending a unit of data between any two processors is the same.

The modules of a modular program must be assigned among the processors in such a way as to minimize inter-processor communication while taking advantage of affinities of certain modules to particular processors. Therefore, there are two kinds of costs that must be considered in the search for a good assignment.

1. Each module has an execution cost that depends on the processor to which it is assigned. Let e_{ij} represent the cost of executing module i on processor j .
2. Any two modules that communicate during program execution incur a penalty if they are assigned to different processors. (It is assumed that the cost of such communication is zero when the reference is made between modules residing on the same processor.) Let the cost of communication between program modules i and k be denoted by c_{ik} .

An optimal assignment is one which minimizes the sum of the execution costs of the modules on the processors and the intermodule reference costs incurred when communicating modules reside on different processors.

It should be noted that the costs e_{ij} and c_{ik} must be measured in the same units of money or time. If these costs are measured in time, then the assignment minimizes the actual utilization of system resources.

Since the distributed program is to be executed in a serial fashion, and therefore all execution costs and communication costs are incurred in disjoint time intervals, the cost of the assignment is actually the minimal completion time of the program.

The problem can be formulated as a zero-one quadratic programming problem as follows:

minimize

$$\sum_{i=1}^m \sum_{j=1}^n e_{ij}x_{ij} + \sum_{i=1}^m \sum_{k=i+1}^m c_{ik} - \sum_{i=1}^m \sum_{j=1}^n \sum_{k=i+1}^m c_{ik}x_{ij}x_{kj}$$

subject to the constraints

$$x_{ij} = 0, 1 \quad \text{for all } i, j \quad (c1)$$

$$\sum_{j=1}^n x_{ij} = 1 \quad \text{for all } i \quad (c2)$$

where m is the number of program modules and n is the number of processors.

Zero-one polynomial programs can be converted to linear programs with nonlinear secondary constraints,¹² but this problem is approached here with techniques which take advantage of the special structure of the problem.

The problem has been solved for $n = 2$ by Stone.²⁹ A model is developed that can be interpreted as a commodity flow network, and an assignment is made by applying a maximum flow algorithm.¹¹ Efforts to extend this method to the general n -processor case have not been completely successful.³⁰

For n -processor problems in which the intermodule reference pattern is constrained to be a tree, an optimal assignment can be obtained by using a shortest path algorithm.^{5,10,31} The graph model developed for this restricted problem has been extended to allow an arbitrary module intercommunication pattern, but a modified shortest path algorithm that has been developed is guaranteed to yield an optimal assignment only when the graph model exhibits certain identifiable structural properties.²⁵

Assignment algorithms, such as the ones mentioned above and the one to be described in the following section, may be used to find a static assignment of modules to processors, but may also be applied repeatedly during the life of a program to reassign modules dynamically as the program's working set changes. (Models have been developed for special cases and an algorithm has been given to handle dynamic reassignment of modules.⁴)

THE ASSIGNMENT ALGORITHM

Solutions to scheduling problems, assignment problems, and transportation problems have frequently been sought through iterative techniques. Examples are the simplex method for linear programming problems,⁸ the "Hungarian" method^{2,16,17} for assignment problems, and the "modified distribution" method²⁴ for transportation problems. Such techniques begin with an initial solution which is then augmented at each step of the procedure until an optimal feasible solution is obtained.

An iterative procedure is defined here, for the multi-processor scheduling problem under consideration, that begins with an initial feasible assignment and repeatedly reassigns modules to processors until no further improvement is achievable by continuing the process. This reassignment of

modules is accomplished by performing a transformation on the assignment matrix X .

An assignment X is an $m \times n$ matrix such that

$$x_{ij} = 0, 1 \quad \text{for all } i, j \text{ and}$$

$$\sum_{j=1}^n x_{ij} = 1 \quad \text{for all } i.$$

The element $x_{ij} = 1$ if and only if module i is assigned to processor j . The set of all assignments X is called A . The cost of an assignment X is defined to be

$$c(X) = \sum_{i=1}^m \sum_{j=1}^n e_{ij}x_{ij} + \sum_{i=1}^m \sum_{k=i+1}^m c_{ik} - \sum_{i=1}^m \sum_{j=1}^n \sum_{k=i+1}^m c_{ik}x_{ij}x_{kj}$$

A transformation is described below that maps the set of all assignments into itself. The procedure determines whether reassignment is advisable and, if so, performs the most advantageous reassignment. The transformation $T: A \rightarrow A$ is defined as follows.

Transformation

1. For each element (i, j) in X , compute a "penalty" which is the cost of executing module i on processor j plus all communication costs for module i , given that all other modules (other than i) are assigned as indicated in matrix X . Thus the penalty matrix P is defined as

$$p_{ij} = e_{ij} + \sum_{k=1}^m c_{ik} (1 - x_{kj})$$

2. For each row i in X , compute the minimum penalty Θ_i as

$$\begin{aligned} \Theta_i &= \text{greatest possible improvement in cost achievable in one step on row } i; \text{ that is, by reassigning module } i \text{ and leaving all other modules unchanged} \\ &= \text{penalty for current assignment of module } i \text{ minus least penalty for any assignment of module } i \\ &= \sum_{j=1}^n p_{ij}x_{ij} - \min_{1 \leq j \leq n} \{p_{ij}\} \end{aligned}$$

and let t be the value of j giving this minimum. Note all $\Theta_i \geq 0$.

3. Select the row that permits the most profitable reassignment by finding

$$\max_{1 \leq i \leq m} \{\Theta_i\}$$

and let s be the value of i giving this maximum.

4. Change the assignment matrix X by setting

$$x_{st} = 1$$

and

$$x_{sj} = 0 \quad \text{for } j \neq t.$$

The transformation T is applied repeatedly until all $\Theta_i = 0$.

The transformation is illustrated by the following example. Let $m = 3$ and $n = 3$. The matrices E , C , and X represent

execution costs, communication costs, and the assignment, respectively, and are initially defined as

$$E = \begin{bmatrix} 4 & 2 & 5 \\ 1 & 8 & 8 \\ 4 & 6 & 3 \end{bmatrix} \quad C = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where X is obtained by assigning each module to the processor for which the execution cost is least (ignoring communication costs). The cost $c(X)$ is 12. In the first iteration, the penalty matrix P is computed as

$$P = \begin{bmatrix} 6 & 5 & 6 \\ 5 & 11 & 9 \\ 6 & 9 & 8 \end{bmatrix}$$

The theta values are

$$\Theta_1 = 5 - 5 = 0, \quad \Theta_2 = 5 - 5 = 0, \quad \Theta_3 = 8 - 6 = 2$$

and of these the maximum is Θ_3 . Therefore $s = 3$ and $t = 1$ and the third row of X is changed to $(1 \ 0 \ 0)$. The cost $c(X)$ now is 10. In the second iteration,

$$P = \begin{bmatrix} 4 & 5 & 8 \\ 2 & 11 & 12 \\ 6 & 9 & 8 \end{bmatrix}$$

The theta values are

$$\Theta_1 = 5 - 4 = 1, \quad \Theta_2 = 2 - 2 = 0, \quad \Theta_3 = 6 - 6 = 0$$

and $\Theta_1 = 1$ is selected as the maximum. Therefore $s = 1$ and $t = 1$ and the first row of X is changed to $(1 \ 0 \ 0)$. The cost $c(X)$ is now 9. In the third iteration,

$$P = \begin{bmatrix} 4 & 5 & 8 \\ 1 & 12 & 12 \\ 4 & 11 & 8 \end{bmatrix}$$

and all Θ_i are zero:

$$\Theta_1 = 4 - 4 = 0, \quad \Theta_2 = 1 - 1 = 0, \quad \Theta_3 = 4 - 4 = 0.$$

Therefore the procedure terminates and the assignment matrix X is

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

The cost $c(X) = 9$, which happens to be the optimum cost for this problem.

It is important to ascertain that the iterative procedure described above does not "cycle" indefinitely, thereby generating assignments that have been previously generated. The following theorem states that the iterative procedure does converge, after finitely many applications of the transformation T , to a local optimum.

Theorem. Let A be the set of all feasible assignments for a particular assignment problem. Then the transformation T has a fixed point, that is, $T(X) = X$ for some X in A .

Proof: There are only finitely many assignment matrices X in A (in fact, $K = N^M$ of them, where M is the number of modules and N is the number of processors). The transformation T is monotone with respect to cost, that is,

$$c(X) \geq c(T(X)) \text{ for all } X.$$

Therefore $c(X_i) \geq c(X_j)$ for $i < j$.

Let X_0 be the starting feasible solution. Then

$$\begin{aligned} X_1 &= T(X_0) \\ X_2 &= T(X_1) \\ &\vdots \\ &\vdots \\ X_k &= T(X_{k-1}) \text{ and } k \leq K-1. \end{aligned}$$

It is always true that $T(X_i) = X_j \notin \{X_l \mid l = 0, \dots, i-1\}$. Since the procedure continues only as long as an improvement can be made in one step, the transformation can be applied only finitely many times.

Optimal assignments are frequently obtained by assigning each module to the least expensive processor and using this as a starting feasible solution, but the following example provides a counter-example to guaranteed optimality.

Let $m = 4$ and $n = 2$. The matrices E , C , and X are initially as follows:

$$E = \begin{bmatrix} 6 & 10 \\ 8 & 6 \\ 8 & 6 \\ 8 & 10 \end{bmatrix} \quad C = \begin{bmatrix} 0 & 10 & 0 & 0 \\ 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

At the first iteration, $s = 4$ and $t = 2$, and row 4 of X becomes (0 1). At the second iteration, $s = 1$ and $t = 2$, and row 1 of X becomes (0 1). At the third iteration, all $\Theta_i = 0$, therefore the procedure terminates with the assignment

$$X = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

and $c(X) = 32$, whereas the optimal assignment is

$$\bar{X} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

for which $c(\bar{X}) = 30$.

The procedure terminates because no improvement is achievable in *one step*. In this example, the communication cost $c_{23} = 30$ is so high, relative to the difference in execution costs for modules 2 and 3, that no improvement is possible by temporarily assigning modules 2 and 3 to separate processors.

A slight modification may be made to the iterative procedure that would guarantee convergence to a global opti-

um, but at considerable computational expense. The modification consists of the following extensions. If $\Theta_i = 0$ for all i , then the procedure continues by considering all two step transformations, that is, all simultaneous reassignments of two modules. This requires that for each of the $\binom{M}{2}$ possible two-module reassignments, the penalty matrix P be computed. Values of Θ are then generated, the maximum is selected (unless all Θ are zero), and the appropriate two rows in the assignment matrix X are adjusted. If all Θ values are again zero, then no improvement is possible by reassigning only two modules. The procedure then considers all $\binom{M}{3}$

three-module reassignments, all $\binom{M}{4}$ four-module reassignments, and ultimately the simultaneous reassignment of all M modules. Clearly, generalization of the iterative procedure approaches an exhaustive search that requires a complete enumeration and evaluation of all reassignments in order to achieve a guaranteed global optimum.

Recall that, in the example shown just above, the difficulty arose because the communication costs were high relative to the differences in execution costs. By contrast, an optimal assignment can be achieved easily when communication costs satisfy the following condition

$$\sum_k c_{ik} \leq \min_{\text{all } j \neq i} (e_{ij} - e_{il}) \quad (c)$$

for all i . If condition (c) is satisfied then the optimal assignment is that which assigns each module to the processor with least execution time. Intuitively, under condition (c), communication costs are sufficiently small that they can be ignored and the assignment can be made solely on the basis of execution costs (i.e., no significant communication cost penalty is paid for distributing the program modules to the processors best suited for them).

It is worth noting that the standard form for the objective function to be minimized in a quadratic programming problem with continuous decision variables is

$$Q(x) = \sum_{j=1}^n b_j x_j + \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n d_{ij} x_i x_j$$

When this quadratic function has the property that its quadratic part is non-negative for all x_i and x_j , then $Q(x)$ is a convex function. And in cases where the objective function is convex, a local optimum is a global optimum. In the problem under consideration in this paper, condition (c) guarantees that the quadratic terms in the objective function are negligible and thus provides a discrete approximation to the convexity condition in the general problem. Thus, under condition (c), a local optimum produced by the iterative transformation algorithm is also a global optimum.

A FORTRAN language implementation of the iterative algorithm has been developed for performance testing. Random test data for experimentation were systematically generated for hypothetical networks of 5, 10, 15, and 20 processors, and 5, 10, 15, and 20 program modules. Nine different networks were generated for each problem size (m, n) .

Average computation times for the algorithm are reported in Table I. The computation time exhibited on these sample

TABLE I—Computation times

	$n = 5$	$n = 10$	$n = 15$	$n = 20$
$m = 5$.0107	.023	.034	.043
$m = 10$.077	.163	.283	.359
$m = 15$.260	.557	.883	1.203
$m = 20$.587	1.407	2.114	2.820

M = number of program modules.

N = number of processors.

Computation times in seconds.

test cases is $O(nm^3)$. As expected (based on the discussion just above), the algorithm's performance varied depending on the nature of the network. In networks having high execution costs and relatively low communication costs, the algorithm runs in approximately one-third the time required for networks having relatively high communication costs.

SUMMARY

The scheduling problem considered here has an efficient solution for $n = 2$. It has been shown that the problem is NP-hard for $n \geq 4$.³⁰ This property has not been established for the 3-processor case.

There are n^m possible assignments to be considered in this scheduling problem and, indeed, actual computational experiments using an enumerative algorithm require time that is $O(n^m)$. An optimal search procedure described in Price²⁶ requires $O(n^m)$ computation time in the worst case, but typically runs in polynomial time.^{15, 20, 22} The shortest path and non-backtracking branch-and-bound algorithms²⁵ and the iterative algorithm (see above) are of low polynomial complexity but generally produce suboptimal solutions.

A variety of techniques have been applied to scheduling problems. For this particular scheduling problem, there remain several interesting alternative approaches, which to date have been explored with only limited success, but which probably deserve further study.

Stone's two-processor network flow approach²⁷ might be extended by using the multiterminal cut techniques of Gomory and Hu.¹³ (The problem of processor load balancing in a two-processor system has also been studied²⁸ and it, too, may be extendable with multiterminal techniques.)

Spanning trees are of interest in various problems which can be studied through graph models.^{9, 18} It may be possible to devise a graph model of a modular computer program in such a way that a minimal spanning tree can be interpreted as an optimal assignment of modules to processors.

This scheduling problem is formulated above as a quadratic programming problem. Perhaps algorithms, such as that of Balas, can be tailored to solve particular problems very efficiently.^{1, 19, 32}

Clustering algorithms³³ might be applied to this problem to cluster program modules having high intercommunication costs together on the same processor.

It is clear that reasonable approaches to scheduling problems include (but are not limited to) techniques from the areas of mathematical programming, network analysis, and graph

theory.^{3, 9, 11, 21} Future practical developments will likely consist of heuristic methods and combinations of algorithms contributed from diverse fields.

ACKNOWLEDGMENTS

The first part of this paper is based on sections of a doctoral dissertation under the direction of Professor Udo W. Pooch at Texas A&M University. The author wishes to thank him, and Professors Don T. Phillips and A. P. Lucido, for their helpful suggestions and constructive review of the material.

REFERENCES

- Balas, E. An additive algorithm for solving linear programs with zero-one variables. *Oper. Res.* 13, 4 (July-August 1965), 517-546.
- Balinski, M. L., and Gomory, R. E. A primal method for the assignment and transportation problems. *Management Science* 10, 3 (April 1964), 578-593.
- Bellman, R. E. *Dynamic Programming*, Princeton U. Press, Princeton, N.J., 1957.
- Bokhari, S. H. Dual processor scheduling with dynamic reassignment. *IEEE Trans. Software Eng.* SE-5, 4 (July 1979), 341-349.
- Bokhari, S. H. Multiprocessor scheduling with shortest path algorithms. Tech. Rep. ECE-CS-77-11, Dept. of Elec. and Computer Eng., University of Massachusetts, Amherst, Dec., 1977.
- Coffman, E. G., Jr. et al. *Computer and Job-Shop Scheduling Theory*, John Wiley and Sons, New York, 1976.
- Conway, R. W., Maxwell, W. L., and Miller, L. W. *Theory of Scheduling*, Addison-Wesley, Reading, Mass., 1967.
- Dantzig, G. B. *Linear Programming and Extensions*, Princeton U. Press, Princeton, N.J., 1963.
- Deo, N. *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, N.J., 1974.
- Dijkstra, E. W. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269-271.
- Ford, L. R., Jr., and Fulkerson, D. R. *Flows in Networks*, Princeton U. Press, Princeton, N.J., 1962.
- Glover, F., and Woolsey, E. Converting a 0-1 polynomial programming problem to a 0-1 linear program. *Oper. Res.* 22, 1 (Jan.-Feb. 1974), 180-182.
- Gomory, R. E., and Hu, T. C. Multiterminal network flows. *J. SIAM* 9, (Dec. 1961), 551-570.
- Greene, W. H., and Pooch, U. W. A review of classification schemes for computer communication networks. *Computer* 10, 11 (Nov. 1977), 12-21.
- Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. of Systems Science and Cybernetics*, SSC-4, (July 1968), 100-107.
- Hu, T. C. *Integer Programming and Network Flows*. Addison-Wesley, Reading, Mass., 1969.
- Klein, M. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science* 14, 3 (Nov. 1967), 205-220.
- Kruskal, J. B. Jr. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Am. Math. Soc.* 7, (1956), 48-50.
- Lawler, E. L. The quadratic assignment problem. *Management Science* 9, 4 (July 1963), 586-599.
- Martelli, A., and Montanari, U. Optimizing decision trees through heuristically guided search. *Comm. ACM* 21, 12 (Dec. 1978), 1025-1039.
- McMillan, C. *Mathematical Programming*, Wiley, New York, 1970.
- Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
- Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. Medusa: an experiment in distributed operating system structure. *Comm. ACM* 23, 2 (February 1980), 92-105.
- Phillips, D. T., Ravindran, A., and Solberg, J. J. *Operations Research: Principles and Practice*, John Wiley and Sons, New York, 1976.

25. Price, C. C. A Nonlinear Multiprocessor Scheduling Problem. Ph.D. Th., Texas A&M University, College Station, Texas, May 1979.
26. Price, C. C. Scheduling algorithms for a distributed computer system. University of Texas at Dallas Technical Report No. 65, September, 1979.
27. Rao, G. S., Stone, H. S., and Hu, T. C. Assignment of tasks in a distributed processor system with limited memory. *IEEE Trans. Computers C-28*, 4 (April 1979), 291-299.
28. Stone, H. S. Critical load factors in two-processor distributed systems. *IEEE Trans. Software Eng. SE-4*, 3 (May 1978), 254-258.
29. Stone, H. S. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Eng. SE-3*, 1 (Jan. 1977), 85-93.
30. Stone, H. S. Private communication, Jan. 1979.
31. Stone, H. S., and Bokhari, S. H. Control of distributed processes. *Computer II*, 7 (July 1978), 97-106.
32. Taha, H. A. A Balasian-based algorithm for zero-one polynomial programming. *Management Science* 18, 6 (Feb. 1972), B328-B343.
33. Zahn, C. T. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comp. C-20*, (Jan. 1971), 68-86.

Software reliability in real-time systems*

by BHARAT BHARGAVA

University of Pittsburgh
Pittsburgh, Pennsylvania

ABSTRACT

This paper investigates techniques to enhance the continuity of operations of the enroute air traffic control system. First the issues of software reliability and fault tolerance in real-time systems are discussed. Next, a list of problems associated with nonstop operations of flight dataprocessing (FDP) subsystem of the enroute air traffic control system are assumed, based on limited knowledge of the system; and possible solutions are suggested and analyzed. Implementation issues of recovery block scheme such as architecture, design of alternates and acceptance tests, and cost vs. reliability are studied. Four architectures of recovery block scheme are analyzed, and results of a simulation study using flight data processing subsystem as a test case are discussed.

INTRODUCTION

Most large software systems are error-prone. It is expected that the many efforts to improve software quality and reliability will reduce failures but it is hard to say if they will completely eliminate them. Sometimes it is believed that maturity can provide freedom from software errors but that is not borne out by the experience of extensively used operating systems. In the Bell Laboratories' Electronic Switching systems (which employ hardware redundancy and thoroughly tested software) software faults accounted for approximately 20% of all failures. There are many types of errors that manifest themselves during some unusual data or machine state and lead to a system failure. Some of these errors are: computational (divide by zero), logical, definitional (array not subscripted properly), operational (wrong transaction entered and accepted), etc. For a complete list see the appendix.

Large software systems are also under constant modification for improving efficiency and this leads to additional faults. In many applications such as computerized air-line reservation system, isolated small breakdowns can be tolerated as long as the overall system remains operational. But in

transportation applications such as air-borne computer, air traffic control systems, mass transit systems, only momentary cessation of service can be tolerated, no maintenance or manual repair activity is feasible, and incorrect results are unacceptable. In addition, the enroute air traffic control system collects and processes extensive amounts of valuable data and safeguarding the data is more important than providing continuity of access with possible damage to such data. The architectures of nontrivial computer systems involve careful considerations of tradeoffs between reliability, performance, and costs.

The need for reliability of operations in large automated real-time system is becoming increasingly important, particularly in transportation applications and nuclear industry. For such systems, it is important to have high confidence that the system will behave as expected for all possible environments. Of course, the development methodology has a great impact on the quality of software as well as the effort required for a thorough validation. An example of utilizing of a proper methodology in nuclear power system is discussed in Ramamoorthy et al.⁸. Though it is nice to start from scratch and develop reliable software with present technology, many systems currently operational in daily use cannot afford to wait for a new software system and architecture. And what if the design and development methodology is further advanced during the time the system is rebuilt? This "catch-22" situation requires us to deal with the present software and its problems in such a way that continuity of operations is allowed in the current system and the cost of providing reity is reduced as parts of the software systems are renovated. So software structures must be investigated that provide fault tolerance in addition to fault avoidance. A survey of fault tolerant techniques is available¹⁰.

FAULT-TOLERANCE (ERROR DETECTION, DIAGNOSIS, RECOVERY, AND BYPASS)

The various steps of fault tolerance are as illustrated in Figure 1. A detailed discussion of these steps is available¹⁰.

The purpose of error detection is to prevent or to recognize system failures. This can usually be achieved by designing proper checks on every critical step. Reading the data, exe-

* This work was performed under contract DOT-RC-92031 with the U.S. Department of Transportation.

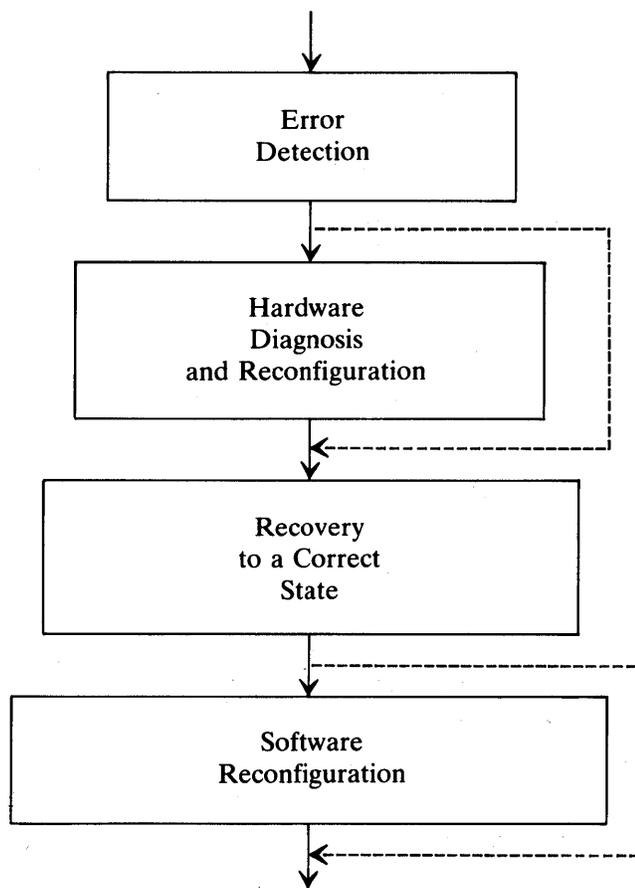


Figure 1—Steps of fault-tolerant processing

ing a loop or a decision branch, conversing with another program, writing the data, are examples of such critical functions. There is usually a high cost associated with such checks, and vigorous checking is usually avoided for normal operations. Only when a failure has already occurred, the complexity of checks must be increased. Some examples of checks used in software are as follows:

1. Dual or triple modular redundancy
2. Reversal check
3. Inference check
4. Watchdog time
5. Address-in-bound check
6. Acceptance check
7. Path testing
8. Branch testing
9. Structure testing
10. Special value testing
11. Symbolic testing

A detected error is only a symptom of the fault that caused it and does not necessarily identify that fault. Usually there is many-to-many mapping between errors and possible reasons.

In real-time systems, on-line diagnosis is not usually possible. It is important to observe that off-line diagnosis can be made easier if mechanisms based on diagnosis techniques are

included in the software architecture to collect necessary information before and during the time the fault occurs and thus create a list of suspicious program modules, data, and messages. This information can be used off-line for thorough investigation of errors and reasons behind them and at the same time can drive the reconfiguration and bypass algorithms.

The variety of undetected errors which can exist in a design of nontrivial software component is essentially infinite. Due to the complexity of the component, the relationship between any such error and its effect at run-time may be very obscure. For these reasons, diagnosis of the original cause of software errors should be left to the humans

A different strategy will be to ignore the fault and try to continue to provide service despite its continued presence. Given that a component has been designated as faulty, whose further use should be avoided, one can use either its replacement (by a standby spare) or reconfigure so that its responsibilities are taken over by other components available in the system. Reconfiguration necessarily involves some degree of performance and/or function degradation.

Once the system goes into an erroneous state its resources (program states, databases) should be brought to a correct state before further processing can be continued. One can use either forward error recovery in which an attempt is made to correct the error states. Compensation is a prime example of such mechanism. Backward error recovery depends on the provision of recovery points, i.e., a means by which the state of processes can be recorded and later reinstated. This is a popular mechanism used by many practitioners and its merit is due to two facts:

1. The questions of damage assessment and repair are treated quite separately from those of how to continue further service.
2. Damage assessment can be made independent of the type of fault.

The recovery block scheme⁹, checkpoints with audit trail¹¹, and complete database dump¹¹, are examples of backward error recovery. The implementation issues of some of these fault-tolerant techniques in the enroute air traffic control center software are the topic of discussion in the following sections.

PROBLEMS OF CONTINUITY OF OPERATIONS IN FLIGHT DATA PROCESSING SUBSYSTEM

The enroute air traffic control center (ARTCC) software consists of programs, databases, and messages (or transactions). We have tried to identify the problems encountered in continuing the operations of the flight data processing (FDP) subsystem of the enroute system by studying the reports^{12,13} and discussing experiences with the staff of Cleveland Enroute Center. We feel that there are four problem areas.

Program Errors

The programs of the FDP system have been designed and coded by a large number of programmers over a period of

time. Though they have reached a mature stage, possibility of some hidden errors still exists. These errors cannot be easily identified and the complete debugging cannot be ensured. There are some problems regarding the maintenance of programs. Only an object version of the program module exists at an enroute center. In case the ARTCC staff identifies an error, the object program is patched to fix the bug. The error is documented and sent back to FAA technical center in Atlantic City for correcting the source program. The programmers in the Federal Aviation Agency (FAA) technical center determine the necessary JOVIAL statements to patch the source. They compile the new source version and send the new object version to the enroute center. Sometimes the new object version is not quite equivalent to the object version that was patched earlier. Moreover, any side effects of the compiler are not quite known to the enroute center staff. The problem arises due to the non-availability of a compiler and source code at each enroute center but the reasons are understandable. Careful patching of bugs with coordinated testing at FAA technical center and enroute center has eliminated some such problems.

Database Inconsistency

There are four types of databases used in the FDP sub-system: static (airspace, airways definitions, bulk flight plan, etc.), dynamic (daily flight plans), real-time generated (changing tracks), Compool tables (interprogram communications). There are two distinct reliability issues regarding databases: they should be protected against a possible loss during a failure and they should be consistent during normal processing. Since the flight plan database is under constant update traffic, it must be thoroughly checked at the time of retrieval (before actual use) and update (before any changes are made).

Unexpected Inputs

Sometimes an incorrect or illegal input (or message) is entered in the system for processing. This sometimes causes an abort to be initiated. Since the checking and abort decisions are not always made at the source of the input, it leads to repeated entry of such input, causing repeated aborts and system failure.

This problem is further complicated by the fact that there are a variety of input types and their sources. Sometimes the erroneous error is passed from another program (where it might be a legal output).

Synchronization of Multiple Concurrent Processes

In the multiprocessing/multiprogramming environment processing sequence errors are a possibility. For example, flight data is supposed to be brought in core for processing. Sometimes processing of data can start but all necessary data

is not yet available. Since locking/unlocking and interprogram communication is done by user programs, the interleaved operations on data by different processes are error prone. There are three dimensions to such interaction: data objects, processes, and time. A complex case arises when m objects are being processed by n processes and interaction is to be governed by a time order. For example, Object i should not be referenced by Process x until Objects i and k have been referenced by Process y . The possibility of failure of one or more processes makes this model all the more complex.

Because of space limitations in this paper, I limit discussion to solutions dealing with program errors, unexpected inputs and database inconsistency. Details of these and possible techniques for ensuring correctness of synchronization of multiple concurrent processes are available⁴.

AVOIDANCE OF FAILURE DUE TO PROGRAM ERRORS

There are various mechanisms to protect against program errors.

First of all, proper specification, design, and testing should lead to improved reliability and availability. An example of this research technique for developing software of nuclear power plant is available.⁸ Secondly, we can study the error diagnosis, and correction mechanisms, but they are quite inefficient for real-time processing. But with the development of concepts of recovery lines⁶⁻¹⁰, two-phase commit⁵⁻⁶, and atomic actions⁶⁻¹⁰, system structures which are more fault-tolerant will be possible. A possibility exists that the system may keep runtime graphs describing the processes, operations, and interleaving interactions. This can provide some suspicion lists which can diagnose and correct errors at runtime. Investigation of these techniques is the goal of our future research.

The third mechanism provides error bypass and software reconfiguration and is most suitable for continuing operations in real-time systems. Recovery block scheme⁹ proposed by Randell is an example of this mechanism. The simplest structure of the recovery block is

```

Ensure AT
  by P
  Else by A1
  -----
  -----
  Else by An
Else Error

```

Where AT is the acceptance test condition that is expected to be met by successful execution of either the primary program module P or by the alternate module A_1 ... or A_n . The internal control structure of the recovery block will transfer to the next alternates if the test conditions are not met by the previous primary or alternate. A hierarchy of acceptance tests based on their complexity can also be used. Moreover different tests for different alternates can be used. The acceptance test can also be augmented by a watchdog timer that monitors that an acceptable result is furnished within a specified period. The timer can be implemented in either hardware or software.

COST/RELIABILITY ANALYSIS OF RECOVERY BLOCK SCHEME

Even though the idea of recovery block scheme is bright, its implementation poses some challenges. Some of these are listed below and have been topics of our research.

1. Analysis of recovery block scheme to identify the selection criterion of alternates and acceptance tests.
2. Selective implementation of recovery block scheme to maximize reliability/cost ratio.
3. Application of recovery block scheme to the software structure and the processing of a real application.
4. Design of proper alternates and acceptance tests for a given primary module.

We have studied the first two questions². The problem is formulated as follows:

1. Given a software structure with cost of processing (in terms of execution time), failure probability, and processing requirements, and
2. Given several alternatives for implementing recovery block scheme (choice of alternate and acceptance test characteristics, granularity of testing etc.),

how to decide the best architectural implementation which has:

1. A low overhead during normal processing, and
2. A high potential for reconfiguration (in case of failure) at a low cost.

The evaluation standard to select implementational choices was as follows:

1. Reliability should achieve some minimum.
2. Absolute processing time should not exceed some maximum.
3. Cost/reliability index (CRI) should be as low as possible.

The first two evaluation standards are dictated by the performance criterion of the ARTCC software. The third reflects the marginal reliability gain and extra processing cost required to obtain it. As a general rule, reliability is increased by early detection of error (or increased granularity of testing) and adding a number of alternates, but the cost of testing, recovery, and running is also simultaneously increased. Hence cost-reliability index (CRI) defined as the ratio of cost of executing the module and its reliability is a suitable evaluation criterion. If CRI' and CRI are the cost-reliability indices of the new and original software architecture, then the new design is considered better if $\frac{CRI'}{CRI} < 1$.

We make the following assumption for the cost/reliability analysis of software architectures utilizing recovery blocks:

1. Acceptance tests are perfect. This assumption is made to eliminate complexity of analysis. Since this assumption will be made in all types of architectures, for comparison purposes, it will factor out. Since perfect tests with a low cost are seldom available, we will relax this assumption in our simulation study.
2. Probability of failure of primary and alternates are independent. This assumption is made because it is very difficult to obtain dependencies between the failures of primary and alternates. Of course this assumption can generally be true for independently designed modules.
3. The cost of recovering states after a failure is small compared to the cost of executing the module.

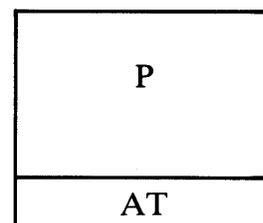
Usually the state recovery cost involves popping stacks and resetting variables. Moreover, use of the recovery cache⁷ makes the state recovery possible at low cost.

We have studied the cost, reliability, and CRI of four types of software structures:

- I. Primary module P with an acceptance test AT .
The probability of failure of P is P_p , and the cost of executing and testing P by AT is C_p .
- II. Primary module P decomposed into m submodules P_1, P_2, \dots, P_m . Each submodule has an acceptance test.
The probability of failure of each submodule is P_{p_i} and the cost of executing and testing P_i is C_{p_i} (for $i = 1, \dots, m$).
- III. Primary module P supported by $n - 1$ alternates A_1, A_2, \dots, A_{n-1} .
The probability of failure of each alternate is P_{A_i} , and the cost of executing and testing the alternate is C_{A_i} (for $i = 1, \dots, n - 1$).
- IV. Primary module P is decomposed into m submodules, P_1, P_2, \dots, P_m . Each submodule has an alternate and an acceptance test. A primary submodule and its alternate form a block M_i (for $i = 1, \dots, m$).
The probability of failure of components of each block M_i are defined as in Type II and III.

The reliability and cost equations for each type of software structure are given as follows. More details on arriving at these equations are available².

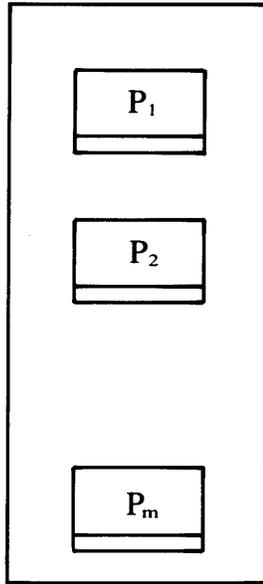
Type I



$$R_I = 1 - P_P \quad (P_P = \text{probability of failure})$$

$$C_I = C_P \quad (C_P = \text{cost of executing and testing } P)$$

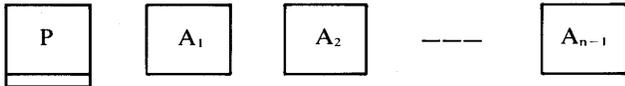
Type II



$$R_{II} = (1 - P_{P_1}) (1 - P_{P_2}) \dots (1 - P_{P_m})$$

$$C_{II} = C_{P_1} + (1 - P_{P_1}) C_{P_2} + \dots + (1 - P_{P_1}) \star (1 - P_{P_2}) \dots \star (1 - P_{P_{m-1}}) \star C_{P_m}$$

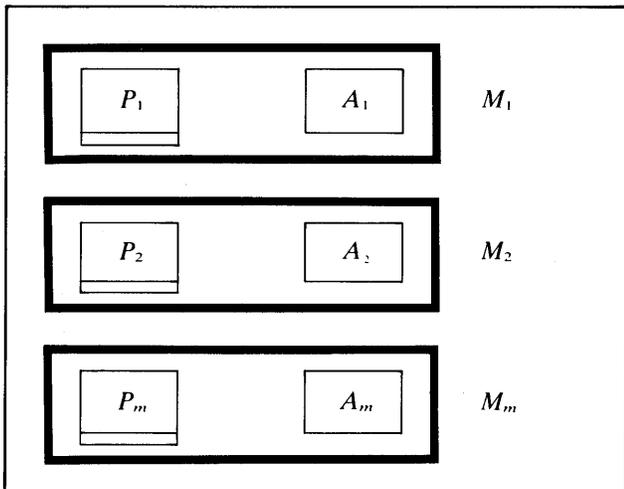
Type III



$$R_{III} = 1 - P_P P_{A_1} P_{A_2} \dots P_{A_{n-1}}$$

$$C_{III} = C_P + P_P C_{A_1} + \dots + P_P P_{A_1} \dots P_{A_{n-2}} C_{A_{n-1}}$$

Type IV



$$R_{Mi} = 1 - P_{P_i} P_{A_i}$$

$$C_{Mi} = C_{P_i} + P_{P_i} C_{A_i}$$

$$R_{IV} = R_{M_1} \cdot R_{M_2} \dots R_{M_m}$$

$$C_{IV} = C_{M_1} + R_{M_1} C_{M_2} + \dots + R_{M_1} \star R_{M_2} \dots \star R_{M_{m-1}} \star C_{M_m}$$

Let us now see under what conditions application of a recovery block scheme will improve the CRI of the software architectures:

Case 1 (Type III vs. Type I)

Selection criterion for employing alternates:

It is obvious that using more alternates will increase reliability as well as the execution cost. So the CRI is the most appropriate evaluation index.

Let

$$\frac{P_{A_i}}{P_P} = X \quad X \leq 1$$

$$\frac{C_{A_i}}{C_P} = Y \quad Y \geq 1$$

then

$$\frac{CRI_{III}}{CRI_I} < 1$$

if

$$(1 - P_P) Y + P_P X < 1$$

This equation is plotted in the following Figure 2.

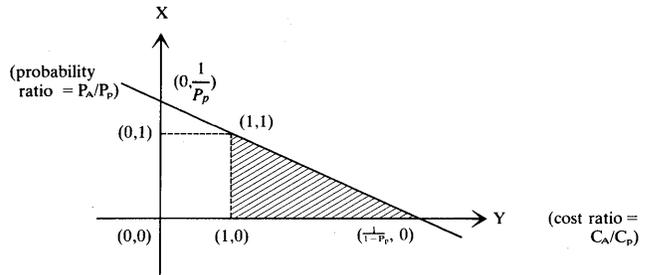


Figure 2—Domains for better cost/reliability index

Note that $CRI_I = CRI_{II}$ when $x = 1$ and $y = 1$; i.e., we do not gain anything on CRI if we put an alternate module with the equivalent power of the primary as the alternate.

We will gain recovery power ($CRI_{III} < CRI_I$) if (X, Y) is in the triangle formed by $(0, 1/P_P)$, $(0,0)$, and $(1/1 - P_P, 0)$. Usually, an alternate will be more reliable ($X < 1$) and cost more ($Y > 1$). This represents the operating range in the shaded area. Moreover, contrary to our intuition, a recovery block may have smaller CRI if (Y, X) is in the triangle $(0, 1/P_P)$, $(0,1)$, $(1,1)$. It implies that a less reliable alternate ($X > 1$) may be used for improvements of CRI.

Case 2 (Type I vs. Type II)

Selection criterion for granularity of testing:

Let

$$P_{P_1} = P_{P_2} = \dots = P_{P_m} = \bar{P}_P$$

$$C_{P_1} = C_{P_2} = \dots = C_{P_m} = \bar{C}_P$$

Note:

$$P_P = (1 - \bar{P}_P)^m$$

$$C_P \geq \bar{C}_P * m$$

$$\frac{CRI_{II}}{CRI_I} = \frac{\bar{C}_P * (1 - P_P)^2}{C_P * \bar{P}_P * P_P}$$

Let

$$C_P = m\bar{C}_P$$

Then

$$\frac{CRI_{II}}{CRI_I} < 1$$

If

$$\frac{(1 - P_P)^2}{m * \bar{P}_P * P_P} < 1$$

Obviously the CRI will decrease as granularity of testing (m) increases.

Case 3 (Type IV vs. Type II)

The selection criterion for employing alternates if the testing granularity can be increased:

Let

$$P_{P_1} = P_{P_2} = \dots = P_{P_m} = \bar{P}_P$$

$$C_{P_1} = C_{P_2} = \dots = C_{P_m} = \bar{C}_P$$

$$\frac{P_{A_i}}{P_{P_i}} = X \quad X \leq 1$$

and

$$\frac{C_{A_i}}{C_{P_i}} = Y \quad Y \geq 1$$

Then

$$\frac{CRI_{IV}}{CRI_{II}} < 1$$

$$\text{if } \frac{(1 + Y\bar{P}_P) [1 - (1 - X\bar{P}_P^2)^m] (1 - \bar{P}_P)^m}{X\bar{P}_P [1 - (1 - \bar{P}_P)^m] (1 - X\bar{P}_P^2)^m} < 1$$

This condition is a function of three variables: X , Y , and m . If $X = Y = 1$ (alternates have the same probability of failure and cost as the primary), then

$$\frac{CRI_{IV}}{CRI_{II}} < 1$$

if

$$\frac{(1 + \bar{P}_P) [1 - (1 - \bar{P}_P^2)^m] (1 - \bar{P}_P)^m}{\bar{P}_P [1 - (1 - \bar{P}_P)^m] * (1 - \bar{P}_P^2)^m} < 1$$

or if

$$\frac{(1 + \bar{P}_P) [1 - (1 - \bar{P}_P^2)^m]}{\bar{P}_P [(1 + \bar{P}_P)^m - (1 - \bar{P}_P^2)^m]} < 1$$

The left-hand side of the equation becomes smaller as m increases, or for a given m , \bar{P}_P approaches zero. Thus the decision must be based on meeting the maximum allowable execution cost. Obviously the reliability of Type IV architecture is always greater than that of Type II but the execution cost of Type IV architecture increases linearly with m .

Case 4 (Type III vs. Type IV)

Selection criterion (more alternates vs. more granularity): This is the most interesting case for implementation of a recovery block scheme.

We know that both adding an alternate or increasing granularity will increase reliability as well as cost. It is more interesting to see how fast CRI grows as the number (n) of alternates increases in Type III architecture and as the granularity (m) of testing increases in Type IV architecture.

Let

$$P_{P_1} = P_{P_2} = \dots = P_{P_m} = \bar{P}_P$$

$$C_{P_1} = C_{P_2} = \dots = C_{P_m} = \bar{C}_P$$

For

$$1 < j \leq m$$

$$\frac{P_{A_j}}{P_{P_j}} = X \quad X \leq 1$$

$$\frac{C_{A_j}}{C_{P_j}} = Y \quad Y \geq 1$$

For

$$1 \leq i \leq n - 2$$

$$\frac{P_{A_1}}{P_P} = \frac{P_{A_{i+1}}}{P_{A_i}} = X$$

and

$$\frac{C_{A_i}}{C_P} = \frac{C_{A_{i+1}}}{C_{A_i}} = Y$$

Let us further analyze cost, reliability, and CRI of Type III and Type IV architecture. Let cost, reliability, and CRI be increments with n as a variable for Type III and with m as a variable for Type IV. Δ CRI is the increment in the cost-reliability index when n or m is increased. For example, $\Delta CRI \Big|_n^{n+1}$ is the difference between CRI with $n + 1$ alternates and CRI with n alternates.

For Type III module, increasing the number of alternates from n to $n + 1$, we have the following increments:

$$\Delta \text{cost (increase in cost)} = P_P^n X^{n(n-1)/2} Y^n C_P$$

$$\Delta \text{reliability (increase in reliability)} = (1 - P_P X^n) P_P^n X^{n(n-1)/2}$$

$$\Delta CRI \Big|_n^{n+1} = \frac{C_P Y^n}{1 - P_P X^n}$$

Δ CRI is a polynomial that will grow very fast as n increases.

For Type IV module, increasing the granularity (which may not always be possible) from m to $m + 1$, we have the following increments:

Let

$$g_m = 1 - (P_P)^{1/m}$$

$$\Delta \text{reliability} = (1 - X g_{m+1}^2)^{m+1} - (1 - X g_m^2)^m$$

$$\Delta\text{cost} = \frac{C_P}{m+1} * \frac{(1 + Yg_{m+1})[1 - (1 - Xg_{m+1}^2)^{m+1}]}{Xg_{m+1}^2}$$

$$- \frac{C_P}{m+1} * \frac{(1 + Yg_m)[1 - (1 - Xg_m^2)^m]}{Xg_m^2}$$

$$\Delta\text{CRI} \Big|_m^{m+1} = \frac{\Delta\text{cost}}{\Delta\text{reliability}}$$

ΔCRI will grow very slowly as m increases.

From the above analysis, we draw the following conclusions.

1. Increase number of alternates till the minimum required reliability is achieved.
2. Increase granularity of testing till maximum cost has been reached.
3. In case additional reliability is required, after a certain value of m and n (depending on X , Y , P_p , and C_p), it will be preferable to increase m .

Relaxing the assumption of imperfect acceptance tests introduces four new parameters for our cost analysis:

AT accepts condition when module is correct:

Probability u ,

AT rejects condition when module is correct:

Probability v ,

AT accepts condition when module is incorrect:

Probability s ,

AT rejects condition when module is incorrect:

Probability t .

We are currently working on generalizing our analysis and this will be the subject of a future paper.

The fourth issue regarding implementation of recovery block scheme is to develop a methodology to systematically design proper alternates and acceptance test for a given primary module. We outline our present understanding briefly in the following paragraphs.

One can design alternates which would

1. Execute in a subset of domain of primary, but the failure probability of primary and alternate are independent of each other (alternate more specialized than primary).
2. Execute in a domain which does not overlap with the domain of primary.
3. Execute in a domain which covers the domain of primary and more (alternate more general than primary). The three cases are shown in Figure 3.

In addition, one would like to design acceptance tests which cost 10 to 15% of the execution of a primary but cover precisely domain of primary (or union of primary and alternates).

We give three strategies for designing an alternate. One strategy can be to utilize the most recently developed module as the primary and utilize the one which has been in use and time-tested as the first alternate. Obviously the primary module will be very efficient and its components optimized but its reliability is questionable.

Another strategy could be to have the primary and alternate modules utilize different methods to achieve the same (or similar function). For example, the primary module may use

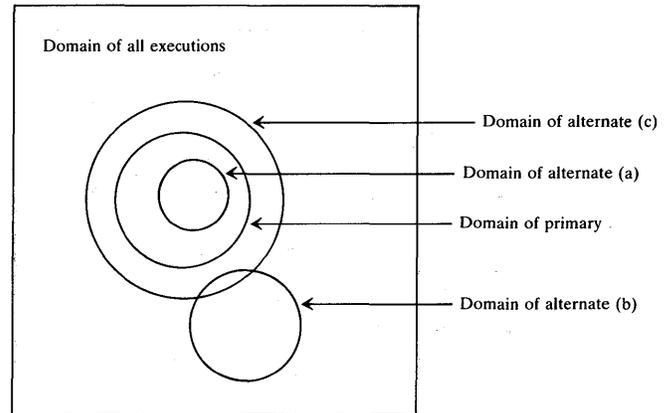


Figure 3—Possible domains of primary and alternate program modules

the complex but efficient quick sort; the alternates may be bubble sort, merge sort, or tree sort.

A third approach could be to allow limited capabilities in the alternates and provide extensive facilities in the primary. Just for the sake of making up a simple example, we can say the primary module can sort negative and positive numbers and also manage duplicates while the alternates can handle only positive numbers, or the primary can handle numerical as well as erroneous (nonnumerical) data while the alternate can work on a limited range of numerical data.

As regards design of proper acceptance tests, we agree that the test has to be simpler than the primary block. Moreover the testing procedure will be simple and extremely reliable. But then how well can the acceptance test check the results? For example, if we use check-sum to be the acceptance test for a sort module, we can be sure that output contains all data that came as input, but we cannot be sure if the numbers are properly sorted. If we use the acceptance test to check if the last number is greater than the middle which is greater than the first, we can be sure that the output is possibly in ascending order but we cannot be sure whether the output contains all numbers which have been properly sorted. If we go to the extreme and check all numbers, then the acceptance test will become as complex as the primary module and will become very costly. A different approach to the design of acceptance tests could be based on the concept of time out, which is used in the enroute system hardware. For example, we could obtain the worst bounds on the algorithms of the primary module (e.g., quick sort takes $2n \log_2 n$ comparisons, or binary search requires $\log_2 n$ comparisons) and translate them in the maximum time required to complete the execution of the module. If there is a fault in the module, the time out mechanism can be used to switch to the alternates.

Another interesting question to study is whether the acceptance test of the whole block should be unique or we should use different hierarchy based on graceful degradation of system's performance and use one level for primary and another level for the alternate module, etc. We also believe that the acceptance tests have to be adaptive and can change as we gain more and more confidence with the system.

While thinking of the feasibility of employing the recovery block architecture to enroute system, we observe that some of the subsystems are computation oriented (e.g. deal with nu-

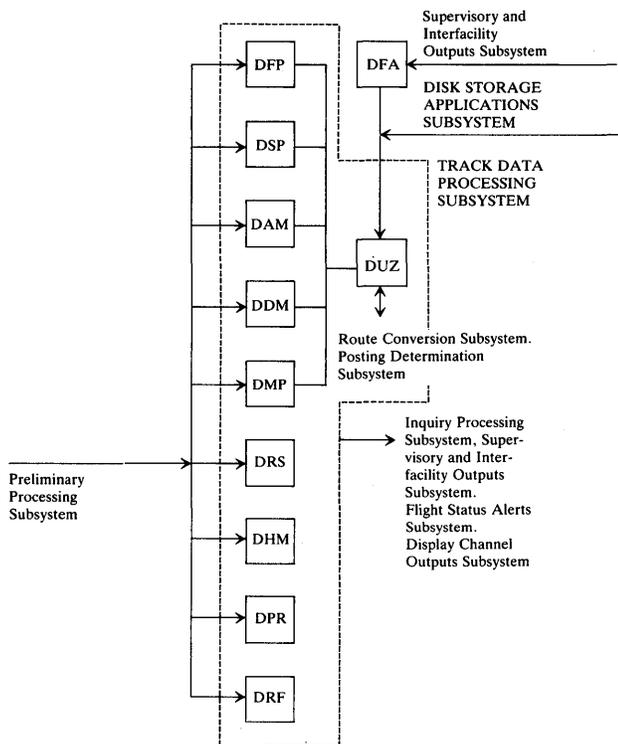


Figure 4—FDP subsystem subprograms flow

merical analysis issues such as correlation, convergence, etc.) while others are database update and transaction oriented. The application of acceptance tests as discussed earlier could be inexpensive for the computation oriented subsystem, but for the heavy traffic database system which is under constant modifications and heavy references, integrity monitoring could be expensive. So the interesting question arises: Is it simpler and cheaper to check the input stream or the program flow? In a report³ I have answered this question partially by concluding compile-time validation is better than run-time or post-execution validation.

APPLICATION OF RECOVERY BLOCK SCHEME ON A SIMULATED FLIGHT DATA PROCESSING SUBSYSTEM

Even though the analytical approach allows us in identifying key variables such as cost of alternate, and acceptance test, probability of failure etc., it does make substantial assumptions about the structure of the system to keep analysis simple. In order to study the impact of recovery block scheme on the software structure of ARTCC, one must relax the following two assumptions:

1. All modules have identical probability of failure and execution cost.
2. Acceptance tests are perfect.

To do this, we adopted a simulation approach in which the above parameters were made variables. We selected the flight

data message processing (FDP) subsystem of ARTCC for simulation study. The FDP consists of 11 re-entrant subprograms which call on a pool of 23 subroutines. The memory limitations available on DEC-10 forced us to consider a part of FDP and the simulated FDP only contained five subprograms: FDP, DAM, DSP, DFA, and DUZ. A detailed description is available¹³. Subsequently only 15 subroutines called by these subprograms were used. It is important to note that only the processing requirements of the subprogram and subroutines were simulated, i.e., subprograms and subroutines were considered black boxes which were triggered by randomly generated messages due to the flight plan generator. Thus no attempt was made to code the actual functions of the subprograms or subroutines. The simulation study undertaken here can be easily used for studying any subsystem of the ARTCC software. Figure 4 shows the flight data message processing subprograms and subroutines.

Details on execution time and size of each subprogram and subroutine are available¹.

Two modes of FDP processing were simulated:

1. FDP processing with acceptance test at each subroutine (Type II module).
2. FDP processing with an acceptance test at each subroutine. In addition an alternate was provided for each subroutine. The probability of failure of primary and alternate were independent (Type IV module).

In the first mode, the following actions are performed.

1. A message is generated randomly by the flight plan generator.
2. Based on FDP processing requirements, appropriate messages are generated to initiate the processing of subprograms. (For example, one message from flight plan generator produces 2.2 amendment messages). Subgrams in turn call subroutines.
3. The processing of subroutine fails randomly.
4. If a subprogram call finds the subroutine in failed state, all partial processing completed by subprogram for the current message is lost and subprogram starts executing from the beginning (as if the current message was reissued). Note that all previous subroutine calls are also repeated. One could also abort the system at this point. In our simulation, the abort simply means reexecution.
5. Action 4 is repeated until the current message execution is completed.

In the second mode of FDP processing, the following actions are performed:

1. Same as in mode 1.
2. Same as in mode 1.
3. Same as in mode 1.
4. If a subroutine fails, its alternate is executed. If the alternate subroutine fails, another alternate can be tried or else the subprogram loses the partial processing and restarts from the beginning. If the alternate succeeds, normal processing continues.
5. Action 4 is repeated until the current message execution is completed.

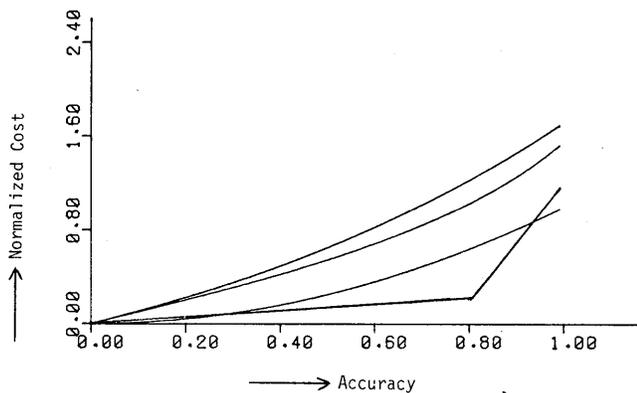


Figure 5—Cost-accuracy function for acceptance tests

In both modes, the acceptance test used to detect the failure of the subroutine is assumed imperfect. If the subroutine failed and this fact is not detected by the acceptance test, it is assumed to be always detected at the end of the execution of the subprogram. In such a case, all processing is lost and execution is restarted. Note that in case of loss of the execution, the time spent is added to the new processing cost.

The following parameters were used as input variables:

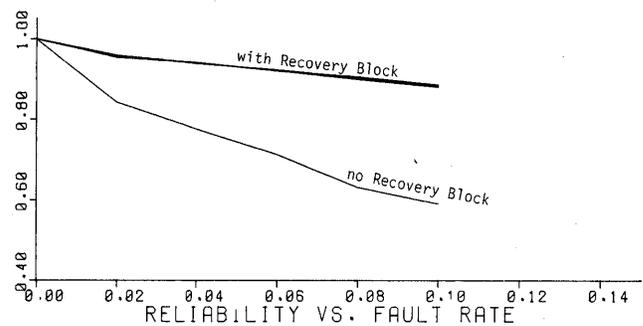
1. Probability of failure of the subroutine or alternate (P_P or P_A).
2. Execution cost ratio of alternate/primary (C_A/C_P). $C_A/C_P = 1.2$ means that the cost of restart and execution of the alternate is 1.2 times the cost of executing the primary.
3. Cost-accuracy function of the acceptance test.
4. Accuracy of the acceptance test (test strength). An accuracy of 0.8 means that acceptance test will catch only 80% of errors. The rest 20% will be caught at the end of subprogram's execution.

The following parameters were measured for the two modes of FDP processing:

1. Execution cost of FDP.
2. Cost of testing.
3. Reliability of the FDP.
4. Cost-reliability index of the FDP.

In the simulation study:

1. The probability of failure of primary or alternates was varied from 0.0 to 0.10.
2. The execution cost ratio of alternate/primary was varied from 1.0 to 1.2
3. The cost-accuracy functions were
 - a. $Y = \text{Exp}(x) - 1$
 - b. $Y = \text{Tan}(x)$
 - c. $Y = X * 2$
 - d. $Y = 0.1 * \frac{x}{k}$ for $x \leq k$
 $= 0.1 + \frac{0.9}{1.0 - k} * (x - k)$ for $x > k$

Figure 6a—FDP reliability vs. probability of failure of a subroutine (Test accuracy = 0.8; $C_A/C_P = 1.2$)

(These functions are plotted in Figure 5; Y is the cost of testing and X is the accuracy of the test [or test strength]).

4. The accuracy of the acceptance test was varied from .7 to .9.

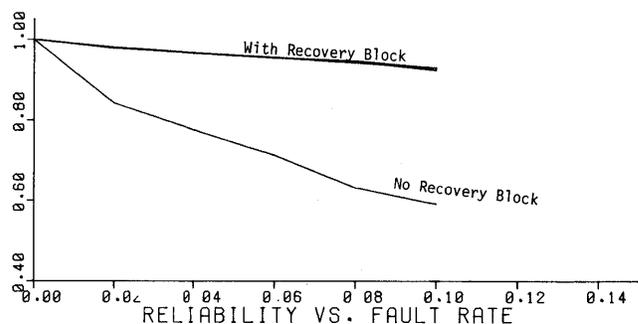
The probability of failure of the largest subroutine was considered to be the variable (failure rate). The probability of failure of all other subroutines were normalized based on the ratio of their execution time with that of the largest subroutine execution cost.

The graphs showing cost-reliability index vs failure rate and reliability vs failure rate for acceptance test strength varying from 0.8 to 0.9 and cost ratio of the alternate to primary varying from 1.0 to 1.2 are as shown in Figures 6a-7d.

We note from our simulation that the reliability of the simulated flight data processing subsystem with recovery blocks exceeds the reliability without recovery blocks (Figures 6a, b) as probability of failure increases. When the probability of failure of the subroutine exceeds 0.10, the reliability with recovery block is twice the reliability without recovery block. Such reliability improvement can increase the mean-time between failure.

We note from our simulation that for cost-accuracy functions of the type exponential and tangent the CRI without recovery blocks does not exceed the CRI with recovery blocks till the probability of failure of the subroutines exceeds 0.15. This is true for test strength range of 0.8-0.9 and C_A/C_P range of 1.0-1.2.

For square and step function for cost accuracy, the CRI without recovery blocks reaches the CRI with recovery block,

Figure 6b—FDP reliability vs. probability of failure of a subroutine (Test accuracy = 0.9; $C_A/C_P = 1.2$)

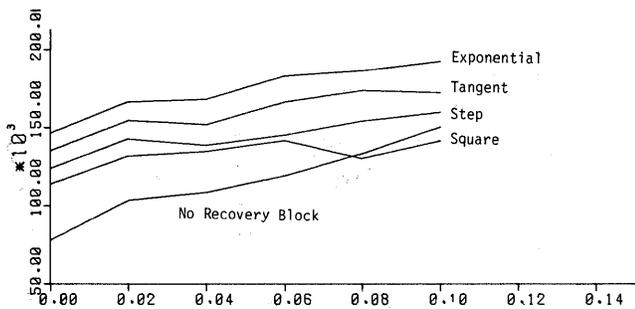


Figure 7a—FDP cost-reliability index vs. probability of failure of a subroutine (Test accuracy = 0.8; $C_A/C_P=1.0$)

when the probability of failure of the subroutines is around 0.10 (see Figures 7a–7d). In particular if $C_A/C_P = 1.0$, test strength = 0.8, and square cost-accuracy function, the CRI without recovery block exceeds the CRI with recovery block, when the probability of failure of subroutines exceeds 0.08 (see Figure 7a). From this, we conclude that recovery block scheme will only give a lower CRI if the probability of failure exceeds 0.08.

One underlying assumption that a failed execution can be restarted in the flight data processing subsystem without any penalty (except the loss of processing up to current state) is really not true in general. One would expect the CRI of the system with no recovery block to be increasing much faster than shown in our simulation. This could result in better justification of recovery block scheme even at lower probabilities of failure.

The simulation model has been developed to try different scenarios regarding the implementation of recovery block scheme in the flight data processing subsystem. The results presented in this report are only illustrative rather than conclusive. More experimentation with actual parameters of the software of FDP is needed. The simulation programs have been written in the language SIMULA and run on DEC-10.

AVOIDANCE OF FAILURE DUE TO DATABASE INCONSISTENCY

One of the key issues in improving the reliability of ARTCC software is to ensure that incorrect data is not stored in the databases (flight plan database, Compool tables, etc.). One way to achieve this is to define integrity assertions on the structure and semantics of the database and surround the database by an integrity monitor. Any access to the database must pass through the integrity monitor for verification. Transactions violating the assertions are disallowed. There are three research questions regarding this approach to ensure integrity of the database:

1. Design of integrity assertions
2. Language of integrity assertions
3. Monitoring of integrity assertions

These issues have been reported by us and others^{3,6} and are briefly discussed below.

There are two types of integrity assertions that can be defined in a database. One type is based on structural constraints. For example, we can declare that duplicate keys or records are not allowed, every table must contain only those items which are fully dependent on the key attributes and no transitive dependencies among attributes are allowed. The second type of constraints concerns the actual values stored in the database. Some examples are as follows:

1. Value of an item must exist between a lower and upper bound; some arithmetic relationship exists among various items (time of flight arrival < time of flight departure).
2. There must be a trend in change of values over a period of time (while an aircraft is ascending, new altitude > old altitude, or when an aircraft is handed over to the next ARTCC, it is not handed back to the old ARTCC).
3. Certain records must exist in the database if some other record was already in the database (flight plan data must exist if the plane is in the ARTCC airspace).

More examples can be found³.

The language to express integrity assertion could be the same as one used for accessing the data. One can always use tables (such as header to a datafile) to describe integrity assertions. These tables are brought in core at the time of access of these files. This mechanism has been used in many other applications⁵ and is a subject of my further research.

The monitoring or validation of integrity assertions can be done before executing the transaction, at run-time, or after executing the transaction. The three methods are briefly described below.

1. *Pre-execution.* The method requires
 - a. simulating the transaction to find results that would be written if assertions are not violated (what is to be written?),
 - b. checking the assertions,
 - c. executing the transaction if all assertions were found true.
2. *Run-time validation.* The method requires
 - a. executing a transaction ignoring its "write" operations,
 - b. checking the assertions,

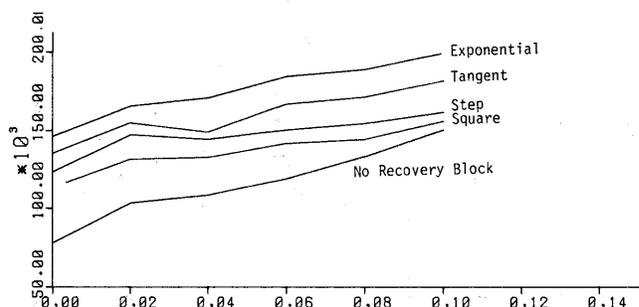


Figure 7b—FDP cost-reliability index vs. probability of failure of a subroutine (Test accuracy = 0.8; $C_A/C_P=1.2$)

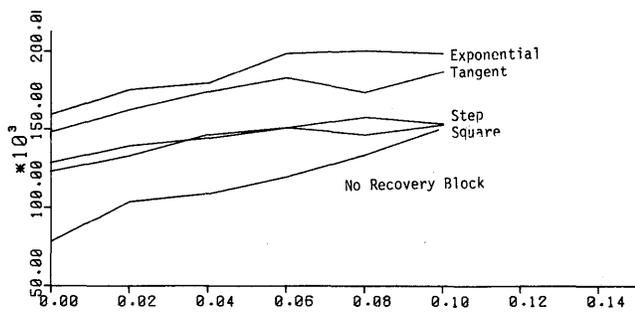


Figure 7c—FDP cost-reliability index vs. probability of failure of a subroutine (Test accuracy = 0.9; $C_A/C_p = 1.0$)

- c. performing the “write” operations if all assertions were found true.
3. *Post-execution validation.* The method requires
 - a. executing the transactions completely,
 - b. checking the assertions,
 - c. performing correction actions.

In ARTCC, we know a priori the types of transactions (or messages) that will be entered in the system. For each message, we have the list of items it will read (readset) and the list of items that it will write (writeset). Under the assumptions that readset and writeset are determined before the transaction starts executing, we found that the pre-execution validation cost is less than or equal to the runtime validation cost which is less than or equal to the post-execution validation cost. These results have been obtained³, and I briefly list three lemmas for comparing the validation methods.

Lemma 1. The cost of pre-execution validation is never larger than the cost of run-time validation.

Lemma 2. The cost of pre-execution validation is never larger than the cost of post-execution validation.

Lemma 3. The cost of run-time validation is never larger than the cost of post-execution validation.

These lemmas are important because we can design our ARTCC software such that no execution takes place without any violation of integrity assertions. The problems of run-time and post-execution validation such as storing the original states for backup are thus avoidable.

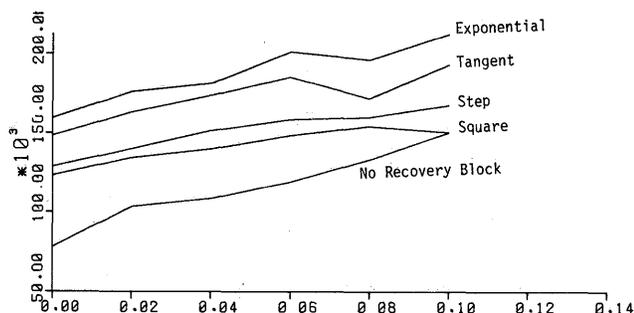


Figure 7d—FDP cost-reliability index vs. probability of failure of a subroutine (Test accuracy = 0.9; $C_A/C_p = 1.2$)

These validations can be very expensive if the probability that integrity assertions are violated is low. The cost of these validations, as a ratio of cost of executing a transaction as the probability error varies, is a topic of further research.

SYNCHRONIZATION OF MULTIPLE CONCURRENT PROCESSES

The problem of synchronizing the interleaved execution of several processes was discussed briefly in the section “Problems of Continuity of Operations in Flight Data Processing Subsystem.” This problem has received increased attention in the last few years because of the need to design highly concurrent processing systems. Such systems contain some form of concurrency control algorithms which ensure correctness of synchronization.

We assume that each process will take the system from an initial consistent state to a final consistent state. We define a serial execution in which each process runs to completion before the next one starts. Thus, serial execution will keep the system in a consistent state. It is possible that some other interleaved execution of the processes can also preserve the consistency. Such executions have the same effect as a serial execution and are called serializable executions.

The flight data processing subsystem employs a synchronization algorithm which is based on the locking/unlocking of resources needed by the process. We find that locking is in general a pessimistic approach because it reduces concurrency and hence reduces efficiency also. Moreover, locking is a type of partial commitment because the system must ensure that a process unlocks all the resources after its completion. If for some reason such as process failure, memory loss, or system abort, unlocking cannot be completed, the system becomes inconsistent. Moreover, there are no general purpose deadlock free algorithms. Thus deadlock detection and resolution becomes an overhead added to the maintenance of locks.

I have proposed an optimistic concurrency control algorithm⁴ which does not commit a process unless it has completed and its effects have been validated. I call the approach optimistic because I believe that in general, few processes will interfere and conflict with each other and hence more concurrent executions can be allowed by the system.

In this paper⁴ I discuss the details of our algorithm and its performance against the locking algorithms in a database transaction processing environment. I note that the optimistic approach will perform as well as the locking approach when no conflicts exist. When conflicts increase, the optimistic approach does better than locking in both crash and non-crash environment. Further research is being done on this approach to study its fault-tolerant capabilities, and this will be a topic of my future research.

CONCLUSIONS AND PLANS FOR FURTHER WORK

The goal of this research is to investigate the required architecture of the automated enroute air traffic control system that will increase its reliability and will provide capabilities to

handle errors and degrade gracefully so that some minimum level of continuity of operations can be maintained. We are interested in design algorithms that are robust and efficient, and have so far investigated techniques that are useful for handling program errors, database loss and inconsistency, and concurrent processing. Our emphasis has been on the implementation and performance issues of such techniques in the flight data processing subsystem.

We have just begun to understand the reliability issues of the enroute system, and plan to investigate software structures that are fault-tolerant and lead to robust processing. My short-range goal is to study the concepts of atomic actions, commitment levels for backup and recovery, and the assurance consistency of database and concurrent processing.

REFERENCES

1. Bhargava, B., H. Chuang, C. Hua, L. Lilien, and T. Altman. "Software and Processing Structures with Performance Requirements of Enroute Air Traffic Control System." Interim report to the Department of Transportation, Department of Computer Science, University of Pittsburgh, December 1979.
2. Bhargava, Bharat, and Cecil Hua. "Cost Analysis of Recovery Block Scheme and Selection Criterion for Alternates." Technical Report, April 1980.
3. Bhargava, Bharat, and Leszek Lilien. "On Optimal Placement of Integrity Assertions in a Transaction Processing System." Technical Report, January 1980.
4. Bhargava, Bharat. "An Optimistic Concurrency Control Algorithm and Its Performance Evaluation Against Locking Approach." Paper presented at International Computer Symposium, Taipei, December 1980.
5. Gray, J., P. McJones, M. Blasgen, et al., "The Recovery Manager of a Data Management System." IBM Technical Report RJ 2623.
6. Gray, J.N., "Notes on Database Operating Systems." In *Operating Systems: An Advanced Course*. Berlin: Springer Verlag, Heidelberg 1978.
7. Lee, P.A., et al. "A Recovery Cache for the PDP-11." *IEEE Transactions on Computers*, 1980, pp. 546-549.
8. Ramamoorthy, C.V., et al., "A Systematic Approach to the Development and Validation of Critical Software for Nuclear Power Plants." Paper presented at 4th International Conference on Software Engineering, September 17-19, 1979.
9. Randell, B. "System Structure for Software Fault Tolerance." *IEEE Transactions, Software Engineering*, SE-1.2 (1975), pp. 220-232.
10. Randell, B., P.A. Lee, and P.C. Treveaven. "Reliability Issues in Computing System Design." *Computing Surveys* (1978), pp. 123-166.
11. Verhofstad, J.S.M. "Recovery Techniques for Database Systems." *ACM Computing Surveys* (1978), pp. 167-196.
12. "Design Specifications—Application Subsystem." U.S. Dept. of Transportation, NASP-5105, Vol. 2.
13. "Subsystem Design Data: Flight Data Processing." U.S. Dept. of Transportation, NAS Enroute State A (Model A3d2.8), NASP-5154-11, April 1979.
14. Zellweger, Andres. "Productivity and Safety of the Control Process." *Proceedings of the Consultative Planning Conference*, U.S. Department of Transportation, March 1978.

ACKNOWLEDGMENT

I would like to thank the members of the software reliability project, Cecil Hua, Tom Altman, Leszek Lilien, Redda Bourna, and Professor Henry Chuang, at the University of Pittsburgh for their help in this study. Professor Chuang also provided the information included in the appendix.

I would also like to thank Roy Smith and Ed Maynard of the Cleveland Enroute Air Traffic Center and David Clapp of

the Transportation System Centre at Cambridge for information about the enroute system.

APPENDIX—SOFTWARE ERRORS AND THEIR FREQUENCY OF OCCURRENCE IN REAL-TIME SOFTWARE

Software errors and their frequency of occurrence in real-time software

The types of errors can be grouped into the following major classes:

1. Computation errors: errors in or resulting from coded equations, equations that produced values directly from the physical problem being solved, and equations used in book-keeping sense. Typical errors are mathematical modeling, index, conversion, and mixed-mode arithmetic.
2. Logic errors: incorrect logic code, missing condition test, flag not tested, etc.
3. Data input errors: format errors, input read from incorrect data file, invalid input read from correct data file, etc.
4. Data output errors: format errors, data written on wrong file, incomplete or missing output, output field size too small, etc.
5. Data-handling errors: errors made in reading, writing, moving, storing, and modifying data, etc.
6. Interface errors: routine/routine interface errors, routine/system software interface errors, wrong routine called, and incompatibilities between database and using routines, etc.
7. Definition errors: errors in specification of global variables and constants, data not properly defined/dimensioned, etc.
8. Present database errors: data not initialized, initialized to wrong values, incorrect data units, etc.
9. Documentation errors: errors in design and operational documents.

10. Operation errors: wrong database used, wrong tapes used, configuration control errors, etc.
11. Others: time limit exceeded, storage limit exceeded, compilation errors, etc.

The frequency of occurrence of each type of error is determined by the aforementioned factors. Conclusive results about error occurrence are difficult to obtain. The extensive Software Reliability Study, performed by TRW for Rome Air Development Center, has revealed the results shown in the table for real-time software, to which ATC software belongs.

The table shows the percentage breakdown of major error types which resulted from analysis of error data obtained in a large state-of-the-art real-time software project. The software was developed using top-down structured programming approach under rigorously enforced standards and procedures. The application software is in FORTRAN, and the operating system is in assembly language.

TABLE—Percentage breakdown of major error types in real-time software (at final stage of development)

Major Error Types	Real-Time Application Software	Real-Time Operating System
Computational (1)	11.2	2.5
Logic (2)	18.1	34.6
Data input (3)	1.1	3.7
Data output (4)	2.2	4.9
Data handling (5)	6.7	21.0
Interface (6)	6.7	7.4
Data definition (7)	7.9	7.4
Database (8)	32.6	4.9
Others (9,10,11)	13.5	13.6

A state- and time-dependent error occurrence-rate software reliability model with imperfect debugging

by J. G. SHANTHIKUMAR

Syracuse University
Syracuse, New York

ABSTRACT

In this paper, assuming a state- and time-dependent software failure rate and imperfect debuggings, we develop a simple binomial model for software error occurrences. Maximum likelihood estimates for the required parameters of this model are also derived. It is established that the Jelinski-Moranda, imperfect debugging and non-homogeneous Poisson process models are all special cases of ours.

INTRODUCTION

In recent years, several statistical approaches have been developed to measure and predict software quality. One of such approaches is to postulate a stochastic model, use its results and the data on error occurrences to estimate the model parameters and forecast the future behavior using the model and the estimated parameters.¹⁻²² In most of these models it is assumed that a software error once detected is perfectly debugged. Recently, in an article⁶ in the proceedings of the National Computer Conference, Goel and Okumoto, however, considered a model in which imperfect debugging is allowed. Assuming a fixed number of initial error content and a constant failure rate for each error, they formulated a Semi-Markovian model for the software error occurrences. Using this model they derived expressions for software performance measures. Since these expressions seem complex, they also suggest some approximation.

In this paper, assuming a state- and time-dependent software failure rate and imperfect debuggings, we develop a simple binomial model for software error occurrences. We establish that the Semi-Markovian model developed by Goel and Okumoto⁶ is a special case of our binomial model. It is also noted that the Jelinski-Moranda⁷ and the Non-homogeneous Poisson Process model⁵ are also special cases of our model.

The basic model and the assumptions are presented in section 2. System performance measures are derived in section 3. The parameter estimation is discussed in section 4 and the expressions for performance prediction are developed in section 5. The generality of our model is demonstrated in the appendix.

THE MODEL

The software reliability model developed here is based on the following assumptions.

1. The initial error content at the beginning of the observation phase, that is at time zero, is an unknown constant N .
2. The probability that an error will cause a software failure in a small time interval $(t, t + \Delta t)$ is equal to $\phi(t)\Delta t + o(\Delta t)$, where $\lim_{\Delta t \rightarrow 0} \left(\frac{o(\Delta t)}{\Delta t}\right) = 0$. These probabilities for all errors are independent of one another and dependent of time. That is, if there are r errors in the software at time t , the probability of a software failure in $(t, t + \Delta t)$ is $r\phi(t)\Delta t + o(\Delta t)$. Note that this assumption, when restricted $\phi(t)$ to be a constant λ , is equivalent to assumption (2) of Goel and Okumoto,⁶ page 769.
3. When an error occurs, it is corrected with probability p . That is, with probability q ($q = 1 - p$), the error is imperfectly debugged (not eliminated).
4. No new errors are created, at most one error is removed at a correction time, and the time taken to correct an error is negligible.

With this set of assumptions we will now formulate our model. We shall do this by considering each error separately. Let us consider an error (out of those N) present in the software at time zero. Let T be the time by which this error is removed from the software. Suppose $F(\cdot)$ is the cumulative distribution function of T . That is $\Pr\{T \leq t\} = F(t)$, $t > 0$. Since

$$\Pr\{T > t + \Delta t\} = \Pr\{T > t \text{ and error is not removed during } (t, t + \Delta t)\},$$

we get

$$\bar{F}(t + \Delta t) = F(t)\{1 - p\phi(t)\Delta t\} + o(\Delta t), \quad (1)$$

where $\bar{F}(t) = 1 - F(t) = \Pr\{T > t\}$ and $p\phi(t)\Delta t + o(\Delta t)$ is the probability that the error is removed during $(t, t + \Delta t)$.

Now dividing (1) by Δt and taking the limit as $\Delta t \rightarrow 0$, we get

$$\frac{d}{dt}\{\bar{F}(t)\} = -p\phi(t)\bar{F}(t), t > 0. \quad (2)$$

Since $\Pr\{T > 0\} = 1$, we have the boundary condition $\bar{F}(0) = 1$. Solving (2) with this boundary condition we get

$$\bar{F}(t) = \exp\{-pG(t)\}, t > 0, \quad (3)$$

where

$$G(t) = \int_0^t \phi(x) dx, t \geq 0. \quad (4)$$

Note that $\lim_{t \rightarrow \infty} \bar{F}(t)$ need not be zero since the limit $\lim_{t \rightarrow \infty} G(t)$ need not be infinity. This means that an error in the software may never be removed. This would then represent the situations in which an error is resident in a part of the code which is never processed or very scarcely processed. In almost all the software reliability models previously considered, it is assumed that all errors would be eventually eliminated. Our results thus represent a more realistic situation. From (3) we also have

$$F(t) = 1 - \exp\{-pG(t)\}, t > 0. \quad (5)$$

Now let X be the time at which this error causes a software failure for the first time. Then, if $H(t) = \Pr\{X \leq t\}$, $t > 0$, using an analysis similar to the above we can show that

$$\bar{H}(t) = 1 - H(t) = \exp\{-G(t)\}, t > 0 \quad (6)$$

and

$$H(t) = 1 - \exp\{-G(t)\}, t > 0, \quad (7)$$

where $G(t)$ is as defined in (4).

With these results (3), (5), and (7), we have binomial distributions for the number of errors remaining at time t , for the number of errors perfectly debugged by time t , and for the number of distinct errors detected by time t with parameter sets $(N, \bar{F}(t))$, $(N, F(t))$, and $(N, H(t))$, respectively.

PERFORMANCE MEASURES

In this section, using results (3)–(7), we will derive expressions for software performance measures that are of interest to us.

Distribution of Number of Remaining Errors

Let $P_{N,n}(t)$ be the probability that there are n errors remaining at time t . From (3) we know that the probability that an error is not perfectly debugged by time t is $\bar{F}(t)$. Then $P_{N,n}(t)$ is binomial with parameters $(N, \bar{F}(t))$. That is,

$$P_{N,n}(t) = \binom{N}{n} (\bar{F}(t))^n (F(t))^{N-n}, n = 0, 1, \dots, N, \quad (8)$$

with mean

$$E(R(t)) = N\bar{F}(t), \quad (9)$$

where $R(t)$ is the number of errors remaining at time t . A software model satisfying conditions given in section 2 with $\phi(t) = \lambda$ a constant, should be identical to the imperfect debugging model of Goel and Okumoto.⁶ Even though the results for $P_{N,n}(t)$ given there seems different from (8), we establish their equivalence in the appendix.

Distribution of Time to a Completely Debugged System

Let T^* be the time taken to completely debug the system. Define $G_{N,0}(t) = \Pr\{T^* \leq t\}$, $t > 0$. That is, by time T^* , the number of errors remaining should be zero. Then $G_{N,0}(t)$ should be equal to $P_{N,0}(t)$. So, from (8) and (3), we get

$$G_{N,0}(t) = (1 - \exp\{-pG(t)\})^N, t > 0. \quad (10)$$

It should be noted that, for reasons discussed earlier, $G_{N,0}(\cdot)$ may be defective. That is, $G_{N,0}(\infty)$ need not be 1.

Distribution of Time to n Remaining Errors

Let T_n^* be the time by which the number of remaining errors is n and define $G_{N,n}(t) = \Pr\{T_n^* \leq t\}$, $t > 0$. Noting that the events $\{R(t) = r, r < n\} \equiv \{T_n^* \leq t\}$ we have

$$G_{N,n}(t) = \sum_{r=0}^n P_{N,r}(t) = \sum_{r=0}^n \binom{N}{r} (\bar{F}(t))^r (F(t))^{N-r}, \quad n = 0, 1, \dots, N. \quad (11)$$

Distribution of Time to Next Software Failure

Suppose there are r software errors remaining just after a recent software failure, say, at time t . Let $Y(r, t)$ be the time to next software failure. Then $Y(r, t)$ is the minimum of the r failure times, each of the r remaining errors. The unconditional cumulative distribution function of these failure times is given by equation (7). Now suppose X_i , $i = 1, 2, \dots, r$, are the failure times corresponding to these r errors. Then knowing that $X_i > t$, $i = 1, 2, \dots, r$, we have from (7) and the laws of conditional probabilities,

$$\Pr\{X_i > t + x | X_i > t\} = \exp\{-(G(t+x) - G(t))\}, x > 0, \quad i = 1, 2, \dots, r. \quad (12)$$

Then

$$\Pr\{Y(r, t) > x\} = \Pr\{X_i > t + x, i = 1, 2, \dots, r | X_i > t, i = 1, 2, \dots, r\} = \exp\{-r(G(t+x) - G(t))\}, x > 0. \quad (13)$$

Clearly (13) is the reliability function of the software when there are r errors remaining at time t . Now to use all these expressions, we need the model parameters N , p , and the

function $\phi(t)$. We shall attend to this problem in the next section.

PARAMETER ESTIMATION

Suppose we have observed the software failure times caused by each of n errors for the first time. That is, we have the observations of the random variables $X_i, i = 1, 2, \dots, n$ (X_i as defined earlier). Let $S_i, i = 1, 2, \dots, n$, be the values of $X_i, i = 1, 2, \dots, n$ in the increasing order. Then from (13) it is easily verified that,

$$\Pr\{S_k \geq t + x | S_{k-1} = t\} = \exp\{- (N - k + 1)(G(t + x) - G(t))\}, x > 0. \quad (14)$$

Now suppose that $f(s_1, s_2, \dots, s_n)$ is the joint probability density function of S_1, S_2, \dots, S_n . Then from (14), the properties of the model, and the laws of conditional probabilities, we can show that

$$f(s_1, s_2, \dots, s_n) = \prod_{k=1}^n \{(N - k + 1)\phi(s_k)\exp\{- (N - k + 1)(G(s_k) - G(s_{k-1}))\}\}, s_i > 0, i = 1, 2, \dots, n, \quad (15)$$

where $s_0 = 0$. Then from (15), for a given sequence s_1, s_2, \dots, s_n of n software failure times caused by n distinct errors for the first time, the log likelihood function L is given by

$$L = \sum_{k=1}^n \ln(N - k + 1) + \sum_{k=1}^n \ln\phi(s_k) - \sum_{k=1}^n (N - k + 1)(G(s_k) - G(s_{k-1})). \quad (16)$$

To use (16) for parameter estimation, we need specific form of $\phi(t)$. We choose

$$\phi(t) = \alpha b \exp(-bt), t \geq 0 \quad (17)$$

following Goel and Okumoto.⁵ Note that several other forms may also be chosen for $\phi(t)$. From (4) and (17), we have

$$G(t) = \alpha(1 - \exp(-bt)), t \geq 0. \quad (18)$$

Using (16) and (18), it can be shown that (see Shanthikumar²⁰) the maximum likelihood estimates $\hat{N}, \hat{\alpha},$ and \hat{b} of $N, \alpha,$ and $b,$ respectively, are the solution of

$$\sum_{k=1}^n \frac{1}{N - k + 1} - \alpha(1 - \exp(-bs_n)) = 0 \quad (19)$$

$$\frac{n}{\alpha} - \sum_{k=1}^n (N - k + 1)(\exp(-bs_{k-1}) - \exp(-bs_k)) = 0 \quad (20)$$

and

$$\frac{n}{b} - \sum_{k=1}^n s_k - \sum_{k=1}^n (N - k + 1)\alpha(s_k \exp(-bs_k) - s_{k-1} \exp(-bs_{k-1})) = 0. \quad (21)$$

These equations (19), (20), and (21) can be numerically solved to obtain these estimates. Next we will look at an estimate for p . Let $u_i = 1, 2, \dots, n,$ be the number of time error i (out of the n distinct errors observed) caused software failures during $(0, t_i)$. Then $u_i \geq 1, i = 1, 2, \dots, n.$ Since the number of times an error causing software failures with imperfect debuggings can be represented by a Geometric random variable (see Shanthikumar¹⁸) with mean $1/p,$ we can approximate \hat{p} by

$$\hat{p} \cong \frac{1}{n} \sum_{i=1}^n \frac{1}{u_i}. \quad (22)$$

PERFORMANCE PREDICTION

Now that we have the estimates $\hat{N}, \hat{\alpha}, \hat{b},$ and $\hat{p},$ we can use equations (8)–(13) for performance prediction. We should note, however, that we have made some observations to estimate the parameters and therefore equations (8)–(13) have to be modified. This is because the current state of the software system has changed. This aspect has been ignored in an earlier paper.⁶ Suppose we have observed n distinct errors during the time period $(0, t_i)$. Also, suppose we have observed a total of l software failures during this time period. Then $l \geq n$ since some or all of these n errors may have been imperfectly debugged. Hence if R_i is the number of remaining errors at time t_i, R_i is neither equal to $N - n$ nor equal to $N - l.$ In fact, the exact value of R_i is unknown because of imperfect debuggings. If $R_i = r,$ then equations (8)–(13) may be used when the time origin shifted to t_i and N replaced by $r.$ Specifically, from (8),

$$P_{r,k}(t | R_i = r) = \binom{r}{k} (\bar{F}^*(t))^{k-1} (F^*(t))^{r-k}, k = 0, 1, \dots, r, t > t_i, \quad (23)$$

is the probability that there will be k errors remaining at time t given that there are r errors remaining at time $t_i.$ In this equation (see (4), (6), and (7))

$$F^*(t) = 1 - \exp\{-pG^*(t)\}, t > t_i$$

$$\bar{F}^*(t) = 1 - F^*(t)$$

and

$$G^*(t) = \int_{t_i}^t \phi(x) dx = G(t) - G(t_i), t \geq t_i$$

due to the shift in time origin.

Similarly, from (10),

$$G_{r,0}(t | R_i = r) = (1 - \exp\{-p(G(t_i))\})^r, T > t_i \quad (24)$$

from (11),

$$G_{r,k}(t | R_i = r) = \sum_{j=0}^k \binom{r}{j} (\bar{F}^*(t))^j (F^*(t))^{r-j}, k = 0, 1, \dots, r, t \geq t_i \quad (25)$$

and from (13), the reliability function

$$\Pr\{Y(r, t_i) > x\} = \exp\{-r(G(t_i + x) - G(t_i)), x > 0. \quad (26)$$

So, in order to use the above equations, we need an estimate for the number of errors remaining at time t_i . As mentioned earlier, the exact value of R_i is unknown and therefore we may develop a probability distribution for it.

Let z_i be the last time an error $i, i = 1, 2, \dots, n$, caused a software failure before time t_i . That is, this error did not reappear during (z_i, t_i) . If the error is imperfectly debugged, the probability that it will not show up during (z_i, t_i) is from (7), equal to $\exp\{-(G(t_i) - G(z_i))\}$, $i = 1, 2, \dots, n$. This probability, when the error is perfectly debugged, is 1. Noting that the probability of perfect debugging is p , and using Bayes rule, we get

$$\beta_i = p / \{p + (1 - p)\exp\{-(G(t_i) - G(z_i))\}\}, i = 1, 2, \dots, n,$$

where β_i is the probability that an error i (one of the n distinct errors) detected is perfectly debugged given that this error did not reappear during (z_i, t_i) . Then

$$\Pr\{R_i = N - n + r\} = \sum_{A \in M_r} \prod_{i \in A'} \beta_i \prod_{i \in A} (1 - \beta_i), \quad r = 0, 1, \dots, n, \quad (27)$$

where M_r is the set of all possible subsets of $\{1, 2, \dots, n\}$ with cardinality r , and A' is the complement of A . That is, $A' = \{1, 2, \dots, n\} - A$. It can be verified, after some algebra, that

$$E(R_i) = N - \sum_{i=1}^n \beta_i \triangleq \hat{r}. \quad (28)$$

Now either \hat{r} can be used as an estimate for the number of remaining errors or use (27) along with (23)–(26) to predict software performance. That is, from (23) and (27),

$$P_{R_i, k}(t) = \sum_{r=k}^N \Pr\{R_i = r\} P_{r, k}(t | R_i = r), k = N - n, \dots, N, \quad (29)$$

from (24) and (27),

$$G_{R_i, 0}(t) = \sum_{r=N-n}^N \Pr\{R_i = r\} G_{r, 0}(t | R_i = r), t > t_i, \quad (30)$$

from (25) and (27),

$$G_{R_i, k}(t) = \sum_{r=k}^N \Pr\{R_i = r\} G_{r, k}(t | R_i = r), k = N - n, \dots, N, \quad (31)$$

and the reliability function from (26) and (27) is

$$\Pr\{Y(R_i, t_i) > x\} = \sum_{r=N-n}^N \Pr\{R_i = r\} \Pr\{Y(r, t_i) > x\}. \quad (32)$$

Note that the above expression for the reliability function corrects an error in equations (23), (24), and (25) of Goel and Okumoto (1979)⁶ (see Shanthikumar,¹⁷ page 71). Since evaluation of equation (27) is of combinatorial nature, a simple binomial approximation is proposed based on $E(R_i) = \hat{r}$. It is

$$\Pr\{R_i = N - n + r\} \cong \binom{n}{r} \bar{\beta}^{n-r} (1 - \bar{\beta})^r, r = 0, 1, \dots, n, \quad (33)$$

where $\bar{\beta} = \frac{1}{n} \sum_{i=1}^n \beta_i$, so that $E(R_i) = N - n\bar{\beta} = \hat{r}$ is preserved. Now equations (23)–(26), (29)–(32), and (33) can be used for performance prediction.

CONCLUSION

In this paper, assuming a state- and time-dependent software failure rate and imperfect debuggings, we developed a simple binomial model for software error occurrences. Maximum likelihood estimates for the required parameters of this model are also derived. For this we use $\phi(t) = \alpha b \exp(-bt)$, $t > 0$ for the time-dependent failure rate function. In the appendix it is established that the imperfect debugging model of Goel and Okumoto⁶ is a special case of this model (specifically when $\phi(t) = \lambda$ is a constant. Then, obviously, when $\phi(t) = \lambda$ and the probability p of perfect debugging is equal to one, we will get the results for the Jelinski-Moranda model.⁷ It can also be shown (see Shanthikumar²⁰) that when $p = 1$, $N \rightarrow \infty$, $\alpha \rightarrow 0$, and $N\alpha \rightarrow \alpha < \infty$, the above model reduces to the non-homogeneous Poisson process model discussed by Schneidewind¹⁶ and Goel and Okumoto.⁵ Because of this generality of this model, it is expected that this model will prove to be versatile.

APPENDIX

In this appendix we will systematically transfer Goel and Okumoto's results⁶ to match a special case of our results. The special case considered here is $\phi(t) = \lambda$, $t \geq 0$. Then $G(t) = \lambda t$, $t \geq 0$ and $\bar{F}(t) = \exp(-\lambda t)$, $t > 0$.

Distribution of Time to a Completely Debugged Software System

The distribution $G_{N,0}(t)$ of time to a completely debugged system is given by equation (12) in Goel and Okumoto's "A Markovian Model for Reliability and Other Performance Measures of Software Systems."⁶ It is

$$G_{N,0}(t) = \sum_{j=1}^N C_{N,j} (1 - e^{-j\lambda t}), t \geq 0 \quad (A1)$$

where

$$C_{N,j} = \binom{N}{j} (-1)^{j-1} \quad (A2)$$

Substituting (A2) in (A1), we get

$$G_{N,0}(t) = \sum_{j=1}^N -\binom{N}{j} (-1)^j + \binom{N}{j} (-e^{-j\lambda t}) = (1 - e^{-\lambda t})^N \quad (A3)$$

is obtained using the combinatorial identity

$$\sum_{j=0}^N \binom{N}{j} f^j = (1+f)^N$$

Note that (A3) agrees with (10) when $G(t) = \lambda t$.

Distribution of Time to a Specified Number of Remaining Errors

The distribution $G_{N,n_0}(t)$ of time to n_0 number of remaining errors is given by equation (14) of "A Markovian Model." It is

$$G_{N,n_0}(t) = \sum_{j=1}^{N-n_0} B_{N,j,n_0} \{1 - e^{-(n_0+j)p\lambda t}\}, t \geq 0 \quad (A4)$$

where

$$B_{N,j,n_0} = \frac{N!}{n_0!j!(N-n_0-j)!} (-1)^{j-1} \frac{j}{n_0+j} \quad (A5)$$

Rewriting (A5) we get

$$\begin{aligned} B_{N,j,n_0} &= \binom{N}{j+n_0} \binom{n_0+j-1}{j-1} (-1)^{j-1} \\ &= \binom{N}{j+n_0} (-1)^{j-1} \sum_{r=1}^j \binom{j+n_0}{j-r} (-1)^{r-1} \quad (\dagger) \\ &= \sum_{r=1}^j \binom{N}{j+n_0} \binom{j+n_0}{j-r} (-1)^{j+r} \\ &= \sum_{r=1}^j \binom{N}{r+n_0} \binom{N-r-n_0}{j-r} (-1)^{j+r} \\ &= \sum_{n=n_0+1}^{j+n_0} \binom{N}{n} \binom{N-n}{j-n+n_0} (-1)^{j+n-n_0} \quad (A6) \end{aligned}$$

Now substituting (A6) in (A4) and interchanging the order of summations, we get

$$\begin{aligned} G_{N,n_0}(t) &= \sum_{n=n_0+1}^N \sum_{l=0}^{N-n} \binom{N}{n} \binom{N-n}{l} \{(-1)^l + (-e^{-p\lambda t})^l e^{-np\lambda t}\} \\ &= 1 - \sum_{n=n_0+1}^N \binom{N}{n} e^{-np\lambda t} (1 - e^{-p\lambda t})^{N-n} \\ &= \sum_{n=0}^{N_0} \binom{N}{n} (e^{-p\lambda t})^n (1 - e^{-p\lambda t})^{N-n} \quad (A7) \end{aligned}$$

Note that (A7) agrees with (11), when $\bar{F}(t) = \exp(-\lambda pt)$.

Distribution of Number of Remaining Errors

The distribution of $P_{N,n_0}(t)$ of n_0 remaining errors at time t is given by equation (17) of "A Markovian Model." It is

$$P_{N,n_0}(t) = G_{N,n_0}(t) - G_{N,n_0-1}(t) \quad (A8)$$

Substituting (A7) in (A8) we get

$$P_{N,n_0}(t) = \binom{N}{n_0} (e^{-p\lambda t})^{n_0} (1 - e^{-p\lambda t})^{N-n_0} \quad (A9)$$

a binomial distribution. Equation (A9) agrees with equation (8), $\bar{F}(t) = \exp(-\lambda pt)$.

REFERENCES

1. Angus, J.E., R.E. Schafer, and A. Sukert, "Software Reliability Model Validation," *Proc. of Annual Reliability and Maintainability Symposium* (1980), pp. 191-193.
2. Basin, S.L., *Estimation of Software Error Rate Via Capture-Recapture Sampling*, Science Applications, Inc., Palo Alto, California (1974).
3. Endres, A., "An Analysis of Errors and Their Causes in System Programs," *Proc. of the 1975 International Conference on Reliable Software* (1975), pp. 327-336.
4. Forman, E.H. and N.D. Singpurwalla, "An Empirical Stopping Rule for Debugging and Testing Computer Software," *J. of American Statistical Association*, Vol. 72 (1977), pp. 750-757.
5. Goel, A.L. and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability*, Vol. 28 (1979), pp. 206-211.
6. Goel, A.L. and K. Okumoto, "A Markovian Model for Reliability and Other Performance Measures of Software Systems," *Proc. of the National Computer Conference* (1979), pp. 769-774.
7. Jelinski, Z. and P. Moranda, "Software Reliability Research," *Statistical Computer Performance Evaluation*, W. Freiberger (Ed.), Academic Press (1972), pp. 465-484.
8. Littlewood, B. and J.L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," *Applied Statistics*, Vol. 22, (1973), pp. 332-246.
9. Littlewood, B., "A Reliability Model for Systems With Markov Structure," *Applied Statistics*, Vol. 24 (1975), pp. 172-177.
10. Miyamoto, I., "Software Reliability in On-Line Real Time Environment," *Proc. of the 1975 International Conference on Reliable Software* (1975), pp. 194-203.
11. Moranda, P., "Prediction of Software Reliability During Debugging," *Proc. of the Annual Reliability and Maintainability Symposium* (1975), pp. 327-332.
12. Moranda, P., "Error Detection Models for Application During Program Development," *Proc. of the Nineteenth Annual Technical Symposium—Pathways of System Integrity* (1980), pp. 75-78.
13. Musa, J.D. "A Theory of Software Reliability and Its Application," *IEEE Transactions on Software Engineering* (1975), pp. 312-327.
14. Schick, G.J. and R.W. Wolverton, "Assessment of Software Reliability," *Proc. Operations Research*, Physica-Verlag, Wurzburg-Wien (1973), pp. 395-422.
15. Schick, G.J. and R.W. Wolverton, "An Analysis of Competing Software Reliability Models," *IEEE Transactions on Software Engineering* (1978), pp. 104-120.
16. Schneidewind, N.J., "Analysis of Error Process in Computer Software," *Proc. of the 1975 International Conference on Reliable Software* (1975), pp. 337-346.
17. Shanthikumar, J.G., "Software Performance Prediction Using a State-Department Error Occurrence-Rate Model," *Proc. of the Nineteenth Annual Technical Symposium—Pathways to System Integrity* (1980), pp. 67-72.
18. Shanthikumar, J.G., "A Binomial Model for Software Performance Prediction," *Proc. of the Eighteenth Annual Allerton Conference on Communication, Control, and Computing*, (1980), to appear.
19. Shanthikumar, J.G. and S. Tufekci, "Optimal Software Release Time Using Generalized Decision Trees," *Proc. of the Fourteenth Annual Hawaii International Conference on System Sciences* (1981), to appear.
20. Shanthikumar, J.G., "A General Software Reliability Model for Performance Prediction," Technical Report, Dept. of Ind. Eng. & Opns. Res., Syracuse University (1980), p. 18.
21. Shooman, M.L., "Software Reliability: Measurement and Models," *Proc. of the Annual Reliability and Maintainability Symposium* (1975), pp. 485-491.
22. Trivedi, A.K. and M.L. Shooman, "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters," *Proc. of the 1975 International Conference on Reliable Software* (1975), pp. 208-220.

†This is obtained using the combinatorial identity (1.5) in page 1 of Gould, H.W., *Combinatorial Identities*, Morgantown Printing and Binding Co., (1972).

On the complexity of measuring software complexity*

by G. MICHAEL SCHNEIDER,
ROBERT L. SEDLMEYER, and
JOE KEARNEY

University of Minnesota
Minneapolis, Minnesota

INTRODUCTION

The term *software complexity* (program quality, program complexity, . . .) has been used by software engineering researchers to denote the testability, maintainability, readability, and/or comprehensibility of a program. Curtis⁷ points out a common bond that unites all these concepts: a program's complexity determines how difficult it is for programmers to work with.

The complexity of a program is an important factor in two parts of the software lifecycle. In the development phase complexity strongly influences the effort required to debug and test program modules and subsystems. In the maintenance phase complexity determines how difficult it will be to locate and correct undetected implementation errors, and also how much effort will be required to modify program modules to incorporate specification changes. Since maintenance comprises the costliest part of the software lifecycle, a program's complexity will have a direct bearing on the ultimate manpower and machine costs associated with the program. Obviously, then, minimizing program complexity is a worthwhile pursuit.

For approximately a decade software engineering researchers have struggled to develop a suitable complexity measure. All previous research has shared a common approach: identify a relationship between an intrinsic program property and programmer performance on a given programming task. Candidate properties can be generally classified as pertaining to control structure^{16,17,30}, data usage⁶, token volume^{12,15,19}, or a combination thereof.^{13,19} Representative programming tasks have included locating a single error in a program listing^{1,7,10,11} implementing a modification^{7,8,23}, and constructing a program from its specifications.^{18,25} Performance measures have usually been based on task completion time or correctness. Having formulated a complexity measure, a validity study may then be performed to ascertain the predictive power of the measure. In essence, such a study should demonstrate that performance differences are due to experimental manipulations of complexity. In many cases, however, these studies are

confounded by performance differences arising from two other sources: the variability of the subjects themselves and the selection of the experimental programming task. In the next section we present an historical summary of such differences. The remainder of the paper discusses results from an experiment to study the effects of programmer and programming task factors on complexity measures and outlines a multi-dimensional approach to measuring software complexity.

SOURCES OF PERFORMANCE DIFFERENCES

Programmer-related differences

Immense individual differences have been noted in almost every software engineering study. In 1968, while examining the effects of time-sharing and batch environments on productivity, Sackman, Erickson and Grant²¹ reported individual differences of up to 28:1. This surprising discovery prompted them to report in their conclusion, "The most important practical finding involves the striking differences in performance." During the same year Schwartz²² concluded that "an order of magnitude difference in capability" may exist within a group of programmers involved in developing a large scale software system. In a series of articles exploring the reasons for the high cost of software Boehm³ stated, "Productivity variations of 5:1 between individuals are common." Significant individual differences have also been reported in studies concerning debugging^{1,10,11,20}, flowcharting²⁸ and comprehension.^{14,27,32} The important point here is not that individual differences among programmers exist, but that the *variability* is so large that experimental results may depend more on individual differences than on experimentally induced differences. With respect to program complexity, we may find that what is incredibly difficult for one programmer may be trivial for the next, and a complexity measure applicable to the former programmer may be useless to the latter.

Task-related differences

Not all programmers may perform all programming tasks with equal facility. Evidence of the dependence of pro-

*This work was partially supported by NSF Grant MCS-8016549 and a research fellowship from the Control Data Corporation.

programmer performance on the programming task can be found in a series of studies conducted at the General Electric Center for Software Management Research.

Curtis, Sheppard, Milliman, Borst, and Love⁷ studied the relationship between Halstead's E, McCabe's V(G), program length (measured by number of statements), and programmer performance on *maintenance* tasks. The first experiment showed negative correlations between the recall of program statements (a popular measure of program comprehension) and the three complexity measures. Using transformed scores the correlations were $-.73$, $-.21$, and $-.65$ respectively for E, V(G), and lines of code. In the second part of the experiment Curtis et al. examined the relationship between the complexity measures and performance on *modification* tasks. Both accuracy and time to implement a program modification were used as performance measures. This time smaller correlations were found between the complexity metrics and two performance measures:

	E	V(G)	Length
Accuracy	-.21	-.36	-.28
Time	.25	.23	.20

The results of this experiment indicated that Halstead's E, McCabe's V(G) and the number of lines of code were all weak predictors of programmer recall and programmer ability to modify a program, but, more significantly, that the correlational strength was dependent on the programming task used!

Curtis, Sheppard, and Milliman²⁴ conducted another experiment examining Halstead's E, McCabe's V(G), and program length. In this experiment the complexity measures were compared with the number of minutes necessary to *locate and debug* a single error in a program. Correlations between the complexity measures and performance were substantially higher in this experiment. If the measures were applied to the subroutine in which the bug was located the following correlations were obtained:

	E	V(G)	Length
Time	.66	.63	.67

In the third of a series of experiments, Sheppard, Milliman and Curtis²⁵ studied the association between Halstead's V, Halstead's E, McCabe's V(G), and the number of program statements with measures of performance in program *construction*. Subjects were given a program specification and asked to write a program. The dependent variable was the time to write a correct program. They reported statistically significant correlations between programming time and all four of the complexity measures.

	V	E	V(G)	Length
Time	.78	.61	.66	.35

The correlational data gathered by Curtis et al. shows evidence of the sensitivity of these program-based measures to the programming task being analyzed—either maintenance, modification, debugging, or construction.

PROGRAM-PROGRAMMER-PROGRAMMING TASK INTERACTIONS AND COMPLEXITY MEASURES

Much circumstantial evidence exists for the premise that program complexity is not a simple function of the program itself. Despite this apparent sensitivity of complexity to both individual and task differences research has continued to concentrate on the program as the ultimate source of differential performance. In this section we present the results of an experiment, which demonstrate the futility of program-based complexity measures for predicting general programmer performance.

Moher/Schneider Study

To explore the sensitivity of complexity measures to programmer and programming task factors we analyzed data collected from a recent study by T. Moher and G. M. Schneider¹⁸. The study, carried out over a seven-month period in 1979, included two program comprehension tasks—one based on a 51-line FORTRAN program (called SR), and one based on a 221-line FORTRAN program (LR). One hundred and sixty subjects participated in the study. The subject pool was composed of a 100-member student group and a 60-member professional group who each filled out an extensive background questionnaire.* All subjects participated in the first comprehension task, the Short Reading (SR) task, while a randomly selected subgroup of 54 students and 26 professionals also participated in the Long Reading (LR) task. Both tasks involved the same operations: each subject was asked to examine the task program for a specified amount of time and then complete a 20-question multiple choice examination. The number of correct responses on the examination was used as the performance measure for both comprehension tasks.

Analysis of Performance Data

Program factors

As a departure point for investigating the effects of programmer and programming task factors on the validity of program-based complexity measures, we first examined the contribution of program factors to performance differences. Table I summarizes the performance data for all subjects and complexity measures** on programs SR and LR.

As expected, the subjects found program LR, on the whole, to be more difficult to understand. Their comprehension scores dropped by approximately 29% (from 13.2 correct out of 20 to 9.4). Such a degradation in performance is easily reconciled with an increase in the three selected complexity

*To avoid bias the subjects were randomly selected from a wide range of schools and corporations. For a complete discussion of the experiment and subject selection procedures refer to Moher and Schneider (1981).

**The complexity measures selected are considered representative of the program-based approaches discussed in the literature. They are McCabe's V(G), Halstead's E, and Chapin's Q.

TABLE I—Overall performance differences on tasks SR and LR

	N	MEAN	STD.DEV.	RANGE	V(G)	E	Q
PROGRAM SR	160	13.2	3.8	1-20	2	8094	2.2
PROGRAM LR	80	9.4	3.7	0-19	5.2	180087	6.0

measures. If we make the assertion that these measures are perfect predictors of performance, then we may also use them to predict the magnitude of the performance drop by applying a simple two-point regression analysis.

Before we continue our data analysis we note that the functional relationship between each complexity measure and performance is based on a particular programmer population performing a particular programming task on two particular programs. However, the implication is that this functionality will generalize to different programmer populations, different programming tasks and different programs. Our objective will be to show that, even if the programs are held constant, this functionality does not generalize.

Programmer factors

A cursory examination of a histogram of the performance data for programs SR and LR reveals a bimodal distribution (see Figure 1), suggesting the presence of two distinct programmer populations.

In order to determine the possible effects of programmer factors on the predictive power of complexity metrics we extracted two disjoint programmer populations. This process was greatly aided by Moher and Schneider's earlier research in which they constructed regression models of programmer performance based on a small number of background variables. For professionals, the single best predictor was found to be the number of years of programming experience. For students, the models included both experimental and aptitudinal factors. The best predictive model was composed of the number of computer science courses taken and computer science

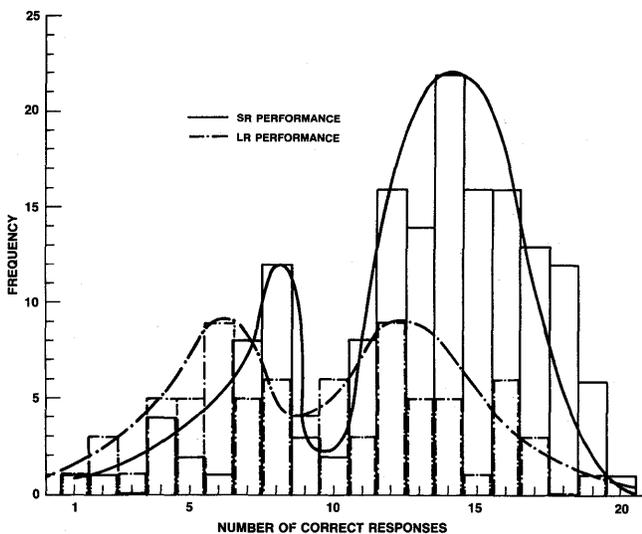


Figure 1—Overall task performance

TABLE II—Operational criteria for novice/expert subpopulations

CLASS	CRITERIA
NOVICE	≤ 4 computer science courses and < 3.00 gpa, or ≤ 2 years programming experience
EXPERT	≥ 7 computer science courses and ≥ 3.50 gpa, or ≥ 5 years programming experience

TABLE III—Novice/expert summary data for tasks SR and LR

	N	SR MEAN	LR MEAN	% DIFFERENCE
NOVICE	45	10.5	5.6	47
EXPERT	30	15.7	13.5	14

grade point average. These models (reported in detail in Moher and Schneider, 1981) explain about 40-55% of the performance variability, lending further credence to the conjecture that individual differences indeed have a significant effect on performance.

Using these models we operationally defined our two sub-populations as "novice" and "expert." The criteria for novice/expert classification is given in Table II.

The novice/expert performance distributions for programs SR and LR are shown in Figures 2 and 3. Table III summarizes the data for these distributions.

The difference between the two groups is quite evident. While both groups found program LR more difficult to understand, the magnitude of the performance decline was 3½ times greater for the novices than for the experts. This is hardly surprising, since increased programming experience almost always involves working on larger programs. The ability to intellectually manage large programs would certainly be indicative of programming expertise. Neither is it surprising that the expert group performed better on both program comprehension tasks. What is surprising is that the three program-oriented complexity measures grossly misjudged the magnitude of performance decline for both groups. The measures

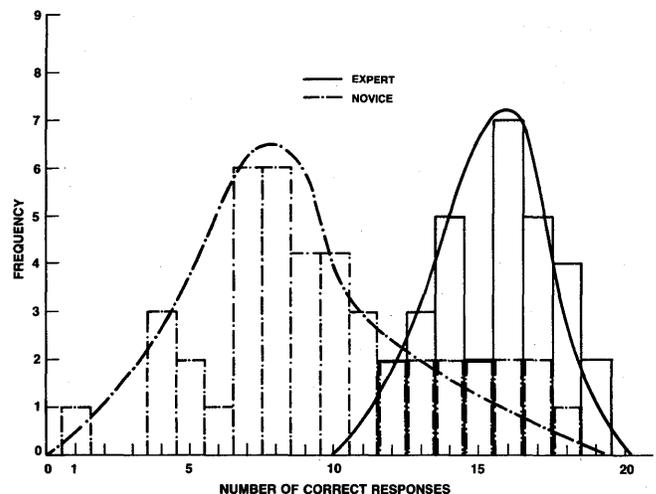


Figure 2—Novice/expert performance on Task SR

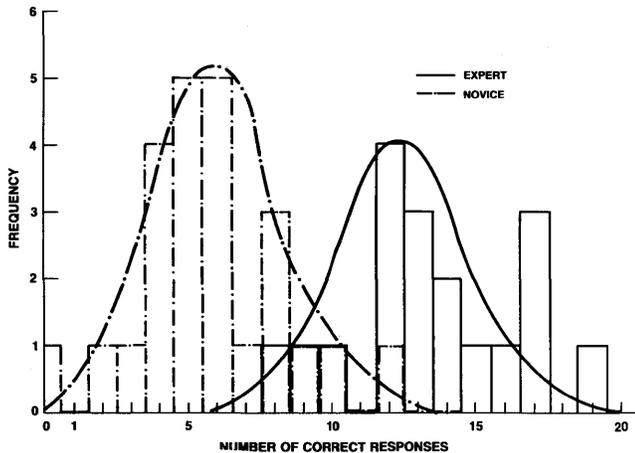


Figure 3—Novice/expert performance on Task LR

overestimated novice group performance and underestimated expert group performance by an approximate factor of two. This result clearly demonstrates that program complexity is not an inherent property of the program source text but a perceptual phenomenon that arises from the interaction between a program and a programmer. It is therefore futile to attempt to predict performance without considering the effect of salient programmer attributes. This point may be made more dramatic if we use the regression models to define a programmer population denoted "super-expert." We define a super-expert as a programmer with over seven years' programming experience or with at least 10 computer science courses and a computer science grade point average of B+ or better.* Using these criteria we extracted the performance data shown in Table IV.

Now we have a performance decline of only 6%. Our performance data show that while LR was a very complex program for a novice (47% decrease), and a slightly complex program for an expert (14% decrease), it was almost a trivially complex program for a super-expert (6% decrease). In contrast, the static, program-based complexity measures predict an equally complex program across all programmer populations.

Programming task factors

In addition to program and programmer factors we also conjectured that programming task factors differentially affect performance. By programming task we mean those distinct operations that programmers must perform on a program. Shneiderman²⁹ previously identified four basic programming tasks: construction, comprehension, debugging, and modification. He further stated that these tasks were not disjoint but, in fact, comprehension played an important role in the accomplishment of all other tasks. Other researchers, most notably Brooks,^{4,5} concur that comprehension is perhaps the most important programming task. In general, the task of program comprehension involves the extraction of semantic

*Our definition of a super-expert is hardly a contrived one. We surmise from our sample population that a significant portion of programmers in large data processing shops fit our criteria.

TABLE IV—Super-expert performance data

	N	SR MEAN	LR MEAN	% DIFFERENCE
SUPER-EXPERT	14	15.8	14.8	6

information from the syntactic representation of the program. Since a certain amount of information must be extracted to find a program bug, locate the correct place to insert a modification, or decide what parameters a given procedure requires, it is easy to see the importance of comprehension.

Analyzing the comprehension task more closely reveals that it is also comprised of subtasks corresponding to extraction of different types and amounts of information. Moher and Schneider proposed five such information classifications.

1. High-level semantics—determine the purpose of a high-level program unit, i.e., function, procedure, program. (We will abbreviate this as HI in succeeding discussions).
2. Low-level semantics—determine the purpose of an individual statement or low-level program unit, i.e., loop or conditional construct. (LOW)
3. Data structure—determine the form, structure, and contents of higher-level data structures used in the program. (DATA)
4. Program flow—determine the sequence of statement execution and/or conditions under which a specific statement or program unit will be executed. (FLOW)
5. Program modification—determine the global effects of a specific program change. (MOD)

We do not claim that these categories represent either a complete set of subtasks or a totally disjoint set of operations. However, we do feel that they represent different facets of comprehension and may require different cognitive operations. If these categories do capture, in some sense, different comprehension subtasks then performance on these tasks may be differentially affected by program and/or programmer factors.

The 20-question comprehension examination used to measure performance on both the SR and LR tasks was, in fact, composed of five separate parts. (Although this was transparent to the subjects.) The 5 parts were intended to measure comprehension within the 5 distinct categories listed above—namely HI, LOW, DATA, FLOW, and MOD—with 4 questions per section. (The questions were shuffled, however, to avoid any type of learning effect.) The comprehension performance data, now detailed by category, is shown in Table V for programs SR and LR.

Table V reveals some very interesting results. For small programs, such as SR, most programmers seem to handle a wide range of tasks with equal facility. The range on the five SR subtasks is 2.2—3.0 questions correct, out of a possible 4. The performance differential on the larger program, LR, across subtasks is much more marked. The ability to perform subtasks for extracting detailed information (LOW, FLOW, MOD) declines much more rapidly than that to perform subtasks for extracting more abstract information (HI, DATA).

TABLE V—Scores on LR and SR by comprehension category

	MEAN SCORE (out of 4)					OVERALL COMPREHENSION
	HI	LOW	FLOW	DATA	MOD	
SR	3.0	2.4	2.8	2.8	2.2	13.2
LR	2.7	1.8	1.0	2.5	1.4	9.4
% DIFFERENCE	10	25	64	11	36	29

The overall effect of program LR is also reflected in the performance range, 1.0–2.7, for these subtasks.

Perhaps more striking is the performance fall-off between identical categories on SR and LR. The overall decline for all subjects was 29%, as mentioned earlier. However, when separated out by task, this decline ranged from 10% to 64%, a factor of 6! The high-level tasks showed a moderate decline of 10% to 11%, in contrast to the steep decline of 25% to 64% for the lower level tasks.

This data again demonstrates the inadequacy of complexity measures based on program factors alone. Even though the original functionality between each complexity measure and performance was based on a comprehension task, we see that they only predict gross comprehension performance. The measures will either underestimate the complexity of carrying out detailed, low-level subtasks or will overestimate complexity for high-level subtasks. Given that all major programming activities (documentation, debugging, maintenance) will require us to carry out one or more of these subtasks, as well as others, it is apparent that static complexity measures cannot predict performance with any high degree of accuracy.

Program-programmer-task interactions

Table VI combines the data presented in Tables IV and V. It contains performance data grouped by both population and subtask.

This table best demonstrates the thesis that program complexity arises from an interaction between the program, the programmer and the programming task and therefore cannot be captured by a static analysis of the program alone. The experts, when dealing with program LR, unexpectedly

showed an increase in performance for the high-level subtasks of HI and DATA. While this anomalous increase may be attributable to experimental error (or to the fact that this is the type of operation most frequently done by professionals) the important fact is that performance did not decrease as predicted by the complexity measures. On the other hand, in at least one case, program flow, expert performance declined almost as much as that of the novices. As expected, the novices fared poorly on all aspects of comprehension when moving from program SR to program LR. Nonetheless, they demonstrated a wide range of degradation with a factor of 3½ between the best (HI) and worst (FLOW) subtasks. The complexity measures were equally abysmal in predicting large and small declines between programmer class and subtask performance, demonstrating their inability to predict performance for a given population and subtask that was not identical to the ones for which they were originally predictors.

While intuitively expecting performance variations between tasks and programmers, the variation range of –8% to +74% shown in Table VIb is probably quite a bit larger than would have been expected, and shows the futility of trying to encapsulate the concept of “complexity” within a single measure—namely the V(G), E, or Q figures of Table I.

CONCLUSION

The usefulness of a complexity measure ultimately depends on its ability to predict programmer performance. In order to reliably predict performance, such measures must be based on factors which determine performance. The implication of previous research has been that programmer performance can be accurately predicted from an examination of the source code

TABLE VIa—Performance summary by programmer class and programming subtask

	PROGRAM SR					PROGRAM LR				
	HI	LOW	FLOW	DATA	MOD	HI	LOW	FLOW	DATA	MOD
NOVICE	2.4	1.8	2.3	2.4	1.6	1.9	1.3	0.6	1.3	0.5
EXPERT	3.5	2.9	3.3	3.2	2.7	3.6	2.4	1.7	3.5	2.3

TABLE VIb—Performance degradation by programmer class and programming subtask

	PERCENT PERFORMANCE DECLINE (from SR to LR)					
	HI	LOW	FLOW	DATA	MOD	OVERALL
NOVICE	27	28	74	45	70	47
EXPERT	–3	17	51	–8	16	14
OVERALL	10	25	64	11	36	29

of a given program. Differences in performance, then, are directly attributable to differences in the program. During the past decade a host of evidence has been gathered to document the effects of program factors on programmer performance. The main controversy in the complexity literature today centers around which factor or factors best capture "complexity." We contend that, while program factors may be one of the determinants of performance, it is not the only one. In fact, the same studies which attempt to demonstrate the validity of program-based complexity measures also provide evidence of two other major performance determinants: the programmer and the programming task. To develop a truly useful complexity measure we must not only continue to study the effects of program factors on programmer performance but must also begin to identify the critical programmer and programming task factors which contribute to performance. Gorsline has developed a tentative taxonomy of program factors. Our future research will be directed toward developing a suitable taxonomy for the other factors involved and formulating software complexity measures as a function of program, programmer and programming task interrelationships.

REFERENCES

- Atwood, M.A. and Ramsey, H.R., "Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging," ARI Technical Report TR-78-A21, August 1978.
- Atwood, M., Turner, A., Ramsey, H.R., and Hooper, J., "An Exploratory Study of the Cognitive Structures Underlying the Comprehension of Software and Design Problems," U.S. Army Research Institute for the Behavioral and Social Sciences, Technical Report 392, July 1979.
- Boehm, B., "The High Cost of Software," *Practical Strategies for Developing Large Software Systems*, Addison-Wesley, 1975.
- Brooks, R., "Towards a Theory of Cognitive Processes in Computer Programming," *International Journal of Man-Machine Studies*, Vol. 9, 1977, pp. 737-752.
- Brooks, R., "Using a Behavioral Theory of Program Comprehension in Software Engineering," *Proceedings of the 3rd. IEEE Conference on Software Engineering*, 1978, pp. 196-200.
- Chapin, N., "A Measure of Software Complexity," *Proceedings of the 1979 National Computer Conference*, New York, 1979, pp. 995-1002.
- Curtis, W., Sheppard, S., Milliman, P.M., Borst, M.A., and Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 95-104.
- Dunsmore, H.E. and Gannon, J.D., "Analysis of the Effects of Programming Factors on Programming Effort," *The Journal of Systems and Software*, 1980, pp. 141-153.
- Gord, R.D., "Measuring Improvements in Program Clarity," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 79-90.
- Gould, J.D., "Some Psychological Evidence on How People Debug Computer Programs," Technical Report RC 4542, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1973.
- Gould, J.D.; and Drongowski, M., "An Exploratory Study of Computer Program Debugging," *Journal of Human Factors*, Vol. 16, No. 3, 1974, pp. 258-277.
- Halstead, M.H., *Elements of Software Science*, New York, Elsevier North-Holland, Inc., 1977.
- Hanson, W.J., "Measurement of Program Complexity by the Pair Cyclomatic Number, Operator Count," *SIGPLAN Notices*, Vol. 13, No. 3, March 1978, pp. 29-33.
- Love, T., "An Experimental Investigation of the Effect of Program Structure on Program Understanding," *SIGPLAN*, Vol. 12, March 1977, pp. 105-113.
- Love, T., and Fitzsimmons, A., "A Review and Evaluation of Software Science," *Computing Surveys*, Vol. 10, No. 1, March 1978, pp. 3-18.
- McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.
- McClure, C.L., "A Model for Program Complexity Analysis," *Proceedings of the 3rd. Conference on Software Engineering*, 1978, pp. 149-157.
- Moher, T. and Schneider, G.M., "A Methodology for Improving Experimentation in Software Engineering," *Fifth Intl. Symp. on Software Engineering*, San Diego, Calif., March 1981.
- Myers, G.J., "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Notices*, Vol. 12, No. 10, October 1977, pp. 62-64.
- Myers, G.J., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Communications of the ACM*, September 1978.
- Sackman, Erikson, and Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, January 1968.
- Schwartz, J., "Analyzing Large-scale System Development," *Software Engineering Concepts and Techniques, Proceedings of the 1968 NATO Conference*.
- Sheppard, S., Borst, R., and Curtis, W., "Predicting Programmers' Ability to Understand and Modify Software," *Proceedings of Symposium on Human Factors and Computer Science*, Washington, D.C., June 1978, pp. 115-135.
- Sheppard, S., Curtis, W., and Milliman, P.M., "Modern Coding Practices and Programmer Performance," *IEEE Computer*, December 1979, pp. 41-49.
- Sheppard, S., Curtis, W., and Milliman, P.M., "Experimental Evaluation of On-line Program Construction," GE Technical Report TR-79-388100-6, December 1979.
- Shneiderman, B., "Exploratory Experiments in Programmer Behavior," *International Journal of Man-Machine Studies*, Vol. 5, No. 2, 1976, pp. 123-143.
- Shneiderman, B., "Measuring Computer Program Quality and Comprehension," *International Journal of Man-Machine Studies*, Vol. 9, No. 3, 1977, pp. 465-478.
- Shneiderman, B., Mayer, R., McKay, D. and Heller, P., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *CACM*, Vol. 20, No. 6, June 1977, pp. 373-381.
- Shneiderman, B. and Mayer, R., "Syntactic-Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *International Journal of Computer and Information Sciences*, Vol. 8, 1979, pp. 219-238.
- Woodward, M.R., Hennell, M.A., and Hedly, D., "A Measure of Control Flow Complexity in Program Text," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, January 1979, pp. 45-50.
- Gorsline, G., and Fainter, R. "Program Complexity Measures." ACM/NBS 19th Annual Technical Symposium. Gaithersburg, Maryland, June 1980.
- Miller, L.G. "Programming by Nonprogrammers." *International Journal of Man-Machine Studies*, 6 (1974), pp. 237-260.

Quantitative measures of MIS quality assurance during hardware conversion

by JOHN W. CENTER

Medtronic Incorporated
Minneapolis, Minnesota

ABSTRACT

The management information systems (MIS) department of Medtronic converted their applications from running on the computer of one hardware vendor to another. A quality assurance (QA) program was instituted to monitor, validate, and assist in the conversion process. Several quantitative measures were developed to determine the status and progress of the conversion from the QA viewpoint. Objective measures included learning curve position, rejection delay, and costs. Subjective measures included problem solution impact, confidence of conversion personnel, and confidence of management. Analysis and evaluation of the measures indicated that the QA program was able to pay for itself.

INTRODUCTION

In January 1980 a formal quality assurance (QA) function was established in the management information systems (MIS) department at Medtronic. During the period April 1980 through April 1981 the business application systems were converted from running on an IBM 370/148 to a Univac 1100/60.

The QA function was just being organized when the commitment was made to convert. There were questions in the minds of most MIS personnel about what the QA function would be. It was decided that the conversion itself would be the first major project where an active role would be played by a QA program. Therefore, a QA program was instituted to monitor, validate, and assist the conversion process.

The scope of the conversion project was large. The original count included 61 systems, 329 subsystems, 825 programs (not counting sorts and utilities), and 750K lines of COBOL code. There were also many programs that were to be converted from assembly, MARK IV, and RPG to COBOL. To contain the financial impact to the department and the company, the project was to be completed during one fiscal year.

The QA program had to be able to demonstrate that it was time- and cost-effective. It was determined that certain measures would be appropriate to demonstrate this effectiveness. These measures would also feed information back to the conversion personnel and to management regarding status and progress from the QA viewpoint.

These measures included a set of objective ones and a set of subjective ones. The objective measures included learning curve position, rejection delay, and costs. The subjective measures included problem solution impact, confidence of conversion personnel, and confidence of management.

APPROACHES TO MEASURE DEFINITION

Measure Selection and Determination

It was obvious that a good approach to the determination of the status and the impact of the QA program on the hardware conversion project was necessary. A search for approaches that were directly applicable was fruitless. This empty search was not surprising. A major conversion from the computer of one hardware vendor to another is a rare event. The establishment of a QA function in MIS is a recent trend.

Approaches were found that were indirectly applicable. An article by Roberts¹ gave some insights if the conversion project is treated as a customer of a software development organization. The articles by Mendis² and Buckley³ gave some rough ideas for the measures of a general software QA program.

A search was initiated for approaches used in other specialties that could prove applicable. The Hetch article⁴ discussed benefits that could be derived by treating software development and operation like hardware development and assembly. The articles that appeared in Juran's "Bible" of manufacturing quality control^{5,6,7} showed promise of applicability for this particular situation.

A set of constraints and objectives was established to plant an "instinct" that would trigger or enhance consideration when a good approach was seen. A series of basic facts had to be remembered: (1) the conversion was a single-shot event; (2) the conversion had a limited time frame; (3) the schedule of the conversion was a major constraint; (4) the conversion personnel had little experience in dealing with QA concepts. These facts pointed to some characteristics the quantitative measures would have to demonstrate. The measures would have to be simple to calculate and simple to understand. The measures are an experiment, and the "formulas" need validation and may be modified during use. Measures and data that

were strictly conversion-oriented had to be recognized as possibly useless or meaningless when the conversion had been completed.

The use of QA measures and concepts is well established in manufacturing organizations and facilities. The attributes of a major conversion project were studied to determine which aspects or functions of the manufacturing environment were analogous. It was determined that the conversion was similar to or had characteristics of the following manufacturing areas (1) movement of manufacturing from one plant to another; (2) control and management of a job shop. In a similar manner, it was determined that the characteristics of the manufacturing QA program applicable to the conversion QA program were (1) material receiving inspections; (2) manufacturing line and final inspections. Therefore, a set of simple measures, based on these concepts, was established. The measures were used by the manager of the conversion project, the conversion team leaders, the MIS QA manager, and the management of MIS.

Objective Measures

Three major objective measures were developed to determine the status of the conversion from the QA viewpoint. These measures demonstrate both the positive and the detrimental impact of the QA program on the conversion.

One objective measure was the learning curve position. This measure indicated how well the conversion personnel knew or understood the operation of the systems in the new environment and understood any inspection criteria. The measure also indicated how well the QA function knew or understood the operation of the new environment and the sensitive constraints. This measure was used primarily as a tool to determine the education status of the personnel.

Another objective measure demonstrated the rejection delay of the QA program on the conversion project. This measure gave a base for comparing the detrimental impact of the QA program to any derived benefits.

An additional measure was the costs of the QA program. This measure gave an indication of the financial detrimental impact to the department and to the conversion project. The cost constraints were not as rigorous as the time or schedule constraints. The measure primarily compared actual to planned costs.

The data used in calculating the described objective measures were also used in generating other measures. These measures were of a custom or curiosity nature. They gave management and project leadership additional looks and views at the status and impact of conversion from the QA position.

Subjective Measures

Subjective measures attempt to measure unmeasurable values. Subjective measures appeared to be a good way to demonstrate the positive or incremental benefits provided by the QA program. The following three seemed to be the most quantifiable.

There was an attempt to measure the problem solution impact. By documenting any problems found and the solutions that were discovered, each problem would only be solved once. The value of the QA inspection could be measured with the quantity and benefit of solutions found.

During the conversion, there are always novel or unexpected situations. The conversion personnel would have to be able to use standards and their own initiative to execute proper procedures. There would have to be confidence on the part of conversion personnel in order to do this. An attempt was made to measure this confidence.

During an effort of this magnitude the strain on management is severe. The QA program was placed in a critical position within the conversion project. If the QA program could be executed smoothly, confidence of management would be demonstrated. Only a subjective measure could be used to determine the level.

OBJECTIVE MEASURES AND EVALUATION

Objective measures minimize the impact of users' bias and action upon the results or the calculations. There is subjectivity in the definition of the measure. However, the presentation of the data and the formula will mean that anyone will come to the same result. There is subjectivity in the way the measure is used in the decision or action process. However, consistent results can be displayed. Interpretation and decisions can then be reasonably made.

Learning Curve Position

The measure of learning curve position was used to determine education status of the conversion personnel. This included the education of the QA function. By monitoring the learning curve position it was possible to determine the need and the impact of education, both formal and informal.

Inspections were performed on each package of subsystems as it was presented to the QA function. The dimension of package sequence or quantity became a time dimension. The result of each inspection was a binary state of accepted or rejected. By combining a series of inspections into a group, a ratio of accept to inspected could be obtained. This ratio was used to represent the position of the conversion project on a learning curve. The size of the group had to be determined. The following criteria or constraints were considered: (1) the ratios would have to be easy to calculate; (2) the group size had to be small enough to produce data points early in the project; (3) the group size had to be big enough to absorb unusual perturbations; (4) there would be no changes in the group size after the first report; (5) there were to be about 300 packages. Analysis was done for a few test cases of different group sizes. A group size of 10 was settled upon. This size seemed to be practical and satisfied the constraints.

The results of the learning curve analysis are shown in Table I. The learning curve can be drawn by using data in the first two columns. The pattern and the rate of increase were typical of learning curves. The shape of the learning curve was used by the QA function to manage the education program. An

TABLE I—Acceptance learning curve

Group	Number Accept	Reject Reasons	Other Problems
1	3	10	10
2	4	10	10
3	6	11	16
4	9	1	5
5	9	1	7
6	10	0	6
7	7	5	5
8	9	1	8
9	6	7	1
10	6	5	1

arbitrary bound of 80% was set as the acceptable minimum of the learning curve position. Any value below this bound required an explanation. Inspections of new types of applications or techniques in the subsystems was a common explanation. This was used as a sign that there was additional, and probably specific, training needed. The results of Groups 7, 9, and 10 show the ratio dropping below the bound. In Group 7 the first subsystems that had database applications reached the point of inspection. The first online transaction programs reached inspection in Group 9. The first subsystems from a conversion team that was formed later in the project reached inspection with Group 10.

The rejection reasons were studied for each low group even before the results were completely known. Training and specific information were given to the personnel that needed it. This included those who had not yet run into the problems. The results of Group 8 show the improvement that was possible with immediate knowledge and corrective action.

The third column displays the total number of reasons for rejection for each group. It was possible to reject a single package of subsystems for more than one reason. The trend of the rejection reasons inversely correlated with the learning curve position. The fourth column displays the number of other significant problems encountered for each group. These problems had no standard on which to base a rejection. The

trend showed how some silly problems found early in the conversion project disappeared. The disappearance was a direct result of informing the conversion personnel of the causes and finding solutions. Sensitivity to the problems was raised in the minds of the conversion personnel. New standards were written to cover situations when the same problem would be found in the future.

Rejection Delay

Since the schedule of the conversion project was a major constraint, it was important to know the schedule impact and rejection delays of the QA program. A measure was developed to monitor the impact. The presentation of the data for this measure is made in Table II.

The concept of the group, as discussed in the learning curve position section, is used with this measure also. The number of calendar days' delay associated with each rejected package was recorded. The average delay for Group 1 showed very quick resolution, primarily due to rejections with easy solutions. With Groups 2 and 3 the delay increased as a result of conversion personnel digging themselves into shallow holes. Education and experience resulted in the delay settling down to about three days.

Data presented in Table III shows how the delays were concentrated in the range of less than a week. The longer delays were a result of more significant problems. The resolution required much more work and associated testing.

The delays for rejection were a detriment to the schedule. However, the conversion management received a large benefit simply by publishing the value of the delay. The management of the conversion project could begin to assume a one-day rejection delay for each submitted package and three days' delay for each rejection. These values were plugged into the scheduling mechanisms. The measure also became a base for comparing any benefits that would be obtained from the inspections and the QA program.

TABLE II—Rejection delay by group

Group	Number Reject	Delay Days	Days/Rej.
1	7	14	2.0
2	6	39	6.5
3	4	21	5.2
4	1	0	0.0
5	1	3	3.0
6	0	0	-
7	3	2	0.7
8	1	1	1.0
9	4	14	3.5
10	4	8	2.0
Total	31	102	3.3

TABLE III—Rejection delay by days

Days	Number
0	7
1	7
2	3
3	3
4	3
5	3
6	0
7	2
8	0
9	1
10	1
11	1
12	0
13	0
14	1
15+	0

TABLE IV—Cost breakdown

Classification	Plan(%)	Actual(%)
Conversion inspection	20	17.6
Conversion support	20	20.1
QA projects	20	22.8
Direct overhead	20	21.7
Indirect overhead	20	16.8

Costs

Instituting and running a QA program has costs that must be properly allocated and managed. The data presented in Table IV display the planned and actual cost experience.

The data collected for the first half of the conversion indicated that the costs were just about on plan. It appears that during the period of hardware conversion approximately 40% to 50% of the time and cost of the QA function should be allocated directly to the conversion. This allocation does not consider the indirect overhead of vacation, sick time, etc. The costs associated with the conversion included (1) cost of the direct inspections of each package of converted subsystems; (2) cost of support not directly associated with a given subsystem or package, including writing standards, attending status and review meetings, measure evaluation, and general conversion discussions. An hour was allocated to making the actual inspection of each conversion package (time recording is confirming this estimate as a good one). At the time of writing this paper, some additional steps or requirements were added to the inspection criteria. This should increase the inspection time for each package by about one quarter hour. The increase will probably bring the actual inspection cost time right into line with the planned one.

SUBJECTIVE MEASURES AND EVALUATION

There are cases where the user of the data places a value on a situation, event, or position based on his or her own experience. There are also cases where attempts are made to measure the value of events or situations that can be avoided. These cases give rise to the subjective measures.

Problem Solution Impact

As part of the QA program, a method was developed to distribute descriptions and solutions of discovered problems. The distribution was in the form of a set of "memos" or bulletins, a table of contents, and a key word index to the set.

An informal survey found that each problem cost three to five days of calendar time to resolve. There were three conversion project teams and a staff of technical experts and specialists. Assuming that each group would eventually run into the same problem, the distribution of the bulletins would avoid nine to 15 days of delay. Assuming that some of the delay was already anticipated in the schedules, a conservative

delay-avoidance estimate of perhaps five days per bulletin was obtained.

At the time of writing this paper 54 bulletins had been issued. Using all the listed assumptions, a simple calculation indicates that the QA program has avoided 270 days of schedule delay so far. This figure will never show up on a schedule or status report. Any presentation of a cost or time avoidance is subjective. The objective measure of rejection delay was 102 days. A net saving of 168 days can be calculated if the solution impact and rejection delays are combined. The impact of the problem solutions produced a net positive or incremental benefit.

Confidence of Conversion Personnel

One of the most unpredictable tasks of the conversion was the test and debug phase. The application programs were written in the various styles used by many departed programmers from the last three to 10 years. When the testing of the converted programs began, the personnel expected that a standard test plan would be applicable in every case. It was not.

The QA function issued some formal test philosophy documents. There were informal sessions held with the testors and the project team leaders. A few months into the conversion, the testing personnel began to approach the QA function with their own ideas on how to test a special case. This was an early indication that the conversion personnel were treating the QA program as a beneficial control function.

By the midpoint of the conversion the testers used the general rules and philosophy to generate their own special test plans. The plan and assumptions were included in the package submitted for inspection. Some of the testing procedures were very ingenious and usually more than adequate. No objective value can be placed on these actions. Only confidence on the part of conversion personnel would allow them to take a chance on an unapproved, though good, special test plan. The confidence eliminated the need to bend the unbendable special case to fit the standard test plan and the need for a lengthy approval cycle for special test plans. The confidence led to a significant avoidance of delay and still produced an adequately tested conversion. Only subjective evaluation can put a precise value on this confidence and delay avoidance.

Confidence of Management

The management of MIS and the conversion project had never lived in an environment with a QA function. The inspection of the converted packages was a critical gate in the process. The initial reaction to their presence was a wait-and-see attitude. In the early stages of the conversion project, the QA function participated with the project team leaders at the meetings for status and review. The QA function also participated in the session with the task force or group of experts and specialists. There was a distinct impression that this was done to satisfy an obligation.

About four months into the conversion some convertors went to the conversion manager. They were worried about the

reaction of the QA function to an inspection of an especially strange case. The conversion manager assured them that any reaction would be reasonable and informative. The system had been accepted with no unusual fanfare for about three weeks before the concern of the convertors came to the attention of the QA function.

About five months into the conversion, the director of MIS forwarded to the QA function some comments he had received from the technical and marketing management of the hardware vendor. The vendor had felt the schedule was ambitious and the conversion was in trouble in the early stages. It was now felt that the schedule would probably be met. Much of the credit was due to the avoidance of delays with an active QA program.

At the midpoint of the conversion a problem had developed: many people were making subordinate inspections and checks of a QA nature. The manager of the conversion project requested that all these checks be consolidated. Only the QA function was to perform the inspection. There was complete latitude given in the structuring of the extended inspection criteria. The inspections were expanded and accepted with the support of the managers and team leaders.

These examples demonstrated that the management had confidence in the QA program of the conversion project. Assume that points were given each time a manager gave a positive comment or requested assistance; also assume that points were taken away each time a manager expressed frustration or doubt. Then a precise measure of management confidence could be calculated. The net value of points given the QA program would be subjective but positive.

SUMMARY

An attempt was made to generate some quantitative measures of the status of the conversion project from the QA viewpoint. The attempt was successful. The measures met the requirements or objectives; they were simple to calculate—there was nothing more complicated to calculate than an average or a

ratio; and they were simple to understand—even management was able to understand and use the measures to monitor the status of the conversion. Though there is a possibility that specific data will be useless to the MIS department after the conversion, the concept was proved. It appears that some form of these measures will continue to be used for the enhancement and maintenance projects. It is likely the concepts will serve also as a base for quantifiable measures of the QA status of new development projects.

These quantitative measures demonstrated that it is possible to show that the QA program is paying for itself. The exact value of the payback is full of subjective measures and assumptions, but it is quantifiable.

The conversion is still in progress at the time of writing this paper. These measures are still being used, and their use has stabilized. It is unlikely that the definitions of those described in the paper will change. However, it is possible that other measures may be developed to help evaluate new situations that may arise.

The development and use of these measures has been a very interesting and rewarding experience. It is exciting to discover that it is possible to quantify, calculate, and use measures to describe what was previously only suspected.

REFERENCES

1. Roberts, T.J. "Maintaining Quality after the Software is Released." *ASQC Technical Conference Transactions*, 31 (1977), pp. 157-166.
2. Mendis, K.S. "A Software Quality Assurance Program for the 80s." *ASQC Technical Conference Transactions*, 34 (1980), pp. 379-388.
3. Buckley, F. "A Standard for Software Quality Assurance Plans." *Computer* 12 (1979), pp. 43-50.
4. Hecht, H. "Can Software Benefit from Hardware Experience?" *1975 Annual Reliability and Maintainability Symposium*, pp. 480-484.
5. Pierce, R.J. "Quality Planning" (Section 6). In J.M. Juran (ed.), *Quality Control Handbook* (3rd ed.). New York: McGraw-Hill, 1974.
6. Ekvall, D.N. "Manufacturing Planning" (Section 9). In J.M. Juran (ed.), *Quality Control Handbook* (3rd ed.). New York: McGraw-Hill, 1974.
7. Seder, L.A. "Job Shop Quality" (Section 45). In J.M. Juran (ed.), *Quality Control Handbook* (3rd ed.). New York: McGraw-Hill, 1974.

Taking the measure of program complexity

by JEAN COCHRANE ZOLNOWSKI*

Bell Laboratories
Holmdel, New Jersey

and

DICK B. SIMMONS

Texas A&M University
College Station, Texas

ABSTRACT

Program complexity is a topic often discussed in the literature. Research is ongoing in verifying existing complexity measures. There is also a continuing effort to produce and validate new approaches to a complexity measure which incorporate ideas from a variety of areas.

Too often, however, approaches to complexity measurement center on a particular aspect of a program, e.g., structures, without incorporating other relevant program characteristics. The question to be answered, then, is, What aspects of a program contribute to its complexity?

This paper presents a first step in answering this question. Preliminary results are presented from a Delphi Survey on program complexity. The survey was sent to a cross-section of programmers, managers and software experts. Respondents rated a large number of characteristics as to their effect on program complexity. The paper summarizes the results and includes preliminary analyses.

INTRODUCTION

Software Engineering literature contains a plethora of references¹⁻²⁴ on program complexity, the difficulty in understanding the program as it stands. Discussions on complexity range from a brief mention of the necessity of its inclusion in measuring other life cycle factors such as programmer productivity to proposals for specific complexity metrics.

Validation of these proposals requires an understanding of how complexity is affected by factors and instability in the design cycle and, in turn, how complexity affects/is affected by factors and instability in the testing and maintenance cycles.

However, it is often difficult to collect accurate and reliable data that could relate factors relevant to the life cycle of a large software system to specific program complexity charac-

teristics. The sheer number of potential program complexity characteristics implies that methods must be employed in order to eliminate some of these possible complexity factors.

One method would be to hold an open forum on program complexity and encourage those with expertise and/or experience to reflect on factors affecting a program's complexity. A means for accomplishing such a forum is a Delphi Survey.

The Delphi Survey is a method for structuring group communication so that a group of individuals can effectively deal with a problem.²⁵ The Delphi provides a method for querying expert opinions on a particular topic in an attempt to arrive at a consensus or simply to ascertain where dissension is centered.

There are three essential features to the Delphi: anonymous response; iteration and controlled feedback; and statistical group response. The features are designed to minimize the biasing effects of dominant individuals, of irrelevant communications, and of group pressure toward conformity.

From December 1979 through the summer of 1980, a Delphi Survey on program complexity was conducted. The purpose of this survey was to determine if there existed a consensus as to what variables have the greatest impact on computer program complexity. The survey was sent to a group of people who were actively concerned with software. This included both authors in the software engineering area and managers and programmer/analysts involved daily in software development and maintenance.

Response to the survey was generally excellent and preliminary results from the survey have been analyzed and sent to the respondents. The sections to follow present specifics as to how the survey was conducted, analyses of preliminary results, and a description of on-going analyses.

METHOD

Two rounds of the survey were sent out. The first survey was sent to approximately 100 people, of whom 62 replied. The

*Work completed by author at Texas A&M University

TABLE I—Profile of participants

	Managers	Programmers	Experts	Total
Number of participants	16	18	21	46
Percent rating themselves well qualified	44%	72%	75%	63%
Percent rating themselves moderately qualified	56%	28%	25%	37%
Average years "involved" with software	16.4	10.1	16.2	13.9
Average years of actual programming experience	9.8	8.8	12.4	10.1
Average years since direct participant in programming project	3.4	.2	.8	1.5
Languages where a majority of the total all have a working knowledge	FORTRAN, ASSEMBLER	FORTRAN, COBOL	FORTRAN	FORTRAN

feedback survey was sent to these 62 people, and 46 of them responded. The essential process was as follows.

A questionnaire was designed and sent to the respondent group. This questionnaire was divided into three parts: profile of the respondent, program complexity in general (non-program characteristics such as amount of documentation, modern programming practices used, etc.) and program complexity in detail (specific program characteristics).

The Profile of Respondents' survey is summarized in Table I for the 46 people who participated in the total survey.

Respondents were asked to rate, on a scale from -7 to $+7$, the relevance of each factor to a program's complexity. A minus sign indicated that complexity would be increased by the factor, while a positive sign indicated that complexity would be reduced by the factor. Zero indicated no effect on complexity. Respondents were encouraged to write on the survey any comments they felt were applicable.

After the first survey was returned, the results were summarized and a feedback questionnaire designed. The feedback questionnaire consisted of two parts: (1) factors that affect complexity (in general and in detail) and (2) factors that are a possible measure (indicator) of program complexity. The latter category was suggested by the comments of the respondents in the first survey.

There were two criteria for not resubmitting questions in the feedback questionnaire: (1) a *consensus* was reached, or (2) a *consensus* was not reached but "*% answers significant*" was less than 25%. (Table II contains definitions for these

terms.) The remaining questions posed in the first survey were repeated in the feedback survey along with several questions suggested by respondents. Information was provided on the median, the quartiles Q1 and Q3, and summarized respondents' comments.

Respondents were asked to reconsider their previous rating and change it if they desired. Whenever an answer (changed or otherwise) fell outside the middle 50% range (Q1→Q3), respondents were asked to briefly state the reasons why their answer was different from that of the majority of respondents. For the feedback survey, the respondents were specifically told to use the sign \pm if they felt that a factor could have both a positive and negative effect on program complexity and to respond to the factor as it increases in magnitude.

RESULTS

The purpose of the preliminary analyses was to summarize the results via a categorization scheme that would clearly delineate the complexity factors in a relevant frame of reference. Table II contains the definitions for the terms used below.

The two major categories chosen were consensus and non-consensus variables. Within each of these, a division was made into significant, sit-on-the-fence, and insignificant variables. Variables were uniquely categorized into one of these six slots. In addition, preliminary work was done to determine possible categories for controversial variables.

TABLE II—Definitions of terms

—	No results for the round (for example, dashes under round 2 data indicate that no feedback round was required).
Range of answers	Each Delphi Survey asked participants to respond on a scale of -7 to $+7$ as to the effect an increase in magnitude of the complexity characteristic would have on complexity
Q1	The 25% quartile—25% of the responses were less than or equal to this number
Q3	The 75% quartile—75% of the responses were less than or equal to this number
Percent answers significant	The percentage of the responses which fell into the significance range—response $\geq +4$ or response ≤ -4
Consensus variable	Variable with an interquartile range less than 3—the range from the 25% quartile (Q1) to the 75% quartile (Q3) is less than 3
Nonconsensus variable	Variable with an interquartile range greater than or equal to 3
Significant variable	Variable where median $\geq +4$ or $-4 \geq$ median
Sit-on-the-fence variable	Variable where $-3 \leq$ median $\leq +3$ and " <i>percent answers significant</i> " > 25
Insignificant variable	Variable where $-3 \leq$ Median $\leq +3$ and percent answers significant < 25

TABLE III—Significant consensus variables

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Number of intersections between loops	44	-7	-5	-4	-6	-5	-4	95.5
Number of knots	46	-6	-4	-3	-5	-5	-4	78.3
Number of possible execution paths	46	-6	-5	-3	-6	-5	-4	78.3
Number of undeclared variables	45	-6	-4	-2	-5	-4	-3	64.4
Number of changes made during operation/maintenance	42	—	—	—	-5	-4	-3	64.3
Cyclomatic number	40	-5	-4	-2	-5	-4	-3	62.5
Type of program (real time)	61	-5	-4	-3	—	—	—	62.3
Number of exits out of loop	45	-6	-4	-3	-5	-4	-3	62.2
Number of conditional statements	62	-5	-4	-3	—	—	—	61.3
Number of breaks in flow	45	-5	-4	-2	-5	-4	-3	60.0
Depth of if nesting	61	-5	-4	-3	—	—	—	57.4

The sections to follow present definitions, results (Tables III through XII) and a brief analysis ("thoughts") of consensus, nonconsensus, and controversial variables.

Consensus Variables

Definition of consensus variables

Consensus variables were delineated into these categories:

- Significant** Those variables for which: median $\geq +4$ or median ≤ -4 and interquartile range < 3.0
- Sit-on-the-fence** Those variables for which: $-3 \leq \text{median} \leq +3$, interquartile range < 3.0 , and % answers significant ≥ 25.0

Insignificant

ian $\leq +3$, interquartile range < 3.0 , and % answers significant ≥ 25.0

Those variables for which: $-3 \leq \text{median} \leq +3$, interquartile range < 3.0 , and % answers significant < 25.0

Tables III, IV, and V contain these data.

Preliminary thoughts on consensus variables

- No significant consensus variables were found that positively affect complexity.
- Eight out of 11 significant consensus variables were related to the structure and control aspects of a program.

TABLE IV—Sit-on-the-fence consensus variables

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Number of control flow statements	45	-5	-3	-2	-5	-3	-3	48.9
Total number of variables used globally	29	—	—	—	-5	-3	-3	48.3
Number of entry points	46	-5	-3	-2	-5	3	-3	45.7
Number of returns from subprograms	46	-4	-3	-2	-4	-3	-2	45.7
Total number of variables used	45	-5	-3	-2	-4	-3	-2	44.4
Number of modules a linkage variable passed to from a single module reference (depth of nesting)	46	-4	-3	-2	-4	-3	-2	43.5
Total number of linkage variables	46	-5	-3	-2	-4	-3	-2	41.3
Number of system program interfaces	45	-5	-3	-1	-4	-3	-2	37.8
Number of modules in total a linkage variable passed to (breadth of nesting)	46	-4	-3	-2	-4	-3	-3	36.9
Number of paths within loop	44	-4	-3	-2	-4	-3	-2	36.4
Breadth of nesting within a loop	44	-5	-3	-2	-4	-3	-2	34.1
Number of application program interfaces	45	-5	-3	-2	-4	-3	-2	33.3
Number of changes made during debugging/testing	42	—	—	—	-4	-3	-2	33.3
Number of operands	46	-4	-3	-1	-4	-3	-2	32.6
Nesting depth of module references	46	-4	-3	-2	-4	-3	-2	30.4
Number of variables referenced in condition statements	45	-4	-2	-1	-4	-2	-2	31.1
How variable used (array, conditional name, etc.)	61	-3	-2	-1	—	—	—	27.9
Experience of the programmer	33	—	—	—	2	3	4	33.3

TABLE V—Insignificant consensus variables

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Number of operators	46	-4	-2	-1	-3	-2	-1	23.9
Number of forward branches	62	-3	-2	-1	—	—	—	22.6
How parameters passed (expression)	61	-3	-2	-1	—	—	—	19.7
Span of a branch (number of statements bypassed)	61	-3	-2	-1	—	—	—	19.7
Nesting breadth of module references	57	-3	-2	-2	—	—	—	19.3
FORTRAN (when arbitrarily chosen)	34	—	—	—	-3	-2	-1	17.7
Number of errors found during debugging/testing	43	—	—	—	-3	-2	-1	16.3
Language program(s) written in (when arbitrarily chosen)	37	-4	-2	2	-3	-2	-2	16.0
Number of changes made during coding	42	—	—	—	-3	-2	-1	9.5
Number of unique module references (= call)	45	-3	-2	0	-3	-2	-1	4.4
Total number of variables used locally	29	—	—	—	-3	-2	-1	0.0
Number of computational statements	61	-2	-1	0	—	—	—	18.0
Type of program (calculation)	61	-2	-1	0	—	—	—	14.8
Span of a loop (number of statements within path of loop)	61	-2	-1	-1	—	—	—	14.8
Type of subprogram	56	-2	-1	0	—	—	—	14.3
Type of statement variable referenced in:								
I/O statement	61	-2	-1	0	—	—	—	9.8
Type of variable (integer, real, etc.)	61	-2	-1	0	—	—	—	8.2
Number of non-executable statements	62	-2	0	0	—	—	—	14.5
Type of statement variable referenced in: sequential	61	-1	0	0	—	—	—	11.5
Costs to run	42	—	—	—	-1	0	0	9.5
PL1 (when arbitrarily chosen)	26	—	—	—	-1	1	0	7.7
ALGOL (when suited to application type)	28	-1	1	2	0	1	2	7.1
PL1 (when suited to application type)	35	-1	1	2	0	1	2	5.7
COBOL (when suited to application type)	45	-1	1	3	0	1	2	4.0
PASCAL (when suited to application type)	28	0	2	3	1	2	3	7.1

- There was a general lack of significance attributed to languages.
- The later the change made in the life cycle, the stronger its relation to complexity is.
- Importance was attached to the *declaration* of all variables. A respondent commented, "...in general, to improve a program all variables should be declared and

initialized in some fashion. ... This one step could provide the greatest help in reducing programming complexity and cutting debug/test time. ..."

- The number of variables close to significance, that is, "sit-on-the-fence," was large. Twelve out of 18 of these variables were in the data reference and interaction categories.

TABLE VI—Significant variables not reaching a consensus

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Number of entry points into a loop (number of ways to enter a loop)	45	-6	-4	-3	-6	-5	-3	73.3
Assembler (when arbitrarily chosen)	36	—	—	—	-6	-5	-3	69.4
Problem to be solved	41	-7	-5	-3	-7	-4	-3	63.4
Number of non-sequential instructions within nested if	46	-5	-4	-2	-5	-4	-2	60.9
Number of "infrequently used" instructions (e.g., ALTER, ASSIGNED GO TO, etc.)	46	-6	-4	-2	-6	-4	-2	56.5
Number of exits from decision statement	46	-5	-4	-2	-5	-4	-2	56.5
Number of instructions	46	-5	-3	-2	-5	-4	-2	52.2
Number of statements	45	-5	-3	-2	-5	-4	-2	51.0
Number of comments	46	2	3	5	2	4	5	52.2
Modern programming practices used for coding/debugging	45	3	5	7	4	5	7	80.0
Design techniques utilized	44	0	5	7	4	5	7	77.3

TABLE VII—Sit-on-the-fence variables not consensus

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Assembler (when suited to application type)	45	-6	-3	-2	-5	-3	-2	48.9
Number of GO TO statements	44	-6	-3	-2	-5	-3	-2	47.7
Depth of nesting within a loop	44	-5	-4	-2	-5	-3	-2	47.7
Number of errors found during operation/maintenance	43	—	—	—	-5	-3	-1	46.5
Number of loops created via GO TO	46	-6	-3	-2	-5	-3	-2	45.7
Number of backward branches	46	-5	-3	-2	-5	-3	-2	43.5
Type of linkage variable: implicit (e.g., (COMMON))	45	-5	-3	-1	-5	-3	-2	42.2
Number of IF statements	46	-5	-3	-1	-5	-3	-2	39.1
Number of manhours to correct errors	42	—	—	—	-4	-3	-1	38.1
Amount of relevant documentation produced	42	—	—	—	-3	-2	2	33.3
Costs to develop	44	—	—	—	-4	-2	-1	29.6
Number of I/O statements	46	-4	-2	0	-4	-2	-1	26.1
Number of modules i.e., subprograms, sections, etc.	42	-4	-1	3	-4	-1	2	33.3
Language program(s) written in when language suited to problem	35	—	—	—	1	2	4	28.6

Nonconsensus Variables

Definitions of nonconsensus variables

Nonconsensus variables were divided into three categories:

Significant	Those variables for which: median $\geq +4$ or median ≤ -4 and interquartile range ≥ 3.0
Sit-on-the-Fence	Those variables for which: $-3 \leq \text{median} \leq +3$, % answers significant ≥ 25 , and interquartile range ≥ 3.0
Insignificant	Those variables for which: $-3 \leq \text{median} \leq +3$, % answers significant < 25 , and interquartile range ≥ 3.0

Tables VI, VII, and VIII contain these data.

Preliminary thoughts on nonconsensus variables

- Most respondents do not care for the use of assembler language.
- For the significant and sit-on-the-fence variables, a certain percentage of the respondents held fast to the lower end through two rounds of the survey.
- Fewer than 50% thought the # GO TO statements deserved significance.
- The modern programming practices and design techniques were deemed positive in their relationship to complexity—however, there was disagreement as to how positive.
- Size has significance but there was no consensus on it.
- Languages are judged not to have a strong effect.
- In general, there was lots of disagreement.

Controversial Variables

Definitions of controversial variables

Preliminary choices of categories for controversial variables were as follows:

- Variables with an interquartile range > 4
- Variables with a negative to positive interquartile range where either one of the quartiles is significant and/or the % answers significant is > 25
- Variables with either one or both of the quartiles significant but no consensus
- Variables where less than 75% of the respondents responded

The variables do overlap. Tables IX, X, XI, and XII contain these data.

Preliminary thoughts on controversial variables

There was no agreement on how the number of modules affected complexity. This variable also had an interquartile range spanning negative to positive. This “number of modules controversy” can best be summed up via respondents’ comments:

- “number is not point, so much as structure of connection...”
- “implying that programs can be made less complex by breaking them into modules...”
- “using available modules helps ... maybe ...”
- “modularity helps to decrease complexity but, if carried too far, becomes confusing ...”

TABLE VIII—Insignificant variables not consensus

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
COBOL (when arbitrarily chosen)	35	—	—	—	-3	-2	0	20.0
Number of simple predicates in condition	62	-3	-2	0	—	—	—	19.6
Distance between references to a variable	61	-3	-2	0	—	—	—	18.0
Number of references to a variable	60	-3	-2	0	—	—	—	16.7
Total span of reference of variable within program	45	-3	-2	0	-3	-2	0	15.6
How parameters passed: return label	61	-3	-2	0	—	—	—	14.8
How parameters passed: variable name	61	-1	0	2	—	—	—	16.4
How parameters passed: constant	60	-1	0	2	—	—	—	10.0
Total number of module references (= call)	61	-3	-1	0	—	—	—	24.6
Number of statements between labels	62	-3	-1	0	—	—	—	21.0
Number of errors found during development	44	—	—	—	-3	-1	0	20.5
Type of linkage variable: explicit (via parameters)	60	-2	-1	1	—	—	—	20.0
Type of program: data manipulation	61	-3	-1	0	—	—	—	14.8
Number of loops defined by language (e.g., DO, PERFORM, etc.)	45	-2	-1	2	-2	-1	1	13.3
Number of changes made during design	42	—	—	—	-3	-1	0	11.9
ALGOL (when arbitrarily chosen)	22	—	—	—	-2	-1	1	4.6
Number of statements between breaks in flow	46	-3	-1	0	-3	-1	0	4.3
FORTTRAN (when suited to application type)	42	-2	0	2	-2	0	2	11.9
Number of labels (statement numbers, paragraph names, etc.)	44	-2	0	2	-2	0	1	11.4
PASCAL (when arbitrarily chosen)	22	—	—	—	-1	0	2	9.0
RPG (when suited to application type)	27	-2	0	2	-2	0	1	0.0
Suitability of the match of the hardware system to application	37	—	—	—	0	2	3	24.0

While the number of *changes* made during operation/maintenance was a significant consensus variable, the number of *errors* found during operation/maintenance raised a controversy.

The amount of relevant documentation, while not significant, did span a negative to positive range, and the respondents' comments reflected this—

- “works in both directions...appropriate degree helps clarify...too much confuses issue...”

- “in general, more documentation → bigger system → more complexity...”
- “surely we have all seen complex programs with no documentation...”
- “very weak pattern in my experience; but those individuals who produce good documentation do tend to produce less complex programs...”

Most questions received an adequate number of responses with the exception of a few new questions in the feedback

TABLE IX—Controversial variables with interquartile range greater than 4

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Problem to be solved	41	-7	-5	-3	-7	-4	-3	63.4
Number of “infrequently used” instructions (e.g., ALTER, ASSIGNED GO TO, etc.)	46	-6	-4	-2	-6	-4	-2	56.5
Number of errors found during operation/maintenance	43	—	—	—	-5	-3	-1	46.5
Amount of relevant documentation produced	42	—	—	—	-3	-2	2	33.3
Number of modules, i.e., subprograms, sections, etc.	42	-4	-1	3	-4	-1	2	33.3
FORTTRAN (when suited to application type)	42	-2	0	2	-2	0	2	11.9

TABLE X—Controversial variables with a significant quartile and with a negative to positive interquartile range

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Amount of relevant documentation produced	42	—	—	—	-3	-2	2	33.3
Number of modules i.e., subprograms, sections, etc.	42	-4	-1	3	-4	-1	2	33.3

round and questions about languages the respondent was not familiar with.

SUMMARY

The survey was long and was ambiguous in places. The respondents not only persevered through it but also provided a large number of comments. Comments written by the respondents raised "questions for discussion" among the participants and also provided many unique and interesting viewpoints on program complexity. Several of the respondents stated that exposure to these diverse opinions was in itself a valuable learning experience.

The preliminary analyses are interesting not simply for the

predictable significance of structure and control flow variables. Both the controversial and sit-on-the-fence categories provide topics for further discussion and analyses.

An example of this is the problem respondents had with how the number of modules affects complexity. Modularization was not the issue so much as the degree of modularity and the structure of the modularity. Modularity, then, is a technique that is considered a plus. However, the degree of "goodness" of this technique and its effect on complexity was undecidable for the respondent group.

The respondents' comments on several of the questions reinforced the dilemma that current metrics are in; a judgment is made that a particular factor affects complexity, but it is very difficult to ascertain the degree of the 'goodness' or 'badness' of its effect.

TABLE XI—Controversial variables with no consensus but having significance at one of the quartiles

Program characteristic	Number of responses	Q1	Round 1 median	Q3	Q1	Round 2 median	Q3	Percent answers significant
Number of entry points into a loop (number of ways to enter a loop)	45	-6	-4	-3	-6	-5	-3	73.3
ASSEMBLER (when arbitrarily chosen)	36	—	—	—	-6	-5	-3	69.4
Problem to be solved	41	-7	-5	-3	-7	-4	-3	63.4
Number of non-sequential instructions within nested IF	46	-5	-4	-2	-5	-4	-2	60.9
Number of "infrequently used" instructions (e.g., ALTER, ASSIGNED GO TO, etc.)	46	-6	-4	-2	-6	-4	-2	56.5
Number of exits from decision statement	46	-5	-4	-2	-5	-4	-2	56.5
Number of instructions	46	-5	-3	-2	-5	-4	-2	52.2
Number of statements	45	-5	-3	-2	-5	-4	-2	51.0
ASSEMBLER (when suited to application type)	45	-6	-3	-2	-5	-3	-2	48.9
Number of GO TO statements	44	-6	-3	-2	-5	-3	-2	47.7
Depth of nesting within a loop	44	-5	-4	-2	-5	-3	-2	47.7
Number of errors found during operation/maintenance	43	—	—	—	-5	-3	-1	46.5
Number of loops created via GO TO	46	-6	-3	-2	-5	-3	-2	45.7
Number of backward branches	46	-5	-3	-2	-5	-3	-2	43.5
Type of linkage variable: Implicit (e.g., (COMMON))	45	-5	-3	-1	-5	-3	-2	42.2
Number of IF statements	46	-5	-3	-1	-5	-3	-2	39.1
Number of manhours to correct errors	42	—	—	—	-4	-3	-1	38.1
Costs to develop	44	—	—	—	-4	-2	-1	29.6
Number of I/O statements	46	-4	-2	0	-4	-2	-1	26.1
Number of modules, i.e., subprograms, sections, etc.	42	-4	-1	3	-4	-1	2	33.3
Language program(s) written in when language suited to problem	35	—	—	—	1	2	4	28.6
Number of comments	46	2	3	5	2	4	5	52.2

TABLE XII—Controversial variables—less than 75% response

Program characteristic	Number of responses	Round 1			Round 2			Percent answers significant
		Q1	median	Q3	Q1	median	Q3	
Total number of variables used globally	29	—	—	—	-5	-3	-3	48.3
FORTRAN (when arbitrarily chosen)	34	—	—	—	-3	-2	-1	17.7
Total number of variables used locally	29	—	—	—	-3	-2	-1	0.0
ALGOL (when arbitrarily chosen)	22	—	—	—	-2	-1	1	4.6
PASCAL (when arbitrarily chosen)	22	—	—	—	-1	0	2	9.0
RPG (when suited to application type)	27	-2	0	2	-2	0	1	0.0
PL1 (when arbitrarily chosen)	26	—	—	—	-1	1	0	7.7
ALGOL (when suited to application type)	28	-1	1	2	0	1	2	7.1
PASCAL (when suited to application type)	28	0	2	3	1	2	3	7.1
Experience of the programmer	33	—	—	—	2	3	4	33.3

Respondents' commentProblem

"...if the name was meaningful..."

How do you judge meaningful?

"...too many interfaces can be confusing..."

What's too many?

"... -5 for too deep, +5 for right depth..."

What's "too deep?"

What's the "right depth?"

The general complexity factors (e.g., number of changes, amount of relevant documentation, problem to be solved, etc.) aroused the most comment. This is a "murky area." Depending on the programming environment, different types of techniques work. Respondents' comments on some specific programming characteristics emphasized another dilemma in programming environments, the "old" school versus the "new" school of programming practices.

The results of the survey are being analyzed further in order to investigate differences across categories of groups: for example, programmers vs. managers vs. software experts; across different levels of programming expertise; etc. Also, several of the proposed complexity characteristics are essentially equivalent except for their description. Analyses are being done on these dependent variables to investigate how "saying the same thing in a slightly different fashion" affected the answers.

In conclusion, the efficacy of some of the new communication tools such as walkthroughs, inspections, peer reviews, etc. has emphasized the importance of attempts at interaction among a cross section of people involved in software development. The survey has been one such attempt.

REFERENCES

1. Belady, L.A., "Complexity of programming: A brief summary," In *Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost*. New York: IEEE Cat. #TH0067-9, 1979, pp. 90-94.
2. Chapin, N., "A measure of software complexity," *Proceedings of the National Computer Conference*. 1979. pp. 995-1002.
3. Chen, E.T., "Program Complexity and Programmer Productivity," *IEEE Transactions on Software Engineering* Vol. SE-4, No. 3. (May 1978), pp. 187-193.
4. Cobb, G.W., "A measurement of structure for unstructured programming languages," *Proceedings of the Software Quality and Assurance Workshop*, November 1978, pp. 140-147.
5. Curtis, B., "In search of software complexity," In *Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost*, 1979, IEEE Cat. #TH0067-9, pp. 95-106.
6. Curtis, B., Sheppard, S.B., & Milliman, P., "Third time charm: Stronger prediction of programmer performance by software complexity metrics." In *Proceedings of the Fourth International Conference on Software Engineering*. New York: IEEE, 1979.
7. Gilb, T. *Software Metrics*. Winthrop Publishers. Inc., 1977.
8. Grace, Alonzo G. Jr., "The dimensions of complexity," *DATAMATION*, September 1977, pp. 315-318.
9. Halstead, M.H. *Elements of Software Science*. Elsevier North-Holland, Inc.: New York, 1977.
10. Hansen, W.J., "Measurement of program complexity by the pair (Cyclomatic Number, Operator Count)," *SIGPLAN Notices* 13.2 (March 1978), pp. 29-33.
11. Jelinski, Z., and Moranda, P. B., *Metrics of software quality*, Tech. Rep. AFOSRTR-79-0128. Washington D.C.: Bolling AFB, AFOSR, 1978. (AD A065196).
12. McCabe, T.J., "A complexity measure," *IEEE Transactions on Software Engineering* SE-2,4 (December 1976), pp. 308-320.
13. McCall, J.A., Richards, P.K., and Walters, G.F., *Factors in Software Quality*, Tech. Rep. 77C1S02. Sunnyvale, CA: General Electric, Command and Information Systems, 1977.
14. McClure, C.L., "Model for program complexity analysis," *Proceedings of 3rd International Conference on Software Engineering*. IEEE Cat. no. 78CH1317-7C, pp. 149-157.
15. Mills, H.D., "The complexity of programs," In *Program Test Methods*, pp. 225-239. ed. W.D. Hetzel. Prentice-Hall Inc.; 1973.
16. Mohanty, S.N., "Models and measurements for quality assessment of software," *ACM Computer Surveys*, 11 (1979), pp. 251-275.
17. Myers, G.T., "An extension to the cyclomatic measure of program complexity," *SIGPLAN Notices* 12. 10 (October 1977), pp. 61-64.
18. Shooman, M. and Laemmel, A., "Statistical theory of computer programs—information content and complexity." *Digest of Papers COMPCON Fall 77*, IEEE Cat. no. 77CH1258-3C, pp. 341-347.
19. Sullivan, J.E. *Measuring the complexity of computer software*. MITRE Corporation, MTR-2648, Vol. V, June 1973.
20. Thayer, T.A., et al. *Software reliability study*. RADC-TR-76-238. August 1976.
21. Weissman, L., "Psychological complexity of computer programs," *SIGPLAN Notices* 9 (June 1974).
22. Woodward, M.R., et al., "A measure of control flow complexity in program text." *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1 (January 1979), pp. 45-50.
23. Zolnowski, J.C., and Simmons, D.B., "Measuring program complexity in a COBOL environment," *Proceedings of the National Computer Conference 1980*, pp. 757-766.
24. Zolnowski, J.C., and Simmons, D.B., "A complexity measure applied to FORTRAN," *Proceedings of COMPSAC 77*. IEEE Catalog no. 77CH1291-4C, pp. 133-141.
25. Linstone, H.A. and Turoff, M. (eds). *The Delphi Method: Techniques and Applications*. Addison-Wesley, 1975.

Salvaging your software asset (tools based maintenance)

by MICHAEL J. LYONS

The Catalyst Corporation
LaGrange, Illinois

ABSTRACT

Software is a valuable asset embodying decision processes of an organization and contributing directly to the means of production. Maintenance is the mechanism for combating deterioration of that software asset, which over time tends to become arthritic and inflexible to change. Maintenance, though extremely costly, is essential to insuring the viability of the organization. Both rewrites and purchased software, with ensuing conversions, are usually not a cost-effective solution to software decay. Structured retrofit is an effective alternative, using a software tools-based methodology for combating decay and the high costs of maintenance. The critical tool is the COBOL structured programming engine. With it, spaghetti code software is mechanically transformed to well-structured programs, whose ongoing maintenance reaps the benefits of the structured programming methodologies.

INTRODUCTION

Software is an asset. It is an owned resource that contributes to the means of production. It is costly to acquire and even more costly to replace. To insure maximum return on one's software investment requires prolonging software's usable life and making best use of that life. It is essential to mine that software asset in order to maximize its role as a major contributor to the means of production and overall organizational productivity.

The function of software is to embody a subset of the enterprises's decision processes and to enable them to be carried out by computer machinery. It is the decision processes of the enterprise that are at issue. They are unique to the organization and vital to its prosperity. It is software's embodiment of the organization's decision processes that makes it a direct contributor to the means of production. Indeed, one might say that the survival of the organization depends on insuring the vitality of software.

Software is not a physical machine, and it therefore does not wear out. By the same token, it is not like a small child, which can improve its capabilities or change its attitudes over time. Therein lies the cause of software deterioration: its inability to change itself to match the changing decision processes of the enterprise. Software progressively loses its pro-

ductive capacity unless it is continually infused with the ongoing changes in the enterprise's decision system. This process of adaptation and enhancement is called software maintenance. Whether it corrects bugs, changes the specifications, or improves efficiency, *maintenance* for the purposes of this paper is any change to any system for any reason. The process of maintaining software is unexpectedly difficult and expensive. The typical Fortune 500 company today spends 70% of its (non-operations) data processing budget on maintenance.¹ I have already said that software, unlike a child, does not grow smarter and more capable; unfortunately, it does seem to grow old and cranky. The very act of changing it tends to destroy or obscure its structure^{2,3} (spaghetti code and untrustworthy documentation) and make it progressively more resistant to change. I sometimes call this condition *software arthritis*—the buildup of deposits in the joints of the organism that make it less and less flexible. Remember that flexibility is the characteristic required to preserve the productivity of the asset. Therefore, preserving flexibility—combating software arthritis—is the key element in protecting the asset; and it is the subject of this paper.

Before a discussion on combating software arthritis, let me first point out why the maintenance function will not go away, and, furthermore, why its costs and complexity are on the increase.

IS REWRITE A SOLUTION?

If software deteriorates over time, why don't we rewrite it? Software rewrite is economically unacceptable. An inventory of one Fortune 500 company's software library, shown in Table I, points out why. These statistics were taken from a Chicago-based diversified manufacturing firm.¹⁸ They point out some interesting facts. First, in this case, software represents a very substantial one-third-billion-dollar asset, assuming a cost of \$10/line to rewrite. Note that this figure is the most conservative one we could find. A more commonly quoted figure is \$25/line.^{4,5}

Second, since all programs are not equal, let us assume a strategic approach to a rewrite. The 80/20 rule states that "20% of the programs cause 80% of the problems and corresponding costs." Assuming the ability to weed the good from the bad, in this case we are left with 10,000 tin gods—my term

TABLE I—Appraisal of a software rewrite

Number of COBOL programs:	50,000
Average number of lines/program:	750
Total lines of COBOL code:	37,500,000
Replacement value of code:	\$375 million

ASSUME 80/20 RULE:

50,000 × 80% programs = 40,000 programs

50,000 × 20% programs = 10,000 programs

COST TO REWRITE THE 20% HIGH-PAYOFF CANDIDATES

10,000 programs × 750 lines = 7,500,000 lines

7,500,000 lines × \$10/line = \$75 million

LABOR TIME FOR HIGH-PAYOFF REWRITE

7.5 million lines/(15 good lines/day × 240 productive days/year) = 2,015 RESOURCE YEARS

for any program most often described as “My god, don’t touch that or it’ll blow up.” If we decided to rewrite the tin gods at an average rate of 15 good-debugged lines/day⁶ it would cost \$75 million. Even if funding were available, it would take over 2,000 resource-years to do the job. When 13% of the data processing jobs in America today are open and there is no one to fill them, when there is a current shortfall of 58,000 programmers,^{1,7} who is going to do such a rewrite? In short, a rewrite is not a viable alternative! It might be worth noting that although the example used here represents a large company, the circumstances are linear. That is, if your particular company is small, then your library is smaller and your rewrite task is smaller, but so is your budget and staff.

THE MAINTENANCE DILEMMA

Notwithstanding the criticality of software to the organization, arthritic software is a special maintenance headache for management. Spaghetti code and untrustworthy documentation are not new; management has been facing them for years. Familiarity, however, is not control. The exponential growth in maintenance costs is directly attributable to our inability to control or improve on the quality and human maintainability of our systems. In 1960 the typical data processing organization spent 30% of its nonoperations budget on maintenance; in 1970 it spent 50%; today it spends 70%.¹ The primary reasons for high maintenance cost are

1. Maintenance is a people-intensive activity. While the cost of hardware plummets, the cost of people is rising. By 1985 the cost of hardware will be at 1/10 the 1979 rate, and people will be at twice the 1979 rate.⁸
2. The number of systems in our inventory has increased substantially, correspondingly increasing the maintenance load. Average systems life has increased from three years in 1960 to five in 1970 and eight today.¹⁸
3. Existing systems were designed to operate in a stand-alone fashion, but today we have new requirements from

TABLE II—The structured programming methodologies

CHIEF PROGRAMMER TEAM
DEVELOPMENT SUPPORT LIBRARIES
TOP-DOWN DESIGN
STRUCTURED PROGRAMMING
STRUCTURED WALKTHROUGHS
STRUCTURED TESTING

middle and top management. We are trying to revise existing operational-level systems in order to support control and planning-level systems.^{9,10}

This last point is the one that most influences maintenance costs for the 1980’s.^{9,10} The primary reason 7 out of 10 programmers are involved in maintenance today is that those lower-level operational systems were designed for hardware efficiency and not human maintainability. Costs of maintenance have become alarming because the lower-level systems can not easily support the higher-level systems demanded today.

Should we scrap existing code and start again? There is little argument that code in most operational level systems today is difficult to maintain, but that difficulty does not make the programs bad. Bad code is not the same as bad programs. It is critical to remember that all code, even spaghetti, meets operational-level user requirements but is now subject to sweeping changes mandated to support control and planning-level systems.⁹ The basic logic is sound and proven; the code reflecting it is not. The question here is whether the structured programming methodologies can be employed to advantage in improving the code (see Table II).

These methodologies are being introduced into new systems every day and have had a substantial impact on subsequent costs of maintenance. Normally, when the structured programming methodologies are used in development, subsequent maintenance costs and effort are reduced by a 3:1 ratio.^{11,12,20} However, it is usually uneconomic to rewrite or convert operational-level systems in order to facilitate development of new control- and planning-level systems.

Fortunately, there is an alternative that preserves the logical integrity of the operational-level systems and at the same time provides a well-structured basis for comprehensive maintenance and future systems growth. It involves introducing the structured programming methodologies and their benefits to existing systems reliably and promptly, *after the fact*, through the use of software tools. It is called *structured retrofit*. Structured retrofit is the application of today’s structured programming methodologies to yesterday’s systems in order to meet tomorrow’s demand. Through this method the organization can combat software arthritis, continue to get payback from existing systems, and still meet demands to build on them, thereby salvaging its software asset.

The remainder of this paper presents a software-tools-based methodology for introducing structured programming into existing code. The structured-retrofit procedures, methodologies, and software tools have all come together for beta testing at FMC Corporation for the past year. FMC Corporation is a 3.5-billion-dollar-a-year diversified corporation. Its

data processing facilities involve a worldwide network tied into large IBM mainframes, supporting a library of approximately 35,000 COBOL programs, from which we have drawn our testing sample population. The procedures for structured retrofit used during beta testing at FMC are described in the remainder of this paper.¹⁸

STRUCTURED RETROFIT

Structured retrofit, a concept and methodology pioneered by Jon Cris Miller,^{13,14,15,16,17} involves the establishment of a task force made up along the lines of a chief programmer team. This team has responsibility for scoring the existing software library, isolating high-payoff candidates for retrofit, conducting the retrofit, and finally, validating its success. The task force has a basic arsenal, made up of the following software tools, assembled from various organizations around the United States: Code evaluators, formatter, structuring engine, optimizer, and file-to-file compare utility. (Other software tools are being considered for future use, including but not limited to automated documenters, job schedulers, and test vehicles.) Their use, described below, minimizes human clerical activity and maximizes mechanical processes.

Scoring

Scoring combines both objective evaluation of the software through the use of code evaluation tools and subjective input from managers and users. The objective evaluation determines the degree of structure in a program, the level of nesting, the degree of complexity, the breakout of verb utilization, and failure analysis; and it presents a concise trace of control logic. With it, we have a clear appraisal of the quality of the code.^{14,18} However, no matter how low a piece of code rates during the objective evaluation, if that code runs week after week without problems and never requires enhancement, then obviously it is a low-priority candidate for retrofit. In short, scoring must involve more than just an appraisal of code. It must also be a predictor of upcoming maintenance, enhancements, and planned replacement. There is no substitute for subjective input from management and users.

Compilation

Once the high-payoff candidates have been strategically isolated, they are compiled. One of the fundamental assumptions behind a retrofit is that programs must compile cleanly and be currently operational. Those that do not compile cleanly are referred to the appropriate department for correction. The retrofit procedures are not a mechanism for making nonoperational systems operational.

Restructuring

The source code is then put through a structured programming engine. For purposes of this presentation, a structuring engine is a software tool with two properties:

- It transforms an executable program written in a given language, but of undetermined structure, into another program written in the same language with a well-defined structure.
 - The resulting program produces the same transformation on any set of input data as does the original program.
- Further discussion of structured programming and of a structured programming engine will follow shortly.

Formatting

Once restructured, the source code is then put through a formatting package in order to enhance the visuals and readability. Following formatting, the newly transformed code is then recompiled to insure that there are no syntactic errors. A formatting package can be expected to substantially enhance the visuals. Standard features of a good formatting package can be seen in Table III. I am also aware of development on a formatting package that will eliminate qualification of data names. For move corresponding, it currently requires that the user manually expand each qualified move before eliminating qualification mechanically.

Validation

Once recompiled, the validation mechanism begins. A set of input data is processed through the old program, then through the new program. The resulting outputs are then compared by a file-to-file compare utility. One certainly does not want to employ a visual scan of output reports to insure that they are identical. A mechanical bit-for-bit comparison is far more accurate, simple, and fast. Ideally, one uses copies of live files for a comprehensive validation.

Optimization

In conjunction with compilation, the program passes through an object code optimizer. Whether restructuring is done manually or through automated mechanisms, one expects to introduce some overhead as a consequence of restructuring. However, experience to date indicates that little net overhead remains if an optimizer is used. Initial experimentation resulted in a 20% increase in core requirements from optimized original code compared to optimized restructured code. However, recent improvements in the structuring engine algorithms indicate only an average of 8% overhead and suggest the possibility of absolute improvements.

Retrofit Results

Retrofit goes beyond description and prescription to produce a completed product: well-structured source code logically equivalent to the original. It cannot, however, eliminate logic errors; determine intent; or react to user requirements, demands, and complaints. It does not solve the maintenance problem, but it does simplify the solution. It provides a base-

TABLE III—Features of a formatter

-
- Indents and formats code
 - Standardizes paragraph prefixes
 - Relevels data division
 - Standardizes field alignment
 - Standardizes reserved words
 - Restricts verbs to one per line
 - Provides global name substitution
-

line for cost-effective maintenance by making existing systems understandable.

WHAT IS STRUCTURED PROGRAMMING?

If I were to ask 10 different people for a definition of structured programming, I would probably get 10 different answers. But, suffice it to say for our purposes, structured programming is a method of programming according to a set of rules that enhance a program's readability and maintainability. Structured programming centers around the concept of a module having a single entry point and a single exit point. Structured programming involves the separating of control from action so that the logic flow becomes clearer to human beings, even though a computer obviously doesn't care.

STRUCTURED PROGRAMMING ENGINES

Structured programming engines could theoretically be developed for any programming language. However, to my knowledge, the only two languages for which they currently exist are FORTRAN and COBOL. The FORTRAN engine, developed by Caine, Farber & Gordon, Inc., has been in existence since 1975 and is used in conjunction with a superset of FORTRAN.¹⁹ The only commercially available COBOL structured programming engine, to my knowledge, is the one developed at the Catalyst Corporation by Jon Cris Miller.¹⁸ A structured programming engine accomplishes two things. First, it cleans up existing language and verb usage; second, it introduces consistent structure to the code. Table IV shows what you can expect from a COBOL structured programming engine. A structured programming engine and a good formatting package cover and correct a multitude of sins. The most important result is an isolated control hierarchy. Isolating control provides clear visibility of the algorithms used in that program. In COBOL, the primary control structures used are loops and decision trees.

Unfortunately, most programs employ them on a global rather than a local basis, to construct algorithms. Being able to see control in tight, small modules allows clear visibility and understanding of the program and its component algorithms. If one ever really expects to introduce the concept of reusability of code, there is no better asset to mine than an operational software library. Structured retrofit potentially leads to an inventory of existing algorithms and a practical mechanism for the reusability of code.

TRANSFORMATION RATE

Can a structuring engine handle any program? Judging from personal experience and the results obtained from the beta testing, our engine can process 60% of programs offered immediately and an additional 20% with some manual intervention. The other 20% we cannot handle cost-effectively now. These percentages seem to be consistent with those of Caine, Farber & Gordon.¹⁹ One example of code that requires manual manipulation in order to restructure it is structurally recursive code. Technically, COBOL does not support recursive code. However, some programmers have discovered that by using switches they can terminate a seemingly endless chain of PERFORM flip-flops. It is sometimes difficult to determine compiler tolerances to syntax violations, as in the case of delimiters, reserved words, and margin alignment. When in doubt, we have always elected to follow the ANS COBOL standards.

NON-TASK-FORCE RESPONSIBILITIES

In addition to tasks performed by the retrofit task force, there are tasks to be done by other members of the participating organization:

1. Provide source code for retrofit.
2. Provide copies of live test data for validation of the retrofit. If already available, comprehensive artificial test data may be employed.
3. Review the dead code list to verify that code is not required.
4. Provide on-site computer time.

While the process is primarily mechanical, there is still a substantial amount of work for the retrofit task force and selected members of the host organization. However, the process offers *no* disruption to the user community.

TABLE IV—Features of a COBOL structuring engine

Cleans up language	Removes alters Eliminates perform through overlap Reduces go tos Increases performs Converts notes to comments Eliminates drop through confusion Removes dead code
Structures	Isolates control hierarchy Highlights looping conditions Bounds action modules Physically groups and standardizes all I/O Consolidates all program termination to a single goback Does not remove logic errors

TABLE V—Reasons to retrofit

-
- Cut maintenance costs substantially.
 - Divert maintenance resources to new development.
 - Meet user requirements on a timely basis.
 - Decrease programmer turnover.
 - Decrease the number of systems designated incapable of cost-effective maintenance.
 - Increase the number of systems capable of sustaining major enhancements without a rewrite or extensive testing.
 - Limit the need for a specialized person to maintain each system.
 - Simplify tuning, reconfiguring, and rewrites to take advantage of cost and technological opportunities.
 - Standardize the multiple programming styles found in a program written or maintained by more than one programmer.
 - Cut research costs when the user says, "I suspect something is wrong."
 - Insure consistency with mechanically verifiable standards
-

SUMMARY

In this paper I have introduced structured retrofit, a complex process. It is not a new process in other production areas of the business world, but it is new to data processing. Table V reviews the primary benefits of retrofit.

In closing, let me emphasize two very important points:

- Software, of all ages, shapes and sizes, is a valuable asset to a corporation.
- The corporation can reap the benefits of the structured programming methodologies from currently unstructured systems, thereby salvaging its software asset.

Structured retrofit of application systems decreases costs, increases productivity, and improves morale. As compared to a manual rewrite, it is virtually 100% mechanical, requires little elapsed time, makes minimal demands on managerial and technical staffs, and is completely transparent to the user. Structured retrofit fights software decay.

ACKNOWLEDGMENTS

Putting pen to paper is a difficult chore for me. I owe thanks to many for contributing to this paper: To Jon Cris Miller, who not only acted as editor, but, more important, introduced me to retrofit concepts. To FMC Corporation, our retrofit

beta test site, which has endured our failures and enjoyed our success. To Nicholas Zvegintzov, for his insights on software as an asset. Last, to my wife, without whose support there would be nothing. The errors and omissions remain my own.

REFERENCES

1. Cooper, J.J. "Software Factory." *Raytheon Data Services*. Burlington, Massachusetts, 1980 p. 13.
2. Brooks, F.P., Jr. *The Mythical Man-Month* (3rd ed.) Reading, Massachusetts: Addison-Wesley, 1975.
3. Belady, L.A., and Lehman, M.M. "A Model of Large Program Development." *IBM Systems Journal*, (Vol. 15, No. 3), 1976, pp. 225-252.
4. Lehman, J.H., "How Software Projects Are Really Managed." *Datamation*, 25 (1979), pp. 118-129.
5. Jones, C. "Optimizing Program Quality and Programmer Productivity." *Proceedings of GUIDE 45*, 2 (1977), pp. 689-705.
6. Yourdon, E.N. *Techniques of Program Structure and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.
7. Editors of Business Week. "Missing Computer Software." *Business Week*, No. 2652 (Sept. 1, 1980), pp. 46-56.
8. Diebold, J. "The Annual Diebold Technology Scan 1979." *The Diebold Computer Planning and Management Service*, 1979, 89pp.
9. Nolan, R.L. "Managing the Crisis in Data Processing." *Harvard Business Review*, 57 (1979), pp. 115-126.
10. Danziger, J., Kraemer, K., and King, J. "An Assessment of Computer Technology in U.S. Local Government." *Urban Systems* 3, 1978, pp. 21-37.
11. Diebold, J. "Improving the Utilization of Personnel Resources." *The Diebold Computer Planning and Management Service*, August, 1979, pp. 44-46.
12. Ryan, H.W. "Structured Methods." *Computerworld* 13 (1979), pp. INDEPTH/1-24.
13. Miller, J.C. "Some thoughts on Structured and Traditional Programming." Unpublished paper, Montgomery Wards Corporate Systems Division, 1975, 4pp.
14. Miller, J.C. "Improved Programming Technologies Retrofit (A Study of the Application of Improved Programming Technologies to Systems Developed without Improved Programming Technologies)." Unpublished report, Montgomery Wards Corporate Systems Division, 1976, 46pp.
15. Miller, J.C. "Sow's Ear: The Structuring Engine (COBOL)." *Yink, The Weekly Memo to Yourdon Instructors*, Nov. 18, 1977. pp. 1-3.
16. Miller, J.C. "S.E.—The Structuring Engine." Unpublished paper, 1979, 14 pp.
17. Miller, J.C. "Structured Retrofit." *Techniques of Program and System Maintenance*. Lincoln, Nebraska: Ethnotech, 1980, pp. 85-86.
18. Lyons, M.J. "Structured Retrofit—1980." *Proceedings of SHARE 55*, (1980), pp. 263-265.
19. de Balbine, G. "Better Manpower Utilization Using Automatic Restructuring." Caine, Farber and Gordon, Inc., *AFIPS Proceedings of the National Computer Conference*, 1975, pp. 319-327.
20. Daly, E.B. "Organizing for Successful Software Development." *Datamation*, 25 (1979), pp. 106-120.

Maintenance is a management problem and a programmer's opportunity

by JOHN REUTTER, III
Dynabyte Incorporated
Menlo Park, California

ABSTRACT

Many installations treat systems maintenance as an afterthought or unwanted stepchild of systems development. The resulting demoralization of maintenance programmers and apparent confusion of maintenance activities can be avoided. Making maintenance visible and placing it in the proper perspective are responsibilities of maintenance managers. This paper defines categories and characteristics of maintenance that highlight the complexities and value of maintenance and its contribution to the corporate bottom line.

There is an abundance of opportunities for programmer creativity, management success, and job enrichment in software maintenance. Suggestions are made in this paper on how to achieve proper perspective and assure successful maintenance through application of a "structured maintenance" methodology. The structured maintenance approach defined in this paper enhances programmer morale, improves system quality, assures user satisfaction, and places maintenance in its proper perspective as a vital, profitable, contributing corporate business function.

THE MAINTENANCE PROBLEM

What do you think is the maintenance problem? Many have answered this question with the following statements:¹

- "Maintenance is treated as a necessary evil."
- "There is no pride or recognition in maintenance work."
- "Maintenance is only an afterthought."
- "Maintenance is used to train beginning programmers."
- "Maintenance never accomplishes anything."
- "Nobody likes maintenance."

These statements are all very negative. In contrast, the facts of the author's own experience do not agree with them. A possible conclusion to be drawn from statements like these would be that those who make them do not understand what maintenance is really all about even though they may themselves be deeply involved in maintenance activities. The author believes the maintenance problem is one of perspective and management focus. This paper will define the categories and characteristics of maintenance and then point out the opportunities for creativity, cost cutting, and job enrichment

that abound in software maintenance. Suggestions will be made on how to achieve proper perspective and management focus on maintenance activities. Finally, this paper will show why programmer morale, system quality, user satisfaction, and aggressive maintenance management are necessarily intertwined in successful systems maintenance.

Restatement of the Maintenance Problem

Treating the negative comments cited above as symptoms of an underlying problem, it seems that the real problem is the manner in which maintenance is managed—at all levels of an organization. Maintenance often has the outward appearance of being a helter-skelter, uncoordinated activity rather than a planned, methodical, controlled, necessary business function of any organization committed to computerized data processing. The problem is not so much related to technology as it is related to execution. The tools and techniques for effective maintenance are readily available. Their actual use seems to be skimpy and undisciplined.

The Real Activities of Maintenance

If maintenance does indeed have a problem, the problem is that it is a misunderstood, mistakenly maligned activity. A major contributor to this problem is management's failure to focus attention on the "real" activities and benefits of maintenance. As a result, many individuals in a corporate structure involved with any data processing activity equate "maintenance" with "repair"—the fixing of software "bugs." This perception of maintenance as primarily a "pest control" exercise is the fault of those in maintenance management positions who fail to make visible to the corporate structure the many types of maintenance that take place—including those that are necessary for reasons other than fixing bugs.

CATEGORIES OF MAINTENANCE

The first step in gaining a proper perspective on maintenance is to identify the various kinds of maintenance that take place. While there are certainly as many ways of categorizing main-

tenance activities as there are categorizers,² the following list seems to apply well to most situations:

1. Emergency repair—the immediate fixing of code that must be done to continue service to the user.
2. Corrective coding—the fixing of code to correctly match specifications or correctly utilize system resources (excluding emergency repair).
3. Upgrades—the modification of code required when upgrades in system resources occur (e.g., new model CPU, new release of operating system, new type of telecommunications software, and so on).
4. Growth—the modification of code and development of new programs necessitated by growth in the number of records maintained and/or number of users supported (e.g., expansion or conversion of files, addition of network nodes and user terminals).
5. Support—the explanation of system capabilities, system outputs, and proper use of system features to end users and their managers; assistance to user planning in areas affected by the systems; research and development of awareness of factors affecting future change requirements; measurement and evaluation of system performance.
6. Change in conditions—the modification of code required when business conditions change, particularly due to regulatory situations or other causes which are beyond control of the corporation but to which the corporation is bound.
7. Enhancements—the modification of code or addition of whole new subsystems due to user requests.

Differences in Management Requirements among Categories of Maintenance

It is important to be able to distinguish among these various types of maintenance because of the vast differences in how they should be managed. The activities within them are quite different, affect different parts of the system, and generally require different skills. For example, emergency repair situations usually require an in-depth understanding of the operating system, file structures, and network protocols of the application affected but not of the intricate data edit relationships that are tied to business requirements, government regulations, or corporate policies. Knowledge of data relationships is much more important for dealing with corrective coding situations or enhancements and is of practically no value at all for dealing with system upgrades or growth maintenance requirements.

Observations on Maintenance Costs

Based on extensive experience in software development and maintenance, including work with three large-scale systems (Pacific Telephone's Service Order Retrieval and Distribution System, Bell Telephone Laboratories Automated Repair Service Bureau System, and VISA, Inc.'s Base II—Worldwide Credit Card Transaction Interchange System),

some of the author's observations regarding maintenance costs are as follows:

1. 70% of the programming costs occur after installation of the initial product.
2. The majority of these maintenance costs were never projected as future costs of the system when the economic feasibility of the system was determined initially.
3. Only a small percentage of these costs (less than 10%) can be attributed to initial programming error.
4. A large percent of maintenance costs are nondiscretionary.

Recent articles generally seem to concur with these observations,^{3, 4, 5}

Distribution of Maintenance Costs

The distribution of maintenance costs in the author's experience against the suggested categories has been as follows (see for contrast Lientz and Swanson⁶).

Emergency repair	2%
Corrective coding	6%
Upgrades	12%
Growth	10%
Support	15%
Change in conditions	10%
Enhancements	45%

Fixing the bugs amounts to only 8% (emergency repair + corrective coding) of the total maintenance costs in this distribution.

The category with the largest percentage of maintenance costs is also the one that typically experiences the most variation because it is so easily affected by availability of funds. Very little can be done to avoid the other categories of maintenance. When a program fails, it has to be fixed. When new users are added, they have to be accommodated. When laws are changed, they have to be followed. Consequently, these types of maintenance are relatively inelastic. Enhancements, on the other hand, are the most elastic. If enhancements were to be totally eliminated from the prior distribution of costs, the costs would be redistributed as follows:

Emergency repair	4%
Corrective coding	11%
Upgrades	22%
Growth	18%
Support	27%
Change in conditions	18%

With the elimination of enhancements, the percentage of costs attributable to the fixing of bugs rises to 15%. This percentage represents an unlikely maximum. In fact, enhancements are rarely cut completely out of maintenance. In lean times, minor enhancements are usually implemented under the guise of corrective coding, upgrades, or change in conditions maintenance.

MAINTENANCE MANAGEMENT

Making user and corporate management aware of the “non-bug” categories of maintenance and of their ultimate contribution to the corporate bottom line is the responsibility of those who manage maintenance activity. Visibility of maintenance can be achieved by publishing monthly, quarterly, and annual reports of maintenance achievements showing money expended, results achieved, and highlighting important changes.

Bottom Line Visibility of Maintenance

One of the benefits of categorizing maintenance activities and tracking costs against them is that it provides an opportunity for promoting the value and contribution of maintenance to the corporate bottom line. As a result, decisions regarding discretionary spending for enhancements, and to some extent upgrades, can be determined by enlightened management. These decisions should be made on the basis of profit contribution to the corporate financial position. There should be no “afterthought” or “necessary evil” considerations involved in such decisions. Since the costs of such discretionary activities typically fall in the 50–60% range of total maintenance costs (they have done so consistently in the author’s experience), there is reason to believe that maintenance is an important factor in the bottom line and potentially a creative benefactor for the corporation.

Management of Maintenance Programmers

One of the reasons that programmer morale may seem so low with respect to maintenance activities, in addition to the activities not being highlighted appropriately, is that they are not being properly managed. Often such important items as changes caused by growth, upgrades, or changes in conditions are treated in a helter-skelter, ad hoc fashion with little or no planning, without proper reviews, and without controlled installation.

The area of maintenance which seems to survive best is enhancement maintenance. The most likely reason for that is that enhancements of any reasonably large scale closely resemble the activities of new development. Systems development has received a great deal of attention and has profited from the evolution of such disciplines as structured programming and composite design. However, enhancement maintenance is really much more complicated than initial systems development. Enhancement development must be extremely creative to simultaneously accomplish the following:

1. Correctly implement the specification required.
2. Smoothly integrate into the existing, but changing, system.
3. Strengthen (improve) the overall quality of the system.
4. Not adversely affect the unchanged areas of the existing system.

The consequence of having to meet the integration requirements is that more software has to be developed to accomplish

specific user design goals than is necessary on an initial development project to meet the same user design goals. There is great need for system cross-references, regression test facilities, maintenance control software, installation software, and management reports of enhancement activity. These requirements should be viewed as an unusual opportunity for programmer creativity.

STRUCTURED MAINTENANCE

Borrowing from the concepts and jargon of the highly successful structured programming technologies, I propose an approach to maintenance to be called “structured maintenance.” The essence of structured maintenance is that it is based on a methodical, planned approach to systems maintenance that uses the concepts of top-down development, system modularity, and structured coding where they apply. This is in contrast to the definitions of “structured maintenance” offered by others.^{3, 4}

The main elements of this proposed structured maintenance are the following:

- comprehensive viewpoint of systems maintenance,
- documented maintenance master plan,
- use of project management techniques,
- automated performance results reporting,
- automated maintenance controls,
- structured coding rules.

Comprehensive Viewpoint of Maintenance

The first necessary element of structured maintenance is a comprehensive viewpoint of the roles and responsibilities of systems maintenance. These responsibilities include planning, reporting, and other strategic management tasks in addition to the ordinary day-to-day management activities. This viewpoint also includes acceptance of accountability for the corporate bottom line. The perspective to be used is a top-down view of postinstallation systems activities. The topmost view is total accountability for all categories of maintenance with respect to specific systems. Lower levels subdivide maintenance activities and management requirements into the various maintenance categories.

The Maintenance Master Plan

The cornerstone of structured maintenance is the maintenance master plan which consists of a set of documented plans, each of which addresses individually one of the categories of maintenance identified in this paper. The purpose of each plan is to specify exactly: what type of maintenance is covered by that plan; how the maintenance activities are to be structured; how activities are to be initiated, completed, documented, summarized, and reported; what the goals of that type of maintenance are; what constitutes its elements of cost;

and what the monthly, quarterly, and annual objectives for maintenance performance are for that category.

The value of such a documented plan is that everyone is informed in advance of exactly what is to be done, how it is to be done, who is to be told about it, who is accountable for it, and how it contributed to the corporate bottom line. Each plan is required reading for all those involved in the management or execution of maintenance. With such plans, the burden of maintenance success is shifted from the maintenance programmers to the maintenance managers, as it properly should be. This frees programmers to apply their skills creatively to the programming situations that abound in all categories of maintenance. It then truly becomes a programmer's opportunity to make maintenance a meaningful and productive experience on a par with, or even superior to, original systems development.

Project Management Techniques

The effective techniques of project management typically used in systems development are often abandoned when the initial target of system installation is met. Subdividing maintenance requirements into tasks with deadlines, assigning resources to the tasks, reviewing the progress of these tasks regularly, and taking notice of critical paths are invaluable management techniques for assuring progress towards successful maintenance results.⁸ The maintenance master plan should identify the goals and objectives of maintenance in such a way that these project management techniques can be tied into overall goals and performance evaluation.

Maintenance Program Structure Rules

Systems maintenance may require making changes to either structured or unstructured systems. A convincing argument in favor of structured development is that it minimizes the likelihood of "bugs" in program code and that it facilitates future modification of code.⁷ However, at this point of time, the vast majority of systems in operation were built before general acceptance of structured design and structured programming. Consequently, maintenance programmers face the dilemma of how to apply the rules of structured programming to an unstructured system.

A pragmatic approach to the use of structured programming must be applied in these situations. To strictly apply structured programming rules to changes being made in an unstructured system usually requires major recoding, even of sections of programs which are performing flawlessly. The costs of redeveloping code must be weighed against the risks that recoding may introduce errors that otherwise would not occur and against the use of those programming resources for other maintenance requirements. A pragmatic rule that could be used is as follows: Where possible, follow the conventions and rules of structured programming in effect at your installation for new systems development. Otherwise, maintain the style and conventions used within the program being modified. Do not, in any case, regress a structured program into a less disciplined convention when making modifications.

Design for Maintenance

When developing new systems from ground up or modifying existing systems, a cornerstone for design ought to be integrating considerations for future maintenance in the design of new code. For example, a system ought to police itself or at least provide performance and critical factor information as an integral part of its output generation. Making error-free modifications to existing systems largely depends upon knowing the internal behavior of the existing systems. Programmers need to know code referencing patterns, high water marks in the file utilization, resource usage patterns, and logic path frequency distributions to make judgments about setting up test cases, designing defensive code, and anticipating future problems. The system ought to be designed to supply this type of information to the developers of the system.

Automate System Performance Reporting

Often, data processing is guilty of the "cobbler's children's shoes" syndrome. We fail to utilize the power and capability of computer systems in the execution of our own jobs. The capturing and reporting of systems success and failure data should be automated into the system. Monthly, quarterly, and annual performance reports should be generated and maintained by the computer system and can be done cost effectively if incorporated within the system being reported on.

Automate Change Installation Process

One of the most beneficial accomplishments that maintenance can achieve is the "routinizing" of the change installation process itself. The ideal to be achieved is for maintenance to become a regularly scheduled production job managed by the computer operations center staff. The controlled movement of program changes from the development environment to the production environment can be best achieved through automated techniques. This automation should be itself treated as a serious, total-systems task with appropriate checks and balances, reports, and conditional actions.

CONCLUSIONS

The maintenance problem is primarily one of perspective; that is, maintenance lacks recognition as a vital, profitable, contributing business function. This problem can be overcome through proper management that incorporates formal planning and reporting disciplines into the maintenance process. Maintenance programmer morale can be significantly boosted by recognition and by planned, methodical application of good programming techniques to systems modification and to the development of automated tools and maintenance control systems. An approach to maintenance that uses the structured concepts currently used for systems development can be effectively incorporated in systems maintenance. Maintenance can be made very stimulating and is, in fact, significantly more challenging than pure systems develop-

ment. Maintenance is truly a management problem and a programmer's opportunity.

REFERENCES

1. Reutter, J. "Student Reviews of Seminars in Structured Maintenance," Golden Gate University (San Francisco, CA, 1980).
2. Swanson, E. Burton. "The Dimensions of Maintenance," *Proceedings of the 2nd International Conference on Software Engineering* (Long Beach, CA: IEEE, 1976), pp. 492-497.
3. Lyons, Michael J. "Structured Retrofit-1980," *Proceedings of SHARE 55* (Chicago: SHARE, 1980), 23 pp. in Session M767.
4. Parikh, Girish (Editor). *Techniques of Program and System Maintenance* (Lincoln, NB: Ethnotech, 1980), 289 pp.
5. Singer, Larry M. "Attacking maintenance costs," *Computerworld*, Volume 14, number 36 (September 8, 1980), pp. In-depth/9, 12-16.
6. Lientz, Bennet P., and Swanson, E. Burton. *Software Maintenance Management* (Reading, MA: Addison-Wesley Publishing Co., 1980), 214 pp.
7. Myers, Glenford. *Reliable Software Through Composite Design* (Van Nostrand Publishing Co., 1978).
8. Gunther, K.C. *Management Methodology for Software Product Engineering* (Wiley and Sons Publishing Co., 1978).

Productivity in software maintenance

by NED CHAPIN

InfoSci Inc.
Menlo Park, California

ABSTRACT

New evidence is presented that certain management decisions play a critical role in determining the level of staff productivity in the maintenance of computer programs and systems. The finding is surprising because the pivotal decision-making process has not previously been identified as a significant influence on productivity in either development or maintenance.

INTRODUCTION

Productivity is the theme of the National Computer Conference this year, and the productivity of the people doing the maintenance of computer programs and systems is the topic of this paper. Some recent work points to some fairly easy and inexpensive ways to make some significant gains. To appreciate this requires looking briefly at some background, then reporting the evidence, and finally interpreting and discussing the findings.

First, however, a few clarifying words should be said about the software life cycle.³ Very roughly, and for the purposes of this paper, that cycle can be split into two main stages: development and maintenance. Development includes conception, feasibility, analysis, design, programming, coding, debugging, testing, and conversion of programs and systems. Once the subsequent phase of regular routine operation has started, then the maintenance stage may begin; it may extend intermittently for many years.

During the maintenance stage the program or system as delivered for operation is corrected to counteract detected bugs, is enhanced to add functions, and is modified to delete or change the existing functions in nature or scope or implementation to match changed conditions or requirements. Hereafter, the term *maintenance* is used in the broad sense to include corrective, adaptive, and perfective work on an existing computer program or system.^{7, 8, 12}

BACKGROUND

The background consists of some fairly common observations. First, the maintenance of programs and systems is consuming a larger percentage of the person-hours of programmers and analysts than it has historically, and the costs of

maintenance are growing as a proportion of the expenditures for software work.^{6, 14} Although the situation is generally worse for organizations that have been computer users for many years, even computer users of a few years' duration note these unfavorable trends. In short, on a current basis, maintenance is claiming a larger fraction of the available personnel resources than it has historically.

Second, on a life-of-software basis, the cost of maintenance is commonly the most expensive phase in the life cycle.^{1, 11} The money that was originally expended to develop the software may have appeared large at the time, but expenditures to maintain the software after delivery appear even larger and often cumulatively exceed after only a few years the original development cost.

Third, the recognition of the value and importance of maintenance work is usually relatively low. Both coworkers and management personnel act as though they held maintenance work in low esteem.^{9, 13} Management rarely rewards good work in doing maintenance as generously as good work in doing development. Coworkers value nonmaintenance assignments more highly than they do assignments involving the maintenance of existing programs and systems. Lower morale tends to go with sagging or stagnant productivity.

Fourth, the challenge and glamor of the new techniques such as software engineering have only rarely been introduced in the maintenance area.^{4, 14} The introduction of new techniques has usually been limited to the development area, and it has trickled over very slowly and not deliberately into the maintenance area. The failure to use powerful tools limits staff productivity.

Fifth, the documentation available during maintenance is notoriously weak, incomplete, and untrustworthy.¹⁰ Usually the source code is available, but sometimes the listing of that is not current. Other documentation, when it even exists, is often not relied on because it has not been kept current and because it does not communicate quickly and easily what the maintenance personnel need to know to do their jobs. This reduces their productivity.

Sixth, the staff actually doing the maintenance work is only rarely the same staff that did the development work. This difference in staffing cuts productivity, because time must be spent relearning the program or system. Often the staff is completely different, with no carry-forward of personnel at all except sometimes at the very first part of the maintenance

TABLE I—Classification of program characteristics affecting maintainability

<i>Low productivity program characteristics</i>
Out-of-date, untrustworthy documentation
Incomplete or missing documentation
No logical order (or rat's nest order) to source code
Large or long program
Much logic or heavy logic in program
Many different inputs or outputs
Variability among inputs or outputs
Arbitrary data names
Arbitrary names for routines, sections, procedures, etc.
Monolithic nonmodular structure
Source code in a low-level language
Complex task to do or difficult mathematics
No or little modularity
No hierarchical structure to program or system
Annotation buried in source code
No annotation or redundant annotation in source code
Tricky source code
Large number of switches or flags
RPG logic in non-RPG programs
Variable-length record formats
Numeric literals in source code ("magic numbers")
Heavy use of indexes or subscripts
Multiple versions of program running
Use of compressed data by program
Scattered input-output statements
Deeply nested or compounded IF statements
<i>High-productivity program characteristics</i>
Short or very modular program
Easy-to-read source code
Independent sharply defined modules
Meaningful data names
Clear functionality of the program
Documentation conformity to standards
Identification of all maintenance changes
Source code in high-level language
Meaningful names for routines, sections, procedures, etc.
Meaningful, clear annotation in source code
No complex calculations
Clear, specific data declarations

stage.⁶ This means that the personnel who know how the program or system works from doing the development are generally not assigned to the project or available to use their knowledge during the maintenance stage.

Seventh, the staff assigned to doing the maintenance work is often the least experienced and the lowest-skilled personnel when the software is application software.¹³ When the software is system software, the gap in skills and experience between the development and the maintenance personnel is usually much less. For application software, newly hired personnel are often assigned to doing maintenance work to acquaint them with the existing applications, the organization's standards, and the way the various systems fit together in operation. With much to learn, productivity is low.

EVIDENCE

Maintenance of programs and systems has only recently begun to receive serious attention in the literature.^{7, 8, 12, 13} A

large number of questions have yet to be answered. In a survey that gathered background information to characterize the nature of the maintenance burden, 20 computer-using organizations in different industries were asked to identify their most hard-to-maintain program where productivity was lowest, and their most easy-to-maintain program where productivity was highest. Then they were asked to identify what in each program made it hard or easy. These responses were then categorized and tallied. A summary of the resulting list of characteristics appears in Table I, ordered roughly by frequency of mention. A scan of the lists in Table I reveals no surprises, for all the entries have a familiar ring to them. These characteristics of programs have been known for years.

Table II is drawn from the same source as Table I, but it introduces two kinds of changes. First, the low list and the high list are combined into a single list of characteristics, all expressed from the point of view of what characterizes programs that are not easy to maintain. Second, the combined list has been broken into two parts, headed respectively C and U. The C portion runs considerably longer.

The criteria used to separate the list entries into the two portions, C and U, are fairly simple ones: Is this item determined by the computer user or the customer? Is this item determined by a third party, such as a hardware vendor? Is this item determined by the need to meet legal, security, physical safety, or auditing requirements? For example, suppose the candidate item is "Program interfaces with the operating system control blocks." The vendor of the operating system thus determines part of the character of the program. If the answer for an item is "yes" to any one or more of the questions, then the item is placed in the U portion. Otherwise, the item is placed in the C portion.

Informal discussion with other persons since the preparation of Tables I and II has led to numerous suggestions for additional items to include on the lists, but it has led to few shifts in portion from C to U or U to C, and little change in the relative lengths of the U and C portions. The most common sources for the suggested additional items have been instances or varieties of items already included, situations particular to systems software, and the characteristics of programming languages used to implement the program or system, such as JCL or FORTRAN.

INTERPRETATION

Table II can serve as a basis for the interpretation of the evidence available, given the background noted earlier. As a first step, the C portion can be interpreted as citing items controllable by those who manage the performance of the analysis, design, programming, coding, debugging, testing, documentation, and maintenance work; and the U portion as citing items not controllable by those management personnel. An example from each portion may help clarify this interpretation.

For example, Portion "C" includes the item "Source code written in low-level language." The productivity of the personnel maintaining assembly language programs tends to be lower than for high-level languages. Who or what determines that the program is to be written in assembly language? Usu-

ally it is the manager of programming, the team leader, or the assigned programmer. Sometimes it is organization policy, expressed in the organization's standards manual. But that expresses a prior determination by the manager, the team leaders, or the assigned programmers. Furthermore, the programmers and the team leaders are all answerable to the manager, who, if he or she does not approve of what the subordinates are doing, can always direct that they are to do it differently and enforce his/her directive. The manager is in the controlling position on the items in the C portion.

As a further example, Portion U includes the item "Difficulty of any mathematics involved." The presence of complex mathematics tends to reduce productivity in maintenance. Who determines the objective or function that the program is to satisfy? Clearly, it is rarely the programmer, the team leader, or the manager, since they are usually concerned with how the program is to accomplish the function the user specifies. The user or customer is the one who normally sets the requirements; and the analysis and programming staff, within that constraint, then does the analysis, design, implementation, and maintenance. In short, the fact that complex mathematics is involved could reflect the algorithm choice made by the assigned programmer, the team leader, or the manager. But it is much more likely that it is determined by the user and is not controllable by the manager.

Table II offers the basis for an additional interpretation. Nearly all of the controllable (C) items that contribute to making programs difficult to maintain originally got into the programs because someone either deliberately put them there or failed to take available action to reduce or eliminate them. It was a manager who was responsible, and one of the results is to hold down productivity.

Table II offers the basis for still another interpretation. This one can be framed first as a question: How many of the controllable (C) items characterized the program or system at the time the developer turned it over to maintenance? A scan of Table II reveals the unexpected conclusion that all the C items may be present at turnover. Given the observation noted earlier about personnel continuity, this means that the development manager creates the maintenance difficulty but is not there during maintenance to take the blame. Instead, the maintenance manager has to take the heat for what he or she inherited. The manager responsible for the mess leaves it for someone else, the maintenance manager, to live with, along with the consequent low productivity.

Table II can also be read that the maintenance manager may also be assigned some blame. How many of the C items are permitted to be perpetuated in, or are even introduced into, the program or system during maintenance? Clearly, all are candidates. In short, some maintenance managers wallow in what they inherit and the attendant effects on productivity; others dig themselves an ever deeper hole; and some act to offset or even reverse what they inherit.

A still further interpretation offers an interesting view of the process of developing programs and systems. This too can be framed first as a question: Do not the C items also cut productivity during development by making debugging and testing more costly and of longer duration? Again, the answer clearly is affirmative, for correcting for bugs and specification reinterpretations and oversights is virtually indistinguishable

TABLE II—Program characteristics adversely affecting maintenance productivity

<i>"U" program characteristics</i>	
Logical complexity of task the program does	
Number of different incoming and outgoing data items	
Variety and variability in incoming and outgoing data	
Difficulty of any mathematics involved	
<i>"C" program characteristics</i>	
Length of program	
Shortfall in documentation's conformance to standards	
Incomplete maintenance of documentation	
Weak modularity	
Nonhierarchical structuring of functions	
Low independence and low functionality of any modules	
Non-mnemonic names for data	
Non-mnemonic names for routines, sections, procedures, etc.	
Source code written in low-level language	
Annotation not meaningful in source code	
Irregular use of annotation in source code	
Tricky, convoluted source code	
Use of switches and flags	
Use of numeric literals in source code	
Level of nesting or compounding in IF statements	
Use of compressed data in program	
Number of versions of program to maintain	
Scattering of input-output statements	
Number of multiple-use indexes or subscripts	
Number of indexes and subscripts	
Vagueness and generality in data declarations	

as an activity from maintenance. Does not this mean that development managers sleep for a while in the bed they themselves made? Do they tolerate it only because they know it will not last long for them, or do they not see the connection? What is the reason for their not correcting the problem?

DISCUSSION

Taken together, the interpretations and observations noted above lead to six comments. One is that the complexity of a program or system affects the cost of maintaining it. Simple designs and simple implementations are possible for difficult tasks, and complex designs and complex implementations are possible for easy tasks. What the personnel do who are assigned to do the design and implementation, and how they do it, are major determinants of complexity.⁵ Complexity can be measured quantitatively and managed in development and maintenance, yet not separately from the matters listed in Table II.² A surprising observation from Tables I and II is not the pervasive influence of complexity on maintenance productivity, but the low recognition of it!

A second comment is about the motivation for adopting or permitting the C items during development. A review of these items in Table II indicates that most are the easy way out, or the path of least resistance. Each can be argued as being the lowest-cost alternative in the phase of the life cycle in which it is adopted or first permitted. The manager is usually taking the alternative offering the least short-run cost. In other words, transitory convenience for small cause dominates the manager's motivation, and offsetting unfavorable future

consequences—significant or even overwhelming over the long run—are usually ignored in developing and maintaining computer programs and systems.

A third comment is that such shortsightedness is rarely well regarded outside the data processing field. For example, in manufacturing automobiles, the need to do the later steps in manufacture in part determines the nature of the earlier steps. Each step is done in a disciplined manner to a formal specification. Thus, windshield windows could be made less expensively if they were made straighter, but what about the cost of bending the glass later to fit the car body? The added cost of that is greater than the cost of making the windshield glass with more curve in it. Or take the case of the machinist who believes providing the high degree of surface smoothness specified for a transmission valve to be an unreasonable use of his time. So he does not do it. Do the transmissions that get the valves he makes work as well as the transmissions that get the valves made to specification? And which transmissions need maintenance sooner? And how often must the parts the valve rubs against also be replaced? In business, the cost to be minimized is the overall cost given the performance desired. Sometimes that means using gold to conduct electricity in spite of the well-known ability of much-less-expensive copper to conduct electricity.

A fourth comment is that who is to pay the later costs is almost always considered carefully. Thus, a person who buys an available house pays to keep it heated, painted, and repaired—the builder does not. But a person who designs and builds his own house and then lives in it pays for everything all the time. In the former case, the builder is little concerned about the costs of use, since he does not pay them. He tries to minimize construction costs. In the owner-builder case, the costs of use and maintenance loom large.

Computer programs and systems fall mostly in this latter category. Yet historically the builder of the program and system have acted as though someone else—the user and the maintainer—were going to pay the costs of use and maintenance. Do not they work for the same organization, or are they indeed totally separate, fiscally and operationally?

A fifth comment is that the technical means of correcting the C items in Table II are widely known and involve largely well-understood technology.⁴ A reexamination of Table I in the light of Table II points the way. Most of the changes in normal operating practice needed are trivial, but they are departures from the traditional “anything goes” practice of most development and maintenance work. An example is the practice during debugging, testing, and maintaining of source code of permitting personnel to change the source code directly. An alternative practice is to state that a change in the source code may be made only to make the source code conform to the design as it is shown in the documentation—an application of the old and respected principle of implementation fidelity. Other examples abound.

A sixth and final comment takes the form of a question: What is the merit in making maintenance work low in productivity, difficult, demanding, and unpleasant?¹² Programs and systems will always need to be changed because organizations and their environments change. Would not our organizations be better served if *any* qualified person could quickly and easily maintain *any* program or system? And

would not our organizations be better served if the qualifications of such a qualified person were kept open enough so that finding people to fill that role would be much easier than it is now?

CONCLUSION

Long ago in most organizations, the executives at the upper levels realized that the process of making things involved tradeoffs. A very common tradeoff is making a bigger initial investment in order to obtain a bigger stream of benefits or cost reductions. As study of Table II indicates, people managing the development and maintenance of programs and systems apparently have not realized this yet. They are minimizing the immediate cost instead of evaluating the tradeoffs. In the old days, when hardware capability was expensive, that may sometimes have been justified. But now personnel costs dominate in data processing.

Somewhere in most organizations is an executive who is concerned about the benefits and costs of programs and systems over the full life cycle. Clearly, in most organizations, that executive has not gotten the word yet that modest additional cost early in development could buy savings in later development and substantial and continued savings in the maintenance of programs and systems. Or, expressing the matter another way, the low productivity that is expressed as the high cost of maintenance, and most of the problems suffered in the maintenance of programs and systems, arise because of the way managers choose to manage the development and maintenance work.

REFERENCES

1. Boehm, Barry W. “Software engineering,” *Trans. Computers*, Volume C-25, Number 10 (Oct. 1976), pp. 1226-1241.
2. Chapin, Ned. “A measure of software complexity,” *Proceedings of the 1979 NCC* (Arlington, VA: AFIPS Press, 1979), pp. 995-1002.
3. Chapin, Ned. “Software life cycle,” *Structured Software Development*, Volume 2 (Maidenhead, UK: Infotech International Ltd., 1979), pp. 17-40.
4. Chapin, Ned. “Structured analysis and design: an overview,” *Systems Analysis and Design* (New York: Academic Press, 1981), in press.
5. Chrysler, Earl. “Some basic determinants of computer programming productivity,” *Comm. ACM*, Volume 21, Number 6 (June 1978), pp. 472-483.
6. Diebold Group. *Improving the Utilization of Personnel Resources* (New York: The Diebold Group, 1979), 17 pp.
7. Ebert, R., et al., Editors. *Practice in Software Adaption and Maintenance* (New York: Elsevier North-Holland Inc., 1980), 455 pp.
8. Lientz, Bennet P., and Swanson, E. Burton. *Software Maintenance Management* (Reading, MA: Addison-Wesley Publishing Co., 1980), 214 pp.
9. Lientz, Bennet P.; Swanson, E. Burton; and Tompkins, G. E. “Characteristics of application software maintenance,” *Comm. ACM*, Volume 21, Number 6 (June 1978), pp. 466-471.
10. London, Keith R. *Documentation Standards* (New York: Van Nostrand Reinhold Co., 1975), 288 pp.
11. Lyons, Michael J. “Structured retrofit-1980,” *Proceedings of SHARE 55* (Chicago, IL: SHARE 1980), 23 pp. in Session M767.
12. McClure, Carma L. *Managing Software Development and Maintenance* (New York: Van Nostrand Reinhold Co., 1981), in press.
13. Parikh, Girish, Editor. *Techniques of Program and System Maintenance* (Lincoln, NB: Ethnotech, Inc., 1980), 289 pp.
14. Zelkowitz, Marvin V. “Perspectives on software engineering,” *Comp. Surveys*, Volume 10, Number 2 (June 1978), pp. 197-216.

Improving software testing in large data processing organizations

by M. A. HOLTHOUSE

The Analytic Sciences Corporation
Reading, Massachusetts

and

C. W. LYBROOK

Chemical Bank
New York, New York

ABSTRACT

Software testing is one of the most critical tasks performed by a large data processing organization. Testing is important in the development of new systems, but it may have an even greater impact on the maintenance of the production systems. In spite of this, testing is rarely approached in the same disciplined manner as are other software production activities. Perhaps as a result of this situation, an organization can often achieve significant improvements in both software testing effectiveness and efficiency through a relatively low-cost investment in testing methodologies, tools, and techniques. This paper describes just such an investment undertaken by the Information Services Group of Chemical Bank. Particular emphasis is placed on how this program for testing improvement was implemented, in addition to what it consists of. Finally, results of the program to date are presented and analyzed.

INTRODUCTION

Major advances have been made in recent years in improving the process of developing complex data processing management information systems. Under the general heading of software engineering, techniques such as structured programming, structured design, and formal requirements analysis have gradually been introduced in Chemical Bank's Information Services Group (ISG) as well as in other large data processing organizations. In general these techniques have gained wide acceptance, and their use has helped produce computer systems of superior quality at lower total life-cycle cost.

Yet one critical aspect of software development and maintenance is often overlooked, or addressed only indirectly: software testing. Testing is rarely approached in the same disciplined manner as other development activities. It is often

the first task to be shortchanged when budgets get cut or schedules begin to slip. Improved analysis, design, and programming techniques can help reduce the amount of error correction necessary during the testing process, and as such, complement thorough testing. Software testing increases in importance for systems undergoing maintenance and minor enhancement, since many of these systems were developed without the benefit of these advanced software engineering techniques. From a management viewpoint, the testing process provides the last (and often the only) opportunity to influence the quality of a system (or changes to a system) about to be placed into production. Hence a well-defined, measurable approach to software testing can provide a major source of understanding and control to top data processing management.

The present lack of attention to software testing in large data processing organizations is not a result of the lack of technology. Over the last decade, software testing has been the subject of intense activity in the research community, and numerous papers, books, and conferences have been devoted to testing methods, techniques, and tools.^{1, 2, 3, 4} Some of this technology has been applied and evaluated on large projects, mostly in defense and related applications.^{5, 6} Even in these communities, however, a recent study reports that 85 to 90% of software engineering project managers consider that testing and reliability continue to be among their most critical problems.⁷

In this paper we will discuss in detail why and how Chemical Bank addressed the process of software testing. In the case of any technological change involving people, it is much easier to define what should be done than how to go about implementing the changes. Accordingly, we will devote more of our attention to the latter. We will present our approach both to implementing the changes and maintaining them as an integral part of the way ISG operates. Finally, although we are not yet finished with the job of improving software testing at Chemical Bank, we will present an analysis of some of our results and progress to date.

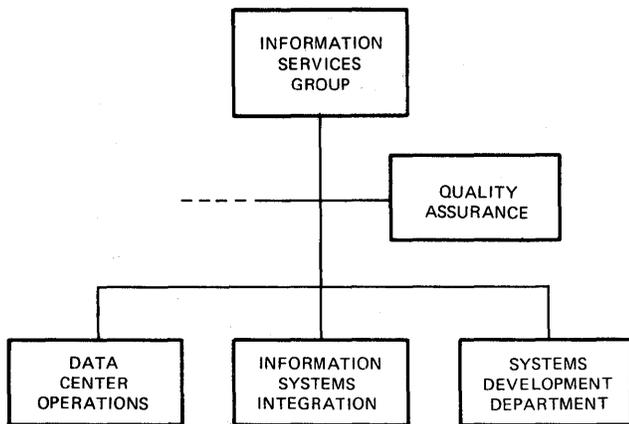


Figure 1—Information services group organization

GOALS FOR TESTING IMPROVEMENT

Chemical Bank had one primary motivation for making an investment in improving software testing: reducing the cost of providing information services. The 700-person Information Services Group has a 1981 data processing and communications budget of over \$65 million, of which approximately \$16 million is devoted directly to software development and maintenance. (This is not the total Chemical Bank data processing budget.) Preliminary analysis shows that around 25% (\$4 million) is earmarked for testing activities, not including overhead expenditures to provide a dedicated IBM System 370/3032 and related support staff. Hence, if one could increase the testing efficiency of ISG (both of people and hardware resources) by as little as 10%, substantial savings could be realized. In reality, of course, these savings would not show up as reduced expenses, but as resources available for new systems development or faster response to requested changes. Hence measurement of increases in testing efficiency would have to be made by examining the before and after testing processes used, rather than on a bankwide, bottom-line basis.

Even more important was a desire to reduce the number of software errors encountered in production. Production errors can be extremely costly to repair and rerun, and the cost of lost services to the bank can be very significant. Obviously, these testing effectiveness issues are related to the efficiency factors described above in that resources spent detecting and recovering from errors in production could be much more effectively used during testing. Studies have shown that an error costs between five and ten times as much to fix when discovered in production as when discovered during testing, even excluding the indirect costs of production problems.⁸

A more detailed analysis of existing testing practices highlighted a major link between efficiency and effectiveness. Much testing was being accomplished using extremely large files, usually copies of old production files. In addition to being extremely inefficient (especially when requiring use of tape files, special output devices, etc.), such tests were very ineffective. A single production run will generally exercise only a small portion of a large system's functions; hence,

errors in untested functions are left to be discovered later. In hindsight, many software errors seemed to have been easily avoidable if only a certain type of transaction or set of conditions had been tested.

It would be ideal to monitor progress in improving testing effectiveness by tracking software error rates over time. However, such a measurement is extremely difficult to make. These measurements are subject to enough uncontrolled environmental conditions as to make almost any improvement or degradation attributable to outside factors. For example, does a constant error rate reflect no testing effectiveness improvement, or substantial improvement coupled with an increase in the software change rate? Hence, as in the case of efficiency improvement, we established process-related goals (reducing use of production files and increasing testing thoroughness), supported by before and after analyses on specific systems.

THE TESTING IMPROVEMENT PROJECT

The Quality Assurance (QA) staff of ISG (see Figure 1), working with The Analytic Sciences Corporation (TASC), took primary responsibility for addressing the testing issues discussed above. As a first step, a project was undertaken to define and implement a standard testing methodology to be used at Chemical Bank. This project consisted of the development of detailed standards, procedures, guidelines, and suggestions for approaching software testing in a disciplined, consistent manner. The standard testing methodology provided the basic framework for directing and implementing two other critical pieces of the overall testing improvement effort: staff training in testing techniques and strategy, and use of automated testing tools.

The Standard Testing Methodology

The standard testing methodology is closely integrated with the project life cycle (PLC) framework in use at Chemical Bank. As a project moves through the various phases of the development or maintenance life cycles, the program guides and identifies testing activities to be performed and possibly documented. The review points established in the PLC provide the opportunity for quality assurance to evaluate testing plans and progress at critical life-cycle milestones.

Figure 2 shows the basic elements of the program and their relationship to the various PLC phases; specific review points for each element are identified in Table I. The two most critical elements are the testing objectives (analysis phase) and the testing strategy (design phase). Together these elements identify what is to be tested and outline a basic plan for accomplishing the testing. The testing strategy is built from a variety of test phases that test a particular piece or pieces of the overall system, and which, according to a planned sequence, culminate in a formal acceptance test phase.

The other elements of the testing methodology associated with design and implementation phases relate to a single test phase and begin with a detailed specification of the type of testing to be performed in the phase. During implementation, this test phase specification is gradually refined to specify

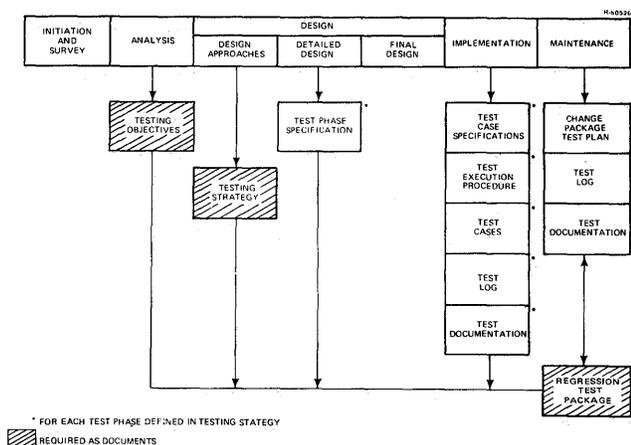


Figure 2—Basic test plan elements

distinct test cases, a detailed procedure for test execution, and the actual test data for each test case. As the tests are executed, their results are evaluated and logged and finally documented as appropriate.

Testing in the maintenance cycle (regression testing) is built around the idea of a regression test manual. Basically, this manual serves as a repository for test objectives, strategies, procedures, data, and results defined during development that can aid in retesting the system when it undergoes change. When each change package is implemented, a change package test plan is followed to run selected tests from various test phases against the system. These tests may be drawn directly from the ones documented in the regression test manual or developed anew, in which case the regression test manual is updated.

It is important to note that although all of these elements are described as documents, some may not be necessary in written form for a particular project. During the development of the testing strategy, a decision is made about which elements are to be written and provided as actual deliverables with the system. *Regardless of whether the elements are documented in written form, they are all considered and applied to every software development and/or maintenance effort, at least informally.* In these cases, a verbal summary of the key information related to each element is presented at the appropriate review point.

Techniques and Training

Symptomatic of the cursory attention that testing usually receives in any software organization is the lack of training courses devoted to testing. Organizations spend hundreds of thousands of dollars teaching or sponsoring courses in systems analysis, design, programming, and documentation, but very little on teaching people how to test. The training at Chemical Bank centers around two kinds of activities:

- Management—planning, organizing, and allocating resources for the testing process

TABLE I—Testing methodology elements

Element	Phases(s)	Review Points
Testing objectives	Analysis	Proposal review
Testing strategy	Design	Internal design review
Test phase specification*	Design	Critical design review
Test case specifications*	Implementation	In-process review
Test execution procedure*	Implementation	In-process review
Test cases*	Implementation	System certification
Test documentation*	Implementation	System certification
Change package test plan	Maintenance	Maintenance cycle and change package reviews
Regression Test Manual	Development Maintenance	Internal and critical design reviews, system certification

- Testing techniques—developing test strategies and specifications, generating test data, and executing and evaluating tests

Exactly which people in an organization need to be introduced to which elements is a function of the organization's structure. At Chemical Bank, systems are developed and maintained through a matrix type of structure, and application groups in the Systems Development Department (SDD) work closely with user or client groups inside another bank division. In this structure, Table II identifies specific training courses and seminars addressed to particular people and concepts.

TABLE II—Training activities

	User	Developer			
		Top Management	Project Managers	Senior Analysts	Programmers
Management					
Standard testing methodology	X	X	X	X	X
Cost/benefit tradeoffs	X		X		
Development testing		X		X	X
Maintenance testing				X	X
Testing techniques					
Strategy design				X	X
Test specification	X			X	X
Test data generation				X	X
Test execution/evaluation				X	X
Building regression tests				X	X

Tools

The most important point to remember when discussing automated testing tools is that they are automated, not automatic; that is, an individual must use them for them to be effective. For maximum impact, tools must be made an integral part of the testing process, and training in their use must be institutionalized as well. Hence, although they are important enough to address separately, they are closely tied to both the standard testing program structure and the training activities described earlier.

The most significant testing tool in use at Chemical Bank is the TRAILBLAZER™* software analysis system. This tool provides both summary and detailed information on how thoroughly tests exercise the logic of a COBOL, FORTRAN, or PL/I program. In addition, it highlights and categorizes the severity of changes to a program in maintenance and identifies whether the changes have been thoroughly tested. This tool is used by project managers and QA analysts to assess testing thoroughness quickly, and it also guides the analyst or programmer in generating more extensive test data. This tool will be discussed in more detail below.

A number of other general tools are also part of the testing program and are used to improve the efficiency of test data generation and test evaluation. These fall under the general categories of

- Data generators
- File/database editors
- Formatted file/database output utilities
- File/database comparators

Some of these tools are used for other purposes as well, but their use in testing is explicitly identified in the standard testing program and taught in the training courses. Finally, with an open-library concept, we encourage project managers and team leaders to share specific, handy tools that they have created to help their own testing but that (possibly after some re-tailoring) may prove useful to other projects as well.

IMPLEMENTING THE PROGRAM

Implementing a program such as the one described above is a difficult task because it involves effecting basic behavior changes in a large number of individuals. Hence each individual needs to be sold on the program; he or she needs to get something out of it. To some extent, the goals of the organization in establishing the program differ from the interests of each individual, as summarized in Table III. Some of these issues are complementary rather than conflicting and thus

Table III—Primary testing concepts

Individual	Organization
Short term	Long term
Efficiency	Effectiveness
Noninterference	Quality assurance
Tools and training	Standardization

*™TRAILBLAZER is a trademark of The Analytic Sciences Corporation.

relatively easily managed (efficiency versus effectiveness, and standardization versus tools and training). Other concerns are tougher to deal with, notably the short- versus long-term orientations, and the need for quality assurance versus a desire for noninterference on the part of system developers or maintainers. To manage these conflicting goals, a key consideration in implementing the program at Chemical Bank was that the people ultimately responsible for the program were part of Quality Assurance, not systems development. By concentrating on selling the efficiency-related tools and training as an integral part of the program, ISG was able to address the short-term concerns of system development and the user groups and still insure that the longer-term organizational goals of standardization and independent quality assessment were met. It is our view that the assignment of organizational responsibility is a vital decision that must be made before implementing such a program.

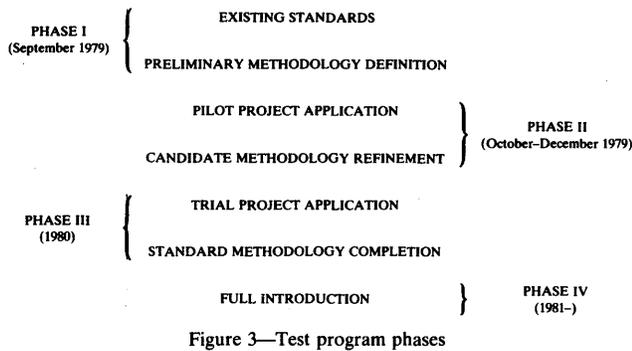
These insights into program implementation may seem obvious, but we did not fully understand many of them until we were well into the process. Likewise, the program itself has changed in numerous ways as a result of the implementation experience. The evolution of the program was in fact planned as part of a four-phase implementation strategy, which is outlined in Figure 3 and described below.

Phase I

The important goals of this phase were to make the overall ISG organization aware of the efforts to improve testing, understand key issues of concern to various groups, and identify and tap interested parties for useful input. In particular, we wanted to get people from all levels of systems development involved in the project. A questionnaire on testing practices and suggestions was distributed and returned by more than 80% of the SDD professional personnel. Along with a number of formal and informal interviews, this information helped identify the ways the program could be tailored to help SDD individuals as well as Chemical Bank as a whole. Feedback on existing standards, training, and documentation related to testing was also valuable in helping capitalize on currently available resources and avoid repeating mistakes. The output of this process was an initial definition of the testing program, similar in form to the description provided above.

Phase II

With a preliminary version of the program in hand, the next step of the process was to try it out briefly with pilot projects. Several such projects were selected, one in each major application area, and one in major life cycle phase (design, implementation, and maintenance). Obviously, the full program was not used in each case. The engagements were limited to three months. The basic technique was to have an outsider actually assist in the testing of the three projects, applying the elements of the program as appropriate. This promise of real assistance made the project managers receptive to the idea, even though they would have to spend some time bringing the tester up to speed on his project.



From our point of view, we were able to identify some major shortcomings in the proposed program and to develop some examples of its use on real projects. Even more important, we were able to solve a few pressing short-term problems (“I never knew you could do that!” was a typical response), and, through these successes, albeit limited, we began to develop some satisfied customers who saw the benefits some new testing tools and techniques could bring. The Candidate Testing Program was then produced, incorporating the necessary revisions and including the examples of program elements as-applied to the pilot projects.

Phase III

In Phase III, the Candidate Testing Program was described throughout ISG and presented to selected user groups as well. Initial versions of the various training courses were developed and used to introduce members of trial projects to the program. In these projects outside assistance was limited to consultation and occasional tool development or tailoring, and the efforts were undertaken over a longer term (three to six months), encompassing either major portions of the development cycle or extended maintenance activity. It is worth noting that we were unable to assist (directly) all the managers who requested that their projects be designated as trial projects. We did, however, hold training courses for all who were interested, and several projects began applying the testing program elements on their own.

Through these mechanisms, we were able to solve some longer-term problems on larger systems and cultivate organizational support. The educational efforts began to take hold, and the use of the various automated tools increased substantially. Finally, the final version of the testing program was produced, taking advantage of all the trial project experience and feedback generated through the initial training courses and seminars.

Phase IV

In addition to putting the various procedures and activities related to the ongoing maintenance of the testing program into full operation (see below), the primary goal of this current phase is to bring all critical maintenance systems into compliance with the program. Basically, this means devel-

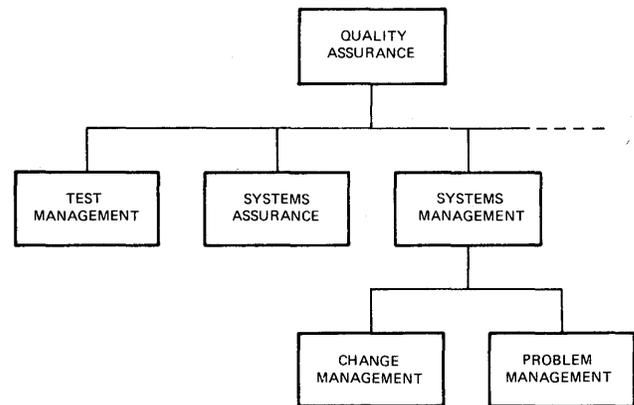


Figure 4—Quality assurance organization

oping a regression test manual for each system’s current practice, then working to improve either the efficiency or the effectiveness of their maintenance testing procedures. This activity requires a significant investment in effort. Nevertheless, for the critical production systems of the bank, this investment will pay dividends many years into the future. In many cases, the break-even point is reached after only a few months.

ONGOING SUPPORT

Any software engineering methodology requires regular maintenance to remain effective, particularly in terms of ongoing training and in adjusting and extending the technology to handle changes in the organizational structure or environment (e.g., distributed data processing). The testing program also requires continued attention to insure that testing does not suffer as a result of project schedule slippage or reductions in project personnel (budget cuts).

Hence, Quality Assurance has defined three progressive levels of increasing involvement with the testing of individual systems in development or maintenance:

- Testing program review
- Testing coverage audit
- Independent testing

Referred to as certification levels, these procedures provide increasing organizational assurance of system reliability through third-party review.

Testing Program Review

This is the certification level for most systems at Chemical. It uses the project life cycle review points to verify adherence to the standard testing program. These reviews are handled by systems assurance in the development cycle, and by change management in the maintenance cycle (see Figure 4). Test management provides technical assistance to both project teams and the QA review teams in preparing for these reviews.

Testing Coverage Audit

At this certification level, QA test management uses the TRAILBLAZER™ tool (including its change analysis feature for systems in maintenance) to assess independently the thoroughness of test data provided by the systems developers. Thoroughness standards ranging from 75% to 95% coverage of (changed) program logic are established as part of a system's testing strategy or regression test manual and agreed to by QA, the developers, and the user(s). If these standards are met, QA certifies the system; otherwise, the detailed reports showing unexecuted logic are returned to the project team for additional testing.

Since a QA analyst, with only a cursory knowledge of the system to be tested, can quickly perform a testing coverage audit, this certification level is relatively inexpensive. This technique can be applied simply and inexpensively to contractor-developed software as well as in-house products. In many cases, the requirement for a specific thoroughness level can be written into contracts with software suppliers. Basically, this technique leads to improved testing quality, for two reasons:

- The awareness of the project team that an activity (QA) is evaluating the thoroughness of their testing improves their own tests.
- Systems that are not thoroughly exercised are returned for further testing.⁹

Independent Testing

This highest certification level transfers system testing responsibility from the development organization to QA. It is relatively costly, since QA analysts must understand the applications area of the system under test in order to do an effective job. In an organization with a high volume and diversity of applications being developed and maintained simultaneously, this approach can only be justified for a few, extremely critical systems.

Use of this technique has the advantage of providing a high level of assurance in the correct operation of a system, *at least to the extent that its functions are understood by the QA analyst*. It also provides an extremely good vantage point for general systems assurance activities, since the analyst is in a position to know exactly where the system stands at any time. When test data are developed in advance, specification changes made in later stages are obvious; indeed, the activity of explaining to an independent party enough about the system that tests can be developed generally encourages better system specification.

EXPERIENCE AND RESULTS

As is clear from the discussion above, we are not yet finished with the job of improving testing at Chemical Bank. Nevertheless, significant progress has been demonstrated. Some notable examples are presented in this section.

The first system to use the testing program from beginning to end (a 40,000-line assembly language, on-line application), went smoothly into production. Moreover, this system had no problems requiring software changes in its first six weeks.

The testing program standards and guidelines have been included in software development contracts with outside vendors and used to define the testing strategy and regression test manual as deliverable items.

A large, complex system in maintenance is being tested, using 700 to 20,000 record test files instead of 750,000 records, with no decrease in testing thoroughness.

The standard test set for another large system in maintenance was found to be exercising only 21% of the system's logic. The test set has now been extended to over 60%, and automated output comparisons have reduced paper usage and have increased early detection of errors.

Perhaps even more important in measuring the success of the testing program is its reception and acceptance by the SDD staff. Although top management was behind the program from its inception, many systems developers were either noncommittal or hostile early on. As the program moved into its third phase, however, we were unable to respond to all the project managers who wanted assistance in applying the program to their systems. As we began to present training courses for project managers, programmers, and users, interest increased, and our hoped-for result occurred: many project managers took the program themselves, tailored it to their systems, and began using the techniques with no coercion and only occasional assistance from Quality Assurance.

For several reasons, acceptance of the program has been much slower in maintenance systems. Not only are personal egos heavily involved in these systems, but it often takes a concerted effort over an extended period to obtain a noticeable improvement in such systems. Gradually, however, maintenance project managers are beginning to believe that results are achievable, and a little healthy internal competition to raise testing coverage figures reported by TRAILBLAZER™ has occurred.

At present we are not able to make any quantitative statements about declining error rates. Although most people involved with our trial efforts would agree to qualitative statements about improved testing effectiveness, we cannot draw impressive graphs showing dramatic decreases in problem occurrences. Even if we were to try, the data presented would be subject to considerable qualitative speculation, as suggested earlier. On the other hand, our experience with several maintenance systems has shown that software errors in production have ceased to be ones which could have been easily found during testing, and month-end processing errors are no longer the regular occurrence they once were.

We are now tracking detailed change and problem rate information for certain critical systems with the goal of providing realistic analyses of system reliability performance and trends. Ideally, such data will form a baseline against which to measure pilot or trial project performance in improved testing; but changes in project personnel, the user environment, or other concurrent software engineering improvement programs could still make such analyses open to debate. Realistically, moreover, in most organizations a proposal to establish baselines first would probably be met with the objection that "we already know the XYZ and ABC systems are problems; we want to do something about it, not measure it." The real tradeoff may thus be whether to devote scarce resources first to measurement and then to improvements that may be subse-

quently quantifiable, or first to critical improvements and later to measurements for tuning and maintenance of the program.

We have chosen the latter course, and we believe our decision is the correct one. It has made the programmers' jobs easier in a number of ways, increased the user confidence level, and, most important, has given all levels of ISG management better visibility into and control over an extremely critical part of their business operations.

REFERENCES

1. Hetzel, W. C. *Program Test Methods*. Old Tappan, New Jersey: Prentice-Hall, 1972.
2. Miller, E. F., Jr. *Tutorial-Program Testing Techniques*. Long Beach, California: IEEE Computer Society, 1977.
3. Myers, G.I. *The Art of Software Testing*. New York: Wiley-Interscience, 1979.
4. ———. *Digest for the Workshop of Software Testing and Test Documentation*. Ft. Lauderdale, Florida: IEEE Computer Society, 1978.
5. Holthouse, M. A., and M. J. Hatch. "Experience with Automated Testing Analysis." *Computer*, 12 (1979), pp. 33-36.
6. Sorkowitz, A. R. "Certification Testing: A Procedure to Improve the Quality of Software Testing." *Computer*, 12 (1979), pp. 20-25.
7. Thayer, R. M., A. Pyster, and R. C. Wood. "The Challenge of Software Engineering Project Management." *Computer*, 13, (1980), pp. 51-59.
8. Boehm, B. A. "Software Reliability—Measurement and Management." *The Abridged Proceedings from the Software Management Conference*. Los Angeles: AIAA, 1976.
9. Sorkowitz, A. R. *op. cit.*

Compiler validation—An assessment

by GEORGE N. BAIRD and L. ARNOLD JOHNSON

Federal Compiler Testing Center
Falls Church, Virginia

ABSTRACT

The Federal COBOL Compiler Testing Service (FCCTS) was transferred from the Department of the Navy to the General Services Administration (GSA) in May, 1979. GSA renamed it the Federal Compiler Testing Center (FCTC) and expanded its functions beyond that of COBOL compiler validation. This paper discusses (1) the effect of the FCCTS/FCTC in the area of COBOL over the past eight years, mainly in the area of the standardization of COBOL; (2) the quality of COBOL compilers today contrasted with those of six and 10 years ago; and (3) the effect of this work on the ADP procurement process and ultimately the end user. Today's COBOL 74 compiler, on the average, is far superior to its predecessors, developed for COBOL 68.

INTRODUCTION

The Federal Compiler Testing Center (FCTC) is a Federal data processing center in the Office of Software Development under the Automated Data and Telecommunications Service of the General Services Administration (GSA). The FCTC represents the older Federal COBOL Compiler Testing Service, which was housed in the Department of the Navy, with a much expanded function, which is necessary for support of the GSA in the area of software management and the enforcement of procurement regulations regarding software acquisition.

Since July 1, 1972, all COBOL compilers brought into the Federal Government must implement one of the four levels of Federal standard COBOL.¹ The GSA has established Federal property management regulations² and procurement policies requiring that all compilers (for which there is a Federal standard) brought into the Federal government be tested for compliance with their respective standards.

The FCTC has the responsibility for the operation of a governmentwide compiler testing center. This responsibility is discharged by the FCTC through the implementation and maintenance of the COBOL, FORTRAN, and BASIC compiler validation systems. (A validation system is a comprehensive set of routines to test programming language compilers for compliance with Federal and national standard programming language. *Compiler*, as used in this paper, refers

also to interpreters and other language processors that process a source program to the point that it can be executed.

This paper addresses the experience gained in the last eight years in the implementation of compiler validation systems, the operation of a facility for generalized compiler testing, and the effect of the testing center on the computer industry in general.

THE FEDERAL COMPILER TESTING CENTER

In January 1979 the Office of Management and Budget ordered the Federal COBOL Compiler Testing Service to be moved from the Navy to the General Services Administration. This was accomplished by May 1979. GSA changed the name of the organization to the Federal Compiler Testing Center (FCTC) and at the same time expanded its functions to include the validation of programming languages and operating system software other than COBOL, research into software development techniques/tools, and test and acceptance criteria for newly developed or updated software.

The Federal Compiler Testing Center provides a major tool to Federal data processing administrators for achieving substantially greater compatibility and interchangeability among COBOL, FORTRAN, and BASIC programs and automated information systems. As a centralized service, it reduces the cost of validation testing by individual agencies and the duplication they might create. Performing a rigorous audit of standards conformance contributes in turn to lower source program conversion costs, practically eliminates programmer retraining, and makes feasible the sharing of software among government agencies.

More important, perhaps, from the viewpoint of the computer industry at large, is the introduction of a uniform, substantive method of quality assurance for systems software. From this starting point may grow standardized quality testing for other software components and more meaningful software standards in the computer industry.

COMPILER VALIDATION

For purposes of this discussion, the term *compiler validation* refers to the process of testing a completed software product

(in this case a compiler) in its operational environment. The validation systems used must be capable of functioning in a variety of dissimilar hardware and operating systems, the staff performing the validation is in no way involved in the development or the maintenance of the products being tested, and the result of a validation could affect the eligibility of the product for procurement by U.S. government agencies. This environment imposes unusual and stringent requirements on the portability of the validation systems and the auditability of the implementation techniques used during the course of a validation.

The purpose of validating a compiler is to test a compiler's acceptance of standard language syntax, and, where unambiguous, language semantics. The latter, of course, is a more difficult area, because we have not developed the appropriate mechanisms for precise semantic definitions for programming language specifications. Compiler validation systems do not evaluate the implementation techniques used for the compiler or its quantitative performance characteristics.

The benefits of compiler validation include a higher degree of source program compatibility between systems, resulting from reduced source code conversion costs. In addition to the protection of an organization's programming investment, the use of compilers that conform to their language specifications (in most cases a national standard) will reduce the need for retraining programmers as they go from system to system. Programmers can concentrate on problem solving and not become involved in specific dialects or idiosyncrasies of different programming language compilers. The principal purpose of a language standard is to provide a disciplined, predictable, efficient framework for software development. A compiler must perform according to that standard if the goal is to be met. The validation of a compiler is required in order to determine the degree to which it conforms to its language standard (specifications). A compiler validation system represents a working, executable interpretation of the programming language being tested.

From the experience gained by the FCTC over the past eight years, it seems that the most successful approach to compiler validation has been functional testing—the process of executing a series of tests against features of a compiler or software product. The FCTC expands on this definition slightly by producing tests combining the use of two or more functions to determine whether the interaction of those functions has any effect on the results of each of the independent functional elements. The best way to describe the techniques used by the FCTC in designing/developing compiler validation systems is a combination of functional and interaction testing.

Functional and interaction testing can be used to test characteristics of software such as performance and integrity, but they are most commonly used for specification testing. Thorough functional testing requires a complete test plan, systematic controls and approaches to the testing effort, and objective measurements of test results. The thoroughness of testing is measured, in terms of number of functions and interactions tested, and the revision and evaluation of test specifications are relatively simple. Functional/interaction testing also offers a high degree of visibility to a customer and is apt to be well understood by that customer. This is important in the operation of a governmentwide operation responsible for the vali-

dation of compilers. All parties involved have a reasonably good understanding of the testing process and the relative standing of a tested compiler or software product.

The most common disadvantage cited in regard to functional testing is that it is generally impossible to insure that all features or decision points of a software product are tested. With regard to compilers, it is certainly true that it is not practical to test all possible combinations of language elements and data types. This has not been found to be a serious shortcoming, since it is certainly possible through interaction to test all reasonable combinations. What is "reasonable" is admittedly a subjective judgment, but such subjectivity regarding test limits is hardly unique to software testing.

A more serious problem, from the FCTC experience, is that functional testing can be only as good as the specifications being tested. Thus, it is an unfortunate fact that many important features of a compiler cannot be tested because the pertinent language specifications are either ambiguous or left to the discretion of the implementer. As a result, one of the major fallouts from producing a compiler validation system is that when vendors first get a chance to review the product, there are generally a large number of requests for interpretations generated for the body responsible for interpreting the language specifications.

VALIDATION SYSTEM DESIGN CONSIDERATION

A compiler validation system that is to be used on a variety of different hardware types must have a high degree of portability. This has been successfully accomplished by producing system-independent validation systems and an executive routine that is used to tailor or edit the programs making up the validation system for a particular computer system. The medium used by the FCTC to distribute machine-readable copies of compiler validation systems is magnetic tape—specifically, 9-track, ASCII,³ 1600 BPI and 2400-character blocks (30, 80 character source images). This has been found to be the most acceptable format used to date. There have been only a few instances where the tape had to be converted to some other format before it could be used.

IMPLEMENTATION CONSIDERATIONS

To implement a validation program or any program on a foreign computer system, the following must be accomplished:

1. The character set used may have to be converted if the validation hardware/software does not support the ASCII code.
2. All external references in the source programs making up the validation system must be resolved. This may require modification of the source programs; alternately, it could be deferred until execution time and handled through job control language or operating system control statements.
3. Operating system control statements must be produced that will compile and execute each of the source programs.

Additionally, the user must have the ability to make changes to the source programs—i.e., delete statements, replace statements, and add statements.

4. The programs must be compiled. Any statements that are not syntactically acceptable to the compiler must be modified or deleted so that a clean compilation takes place and an executable object program is produced.

5. The compiled programs must be executed. Any execution time aborts must be resolved by determining the cause of the abort. After deleting or modifying the particular test or COBOL element that caused the abnormal termination of the program, Steps 3 and 4 must be repeated until a normal program termination exists.

DESIGN PHILOSOPHY

The basic premise used in the design of a compiler validation system is that of small building blocks which when taken collectively become more complex and represent the total of the language being tested. The primitive elements of the language being validated must be defined. (It is safe to say that if a compiler does not support these primitives or incorrectly implements them, then validation of that compiler will be meaningless.) These language primitives will be used to define the source code that will support specific tests.

The programs in a validation system that test the primitive language elements are compiled and executed first during the validation process. This should insure the correctness of the implementation of the primitive language elements. If problems exist, it may be questionable to continue the validation, since these primitives are both the building blocks of the system and the foundation upon which the rest of the system is built. There are, then, three types of source code contained in a validation program.

First, there is what can be called boilerplate source code. This code is present in each of the validation routines and is generally the same in each routine. This source code handles housekeeping functions, which include initiating and terminating an execution time report produced by the program and generalized code used by the program in establishing the truth value of the results of each of the tests.

Second, there is test support source code for each test in the program. This source code is necessary for initiating and setting up conditions prior to the test, for providing supplementary/support code used during the test, and finally for providing the code for evaluating the results of the test. Information regarding the test is passed to the boilerplate code, which then produces a line entry on the execution report for that test.

Finally, there is the test code that represents the language element being tested. This will always be the most complex or highest-level code contained in the program. The support or boilerplate code is always made up of the primitive language elements and is the least complex code in the program. The tests included in a program go from simple to increasingly complex.

The programs making up the validation system are designed so that a single program does not attempt to test more than one language element or feature. The reason for this is to

implement the validation system as simply as possible. If a program tested two language elements, and one of them was not supported by the compiler being validated, then the program would have to be modified in order to successfully complete the test for the language element that was supported. If, in the case above, a separate program had been used to test each of the two language elements, then the program that tested the language element not supported could be discarded with little or no effort.

In the testing program produced for a language element/feature, the tests should begin by using the simplest form of the language element/feature and grow progressively more difficult until the testing is complete. It may be necessary, depending on how many specific tests are required to test a language element/feature satisfactorily, to produce more than one program for a given language element/feature. Again, the first program should contain the simplest tests, and the more complex tests should appear in the second or third programs. This permits the testing of the simple forms of a language element/feature even though the full language element/feature may not be supported.

The test programs should be designed so that when the source code in a given test is rejected by the compiler or causes an execution time abort, the code can be easily deleted in order to resume testing. The solution to this problem used by the FCTC is to include source code following each test that will be executed if the test was deleted and indicate on the execution report the test was deleted. For example,

identify test	(Support code)
initialization	(Support code)
test	(Test code)
branch to next test	(Support code)
test delete	(Support code)

If the “test” code and “branch to next test” code are eliminated, then the “test delete” code is executed, which causes the execution report to document the test as having been deleted.

Programs should be self-checking if possible. (There are some cases where this is not possible and visual checking is required, but these are few and should be kept to a minimum.) After the execution of each test, the program should internally check the results of the execution of the test code and determine whether the results are within an acceptable tolerance. With large validation systems where the number of tests are in the tens of thousands, this is a must if the validation is to be accomplished in a reasonable amount of time and with a minimum of judgmental error.

The execution time report produced by each program making up the validation system must provide as much information as possible, including the number of tests in the program, the number of tests that passed, the number of tests that failed, the number of tests that require visual checking, and the number of tests that were deleted. The execution report should identify each test and point to the location of the test within the source code. For each test failure, the report should contain the results expected by the validation system and the results obtained by the compiler, to be used in further analyzing the failure.

TEST SELECTION/DEFINITION

The most difficult problem in designing a compiler validation system is that of selecting specific tests. The number of tests must be finite, whereas the number of possible tests is roughly infinite. The number of tests must be manageable and represent a respectable cross-section of the language being validated.

The approach used by FCTC is one of attempting to insure that the validation system will test the limits of the compiler in such a way that programmers, for the most part, will be comfortable working within these limits. Programs that are designed to fall within this category will be highly portable across systems and allow users to reap the benefit of programming language standards.

Where there are defined limits in the language (e.g., size of numeric variables, length of programmer-supplied words), they are tested at both the maximum and minimum permitted by the language specification—for example, the following COBOL statement:

ADD A TO B

Where A and B are variables whose characteristics can be:

- Signed or unsigned (can be positive/negative value or absolute value)
- Integer or decimal
- Synchronized or unsynchronized (within word boundaries)
- Binary or character data representation
- 1 to 18 digit positions in size
- Direct or indirect reference
- Qualified or unique names
(1.5 billion possible test cases)

A manageable number of tests (say 100) must be designed to cover the testing of as many as possible of the 1.5 billion possible combinations of tests. As many combinations of attributes as possible must be used in the tests, to the point that the designer feels that the COBOL ADD statement has been adequately tested.

Myers⁸ discusses boundary-value analysis for software testing much like the FCTC guidelines. He goes one step further in that he tests beyond the specified limits to determine whether the software product handles the situation correctly. Thus far the FCTC has not built a compiler validation system that does negative testing or goes outside the defined limits of the language specifications. It is likely that future validation systems will have tests to determine whether a compiler permits programs to go beyond the limits established by programming language standards. Permissive compilers can have a costly impact on software sharing and software conversion.

SYSTEM IMPLEMENTATION— EXECUTIVE ROUTINE

Compiler validation systems must by their nature be highly system-independent. Therefore the systems produced and used by the FCTC are generalized products, almost in the same respect that an operating system is generalized when it

is received by a customer. Prior to the validation process a generation process must take place in order to produce a validation system tailored to the software/hardware environment in which the validation will be performed.

The generation process is accomplished by using an executive routine (written in the same language as is being validated) that is provided with the validation system. The executive routine resolves implementer names in the source code of each of the programs and generates operating system control statements necessary to compile and execute each of the programs. The initial generation, including preparing all the input to the executive routine, can be performed in advance of the validation. Once the executive routine inputs have been verified as being correct, it is a relatively simple process to generate a working validation system suitable for the validation environment. For the ongoing process of subsequent validations it is also easy to work with new releases/versions of the validation system, since only the generation process need be performed to have a new working system.

The executive routine is a software interface between the computer system being validated and the person conducting the validation. Although a suite of programs, through some unwieldy process, could be manually implemented on any given system, this method does not lend itself to the validation process. Using this method, the implementation of the validation system on two different computer systems could vary drastically and produce differing results. As in any method of software testing or validation, consistency of testing and controls for the process is of the utmost importance. In the case of testing different compilers with the same validation system, the implementation of the programs making up the validation system must be the same if the results of the validations are to be used in a meaningful comparison.

The executive routine provides a tool necessary for insuring consistency and providing the controls necessary to perform equitable validations. In the case of the FCTC it also provides a generalized interface to any system/compiler to be validated. FCTC personnel need only be familiar with the executive routine in order to select programs for compilation/execution, make updates to the programs being selected for compilation/execution, select or not select source code that can be optionally included, and modify the operating system control language generated for each program. A hard-copy audit trail is provided that identifies what source programs were selected and what changes, if any, were made to each of the programs. This audit trail and the execution reports from each of the audit routines are used in preparing the validation summary report that sums up the results of the validation.

FUNDING COMPILER VALIDATIONS

The question of who pays for a compiler validation is addressed frequently. The current government procedures require that the requester, generally a computer vendor, pay for all direct costs associated with each validation. This includes salaries, travel costs, and the production of the validation summary report. The rates being used for salaries will allow the center to recover approximately 55% of its total cost if all annually scheduled validations are performed. If any valida-

tions are cancelled, recovery rate is less than 55%; if any additional unscheduled validations are performed, it can be greater than 55%. The average cost of a validation for the period 1978–1980 was \$3000 for labor plus travel and other direct costs. The average number of hours spent by the FCTC staff performing a validation and preparing the validation summary report was 78 hours.

COBOL 68⁴

Early experiences with COBOL compiler validation were reported in 1974 by Baird and Cook.⁶ Many of the problems identified at that time could be attributed to poor planning on the part of vendors producing COBOL compilers. COBOL 68 has been implemented in one form or another by most vendors.

Many errors reflected sloppiness and poor judgment on the part of implementers. Seeing a compiler perform syntax analysis and issue diagnostics on the content of comments in the source program is humorous but of no use whatsoever. Other errors were a result of trying to take a pre-COBOL-68 compiler and upgrade it to meet the 1968 COBOL standard. Several problems could arise from such an attempt as a result of the difference between most existing COBOL compilers at that time and the requirements of the 1968 COBOL standard.

Prior to the publication of the COBOL 68, there was no stable base of specifications the implementer of a COBOL compiler could use. The specifications for COBOL were maintained by CODASYL⁷ and were changed about six times a year. Some of the changes were incompatible with previous issues of the CODASYL specifications. Vendors of COBOL compilers could not make these changes to their compilers as they were made by CODASYL because of their frequency of appearance and the fact that their existing customer base would not put up with compilers that changed six times a year. The potential conversion problems would have killed any hope for COBOL to become the widely used language it is today.

When COBOL 68 was adopted, it was based on the 1965 CODASYL COBOL specifications. Compilers that had evolved over the years may or may not have been up to date in comparison with the 1965 specifications. In this regard many compiler errors were a result of the CODASYL specifications having changed between the time the compiler was developed and the time the COBOL standard was adopted. Most of these errors were a surprise to the vendors who thought their compilers were in line with COBOL 68.

Other problem areas were a result of trying to modify a piece of software to do something it was never intended to do. This produced many strange and seemingly unrelated compiler errors. In some instances correcting one error would cause several others to appear in what one would assume were different parts of the compiler. For example, correcting a problem with the REMAINDER phrase of the DIVIDE statement in one compiler caused a problem to surface with the COPY statement, which introduces text into the program from a library. The COPY statement has nothing to do with arithmetic and even less to do with the DIVIDE statement.

Needless to say, there was a high error rate associated with COBOL compilers that were based on COBOL 68. The disappearance of these types of errors suggests that the availability of an independently administered compiler testing facility has had an impact on the development of COBOL compilers.

COBOL 74⁵

When COBOL 74 was adopted, there were several things that had taken place in the computer industry that would provide for better compilers. The techniques for producing software in general and compilers in particular had been refined and become more mature since the first compilers had been developed fifteen years earlier. The vendors were more familiar with the COBOL standard in 1974, and the majority chose to produce COBOL 74 compilers from scratch rather than modify an existing compiler. The vendors also took the COBOL standard more seriously, since they were aware that there would be a requirement that all COBOL compilers offered to the government be validated by the Testing Center. The COBOL Compiler Validation System for COBOL 74 was available from the Testing Center relatively soon after the standard was adopted.

The Federal implementation of COBOL 74¹ identifies four levels of implementation that are acceptable without an agency's having to grant a waiver. They are low, low-intermediate, high-intermediate, and high. (For COBOL 68 four levels existed, but most compilers supported the high level or the full standard.) As of this writing 35 compilers have been validated within the last year, and their validation summary reports are current. Eighteen have been validated at the high level, none at the high-intermediate level, 10 at the low-intermediate level, and seven at the low level. This represents support of a greater variety of levels than was true for COBOL 68.

COMPILER ERRORS

The number of compiler validation errors have dropped significantly between COBOL 68 and COBOL 74:

COBOL 68	
Maximum	50
Minimum	0
Average	18
COBOL 74	
Maximum	49
Minimum	0
Average	5.8

The average number of errors for a validated COBOL 74 Compiler was 9.6 in 1977. As of this writing, the average has dropped to 5.8 errors per compiler. During the period that COBOL 68 compilers were validated, three compilers were validated as having no errors. Currently 10 COBOL 74 compilers have been validated with no detected errors. When it is considered that COBOL 74 is roughly twice as large and

complex as COBOL 68, this is a remarkable statistic. This decrease in the number of errors was accomplished during a period of the development of newer compilers, while most COBOL 68 compilers were validated during the maintenance phase of their lives.

The compiler errors identified for COBOL 74 fall into two categories: First, that in which the compiler does not produce the correct results, caused by an implementation error or a decision not to follow the language specifications; and second, that in which the compiler simply does not support a feature or a language element.

The number of errors would be somewhat lower if non-support errors were ignored—and from an evaluator's point of view in trying to determine the quality of a compiler, this should be done. (Choosing not to implement a language element can be inconvenient to the user; implementing a feature incorrectly can be disastrous.) This would bring the average number of compiler errors down to 5.5 errors per compiler.

The areas of nonsupport identified in a validation summary report generally result from a compiler's being validated at a level of COBOL higher than the implementation level of the compiler. Federal procurement regulations² permit up to one year to correct the errors in a delivered compiler unless the procuring agency requires a shorter period. Therefore, compilers validated for offering to the government sometimes have entire modules missing.

The features that fell into the nonsupport areas the most were the RERUN statement (19 times) and the Debug module (13 times). Switches (software), CLOSE REEL/UNIT, and program collating sequence followed closely behind.

The compiler errors associated with the incorrect implementation of language features have a breakdown as follows:

Module	% of Total Errors
Nucleus	45
Table handling	2
Sequential I/O	24
Relative I/O	12
Indexed I/O	5
Sort-merge	.5
Segmentation	2
Library	2
Debug	8
Interprogram communication	.5
Communication	Not tested

The percentage of errors in the I/O modules is less than for COBOL 68 compilers, probably because the I/O modules in COBOL 74 are much better defined. The large percentage of errors in the nucleus is not surprising, since most of the other modules are dependent on interfacing with other systems' software. The internal processing represented by the nucleus is wholly controlled by the compiler, with no support from outside operating system software.

SUMMARY

The impact of the FTC on the computer industry has been a positive one. The vendors per se see the FTC as a mixed blessing, providing an independent appraisal of their product, and then requiring that deficiencies be corrected in accordance with Federal regulations. The quality of COBOL compilers based on COBOL 74 is much better than the quality of those based on COBOL 68. The FTC can take some of the credit for this change in quality of compilers. As the FTC begins to expand its testing of FORTRAN compilers and include the testing of BASIC compilers, it will be interesting to note whether the trends are similar to COBOL's growth in the face of an independent validation service.

REFERENCES

1. "Federal Standard COBOL (COBOL 74)," Federal Information Processing Standard Publication 21-1. Superintendent of Documents, U.S. Government Printing Office, Washington D.C. 20402.
2. Federal Property Management Regulation, Subpart 101-36.1305-1. "FIPS PUB 21-1, Federal Standard COBOL," General Services Administration, Washington, D.C. 20405
3. American National Standard Code for Information Interchange, X3.4-1977, American National Standards Institute Incorporated, New York 1977.
4. American National Standard COBOL, X3.23-1968, American National Standards Institute Incorporated, New York 1968.
5. American National Standard Programming Language COBOL, X3.23-1974, American National Standards Institute Incorporated, New York, 1974.
6. Baird, G. N. and Cook, Margaret M. "Experiences in COBOL Compiler Validation," *Proc. NCC 74*, AFIPS Press, Montvale, NJ, pp. 417-421.
7. Conference on Data Systems Language. (CODASYL) COBOL COMMITTEE JOURNAL OF DEVELOPMENT, P.O. Box 1808, Washington, D.C. 20013.
8. G. J. Myers. "The Art of Software Testing," New York: Wiley-Interscience, 1979.

An approach to transfer verification and validation technology

by MARK K. SMITH and LEONARD L. TRIPP

Boeing Computer Services Co.
Seattle, Washington

LEON J. OSTERWEIL and RICHARD N. TAYLOR

University of Colorado
Boulder, Colorado

WILLIAM E. HOWDEN

University of California at San Diego
San Diego, California

ABSTRACT

The National Bureau of Standards' Institute for Computer Sciences and Technology (NBS/ICST) has sponsored the development of a general guideline for computer software verification and validation (V&V). The guideline is to be used by government and commercial personnel to plan, carry out, and assess V&V activities. This paper outlines the scope, direction, and status of this effort. A lifecycle V&V model which forms the basis of the guideline is described. It is the author's conviction that planning for V&V is an aspect of software development management, and that substantial, cost-effective technology is readily available for general application. The guideline to disseminate this information has been developed.

INTRODUCTION

The U.S. National Bureau of Standards (NBS) is charged with the responsibility of promulgating standards for the development of quality software. NBS was formally entrusted with this responsibility under Public Law 89-306 (Brooks Bill). The Brooks Bill requires NBS to develop standards that will allow the "economic and efficient purchase, lease, maintenance, operation and utilization of automatic data processing equipment by federal departments and agencies." V&V is the set of procedures, activities, and tools used to increase confidence in software, an essential aspect of quality software development. NBS has devised a four-step program for determining and promulgating high quality V&V procedures and tools:

1. Prepare a guideline for sound developmental V&V practices.
2. Circulate the guideline as an NBS Special Publication.
3. Demonstrate the utility of the guideline through a deliberate analysis of costs and benefits.

4. Publish the beneficial parts of the guideline as a software V&V standard in the Federal Information Processing Standards (FIPS) series.

To accomplish the first step of this program, NBS's Institute for Computer Science & Technology (NBS/ICST) has contracted with Boeing Computer Services Company (BCS) for the development of a guideline (hereafter referred to as the V&V guideline) to perform software V&V.

The purpose of the guideline, like that of an engineering handbook,¹ is to supply the practicing software engineer and the student with an authoritative reference which covers the field of software V&V in a comprehensive manner. The guideline has been prepared by specialists. Its contents were confirmed by field studies. The guideline is to describe the concepts, techniques and tools of modern V&V and to integrate them into an overall V&V methodology for the mainstream software developer. Planning for V&V is an integral aspect of software development management. It is the authors' contention that the technology necessary to do this is available, cost-effective and proven. Projects of all sizes and application areas share similar needs. The V&V technology to meet these needs can be tailored to individual situations. Some projects may use simple manual techniques while others may add a set of automated techniques. The key point is that software quality issues should not be ignored or inadequately considered.

The guideline is not intended to be a project cookbook but does present essential V&V techniques and principles. Since projects vary in size, complexity, area of application, and importance, different V&V methods are appropriate for each. Thus the guideline can be used to build a V&V plan for any specific project.

The V&V guideline is designed to be used by a variety of personnel. Customers and management can use it to define the tools and techniques to be used. Programmers and analysts can use it as a tutorial, as it presents the concepts

underlying individual methods and gives references to more detailed discussions of each method.

This paper discusses the preparation, rationale, content and uses of the guideline and indicates how the initial guideline will be updated to include new V&V practices and eventually become a FIPS standard.

PURPOSE

Typical Audience

The guideline was developed for a well specified audience, both technical and managerial. It was designed to assist the policymaker, planner, selector of techniques/tools, and the implementor. It can apply to a project team, an independent V&V team or a quality assurance organization. For each audience group, the guideline's intended uses are described below:

- **Policymaker**—The guideline presents certain fundamental concepts, elements of general V&V approach, and a basis for sound V&V policies.
- **Planner**—The guideline presents a process for developing a V&V plan customized to the needs of a particular environment or a specific project.
- **Selector**—The guideline discusses the selection and integration of techniques and tools. It presents examples of alternative ensembles of capabilities to fit various environmental and project needs.
- **Implementor**—For the novice V&V analyst, the guideline explains principles and concepts of V&V. It also provides guidance in the application of each technique and tool. For the experienced V&V analyst, the guideline will function as a reference handbook for the selection, application and integration of techniques and tools.

Typical Project

The 'typical' project for which the guideline is intended is a mid-sized, in-house development or a significant enhancement to an existing system. The project would involve a team composed of a lead manager and a group of analyst-programmers (as opposed to a project requiring multiple levels of management). Project duration might range from two months to 24 months. Project size and budget are such that cost effective techniques and tools can be employed. The application does not involve unusual environmental or performance constraints that would require specialized V&V techniques or tools.

This definition gives an intuitive feel for a typical project. The concepts presented and the techniques and tools described are applicable to a much broader range of projects. However, the guideline does not directly address projects having unusual characteristics such as real time, embedded or concurrent software, or projects which need special techniques for managing multiple resources or products.

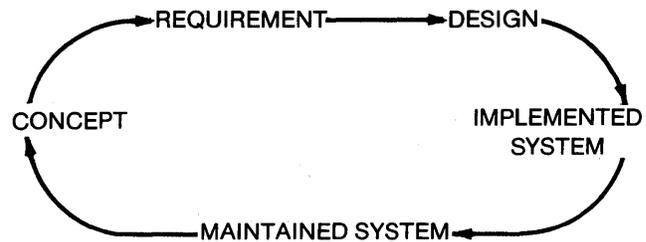


Figure 1—Software lifecycle

GUIDELINE ORGANIZATIONAL THEME

The basic axiom of the guideline is that all software activities should be organized around the lifecycle concept. Within this concept, V&V is a natural and essential activity. This is illustrated by analogy in the next section.

Lifecycle Software V&V

In the dim reaches of the past, men constructed complex physical structures without the benefits of modern engineering. Palaces, temples and entire cities were built over decades without the use of precise design documents or detailed building procedures. In time, architectural principles, tools and methods were consciously formulated and the day of the artisan who carried everything in his head came to end. Builders searched for more challenges, new limits to their abilities. It was no longer imperative that they cling to the past; they were confident that they could plan and design completely new styles, devise new methods of construction and then successfully implement their creations.

In less than fifty years, software development has passed through an engineering development phase for which the analogous phase in palace and temple building took place over thousands of years. There is enormous economic and social pressure to devise sound engineering concepts which will allow software systems to be planned, designed and implemented with the same success and confidence that skyscrapers and bridges are constructed. The basic organizational structure in the emerging software discipline is that of the *lifecycle* (see Figure 1).

The software lifecycle breaks the software development process up into phases which separate intent, planning, construction and use. This division into phases is indispensable to the construction of a "valid" or reliable product. Without an unambiguous statement of intent, planners and builders will construct the wrong product. Without planning, disastrous errors will occur in the building stage that will wipe out months or years of effort. Failure to consider different aspects of the use of a product can also be disastrous. Ill-conceived modifications or corrections may result in unforeseen structural weaknesses in a system, causing it to collapse. It is clearly most appropriate to also view the software lifecycle as the conceptual basis for software. V&V is not a separate phase in a software lifecycle.^{2, 3} It is an integral part and a principal motivating factor in each phase. At each stage of development the product is reviewed to make sure it is a

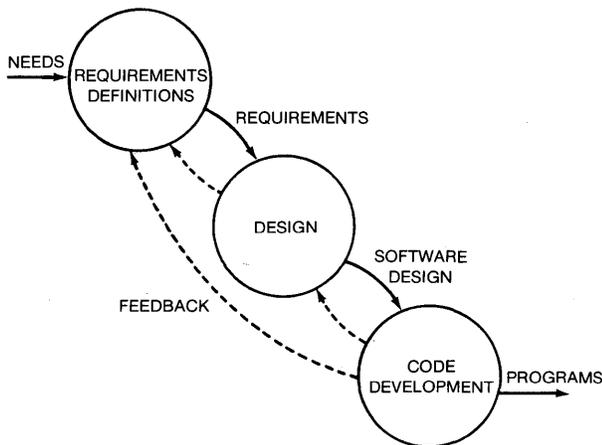


Figure 2—Lifecycle product flow

correct elaboration of the previous stage (see Figure 2). The succession of reviews and analyses from the current stage back to the requirements is the only way both user and builder may have confidence that a fully satisfactory result will eventually emerge. V&V is the process of “confidence raising.” Clearly it is integral to each phase.

The underlying theme of the V&V guideline is the software lifecycle. The procedures which are carried out in each lifecycle phase, their interactions across phases, and their use as basis for a total lifecycle V&V effort are described in the guideline.

Lifecycle Products and V&V Activities

Several recent studies⁴ indicate the vital importance of early V&V. Studies also indicate that the only effective way to motivate early V&V is to formally require the generation of intermediate products involved in the V&V process. This includes both products on which the V&V activities are carried out and products which are generated by the V&V activities

themselves. It is possible to describe V&V techniques and methods independent of intermediate products but the discussion lacks focus and it is difficult to define formal procedures for auditing or enforcing the application of the techniques. The scope of a V&V activity can be clearly delineated by defining it in terms of the product which it analyzes. The scope can also be defined by the results of the V&V activity when the V&V results are formally required as intermediate products. They then can be used to enforce standards and to audit V&V activities (see Figure 3).

The V&V guideline stresses the importance of intermediate lifecycle products. The basic concepts of, and the motivation for, individual methods and techniques are described in terms of these products. Some methods involve the direct analysis of intermediate products for properties such as consistency or completeness. Other techniques may use an intermediate product to generate a test plan which is used later in the lifecycle. The emphasis on lifecycle makes the guideline practical and enforceable.

GUIDELINE CONTENT

Since V&V is not yet a widely understood discipline, this guideline has been written in a self contained manner—all necessary principles and techniques are included. The guideline defines ways to customize the principles and techniques for specific uses. The software lifecycle model enables one to view these customized techniques of a coherent V&V plan as individual instances in an overall V&V paradigm.

The guideline consists of three major sections (see Figure 4): a discussion of the lifecycle V&V strategy, a guide to planning customized V&V, and a detailed presentation of 30 basic V&V techniques. Examples are used to aid in presentation.

Lifecycle and Integration Framework

The first section is composed of three chapters which present the principles of V&V. An overview of software development follows the introduction. The overview makes the key point that many catastrophic problems in software devel-

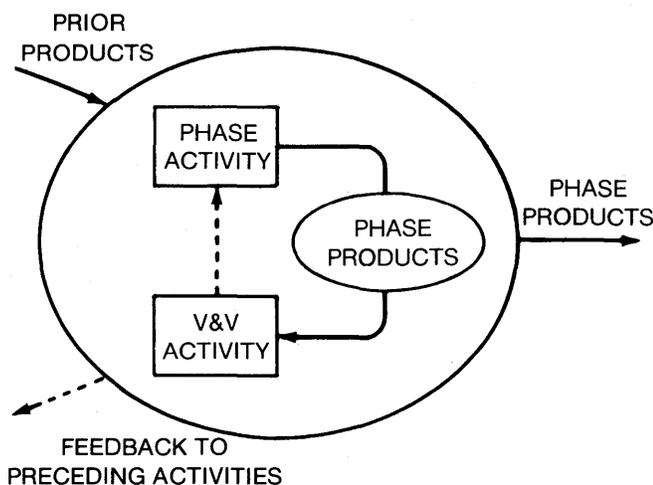


Figure 3—Lifecycle V&V

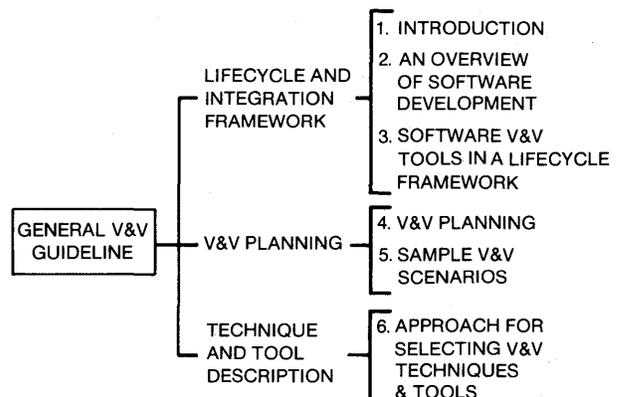


Figure 4—Guideline summary outline

Table I—Technique description format

Name	Effectiveness
This is the accepted title, or when an appropriate one does not exist an invented title.	A brief assessment of the effectiveness and usability including underlying assumptions and difficulties that can be expected in practice.
Basic Features	Applicability
A short description of the technique or tool.	An indication of the situation in which the technique is likely to be useful.
Information Input	Learning
A description of the input required for use.	An estimate of the learning time and training needed to use the technique successfully.
Information Output	Cost
A description of the results of the technique or the output of the tool.	An estimate of the resources needed.
Outline of Method	References
A brief list of the actions that a user is expected to perform.	Sources of additional information.
Example	
An example to illustrate the inputs, outputs, and the method.	

opment and maintenance are preventable. Analogies to other disciplines (eg. home construction and law-making) are used to motivate adoption of the lifecycle approach and give an intuitive feel for the character of V&V.

The software lifecycle model is also presented in the second chapter. The essential properties of a lifecycle model are explained and a discussion of the problem of deciding when to exit a phase is included.

The third chapter discusses V&V technology and how the technology applies in each of the lifecycle phases. V&V is presented as a confidence raising process. The different needs for confidence raising are discussed. The critical role of specifications in the V&V techniques are briefly described and degrees of possible automation are discussed. Manual techniques and automated capabilities are considered. A firm grasp of the material in these chapters will allow the reader to make the transition from the principles of V&V to the specifics required for raising confidence in everyday programs.

V&V Planning

The second section of guideline aids in the transition from principles to practice. Two chapters with different focuses are devoted to this task. Differences between programming environments are considered in the first chapter of this section, and the notion of a customized V&V plan is introduced. The second chapter contains examples of V&V technology.

Programming environments (used in a broad sense) can differ significantly even though the size and kind of software being produced may be similar. Some of the differences discussed are: the support software environment, configuration management practices, existence and quality of interim lifecycle products (such as program design documentation), budget for V&V, availability of tools, training of project personnel, and of course, confidence level.

The notion of the V&V plan is introduced here as an aspect of sensible software management, planning, and cost estima-

tion. The plan is simply a document which details the V&V practices which will be used during the project. Such a plan details the information by project phase, indicating personnel involved, customer interface required, acceptance criteria, etc. The plan considers all the factors which distinguish the project, the programming environment, and the V&V requirements.

To aid in the communication of these principles, the second chapter in this section shows V&V technology applied to some common software applications in direct examples. The two application areas are general ledger transaction processing and a graphics application. Within each, the development of several functions is traced through the lifecycle. Performing V&V in a typical maintenance situation is illustrated, using a general ledger system example.

To illustrate how differing environments can affect the application of V&V techniques, the examples use three levels of technology. The most basic level consists of only manually applied techniques. This is followed by a minimally automated toolset, which includes a requirements representation scheme. Lastly, use of a full toolset is presented.

Technique and Tool Selection and Descriptions

This section contains an index of the V&V tools and techniques described. Each technique useful in one or more lifecycle phases can be referenced by either lifecycle phase, degree of automation, or resource requirements. Based on the requirements for V&V, as identified in a project V&V plan, this index will facilitate the identification of applicable techniques. This section also presents a comprehensive enumeration of V&V tools and techniques (Table I). Each tool and technique is presented in an identical format.

At present, 30 techniques are described in this section. General classifications of techniques and tools are utilized; particular products or narrow instances of tools are not given individual attention. Table II lists the current contents of this

section. (It is anticipated that this list will be updated periodically to reflect new developments and changes in the V&V field.) The guideline concludes with a general bibliography on V&V and an index to the complete guideline.

Name

This is either the originators own title or else an invented title when no accepted name exists or when the original title is misleading.

Basic Features

A short description of the results of using the technique.

Information Input

A description of the input essential to achieving lifecycle integration.

Information Output

A description of both the results of the tool and the information vital to integration.

Outline of Method

A brief list of the actions that a user of the technique is expected to carry out. This outline may not be understood on first reading because it is expressed in abstract terms; it should, however, be clear after the example has been followed.

Example

These are taken where possible from case histories. If none exist examples will be invented to illustrate each of the technique's actions. It is recognized that any example chosen will not be familiar to all readers. If an example is unfamiliar, a reader should read each example with their own application area in mind. They may then find that the principles discussed will suggest applications to situations with which they are familiar.

Effectiveness

These are brief assessments of the effectiveness and usability of the technique with emphasis on the underlying assumptions and on difficulties that can be expected in practice.

Applicability

An indication of the kinds of situations in which the technique is likely to be useful.

Learning

An estimate of the learning time and training needed to use the technique successfully.

Cost

An estimate of the total resources needed for various situations to carry the techniques out.

References

Additional ideas on "technique" format

Algorithm complexity analysis

Assertion generation

Cause-effect graphing

Comparator (code, file, test results)

Concurrency analyzer

Cross-reference

Data base assertions/validation analysis

Data flow analyzer

Design based functional testing

Dynamic assertion processor

Execution time estimator/analyzer

Formal reviews and audits

Formal verification

Inspections

Interface checker

Mutation analysis

Numerical error analysis

Peer review

Requirements tracability aids

Requirements-based functional testing

Simulation modeling (analytical)

Software monitor (spy program)

Specification simulation

Standards analyzer/auditor

Structure analyzer

Table II—List of techniques and tools

Algorithm complexity analysis	Interface checker
Analytic modeling of software designs	Mutation analysis
Assertion generation	Peer review
Cause-effect graphing	Physical units checking
Code auditor	Regression testing
Comparator	Requirement/specification traceability aids
Cross-reference generator	Software monitor
Data flow analyzer	Specification-based functional testing
Dynamic assertion processor	Specification simulation
Execution time estimator/analyzer	Structure analyzer
Formal reviews	Symbolic evaluation
Formal verification	Test support aids
Global round off analysis of algebraic processes	Test coverage analysis
Inspections	Test data generators
Interactive test aids	Walk-throughs

Symbolic evaluation
 Test bed
 Test coverage analysis
 Test data generation
 Walk-through

THE GUIDELINE DEVELOPMENT APPROACH

The Development Team

The individuals producing the guidelines have experience in both the practical and theoretical aspects of software V&V. This combination of skills and knowledge combined with the checks and balances offered by the team approach yields a high quality product.

The Development Approach

The development has been divided into phases similar to those of the software lifecycle. The first phase, the requirements analysis, resulted in the identification of the primary objectives and the intended audiences for the guideline. From this point, the behavioral, environmental, interface, packaging, and performance requirements for the guideline were specified. During the preliminary design, a high level outline was created. Next the objectives for each proposed chapter were added to the outline. Detailed design resulted in the comprehensive annotation of each chapter. During the construction phase, chapter drafts were produced first, then integrated into a complete draft.

As with software development, this process emphasized the development of interim products and their review. There were check points where the evolving product was examined to make sure it was still 'on target.' These V&V steps included both team and sponsor review of the statement of the requirements, the high level outlines and chapter objectives, the annotated outline, each draft chapter, and finally, the complete draft.

Supporting Work

The development of the guideline is supported by two additional tasks (see Figure 5). The first is a survey of V&V practices at ten sites. The second is an analysis of the survey results, and an identification of the major factors affecting the application of V&V technology. These tasks will help assure the broad usability and applicability of the guideline.

The survey task

The selection of the survey sites was governed by several criteria. Five sites, to be representative of the intended guideline audience (and ultimately the resulting FIPS standard) were from the federal ADP community. One of the major objectives of the survey was to characterize the user community to ensure applicability of the guideline.

The remaining five sites were drawn from the private sec-

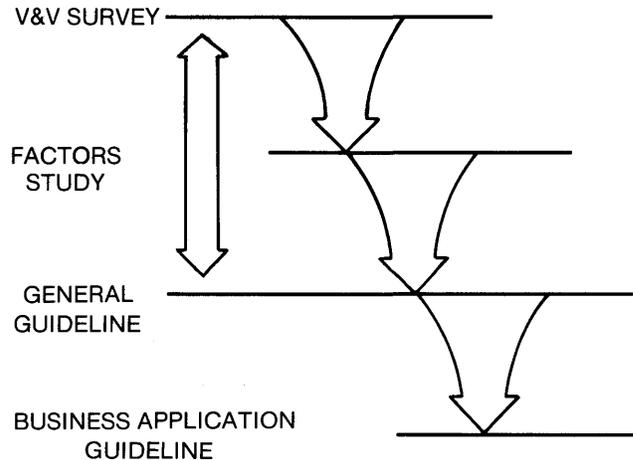


Figure 5—Information flow between tasks

tor. They were to include representatives from the financial, insurance and public utility communities. The intent was to pick several business-oriented sites where there was a significant volume of data processing and where accuracy and timeliness were important factors. The primary reasons for surveying commercial sites were to investigate the basic differences between the two sectors and to look for new emerging V&V technology in the commercial sector that is applicable at similar federal ADP sites. These sites were also selected to represent the expected audience for the guideline.

The survey collected information in four major areas:

1. Size, application areas, percent of time in development vs. maintenance, etc.
2. The lifecycle, its phases, the products produced, reviews held, etc.
3. Specific V&V techniques and tools utilized with a description of each including information about its utilization
4. Formal or documented standards, guidelines and procedures of each site; applicability and prevalent attitudes toward standards in the environment

The survey was administered in four steps. A kickoff meeting was held to explain the survey and pass out the questionnaires which were then filled out by personnel at each site. The questionnaires were collected and participants interviewed to clarify and elaborate on responses. To conclude, a summary report was prepared for each site and reviewed by the participants for accuracy. These site reports and other summary information from the survey will be published in an NBS special report.

The factors study

The purpose of the study is to analyze information gathered by the survey. The primary result will be the identification of major factors affecting V&V technology in different application areas and programming environments. The study has not yet been completed.

We started by looking at methods of categorizing environments and the influences within these environments which impact the success of V&V. These influences span a broad spectrum and include language, application, local standards, testing and quality assurance budget, tool availability, system performance and availability, staff capability, and training requirements. It is important to determine how these factors will be taken into consideration in the development of V&V guidelines and standards. The results of this study will be essential in developing the guideline chapter which addresses V&V planning for a specific environment or project. The study and the related guideline chapter will help customize V&V planning, technique/tool selection and V&V integration for specific projects.

This study should also be useful in identifying techniques/tools which are needed for V&V, but are currently unavailable. These may be general purpose or more specific tools. The use of these tools may be dictated by factors unique to a particular application area or programming environment.

This study may also identify the need for other V&V guidelines or software quality standards. The reports will be used in determining such future directions.

GUIDELINE UPDATING

A guideline such as this one must not remain static. The first draft of the guideline will be submitted to the sponsor on November 1, 1980. During the next two years, the guideline will undergo several changes to incorporate new V&V techniques, V&V application techniques, and empirical analysis of the utility of V&V techniques. Based on government and public review of the draft, a revised draft is planned for May 1981. The resulting NBS special publication is to be available by the first quarter of 1982. Every effort will be made to obtain the widest dissemination of the draft guideline to ensure that the special publication will be of maximum use to software engineers and nontechnical managers.

CONCLUSION

There are at least two major effects that can result from this work. First, the guideline will be an important educational vehicle. It covers the fundamentals of the emerging discipline of software V&V. It describes the lifecycle approach to software development and the integral and pervasive role of V&V in the process. The guideline will promote the importance of V&V in software development and maintenance and aid in

the establishment of agreed upon nomenclature for this discipline. It will provide ammunition for advocates and policy-makers and the impetus for the establishment of V&V policies, standards, and procedures.

Secondly, the guideline will have broad effects throughout the software industry. Through its adoption within the federal government and its presence throughout industry, it will help clarify the fundamentals of software V&V. The guideline presents a comprehensive survey of techniques and generic tools. An implicit result of this compilation will be the realization, or at least confirmation, of gaps in the current technology. This will probably spur the development of needed tools and techniques. The V&V guideline is part of a series of V&V documents sponsored by NBS^{5,6,7}.

We hope that widespread distribution and review of the draft guideline by government, industry, and university people will lead to a FIPS V&V standard that will be widely used and relevant. Any experiences in performing software V&V, or comments and recommendations concerning the content of the V&V guideline should be addressed to the authors.

ACKNOWLEDGMENTS

Many of the ideas for the format and content of the draft guidelines came from discussions with BCS and NBS personnel. Especially helpful were Martha Branstad of NBS, Rick Adrion of NSF (formerly of NBS), and Leon G. Stucki and John R. Brown of BCS.

REFERENCES

1. R. H. Perry and C. H. Chilton, *Chemical Engineer's Handbook*, Fifth Edition, New York, McGraw-Hill, 1973.
2. L. J. Osterweil, J. R. Brown, and L. G. Stucki, "ASSET: A life cycle verification and visibility system"; *The Journal of Systems and Software*; 1, 77-86, 1979.
3. W. E. Howden, "Life Cycle Software Validation", In *Life Cycle Management* Infotech State of the Art Report (to appear).
4. B. Boehm, "Some experience with automated aids to the design of large scale reliable software", *IEEE Trans on Software Engineering*, Vol SE-1, 1, 1975.
5. M. A. Branstad, J. C. Cherniavsky, and W. R. Adrion, *Validation, Verification and Testing for the Individual Programmer*, Institute for Computer Science and Technology, U.S. National Bureau of Standards, October 1979.
6. W. E. Howden, *Validation of Scientific Programs*, Institute for Computer Science and Technology, U.S. National Bureau of Standards, Washington, D.C. 1980.
7. M. A. Branstad, J. C. Cherniavsky, and W. R. Adrion, *Validation, Verification and Testing for Computer Science and Technology*, U.S. National Bureau of Standards, Washington, D.C. (Draft) April 1980.

Easy interactive access to batch image analysis software*

by RONALD L. DANIELSON

University of Santa Clara
Santa Clara, California

ABSTRACT

Effective use of batch software requires a relatively high level of knowledge from the end user. Such a threshold can preclude application of existing software by potential users who are unfamiliar with computer use. Image processing is an example of an area with large amounts of existing batch software. An instance of the threshold problem occurs when natural resource management experts, with minimal computer application experience, want to use satellite imagery to aid management decisions in their discipline areas. This paper discusses the design of an interactive system to solve the problem by allowing inexperienced users to employ existing batch image analysis software with minimal supervision. The design provides for online explanation of terms and functions, default values for most parameters, and allocation of necessary computer resources (e.g., disk files). The user-system dialog may be conducted at any of three levels, allowing continued use as experience is gained. The design considerations and techniques used are generally relevant to accessing batch software for any application area. Difficulties of implementation and expansion are also discussed.

INTRODUCTION

In any computer application there are two possible modes of operation: batch, in which the user submits a task to the computer and receives the results some time (minutes, hours, days) later; and interactive, in which the user works on line with the computer system and receives a rapid response to his inputs. It is generally agreed that batch operation makes most efficient use of computer time, whereas interactive operation makes most efficient use of the (human) user's time.¹ As the cost of computer hardware has decreased, interactive use of computers has enjoyed a corresponding increase.

At the same time there are a number of application areas with a considerable amount of existing batch-oriented software. Such batch software typically requires a relatively high level of knowledge from the user before it can be effectively employed. In particular, the user must know which programs (by name) perform which application functions, what data

formats are needed, what intermediate disk or tape storage is required, and what job control statements are needed to invoke the appropriate routines and allocate the necessary computer resources. This knowledge threshold often precludes use of batch software by unsophisticated users.

An example of an application area with a large body of existing batch software is image processing. The NASA-Ames Research Center has access to a number of batch image analysis routines on several different computer systems (SEL 32, IBM 360/67, CDC 7600, Illiac IV). Ames is also experiencing a growing interest, from state and local governmental agencies, in applying analysis of satellite imagery to a variety of natural resource management activities (e.g., forestry, agriculture, urban planning).

This combination has created a demand for access to image-processing capability by resource management experts with little previous knowledge of computer use or image analysis. Since guiding sizable numbers of new analysts through the image processing tasks could quickly consume available staff time, a means of allowing unsophisticated users to effectively employ batch software with minimal supervision was desired. Some form of interactive approach seemed most suited.

One technique, available at many computer facilities, is to use interactive editors to compose batch job streams for later submission. However, this approach does not address the lack of knowledge of the end users, since the editors involved are not application-area-specific.

Another possible solution would be to convert all the existing batch software to an interactive format. This solution has several drawbacks. First, it would be prohibitively expensive. Second, many image analysis tasks are CPU-bound and not truly suited to completely interactive use. Finally, there is some evidence that much of the insight needed for successful use of computers for problem-solving activities occurs off line,² suggesting that an easy-to-use batch processing approach may be very effective.

The remainder of this paper outlines the design of the batch analysis setup system (BASS), developed at Ames to allow unsophisticated users easy access to batch image analysis software. The design employs current principles of person-machine interaction to insure a friendly user environment, and provides online, process-specific explanations and default values to reduce the knowledge required of the end user. At the same time the design is simple enough to allow rapid

* This work supported in part by NASA-Ames Research Center under University Consortium Agreement NCA2-0R685-810.

implementation and easy expansion. The technique is applicable to virtually any existing batch mode software.

INTERACTIVE SYSTEM CONSIDERATIONS

It is an all too familiar experience that computer professionals, in attempting to provide a tool for general usage, design a system suited to their own habits and needs and poorly suited to those of the end user.³ It is equally true that the impact of a poor interface on the performance of the end user is seldom considered a real system cost, although it obviously should be.⁴ Prior evaluation of established design principles for interactive systems can point out relevant areas for any system design.

Perhaps the primary design consideration is to know the user.⁵ In the instance at hand, the users will almost always be experts, in several discipline areas, but they will have little previous experience with computer applications. They will also generally not have had much experience with image analysis, although they are highly motivated to gain such experience, since they typically have application projects of immediate interest on which they will be working.

It is desirable that the interface behave uniformly in all possible circumstances,⁵ and that the interface be flexible, to maximize satisfaction across a wide spectrum of users. Kasik,⁶ however, warns that too much flexibility may be overwhelming, particularly for new or inexperienced users.

The desire to accommodate a spectrum of users implies satisfying users with different levels of experience. This capability will also allow users to continue using the system as their level of sophistication increases. Bennett² describes three relevant phases in the use of an interactive system as a tool: uncertainty, in which the users overcome their hesitancy and gain confidence; insight, in which they decide how to best use the tool for their special needs; and incorporation, in which they use the tool as an integral part of their problem-solving behavior. Bennett further warns that users are generally ready to use added power only after they are familiar with the previous level. It also seems true that users learn different aspects of a system at different rates,⁷ and that user inputs become more terse as experience with portions of a system increases.⁸

Several authorities advocate using different areas of a display screen for different purposes in the user-system dialog.^{9,10,11} Access of both system capabilities and user data by function, rather than by name,¹¹ is important to allowing the user to concentrate on the task at hand, rather than getting lost in detail.

Finally, robustness in the face of user errors is an essential attribute of any interactive system. Not only should the system not fail in the event of an error; it should also provide meaningful error messages and a means for the user to obtain full explanations if needed.

SYSTEM DESIGN

Constraints

Several constraints were applied before the design of BASS began. For the most part, these constraints had little signifi-

cant impact on the design itself (although they complicate implementation in several areas). The primary constraints were ease of maintenance and possible portability. Consequently, BASS is designed with a high degree of modularity (a good practice in itself) and an eye toward expansion of the number of image analysis functions implemented. Portability dictated a design for a FORTRAN IV implementation and isolation of all host-system-dependent features to minimize difficulties in implementing BASS on a different host. Portability also implied minimal assumptions about peripherals on the host. In particular, the terminal is assumed to be a simple line-oriented device (hard copy or CRT). This precludes reserving various areas of a display screen for different uses in the user-system dialog.

Minimal supervision of users was another consideration. Consequently, the various image analysis tasks are organized by function (e.g., classification) rather than by the names of the particular batch programs implementing those functions. There is online reference material, easily accessible to the user, to provide quick answers for many probable questions. There is also a log of all transactions performed during a terminal session. Such a log provides a history of how various image files were created and what is contained in different parameter files, as well as providing a functional outline of the image analysis tasks for new users.

Finally, it was assumed that all users would receive minimal training, enough to be familiar with both image analysis (e.g., image characteristics, the analysis process, applications to remote sensing) and computer capabilities (e.g., I/O capabilities, what a disk file is, how to use the terminal).

Overview

BASS is designed to function as a complete intermediary between the user and batch mode image analysis programs already existing on the host computer system. This function involves creating files of job control statements to invoke the batch programs and linking those job control files to the host's dynamic batch reader. It also includes prompting the user for any parameter values needed by the analysis program (and supplying default values if necessary) and creation of disk files for storing parameters, communication of data between segments of the analysis programs, and storing output. The log maintained by BASS is in easily readable English and specifies all files and parameter values used in a particular execution of an analysis function. The user may obtain a listing of this file at the end of the session.

The overall structure of BASS is a two-level hierarchy (Figure 1). The top level consists of a continually resident kernel, composed primarily of logon/logoff routines, a single user interface routine that handles all direct communication with the user (both input and output), and an operating system interface to handle all communication between BASS and the host operating system.

The second level consists of several mutually exclusive modules that are loaded as overlays beneath the kernel at various times during a session. One module (HELP) contains the online reference material available to the user. The refer-

ence material explains the use of BASS, the various image analysis functions available, and the parameters needed by each image analysis program. The HELP module is organized as a tree of menu displays.

Each of the other second-level modules contains routines to create the necessary job control files to invoke the available analysis programs and to prompt the user (through the interface routine in the kernel) for desired functions, parameter values, and file names. There are modules for three different modes of user-system interaction, roughly corresponding to Bennett's three phases of use. LOW provides naive users with the detailed prompts and explanations necessary to enable them to perform analysis of image data. MID provides moderately sophisticated users with more rapid interaction, reducing the volume of output by providing more succinct prompts. HIGH allows experienced users to control the dialog even more by inputting parameter values and file names without waiting for a prompt. The LOW, MID, and HIGH modules are each organized as a two-level hierarchy, with a single module executive routine and one functional routine for each of the image analysis functions implemented via batch programs.

Since the function routines are completely independent of one another, modifying any function routine (due to user demands or changes in the batch program that implements the corresponding image analysis task) has no effect on the performance of other function routines. Similarly, modifications to one module do not affect the performance of other modules.

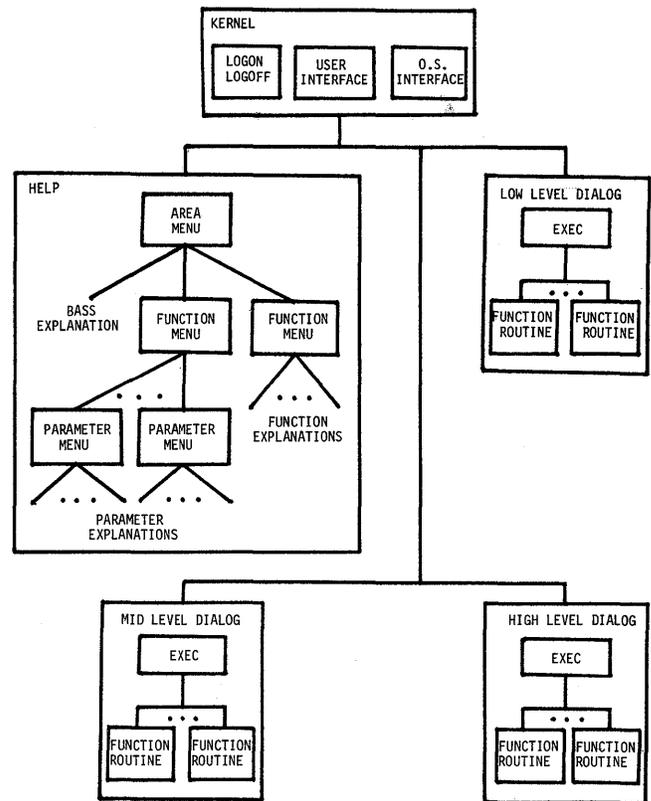


Figure 1—BASS structure

Kernel

The kernel design includes three collections of subroutines that implement functions common to all three second-level interaction mode modules.

The logon/logoff routine welcomes the user to the system and obtains a user name. The user name accesses a file containing the appropriate interaction mode for each user. If the current user's name is not in the file (i.e., the user has never interacted with BASS before), it is inserted, the mode is set to LOW, and the user is guided through a short description of BASS. The appropriate interaction mode module is then loaded, and control passes to the module executive. On return from the module, the user is given a chance to list the history file for this session and is then logged off.

The user interface subroutines implement all direct contact with the user. This isolation insures that the user always works in a uniform environment with regard to matters such as control functions or error detection and announcement, regardless of when the particular functional routine was written. It also frees the programmer of a BASS functional routine from those same concerns. The user interface is employed by all routines in the interaction mode modules, as well as other kernel routines, and is invoked by subroutine call.

Three parameters suffice for communication between the invoking subroutine and the interface routine. One is a character array containing a prompt to be displayed to the user before asking for input. The design provides a separate entry

point for accepting input with no prompt display. The second parameter is an array for returning values to the calling routine. The third parameter indicates the type of input expected, or that the interface should simply display the prompt and return. The interface can return values of type character (arbitrary length), integer, real, or logical (yes/no). More than one distinct input value can be returned on any call, but only one type of input. The interface checks all inputs to insure that they are of the appropriate type. If not, a message is displayed to the user, and new input is solicited.

The interface also scans each set of input for a user request to access the online reference material (the characters HELP). If such a request is found, the interface loads the HELP module and transfers control to that module's root menu display subroutine. On return from HELP, the interface reloads the correct interaction mode module, displays the prompt that led to the help request, and awaits the desired input.

The operating system interface performs operating system functions for all other routines in BASS. Functions needed include dynamic file creation and deletion, allocation and deallocation of files and peripheral devices, opening and closing of files, linking job control files to the batch reader, and determining the status of previously submitted tasks. If the host system and batch software permit, all files should be created in the user's file space to minimize conflict with existing file names. The system interface checks for error condi-

```

What do you want help with?
  1. How to use this system.
  2. Information about image processing tasks.
  3. Information about parameters.
  4. Return to normal processing.

> 3
Which task's parameters do you want more information on?
  1. Destripe
  2. Cluster
  3. Classify
  :
  7. Return to main help selection display

> 2
Cluster function: which parameter?
  1. Parameter file name - PFILE
  2. Input tape file number - ITPN
  3. Separability value - DVM
  :
  16. Return to function selection display.

> 2
< explanation of ITPN >
Cluster function: which parameter?
  1. Parameter file name - PFILE
  :
< 16
Which task's parameters do you want more information on?
  1. Destripe
  :

< 3
Classify function: which parameter?
  1. Threshold value - THR
  :

```

Figure 2—Sample interaction with HELP module

tions associated with its various functions and attempts to resolve the errors by interacting with the user via the user interface. The cause of the error and suggested remedial action (e.g., duplicate file name on creation, select a different name) are reported to the user. Once the error is resolved, the system interface returns to the function routine.

Isolating all operating system tasks in this manner frees function routine programmers from checking and resolving all possible errors. In addition, it is easier to maintain comity between BASS and future releases of the host operating system, to implement new operating system functions as they become available, and to transport BASS to a different operating system.

HELP Module

The HELP module is designed to provide online reference material for the user in three separate areas: the use of BASS, the objectives of and functions involved in image analysis, and the role played by (and typical values of) the individual parameters required by each of the image analysis functions implemented in BASS.

The user enters the HELP module from the kernel's user interface routine by typing the characters HELP in response to any prompt provided by the system. The combination of entry via the user interface and a FORTRAN IV (i.e., non-recursive) implementation implies that the HELP module contains its own simplified user interface, rather than using that in the kernel. This simplified interface accepts single-integer inputs, checks to insure that they satisfy bounds constraints (within the choices allowed by a particular menu), and traps requests for help issued within the HELP module.

The basic format of user interaction within the HELP module is menu selection, in which the user is presented a series of options and must type in a single number to make the corresponding choice. Menus are arranged in a treelike hierarchy. The top-level menu selects which of the three available areas the user wants help with. Second-level menus allow choice of a particular subarea; for example, in the parameter description help sequence, the second-level menu selects the image analysis function that requires the parameter. Third-level menus narrow the selection still further (by selecting a parameter within an image analysis function, for example).

After completing the actions resulting from a choice at a particular level, the user is returned to that menu to make another selection, on the assumption that help may be desired on several related topics. One choice on each menu is return to the next-higher-level menu. This allows the user to move about in the HELP module control hierarchy, clarifying understanding of a number of topics before leaving the HELP module (see Figure 2 for a hypothetical protocol). As described in the preceding section, on return from the HELP module the user is again given the prompt that was on the screen when the help request was entered, and BASS waits for entry of the originally expected value.

LOW Level Interaction Module

All three interaction mode modules perform the same basic function within BASS and have similar structures. The function is to create files of job control statements to invoke one or more batch image analysis programs. The structure is that of a local executive and one subroutine (functional routine) for each of the image analysis programs available on the host processing system.

The differences between the three modules lie in their expectations of users' understanding of the various analysis functions and the changes such expectations make in the user-system dialog. The LOW module expects users to have only a little knowledge of image analysis and essentially no knowledge of the specific programs available on the host system. As a consequence, it employs menu selection techniques and relatively long prompts to provide as much guidance as possible to users. This allows new users flexibility regarding which tasks they want to perform and the rate at which interaction occurs, yet provides a controlled dialog in which the users' choices are clearly and explicitly displayed. Figure 3 contains portions of a user-system protocol at the LOW level, which should be consulted for examples during the ensuing discussion of the module.

The executive routine displays a menu with choices for each

of the analysis functions, plus a choice to terminate the session. On the basis of the user input, the appropriate function subroutine is invoked or, if termination was selected, the LOW module returns to the kernel logoff routine.

Each function routine first determines the names of any disk files needed by the associated analysis program. These files may be input or output data files, input parameter files, or intermediate files between two or more programs coming one analysis task. The user is given a prompt describing the purpose of the file and asking for a file name. A subsequent prompt (if the file is an intermediate, output, or parameter file; input data files must already exist) asks the user if the file already exists. If not, the function routine creates the file via the operating system interface.

For preexisting parameter files, the user is asked if the file contains the desired parameters, and if not, the user is prompted for each parameter. The prompt includes a short description of the parameter and an associated mnemonic. Default values are provided for most parameters if the user simply enters a carriage return. When all parameter values have been collected, the function routine allows the user to review and correct them, if necessary, and then writes them to the parameter file in the format expected by the image analysis program.

After properly instantiating all parameter files, the function routine writes the sequence of job control statements needed to invoke the desired image analysis program(s) to a job control file, and, using the operating system interface, streams the job control file to the dynamic batch input device.

Next a log of all transactions is output to the session history file. The history file includes the analysis function performed, the names of all files created and their purpose, all parameter values used, and the file transactions that will occur during analysis. Figure 4 is a portion of the history file produced by the user-system dialog in Figure 3.

Finally the function routine returns to the module executive to allow the user to select another analysis function. Recall that, at any time during this process, the user may enter the HELP module to clarify understanding of the analysis function or a particular parameter, then return to the function routine to continue processing where it was interrupted.

The function routines are also aware of sequencing restrictions between the various image analysis steps. For example, the BASS classification routine knows a cluster statistics file must exist before classification. If it does not, the classification routine calls the cluster function routine to allow the user to create that file. On return, a message is displayed telling the user to wait for completion of clustering; then the classification routine returns to the module executive. Similarly, the module executive issues a warning on return from the clustering function that the clustering task must be completed before the user begins work on classification.

There are, of course, utility functions each interaction module must provide, in addition to access to the image analysis functions. These include relinking a JCL file to the batch reader (to repeat a function), checking for completion of previous tasks, editing existing parameter and JCL files, editing control point and statistic files, and file transfer functions. Such utility functions are placed in a second-level menu accessed via a catchall utility entry on the initial level.

```

What task do you wish to perform?
  1. Remove banding from an image (DESTRIPE)
  2. Perform unsupervised clustering (CLUSTER)
  3. Group the pixels of an image into distinct classes (CLASSIFY)
    :
  9. End this terminal session.
> 2
Cluster function setup procedure:
Has banding been removed from the input image?
> YES
Enter the name of the parameter file (PFILE)
> CLUSPFIL
Does the file already exist?
> NO
Disk file cluspfil created
What file number on the input tape contains the image (ITPN)?
> 2
What is the separability value for merging clusters?
> 3.0
    < remainder of parameter prompts >
Is there another image to be clustered?
> NO
What task do you wish to perform?
  1. Remove banding from an image (DESTRIPE)
    :

```

Figure 3—Sample user-system protocol at LOW level

MID Level Interaction Module

As previously indicated, the MID module performs the same basic functions as the LOW module, except that users are now assumed to be moderately familiar with BASS. For

```

*****
cluster function history log
!!!!!!
file creation service: file cluspfil created for user rond
!!!!!!
!!!!!!
file creation service: file clusstat created for user rond
!!!!!!
input parameter file cluspfil
and input from mag tape unit 10
were used to create cluster statistic file clusstat
the following parameters were entered in cluspfil:
    input file number = 2
    separability value = 3.0
    window size = 6 x 6 scaled distance
    :

```

Figure 4—Sample history log corresponding to Figure 3 protocol

```

What task?
> CLUSTER
already destriped?
> YES
pfile?
> CLUSPFIL
already exist?
> NO
created
ITPN?
> 2
DVM?
> 3.0
ISW?
> 1
      < remainder of parameter prompts >
another image?
> NO
what task?
>

```

Figure 5—Sample user-system protocol at MID level

such users, who know what actions the system can perform and roughly what order activities occur for each of the image analysis functions, the large amount of verbiage generated by the LOW module can become a hindrance to good person-machine interaction, rather than a support. The presence of online reference material further reduces experienced users' dependence on lengthy system prompts, enabling them to easily fill in gaps in knowledge.

To combat this, the MID module executive routine does not display an initial menu of possible tasks, but simply asks the users which function they wish to perform and expects an image analysis function as a response. The corresponding function routine is then invoked. Note that a list of all functions is available to users in the second-level menu on image analysis in the HELP module (Figure 2).

Similarly, each function routine prompts users for input, using only one- or two-word prompts (typically the mnemonic name used as part of the LOW-level prompt, thus providing continuity with that LOW-level prompt). This greatly reduces the amount of text displayed at the terminal and correspondingly speeds up user-system interaction. Figure 5 shows a MID-level protocol for the same image analysis activities as Figure 3.

Online help is still available at any time, but with increased flexibility for users. Entering HELP in response to any prompt gets the root menu, as in LOW-level use, but with less verbose descriptions of choices. Movement in the tree control structure proceeds as on the LOW level. However, users may append a function name to the HELP request, which directly accesses the third-level parameter selection menu for the indicated function. The parameters are listed by mnemonic on

these menus, again reducing the number of characters displayed and speeding up interaction. Implementing these changes requires a second (more terse) set of HELP information.

The history log is maintained with the same degree of detail as provided by LOW-level function routines.

HIGH-Level Interaction Module

Again, the change between MID-level and HIGH-level interaction is toward reduced amount of display to users and increased user control of the pace of the dialog, assuming users to be very knowledgeable about BASS.

The HIGH-level executive expects users to input a function name in response to a WHICH FUNCTION? prompt, which results in invoking a particular function routine. In this case, however, users may add (in parentheses) a name list of parameter mnemonic and value pairs. Use of keyword arguments with commands typically produces much lower error rates than simply inputting values.¹⁰ It also allows users to specify values in any order. The function routine then provides short MID-level prompts only for the missing parameters (see Figure 6). This allows users to input values for the parameters they remember are required, trusting to BASS to solicit values for the others and place all parameters in the format required by the image analysis program. In effect, this permits users to proceed very rapidly in setting up the analysis tasks they know very well, and to move more slowly with less familiar tasks.

The history log and help functions work as in MID-level interaction.

IMPLEMENTATION AND EXPANSION

Existing Implementation

As of this writing, a subset of the complete BASS is operating at NASA Ames Research Center on a SEL 32 system. This subset contains the complete kernel, a complete HELP module, and a LOW module with executive and function routines for the following image analysis tasks: destripping, registration, clustering, classification, color assignment to classes, CRT image display, and various hard copy (line-printer and color print) functions. All elements of this subset function as described above.

All routines are written in FORTRAN, with the majority of the code meeting ANSI standards. Exceptions include the use of T format in the user interface routine to aid in checking for help requests in response to a prompt and use of INTEGER*1 arrays to facilitate character storage and manipulation. Operating system services are implemented via calls to monitor service routines provided as built-in functions in SEL FORTRAN. The complete system, with the kernel and either the HELP or LOW modules loaded, occupies less than 20 Kbytes of memory. Total effort for system design and the partial implementation was less than six person-months.

Expansion

The decision was made to implement only a subset of the complete BASS design, since virtually all current users of the system fall into the naive user classification. However, considerable thought has gone into exactly what must be done to expand the current implementation, either by adding additional functions at the LOW level or by adding interaction mode modules as the user community grows in sophistication.

LOW Level Expansions

To add an additional image analysis function to the LOW-level module, three actions need to be taken. The first is simply to append the function to the initial menu in the module executive and modify the function routine invocation logic appropriately. The second action is to write the function routine itself. This requires creating meaningful prompts to solicit needed information from the user; gathering that information by a series of subroutine calls to the interface routine; and outputting the information in the proper format to any parameter files, the job control file, and the history file. Finally, the HELP module must be modified (simple additions to menus and branching logic, plus several WRITE statements to actually display the new information) to describe the purpose of the new function in image analysis, and to describe any needed parameters. All of the above, of course, assumes the prior existence of the batch program that actually implements the analysis function.

The use that BASS makes of the user information file could be greatly expanded. Users should be able to access data by name. In a remote sensing context, such data would probably be accessed by geographic area covered by a given image. Each area description should include data on files containing the image (both raw and processed) and existing parameter and JCL files used in that processing. Such a database was omitted from BASS, since a shortage of disk space on the host system precludes long-term online data storage. It would be a straightforward matter to include a modest file management module within the kernel.

Adding Interaction Mode Modules

Implementing the MID-level modules requires substituting a simple keyword matching parser in the module executive routine for the menu displayed by the LOW-level executive. Each function routine requires new, shorter prompts for the user, but the control flow is the same as for the LOW-level function routines. Using a text editor, such changes are simple. Implementing a second HELP module (HELP2) involves using shorter descriptions in all menus and adding a keyword matching procedure (similar to that in the executive) to the root menu routine to determine which function (if any) the user specified in the help request. The actual explanatory descriptions need not be changed. Additionally, the user interface must be enhanced to determine the presence or absence of a function name following HELP, and invoke HELP,

```

What task?
> CLUSTER(DVM=3.0,ITPN=2)
pfile?
> CLUSPFIL
already exist?
> NO
created
isw?
>

```

Figure 6—Sample user-system protocol at HIGH level

or HELP2 with the function name as a parameter, respectively.

Implementing the HIGH-level module requires a parser in the module executive that can determine which parameter values have been specified by the user in invoking a function. The parser could be passed an array containing the parameter mnemonics and return two arrays to the function routine: one containing values (in the same order as the corresponding mnemonics) and one containing logical values indicating whether a value has been supplied. The function routines could use the logical array to modify control flow based on values actually input.

CONCLUSION

Providing effective access to batch image analysis programs is a problem for many organizations engaged in transferring image analysis technology to new user communities. An ideal solution should require minimal supervision of users by training personnel, yet provide support and guidance to foster confidence in those new users. The solution should also provide skills and an environment that continue to be suitable for the users as they grow in sophistication.

This paper has described the design and partial implementation of an interactive system to create batch job streams for inexperienced users. The system allows users to select image analysis programs by function rather than name, guides users in selecting parameter values required by the batch programs, creates and allocates necessary disk files and other system resources, and provides online explanations of functions and individual parameters. Further, the system provides these capabilities at three different levels of user-system interaction, which allows users to maintain a high degree of efficiency as their sophistication increases, yet remain in a familiar, friendly environment.

The design reflects current viewpoints of important characteristics of the user-system interface. In addition, modularity and ease of expansion have been considered at all phases of the design. This has permitted a relatively inexpensive implementation of a system kernel and modules to provide interaction with a community of naive users, while insuring that additions to accommodate more sophisticated users will dovetail smoothly with that implementation.

Initial reactions to the existing system are strongly positive. As we gain experience with user interaction with BASS (and, conversely, as users gain experience with the system) implementation changes within the design framework are anticipated, and an eventual implementation of the complete BASS is expected, paced by user demand.

ACKNOWLEDGMENT

The author would like to thank John Nunneley, who implemented most of the function routines for the LOW-level module, and Larry Hofman and Dave Hein, who provided consultation on the SEL operating system and implementation.

REFERENCES

1. Hansen, J.V. "Man-machine communication: an experimental analysis of heuristic problem-solving under on-line and batch-processing conditions." *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, pp. 746-752, November, 1976.
2. Bennett, J.L. "The user interface in interactive systems." in *Annual Review of Information Science and Technology*, vol. 7, C.A. Cuadra, Ed., American Society for Information Science, Washington, D.C., 1972.
3. Mann, W.C. "Why things are so bad for the computer naive user." *National Computer Conference, AFIPS Conference Proceedings*, 1975.
4. Rouse, W.B. "Design of man-computer interfaces for on-line interactive systems." *Proceedings of the IEEE*, 63, (1975), pp. 847-857.
5. Hansen, W.J. "User engineering principles for interactive systems." *Fall Joint Computer Conference, AFIPS Conference Proceedings*, 1971.
6. Kasik, D.J. "Controlling user interaction." *ACM SIGGRAPH/Computer Graphics*, 10 (1976), pp. 109-115.
7. Nickerson, R.S. "Man-computer interaction: a challenge for human factor research." *IEEE Transactions on Man-Machine Systems*, (1969), MMS-10, pp. 164-180.
8. Sackman, H. "Experimental analysis of man-computer problem solving." *Human Factors*, 12 (1970), pp. 187-201.
9. Martin, J.D. *Design of man-computer dialogues*. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.
10. Miller, L.A., and Thomas, J.C., Jr. "Behavioral issues in the use of interactive systems." *International Journal of Man-Machine Studies*, 9 (1977), pp. 509-536.
11. Nievergelt, J., and Weydert, J. "Sites, modes, and trails: telling the user of an interactive system where he is, what he can do, and how he got there." *Berichte des Instituts für Informatik/Nr. 28*, Edgenössische Technische Hochschule Zürich, January 1979.

A unified approach to online assistance

by NATHAN RELLES and NORMAN K. SONDEIMER

Sperry Univac
Blue Bell, Pennsylvania

and

GIORGIO INGARGIOLA

Temple University
Philadelphia, Pennsylvania

Help! I need somebody.
Help! Not just anybody.
Help! I need someone. Help!
... Won't you please, please help me?
Help me! Help me!

© 1965 Northern Songs Ltd.
John Lennon and Paul McCartney

ABSTRACT

Many interactive computer systems have some form of HELP or assistance commands. Effective online assistance requires a well-defined framework that addresses the needs of both the end-user and the assistance provider. This paper presents such a framework, whose generality and usefulness come from an application-independent assistance processor and a highly structured database of assistance information. Major considerations are (1) the types of assistance interactive users need, (2) the data structures and relationships required to provide comprehensive assistance, (3) software architectures that encourage and support effective forms of assistance, and (4) the programming effort required to include and maintain online assistance. To make online assistance effective and economically feasible, the paper proposes a way to integrate assistance into other phases of the software life cycle.

INTRODUCTION

Ease of use is now recognized as a paramount goal in developing interactive software. The typical interactive user is no longer a data processing professional acting as an intermediary between problem solvers and a computer. Rather, systems are being used by the problem solvers themselves. These users want to interact with a computer without extensive training or programming skills. One way to meet these needs is through online assistance: useful reference informa-

tion, descriptions of possible actions, explanations of results, recognition of errors, and indications of recovery strategies.

Online assistance offers several advantages over conventional reference manuals and user guides. First, the physical distribution of systems and users and the requirement for regular updates can make online assistance more timely and economical. Second, some types of assistance can be provided online in a way that is awkward or impossible in written documents. For example, with online assistance a user can easily follow a chain of cross-references that would otherwise require considerable physical and mental dexterity with a written manual. Finally, the use of online assistance makes it easier to monitor a system's usability and identify those aspects that most frequently puzzle users.

Many commercial systems have some form of HELP or assistance commands.¹⁻⁶ These systems generally provide only reference assistance: summaries and elaborations like those normally found in a manual. Some experimental systems have provided more interactive assistance in the form of menu-selection, intelligent intervention, user-dependent protocols, and natural language interfaces.⁷⁻¹²

Effective online assistance requires a well-defined framework that addresses the needs of both the end-user and the assistance provider. In this paper, we describe a framework being developed to integrate online assistance with other elements of the software life cycle. The generality and usefulness of the framework derives from its use of an application-independent assistance processor and a highly structured database of assistance information. Major considerations are

the types of assistance interactive users need, data structures and relationships required to provide comprehensive assistance, software architectures that encourage and support effective forms of assistance, and the programming effort required to include and maintain online assistance.

PROBLEMS AND CHALLENGES

Despite the existence of some online aids, users are often forced to turn to conventional sources of information such as manuals, human consultants, and the tutorial graffiti that adorn so many terminals and their adjacent walls. One reason for this is the absence of policies and guidelines that encourage the provision of online assistance and software tools that simplify their implementation and maintenance. As a result, online assistance is not always available; where it is available, its characteristics do not always lend themselves to effective use.

To be used effectively, online assistance must have the following characteristics:

- **robustness**—the ability to answer a broad range of questions, not only about the system being used, but about related systems. While editing a program, for example, the user should be able to ask with equal facility about editing procedures, the programming language being edited, or the operating system.
- **flexibility**—assistance should be provided at a level of detail appropriate to a user's needs. Users are often hampered by having to look through several paragraphs to find the single fact they require. It should be possible to request concise descriptions or successively more detailed explication.
- **context sensitivity**—the ability to provide assistance relevant to the user's current situation. For example, when an error has occurred, assistance should be obtainable without requiring the user to identify the error or related conditions that caused it.
- **unobtrusiveness**—the ability to request assistance without interrupting the task at hand. Many assistance systems are implemented as independent job control statements. Having to terminate one's current task to ask for assistance is not only a distraction and a nuisance; it can also cause inefficiencies from having to save and restore the working environment. In fact, some programs cannot be suspended at all, precluding the availability of online assistance altogether.
- **consistency**—the ability to obtain assistance in similar ways on all of the interactive programs that comprise a larger system. The best way to achieve consistency is by integrating assistance facilities at the operating system level. Indeed, the absence of such integration is the major reason that assistance is often provided for an operating system's command language but not for its constituent application programs. Application developers have either omitted online assistance altogether, have devised their own inconsistent schemes for providing assistance, or have had to mimic as best they could the operating system's assistance processing capabilities.

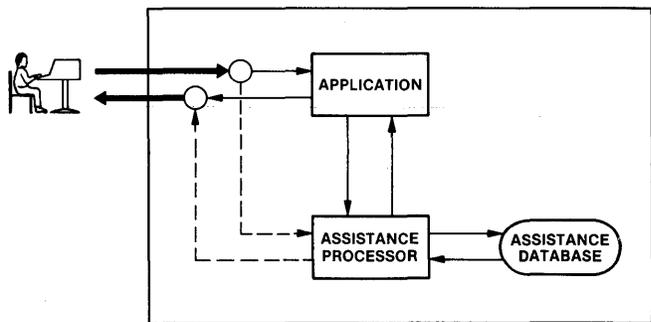


Figure 1—The assistance environment

- **cooperation**—the ability to perform an operation automatically after certain requests for assistance. For example, after a user asks how certain attributes of a file can be changed, it should be possible for the user to say, in effect, “okay—do that for me.”

The importance of these features is clear to anyone who has used (or watched others use) ineffective assistance systems. After a few unsuccessful attempts to get answers to their questions online, users revert to their more reliable sources of information, and the assistance system falls into disuse. Some empirical evidence suggests that the degree to which the above features are present has a significant effect on user performance and self-confidence.⁸

A UNIFYING FRAMEWORK

Overview

Given the above problems and challenges, we propose that an operating system include an Assistance Database (ADB) and an Assistance Processor (AP). The ADB is a highly structured database that can be used to represent the commands, concepts, and functionality of any application and of the operating system itself. The AP is an application-independent processor that interprets assistance queries against the ADB. A single processor can meet the challenge of consistency. With suitable software interfaces, it can be unobtrusive, context-sensitive, and cooperative. An appropriately structured database can make assistance information robust and flexible.

Figure 1 depicts the interaction between a user, an application program, and the Assistance Processor. A statement entered by the user is directed either to the application or to the AP. Requests for assistance are transparent to the application. Where appropriate, the AP may use menus or prompts to cope with ambiguities and potentially long explanations. The application program and the AP may exchange messages for several purposes, such as

- to establish the state of the user, e.g., the objects created, the sequence of commands used, or the last error made;
- to have the AP display explanations, default values, recovery strategies, and the like;
- to record information about users' requests for assistance; and

- to cause the application program to take specific actions corresponding to certain assistance requests (viz., “do that for me”).

The Assistance Database consists of three major parts, shown in Figure 2: fixed assistance data, fluid assistance data, and long-term (monitoring) data. Fixed assistance data comprises the permanent, largely invariant data created and maintained by an assistance administrator. The assistance administrator should be viewed as a role, not a person; it may be a single person, a group of “assistance providers,” or the combined activities of programmers, technical writers, and application experts. Fixed assistance data corresponds, for the most part, to the information normally found in user guides and reference manuals. The dictionary portion of the fixed data contains terms that can appear in assistance queries. The network of assistance information represents the concepts pertaining to an application and how they are interrelated. These concepts and relationships are also associated with segments of text, such as tutorial paragraphs, explanatory messages, command descriptions, and examples.

Fluid assistance data, possibly null at the beginning of each user session, reflects the state of the user’s interaction with the application and its assistance information. This contextual information enables the assistance processor to focus on desired information by using clues from the current query, prior queries, and previous interaction with the application. As is illustrated in Figure 2, fluid assistance data can also include a profile of the user’s experience level, frequency of use, interaction style, etc. Both the application program and the assistance processor can update the fluid assistance data, part of whose structure is linked to the permanent network of assistance data.

Long-term data consists of assistance queries that could not be satisfied, frequency tabulations for referenced messages and nodes of the network, and other statistics concerning the use of the assistance processor. This information is not intended so much for the AP as for the assistance administrator, who uses the information to maintain and improve the ADB.

The Assistance Database

A rich structure of information in the assistance network is crucial to providing effective assistance. We believe that research in knowledge representation has given us a basis on which to build such networks.¹³⁻¹⁵ This section sketches the structure of a network representation we are developing.

The most elementary forms of assistance are provided by a database of unrelated nodes. Each node represents a concept such as a command, parameter, or definition, and is linked to some portion of text. In Figure 3, we show the associations between concepts and their displayable text fragments as arcs labeled TEXT. This simple structure makes it possible to display assistance information associated with concepts the user can ask about. For example, a request for information about random files causes the assistance processor to display the text associated with that node.

To provide more diverse types of assistance, nodes can be interconnected in hierarchies. Several existing HELP systems

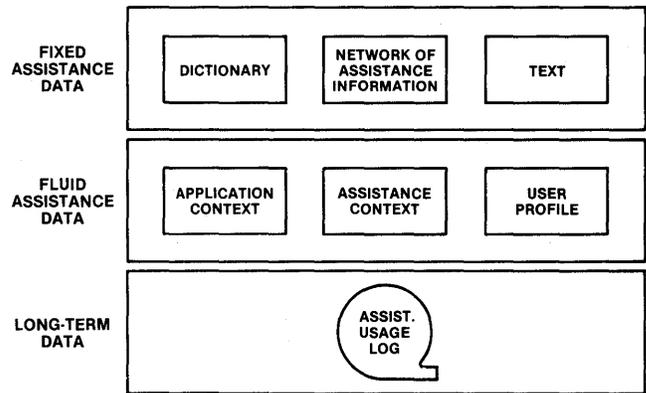


Figure 2—The assistance database (ADB)

incorporate such hierarchies in their assistance databases.^{1,3,4,6,7} The @@HELP system at the University of Wisconsin, for example, has a node for each program in the system library. Each such node has descendant nodes that describe alternative command formats; each format node has descendant nodes that describe the parameters and options in detail.⁶

Important improvements can be made to the way that hierarchical relations are currently provided in assistance databases. By having distinct hierarchical relations, a user can focus quickly on required information. Rather than having to display all of the descendant information related to a concept, a user can obtain just those parts that are related in a particular way. A fixed set of well-defined relations also enforces consistency in the network and directs the retrieval of assistance information. In an associative network, these hierarchies are represented as differently labeled arcs.

One of the most useful hierarchical relations is the IS-A relation, which associates several concepts with a more general or encompassing concept. In Figure 3, for example, the IS-A relation shows that all random files are files, as are all sequential files. Viewed as the subset relation, the IS-A re-

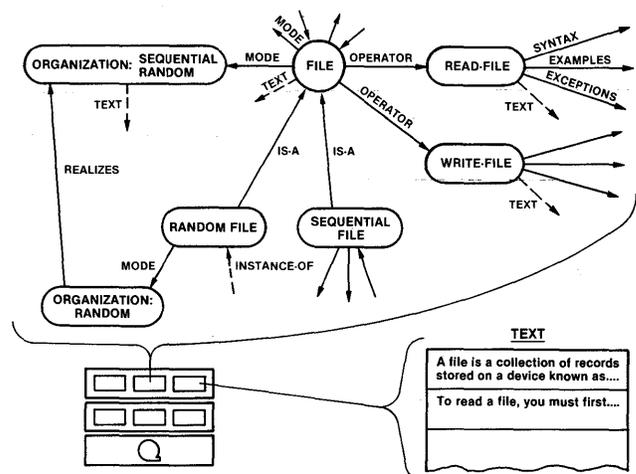


Figure 3—Sample of assistance data

```

COMMAND:  ▷ ARCHIVE
SOURCE FILE:  ▷ PROGTS
READ KEY:  ▷ R2D2
TERM DATE:  ▷ _

```

Please specify the date
after which the archive
entry may be deleted.



Figure 4—Context-sensitive online aids

lation can be used to describe many other types of concepts: commands, parameters, allowable values, and so on. Such relations enable a user to obtain descriptions at varying levels of generality.

Two other relations that are associated with the IS-A hierarchy are *MODE* and *REALIZES*. *MODE* relations introduce attributes of entities. Their labeling shows the limits on how these attributes must be realized. For example, in Figure 3 a *MODE* relation shows that files have *SEQUENTIAL* or *RANDOM* as allowable values for the attribute *ORGANIZATION*. The *REALIZES* relation indicates how an attribute for one entity is a limitation on the same attribute of a more general type of entity. In our example, the *ORGANIZATION* attribute of random files is shown to realize the organization of files in general. These relations make it possible to explain to a user the attributes of entities, the range of values they allow, and the way attributes differ between related entities. In addition, the absence of *REALIZES* relations can be used to allow inheritance of properties. For example, random files can be seen to allow the attributes and values of all other *MODEs* that the file concept may have.

Several other hierarchical relations are suggested in Figure 3. The *OPERATOR* relation identifies commands that may operate on an entity. For example, files are shown as allowing the *READ-FILE* and *WRITE-FILE* commands. The *SYNTAX* relation can be used to index textual descriptions of a command's syntax, *EXAMPLES* to access informative examples, and *EXCEPTIONS* to describe potentially unexpected behavior.

INSTANCE-OF relations provide another major improvement in the assistance network. This relation, absent in most *HELP* systems, associates actual user objects with the more general and abstract concepts represented in the assistance network. This can be seen as the set membership relation. In Figure 3, for example, the *RANDOM FILE* node is linked to an example file in the fluid portion of the *ADB*, through an *INSTANCE-OF* relation. Assistance can then be given in terms of a user's particular random files.

If online assistance is to be more effective than the index that appears at the back of a manual, we must allow more than the usual tree or lattice structures. The structure must provide arbitrary cross-referencing among related concepts. All of the nodes in Figure 3, for example, might be related to similar nodes in the database for a different programming environment. Interconnections are also necessary for representing error conditions. Each error state can be represented by a single node whose associated text describes the error and whose other relations identify the causes, prevention, and correction of that error. An error encountered in trying to write on a random file, for example, should have connections to the *RANDOM FILE* and *WRITE-FILE* nodes of Figure 3.

An assistance network of the type described above can be used to represent the functionality of many different systems. While the relations described are by no means exhaustive, they are sufficient for providing most forms of assistance currently available in commercial systems and some forms that are not yet available anywhere. Assistance based on such structures can be provided with robustness, consistency, flexibility, and context sensitivity.

The User's View

The assistance network is used by the *AP* to provide many different forms of online assistance. Traversal of the network, and the display of associated assistance information, is governed by interaction among the user, the application program, the Assistance Processor, and the Assistance Database. While it is possible to give the user "free reign" over the network's data, it is more useful to establish several positions in the database reflecting the user's current state.

Figure 4 illustrates a possible view of such an assistance environment from the user's standpoint. At any time during a session, the accessible assistance information is associated with suitably labeled function keys. By repeatedly pressing these keys, a user can obtain successively more detailed information sensitive to the current situation.

HOW TO INTEGRATE ASSISTANCE INTO THE SOFTWARE LIFE CYCLE

The availability of an assistance framework does not ensure that it will be used, let alone used effectively. Good assistance, like good documentation, does not come easily, and requires considerable resources. Some recent evidence on the cost of simplifying user interaction comes from the *PROMIS* system,⁷ where it is reported that menu screens are produced at a rate of about 500 per man-year. In another interactive system,⁸ the composition and maintenance of assistance messages comprised more than one-third of the system's total implementation time. It is clear that assistance information must be more economical to develop and maintain if it is to be included in the development of any software product.

It is equally clear that online assistance information must be consistent with written documentation. It should also go with-

out saying that both written and online documentation must be correct; i.e., they must be consistent with the functional requirements and specifications of a system. These goals cannot be met economically if information is duplicated in independent activities: requirements analysis, system specification and high-level design, documentation, and development of assistance information.

To integrate these activities effectively, their respective databases can be realized as subsets of a more generalized collection of data, which we call the Software Product Database. Because of considerable overlap in assistance information, documentation, and design specifications, such integration seems possible. Some work has already been done on the integration of requirements analysis and document preparation.¹⁶ Progress has also been made in combining online assistance facilities with the preparation of written documents.¹⁷ The total integration of assistance information with these activities is depicted in Figure 5. Paired broken lines represent interfaces to the Software Product Database. One interface translates parts of the database into structures that can be used in requirements analysis and system design. The interface also performs corresponding transformations from the requirements and design database back into the Software Product Database. Similarly, there is an interface for creating and maintaining a subset of the database to be used by a document preparation system. Finally, there is an interface that extracts information from the database to produce the dictionary, network, and text that constitute the assistance database.

The Software Product Database, then, is the central structure of the framework for online assistance. It must represent in an easily accessible form the information needed to construct cogent replies to assistance queries and it must be compatible with the information needed during other phases of the software life cycle. Our preliminary definition of this database is being developed as a series of interconnected units, each a structured entity describing an abstract notion or concrete object. We are particularly attracted to high-level languages like Ada¹⁸ that provide for just such entities and relations in their notion of a *module*. Modules can be used to describe tasks, abstract data types, and libraries of related declarations. With appropriate extensions, such modules can provide a basis for integrating requirements analysis, high-level design specifications, and assistance information in a single representation.

Figure 6 illustrates how some of the assistance information for files might be represented in an Ada-like language. Figure 3 contains the network representation of similar information. The PACKAGE module represents the notion of a file. The IS-A, SYNONYM, and RELATED clauses indicate that files are a kind of data structure, that a file is also known as a dataset or data-set, and that secondary storage and peripherals are related notions. The TYPE clause describes one attribute of a file, its organization. This clause specifies that a file may be either sequential or random, that organization remains constant throughout the life of a file, and that the default organization is random. The various TEXT clauses provide explanations for their respective concepts. Descriptions of the notions "record," "data structure," "secondary storage," etc., would appear in similarly structured units.

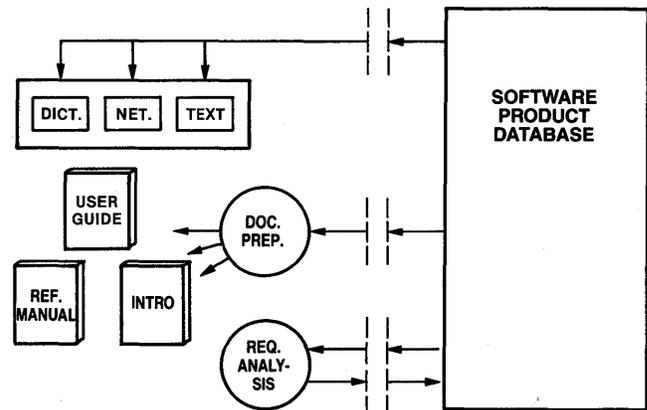


Figure 5—Assistance in the software life cycle

SUMMARY

Although online assistance facilities are common on most commercial systems, they are often limited and cumbersome to use. Today's user needs online assistance that is unobtrusive, robust, flexible, context-sensitive, and consistent. At the same time, it must be easy and economical to maintain the complex information that goes into online assistance. Ad hoc techniques are not adequate for providing effective online assistance.

We have presented a framework being developed to provide online assistance, based on an associative network of assistance information accessible through an assistance processor. Because of the network's generalized structure, assistance can be provided for a wide range of applications and for experienced as well as naive users. The assistance processor must be an integral part of the operating system, so that information can be obtained consistently, easily, and in terms of a user's

```
PACKAGE[file;
  TEXT "A data structure for storing and retrieving";
  "objects to and from secondary storage"] is
  IS-A data-structure;
  SYNONYM dataset, data-set;
  RELATED secondary storage, peripheral;
  :
  TYPE kind-of-organization is (
    [sequential;
      TEXT "A form of organization that allows only"&
        "sequential access to records"],
    [random;
      TEXT "a form of organization that allows both"&
        "sequential and random access to the "&
        records of a file"]);
  organization: CONSTANT kind-of-organization:=
  random;
  :
```

Figure 6—Ada-like Assistance Information

current environment. To simplify the creation and maintenance of assistance information, a centralized database can be used in several phases of software development: requirements analysis, high-level design specification, and documentation. Extensions and modifications to Ada are suggested as an effective way of representing the information in such a database.

While much can be done with current technology to improve online assistance, many issues merit further investigation. The generality and usefulness of the associative network depend on a set of relations that is suitably complete but not prohibitively complex. Arriving at this optimal set will become easier as we learn more about the kinds of assistance users require in various interactive settings. We also need to know more about classes of users and how their needs change as they become more experienced. The provision of intelligent assistance or system-initiated assistance also requires a better understanding of such user characteristics as experience, frequency of use, and programming skill. Some of this information is becoming available as a result of controlled experiments,^{19,20} other insights may be gained by simply monitoring the use of assistance facilities. Constructing assistance databases and integrating them with other development activities are complex tasks. How complex the tasks and with what improvements to ease of use can only be determined through continued research and the implementation of prototype systems.

REFERENCES

1. SPERRY UNIVAC 1100 Series Conversational Time Sharing (CTS) System: Programmer Reference; UP-7940, Blue Bell, PA: Sperry Univac Computer Systems, 1977.
2. Holg, Chloe, *The Joy of TENEX and TOPS-20 . . . in Two parts*, University of Southern California: Information Sciences Institute, Technical Report ISI/TM 79-15, January 1979.
3. *Interactive Facility Version 1 Reference Manual, CDC Operating System NOS I*, St. Paul: Control Data Corporation, 1978.
4. Thompson, K. and D.M. Ritchie, *UNIX Programmer's Manual: Sixth Edition*, Murray Hill, NJ: Bell Laboratories, 1976.
5. *IBM System/38 Technical Developments*, ISBN 0-933186-00-2, IBM Corporation, 1978.
6. Anderson, Jess, "@@HELP: Online Documentation System", in *Technical Papers, USE Spring Conference*, 1979, Bladensburg, MD: USE, Inc., pp. 215-235.
7. Robertson, G., A. Newell, and K. Ramakrishna, *ZOG: A Man-Machine Communication Philosophy*, Pittsburgh, PA: Carnegie-Mellon University, Dept. of Computer Science, August 1977.
8. Relles, Nathan, *The Design and Implementation of User-Oriented Systems*, Computer Sciences Technical Report #357, University of Wisconsin-Madison, July, 1979.
9. Roberts, R., "HELP—A Question Answering System," in *Proceedings, 1970 Fall Joint Computer Conference*, pp. 547-554.
10. Shapiro, Stuart C. and Stanley C. Kwasny, "Interactive Consulting Via Natural Language," *Communications of the ACM*, 18:8, 1975, pp. 459-462.
11. Ash, W., R. Bobrow, M. Grignetti, and A. Hartley, *Intelligent On-line Assistant and Tutor System*, Technical Report No. 3607, Bolt Beranek and Newman, Inc., January, 1977.
12. Burton, Richard R. and John Seely Brown, *An Investigation of Computer Coaching for Informal Learning Activities*, BBN Report No. 3914, ICAI Report No. 12, Cambridge, Massachusetts: Bolt Beranek and Newman, Inc., August 1978.
13. Fahlman, Scott, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, 1979.
14. Findler, N. V., *Associative Networks: Representation and Use of Knowledge by Computers*, New York: Academic Press, 1979.
15. Brachman, Ronald J., *A Structural Paradigm for Representing Knowledge*, Ph. D. Dissertation, Harvard University, 1977.
16. Funk, Susan, "Putting PSL/PSA to Work," in *Technical Papers, USE Spring Conference*, 1979, Bladensburg, MD: USE Inc., pp. 57-78.
17. Price, Lynne A., *Representing Text Structure for Automatic Processing*, Computer Science Technical Report #324, Madison: Computer Sciences Department, University of Wisconsin-Madison, May 1978.
18. Ichbiah, J.D., "Preliminary Ada Reference Manual," in *SIGPLAN Notices*, 14:6, June, 1979.
19. Shneiderman, Ben, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.
20. Miller, L.H., "A Study in Man-Machine Interaction," in *Proceedings, 1977 National Computer Conference*, pp. 409-421.

An experimental system to support a very high level user interface

by WILLIAM L. BATCHELOR and
LUCIAN J. ENDICOTT, JR.

IBM Information Systems Division
Rochester, Minnesota

ABSTRACT

An experimental project is described that developed a very high level end user interface to provide the capability for non-DP trained individuals to automate business procedures.

INTRODUCTION

The Nonprogrammer Interface (NPI) project is an experimental project at the IBM Information Systems Division development laboratory in Rochester, Minnesota. This paper describes the environment in which NPI arose, NPI itself, and the system requirements posed by NPI. It should be noted that this paper describes experimental activity only. No inferences should be drawn in regard to planned or future IBM products.

ENVIRONMENT

Today's data processing environment is characterized by the continually increasing use of computers by individuals: at home, in school, at the office, and in the factory and other work places.¹ Representative of this is the current interest and activity in the area of office automation.^{2,3} This increasing use has produced, and will continue to produce, numerous problems, both social and technical. At the moment, the overriding problem appears to be the ability and training required for an individual to make significant use of the function provided.⁴⁻¹¹

In the early and middle 1970s, roughly from the introduction of the pocket calculator to the announcement of such machines as the TRS-80, it became apparent that the limiting factors in the use of data processing equipment were primarily in programming and systems planning and design. Since this was a common concern throughout the industry, numerous research and advanced technology efforts were established within IBM and most other companies faced with the problem to reduce the extent of one or both of these limitations.

The IBM Rochester Laboratory has had considerable experience in development of systems and products for what was then considered to be the small user. These systems and prod-

ucts include the IBM System/3, the IBM System/32, the IBM 5100, and the IBM 3741, among others. It was felt within the laboratory that this experience, coupled with an empirical rather than a theoretical approach to system design, should be applied to attempt a quantum improvement in ease of use at the very low end of the data processing marketplace. This led to the establishment of the NPI project.

The Nonprogrammer Interface Project

The objectives of the NPI project were twofold:

1. To develop a very high level interface to make possible the preparation of business forms and reports of all types with associated file creation, access, and maintenance, as they relate to billing, inventory control, accounts receivable, sales analysis, etc.
2. To test the use of this interface by non-DP trained business people, both professional and nonprofessional, and by DP trained people in both casual and noncasual use.

The conclusions drawn by the NPI project will be outlined in the following section. The remainder of this section will offer a brief description of the NPI experimental system.

The NPI experimental system was implemented in APL on the IBM System/370. IBM 5100 and 5110 systems were used as terminals. Implementation in APL on the System/370 was chosen because of ease of both implementation and modification since a largely empirical approach to the problem was planned. As indicated above, the system did evolve over a period of time. No attempt will be made here to trace this evolution; only the final version of the experimental system will be described.

The final version of the system was interactive, output-oriented, and nonprocedural in nature (other than for pocket calculator type functions). Information could be entered into the system only in response to a system-posed question, and only described data could be entered into the system.

Basically, a set of menus and prompts were provided—the sequencing and tailoring were provided by the system in response to user answers, and user names were substituted where appropriate. The system required fewer than 20 menus

and fewer than 75 prompts or questions. Most of these were seen by the user only when an invalid response was provided.

Some other significant characteristics of the system were

- A specific sequence of menus and prompts would not terminate until a legitimate function had been performed or a legitimate algorithm defined. For example, NPI could not generate an application that would not run, although the user could produce a result other than that intended.
- The user could not "look" inside the system. Information and data were returned to the user only in the format in which they were entered or described to the system.
- A complete set of system documentation was always available on line to the user, and the "HELP" response was almost always a valid response.
- Manual office procedure concepts and terminology were employed wherever possible. Traditional DP concepts and terminology were kept to an absolute minimum (some concepts were particularly troublesome—the name of a field versus its content, key fields, control breaks, transaction files, etc.—but acceptable approaches were finally found).
- Processing and file activity were generally derived by implication, but specific manipulation was permitted on request (but only via question and answer, again apart from pocket calculator type functions).
- Questions were only asked in their "natural" (to the user) sequence (this sequence was determined empirically and is not "natural" to all users). The user was not asked a specific question unless it was one that he would have asked of himself at the corresponding point in manual processing.

Application Generation: The user is oriented toward the output desired. Application generation is accomplished by a dialog with the user in regard to output. Generally, this will be the completion of some form (e.g., an invoice) or the generation of some report or query. Other types of applications can be written (e.g., modify files), but in general they are not necessary since the system accomplishes this implicitly and also provides capability for the user to accomplish these functions explicitly via menus.

Basically, for application generation the user is asked to describe the form or report. It may or may not contain pre-printed information. The user can describe the form such that the printer will include the form on the output report. This form information can be named and used again for future applications. The system then interrogates the user about the variable content of the report: whether the content occurs one time or is replicated, whether and where it appears on the report (printing can be conditionally suppressed), how long it is, whether it is alpha or numeric, decimal position, where the information is to be obtained, and whether or not it is to be permanently saved. (Additional editing options are provided to the user on request at a later point.)

Information can be obtained from four sources:

1. It can be keyed in. At run time the system will interrogate the user for the information.

2. It can be looked up. The system will obtain the information from its files. Since initially there are no files in the system, empty files are created to correspond to the application generated if look-up is specified as the source. At (first) run time, the system will interrogate the user for missing file information, thus automatically filling in the file while executing the application.
3. It can be computed. This again is accomplished by question and answer using hand calculator type functions plus conditionals ($>$, $<$, $=$, etc.).
4. It can be obtained from previous application executions (i.e., transaction history files).

Application Modification: Applications are modified via menus. Almost any type of modification is possible. After modification, renaming the application will result in both versions being retained. Not renaming the application will cause the unmodified version to be replaced by the modified version.

Application Execution: Applications are executed via menus. Applications not requiring user input will execute in batch mode. Applications requiring user input will execute in interactive mode. A complete journal and history file is maintained.

Files: As previously stated, files can be created and updated either implicitly or explicitly. The most common approach is implicit. Files are created and maintained as a byproduct of application generation, reflecting the user's visualization of the filing system (e.g., customer file, parts file). All fields are named, and associations are generated as implied or stated by the user. Any field can be used for accessing of data; however, the primary key must always be used for updating the file.

Files can be updated explicitly by applications or by the user. If by the user, it is accomplished via menus as is explicit creation (files cannot be explicitly created by an application). Inquire and query functions are also supplied by the menus. Query functions named and saved can be executed as applications.

Other Functions: There are several other functions in the system. The most important of these have already been referenced:

1. System documentation is implicitly produced and presented to the user on request. Portions may be presented as "help" to the user under error conditions. These include application listings, file descriptions, etc., which closely parallel the question and answer creation process.
2. Trace provides a debug tool. The application steps are traced exactly as they are documented in the system documentation.
3. Audit permits an application to be rerun in virtual mode (i.e., no permanent changes are made to the file). This has the added advantage of providing a spool-type function if the printer is not available at execution time.

CONCLUSIONS DRAWN FROM THE NPI PROJECT

A number of different conclusions came out of the NPI project. Some of the more significant ones follow:

- It appears possible to provide significant DP capability to non-DP trained individuals. These individuals must have a good knowledge of the manual procedures to be automated. (The NPI project addressed only the automation of existing manual systems and procedures—it did not address the more difficult problem of developing new systems and procedures.)
- An empirical approach to human/machine interface design is more productive than presently understood theoretical approaches. In particular, it appears now that ease of use involves hundreds, perhaps thousands, of nuances as much as one or a few basic principles.
- For an individual or an organization to make effective use of DP capability without expensive training or use of consultants, it is necessary to provide the capability to automate existing manual procedures with absolutely minimal modification. Particularly, the introduction of new concepts and vocabulary must be held to a minimum.
- People learn most easily and quickly by doing, rather than by reading or being told. For a DP system to be successful at the very low end, productive use must be possible within a few minutes—at the most one to two hours. It must be possible to generate “running” (i.e., one that produces output) applications very quickly and then iteratively modify them until the desired results are obtained. Associated with this are three specific requirements:
 1. It must be impossible to generate applications that hang or crash.
 2. Applications (as well as application generation) must be able to run backwards, in effect, back out the database.
 3. It should be possible to run applications in test mode.
- Basic menus for application generation must be very few—the beginning user should never be concerned with, or even aware of, most of the system’s capability (for example, all standard forms layout and editing capability, including picture clause, must be provided; however, the beginning user is not aware that these exist and need only find out about them when he or she decides that the default layout and edit functions are not adequate).
- Backup and recovery functions must be as automatic as possible and nonoptional. At the best, the user should only be concerned with mounting, demounting, and preserving, through some period, detachable media.
- To provide the functions and characteristics described above, it is necessary to control very carefully input into the system. Information and data can be entered into the system only in logical and unambiguous fashion—it is for this reason that NPI evolved into a menu-driven, non-procedural system permitting only described data. In addition, individual solutions must be provided for a large number of problems, e.g., control break, page overflow, field overflow, field overlap, horizontal versus vertical spread, time and date handling (formatting, sorting), multiple key sorting, aliases, duplicate names, currency specification, etc.—the nuances to which allusion was previously made.
- The keystone of any system designed to provide all or a significant number of the functions available in the NPI experimental system is a system-wide dictionary. The user must be able to name elements in the system, and he must be allowed both duplicates and aliases, and these elements must be known unambiguously to the system. From this the characteristics of the dictionary and the database naturally follow:
 1. Everything in the system (atomic elements, lists, lists of lists, etc.) is an object in the database. Each object may have a name, which may or may not be unique. Each object does have an object ID, which is unique, and may have an object description, which should be unique.
 2. Every object is listed in the dictionary. Access paths are built using the dictionary. Access to an object is via either the dictionary or an existing access path.
- Most of the functions provided by the NPI experimental system are available in one form or another in most DP systems today, yet these systems are strikingly lacking in the ease of use observed in NPI. Why? The answer would appear to be in structure. Ease of use is determined by the human/machine interface. The interface is determined by the underlying structure. The structure may be pure, i.e., the only structure in the system, or it may merely be interposed, i.e., masking the “real” structure of the system from the interface.

SUMMARY AND CONCLUDING OBSERVATIONS

Characteristics of the end user interface included:

- All documentation online.
- Educational mode, novice mode, and experienced user mode available.
- Learning by doing facilitated in several ways: output always generated, ease of application modification assured, test mode provided, backout and rerun always available.
- Information entered into system only by answering questions and responding to prompts.
- Apart from hand calculator type functions, only non-procedural information requested.
- Only described data permitted into the system.
- Data returned to the user only in the manner in which described to the system.

The key system design requirement posed by the end user interface was for a complete system-wide dictionary supported by a database with full relational capability. No characteristic of the dictionary or database could be permitted to show at the end user interface.

In particular, it was asserted that, at the present, ease of use appears to be composed of a large number of small factors, that these factors can best be addressed empirically rather than theoretically, and that they apparently can be satisfactorily provided only by restructuring the design of DP systems.

ACKNOWLEDGMENTS

In addition to the authors, the members of the NPI project were Roger F. Dimmick, George G. Gorbatenko, E. R. (Jed) Harris, Griff H. Rees, Walter S. Schaffer, and Phil C. Schloss.

REFERENCES

1. Press, Larry, Rothenberg, Jeff, and Carlstedt, Jim, "The Next Generation of Personal Computers: A Position Paper," *ACM SigPC Notes*, Vol. 1, No. 4, Spring 1979.
2. A Status Report on the Activities of the CODASYL End User Facilities Committee (EUFC), EUFC, P.O. Box 1808, Washington, DC, 20013, February 1979.
3. Ellis, Clarence A. and Nutt, Gary J., "Office Information Systems and Computer Science," *ACM Computing Surveys*, Vol. 12, No. 1, March 1980.
4. Athey, Thomas H., "Small Business—A Gold Mine for DP Educators," *Proceedings of the ACM 1978 Annual Conference*, Washington, DC, December 4-6, 1978.
5. Embley, David W., "Forms-Based Automatic Program Generation," *Proceedings of the ACM 1978 Annual Conference*, Washington, DC, December 4-6, 1978.
6. Fiszer, Max, "A View of Language Futures (The User Interface)," *SHARE Proceedings Number 51*, August 20-25, 1978, Boston, MA, Vol. 1, pp. 549-560.
7. Newman, I. A., "Personalized User Interfaces to Computer Systems," *Proceedings of the European Computing Congress*, London, England, May 9-12, 1978, pp. 473-486.
8. Shneiderman, Ben, *Software Psychology*, Winthrop Publishers, Incorporated, Cambridge, MA, 1980.
9. Thomas, John C., "Psychological Issues in Data Base Management," *Proceedings of the Third International Conference on Very Large Data Bases*, Tokyo, October 6-8, 1977.
10. Zloof, Moshe M. and de Jong, S. Peter, "The System for Business Automation (SBA): Programming Language," *Communications of the ACM*, Vol. 20, No. 6, June 1977.
11. Infotech State of the Art Report, *Man/Computer Communication*, 2 Volumes, Infotech International Limited, Maidenhead, Berkshire, England, 1979.

Principles of good software specification and their implications for specification languages*

by ROBERT BALZER and NEIL GOLDMAN

USC/Information Sciences Institute
Marina del Rey, California

ABSTRACT

Careful consideration of the primary uses of software specifications leads directly to three criteria for judging specifications, which can then be used to develop eight design principles for "good" specifications. These principles, in turn, result in eighteen implications for specification languages that strongly constrain the set of adequate specification languages and identify the need for several novel capabilities such as historical and future references, elimination of variables, and result specification.

INTRODUCTION

Many computer languages have been designed, without explicitly stated goals. We attempt to reverse this process by carefully developing the goal structure before designing a language to satisfy it.

A few other languages have been constructed using a similar paradigm, most notably PASCAL¹ and EUCLID.² Each was strongly influenced by the unique goal structures chosen (simplicity for PASCAL and provability for EUCLID). So too do we expect our language, not yet designed or implemented, to be strongly influenced by the goal structure chosen.

Our contribution lies in the extent to which we have developed the explicit goal structure and the implications of this for the structure and features of any language that would satisfy these goals.

CRITERIA FOR JUDGING SPECIFICATIONS

In order to establish the criteria to be used in judging software specifications, we begin by considering their primary uses. First, (and most important), a software specification is a contract between the specifier and the implementor defining the system to be constructed. It therefore must be clearly and

unambiguously understandable by both parties. Thus, understandability is the first criterion for judging specifications.

Second, it must be possible to ascertain whether an implementor has fulfilled such a contract; that is to test, in the broadest sense, whether a specification and an implementation are equivalent. In addition, before entering into an implementation contract, a specifier must be able to ascertain that the specified system meets the needs for which it was designed. Thus, the specification itself must be testable. Hence testability is the second criterion for judging specifications.

Finally, because this contract will change over time, it must be easy to modify the specification. Hence, maintainability is the third and final criterion. Optimization is conspicuous by its absence from the list of criteria. This is intentional. Not only is optimization the proper concern of the implementor, but it conflicts with each of the identified criteria. Optimization represents the spread of information which, by increasing the interdependence of the components on each other, increases the complexity of the whole. This reduces the understandability, maintainability, and testability of the specification. Thus, not only is optimization *not* a proper criterion for judging specifications, but specification languages should actively attempt to preclude the optimizability of specifications.

PRINCIPLES OF GOOD SPECIFICATION

Principle 1: Separation of Functionality from Implementation

First, by definition, a specification is a description of WHAT is desired, rather than HOW it is to be realized (implemented). These specifications can assume two quite different forms. The first form is that of mathematical functions: Given some set of input, produce a particular set of outputs. The general form of such specifications is find [A/THE/ALL] result such that P(input), where P represents an arbitrary predicate. In such specifications, the result to be obtained has been entirely expressed in a WHAT (rather than HOW) form. In part this is because the result is a mathematical function of the input (the operation has well defined starting and stopping points) and is unaffected by any surrounding environment.

* This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DAHC15 72 C 0308.

©1979 IEEE. Reprinted, with permission, from *Proceedings Specifications of Reliable Software*, April 30, 1979, Cambridge, MA (79CH1401-9C).

Principle 2: A Process-Oriented Systems Specification Language Is Required

Consider instead a situation in which the environment is dynamic and its changes affect the behavior of some entity interacting with that environment (as in an "Embedded Computer System"). Its behavior cannot be expressed as a mathematical function. Rather, a process-oriented description must be employed, in which the WHAT specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.

Such process-oriented specifications, presenting a model of system behavior, have normally been excluded from formal specification languages, but they are essential if more complex dynamic situations are to be specified. In fact, it must be recognized that in such situations both the process to be automated and the environment in which it exists and must be described formally. That is, the entire system of interacting parts must be specified, rather than just one component.

Principle 3: Specification Must Encompass System Of Which Software Is a Component

A system is composed of interacting components. Only within the context of the entire system and the interaction among its parts can the behavior of a specific component be defined. In general, a system can be modeled as a collection of passive and active objects. These objects are interrelated with each other and over time the relationships among the objects change, which provides the stimulus to which the active objects, called agents, respond. The responses may cause further changes and hence additional stimuli for the agents to respond to.

Principle 4: Specification Must Encompass Environment In Which System Operates

Similarly, the environment in which the system operates and with which it interacts must be specified.

Fortunately, this merely necessitates recognizing that the environment is itself a system composed of interacting projects, both passive and active, of which the specified system is one agent. The other agents, which are by definition unalterable, limit the scope of the design and implementation to follow.

It should be noted that the picture of system specification presented here is that of a highly intertwined collection of agents reacting to stimuli in the environment (changes to objects) produced by each other. Only through the coordinated actions of the agents are the goals of the system achieved. Such mutual dependence violates the principle of separability (isolation from other parts of the system and environment). But this is a DESIGN principle, not one of specification. Design follows specification and is concerned with decomposing a specification into nearly separable pieces in preparation for implementation. The specification, however, must accurately portray the system and its environment as perceived by its user community in as much detail as required by

the design and implementation phases. Since this level of required detail is difficult, if not impossible, to foresee in advance, the specification, design, and implementation processes must be recognized as an iterative activity. It is therefore critical that technology exist for recovering as much of this activity as possible as the specification is elaborated and modified (during both initial development and later maintenance).³

Principle 5: System Specification Must Be a Cognitive Model

The system specification must be a cognitive model rather than a design or implementation model. It must describe a system as perceived by its user community. The objects it manipulates must correspond to the real objects of that domain; the agents must model the individuals, organizations, and equipment in that domain; and the actions they perform must model those actually occurring in the domain. Furthermore, it must be possible to incorporate into the specification the rules or laws which govern the objects of the domain. Some of these laws proscribe certain states of the system (such as "two objects cannot be at the same place at the same time"), and hence limit the behavior of the agents or indicate the need for further elaboration to prevent these states from arising. Other laws describe how objects respond when acted upon (e.g., Newton's laws of motion). These laws, which represent a "physics" of the domain, are an inherent part of the system specification.

Principle 6: Specification Must Be Operational

The specification must be complete and formal enough so that it can be used to determine whether a proposed implementation satisfies the specification. That is, given the results of an implementation on some specific set of data, it must be possible to use the specification to validate those results. This implies that the specification, though not a complete specification of HOW, can act as a generator of possible behaviors among which must be the proposed implementation. Hence, in an extended sense, the specification must be operational.

This operability may exist only in a theoretical sense, since it involves replacing existentially and universally quantified objects in the specification by brute force generation and testing (the British Museum algorithm) of all possibilities (which may be infinite). But given specific possibilities to test (as generated by the proposed implementation), the specification becomes a validation filter for them (it does not, however, guarantee that all valid possibilities will be generated by the implementation, only that those generated are valid).

Principle 7: The System Specification Must Be Insensitive to Incompleteness

No real specification can ever be totally complete. The environment in which it exists is too complex for that. A specification is always a model—an abstraction—of some real (or envisioned) situation. Hence, it will be incomplete. Fur-

thermore, as it is being formulated it will exist at many levels of detail. The operability required above must not necessitate completeness. The analysis tools employed to aid specifiers and to test specifications must be capable of dealing with incompleteness. Naturally this weakens the analysis which can be performed by widening the range of acceptable behaviors which satisfy the specification, but such degradation must mirror the remaining levels of uncertainty.

Principle 8: Specification Must Be Localized and Loosely Coupled

The previous principles deal with the specification as a static entity. This one arises from the dynamics of the specification. It must be recognized that although the main purpose of a specification is to serve as the basis for design and implementation of some system, it is not a precomposed static object, but a dynamic object which undergoes considerable modification. Such modification occurs in three main activities: formulation, when an initial specification is being created; development, when the specification is elaborated during the iterative process of design and implementation; and maintenance, when the specification is changed to reflect a modified environment and/or additional functional requirements.

With so much change occurring to the specification, it is critical that its content and structure be chosen to accommodate this activity. The main requirements for such accommodations are that information within the specification must be localized so that only a single piece (ideally) need be modified when information changes, and that the specification is loosely structured (coupled) so that pieces can be added or removed easily, and the structure automatically readjusted.

IMPLICATIONS FOR SPECIFICATION LANGUAGES

Having set forth the principles of good specification in the previous section, we now derive the implications of these principles on specification languages. References to the principles, and to earlier implications, are embedded in parenthesis and are referenced by principle (P) or implication (I) number.

Implication 1: Logical Data Specification and Access

Since a specification must deal with functional behavior rather than the implementation (P1), the data manipulated in the specification must be representation-independent. The specification must thus be described at the logical level by defining the methods of getting from one data item to another (access paths) and the operations that can be performed upon a data item.

Implication 2: Uniform Data Specification

Since the logical data specification should make no implications about data representation (P1), the principle of parsimony requires that a single uniform data specification be used

for all data and that this specification not preclude any possible representations.

Implication 3: Relational Data Model

This requirement of uniformity (I2) has strong implications. It forces a quite unconventional specification of data to be adopted. The conventional view of data as having a "value" and being composed of a collection of parts is fraught with difficulty. What is the "value" of a data item, what is its range, when is the "value" used rather than the item, and how far does the "boundary" of the data item extend? There are no easy answers to these questions, which arise from choosing a representation-oriented view. Instead, a functional view leads to a very simple, yet general data specification which avoids these difficulties.

Objects are associated with one another, and their relationships can be used to access them. An object has no "value" or "boundary." Rather it is defined by the set of associations it forms with other objects. This definition is necessarily circular, but such circularity causes no problems because after some point further chains of associations become irrelevant for the processing being performed.

Thus data is defined simply by specifying the relations existing among the objects. There are only five basic operations that can be performed on such data. Objects can be created or destroyed. Similarly, relations among two or more objects can be created or destroyed. Finally, one object can be accessed from another via one of these relations. The symmetry of the relational specification is particularly nice since it is just as easy to access one object from another via a particular relation as it is to use the second object to access the first via that same relation. Using Data Base terminology, this means that the data base is fully associative, or equivalently, fully inverted.

It should be noted here that although this type of data specification is quite unconventional in software and system specifications, it is becoming prevalent within the data base community. Unfortunately, because this community is concerned with efficiency, it has adopted a particular canonical form of relational specification (Third Normal Form⁴) rather than allowing the full generality of the formalism. Clearly, for the purposes of functional specification such a restriction should not be included. Instead the recent data base work on semantic models⁵⁻⁹ more closely match the general relational model required.

It should also be noted that the general relational specification is entirely equivalent to the Semantic Net representation¹⁰ widely used in the Artificial Intelligence community. The reasons for its adoption by this community are quite instructive. Artificial Intelligence systems are designed to deal with uncertainty which arises in the data to be processed and/or the processing to be applied to it, which prevents optimization of the data structures and necessitates a very general expression of its functionality. These reasons are very similar to our own, although the motivation is distinct. We need to express the functional characteristics of the data while delaying consideration of representation and optimization to the implementation phase of development.

Implication 4: Global Model

A model of the objects, both passive and active, manipulated by the agents and providing stimuli for them must be maintained (P3). Since new agents can be added to the specification or their stimuli changed (P8), any object may serve as part of an agent's stimuli, and hence must be globally maintained.

This global data base represents a dynamic model of the environment in which the system operates. The model's dynamics are governed by the sequence of actions performed, and these actions, together with the changes made to the objects in the model, constitute the observable behavior of the agents. The agents also gather information from the global model to make decisions of whether or not to perform an action, and if so, which one. These information-gathering and decision-making processes constitute a model of the agent; it is just such a model that can be embodied in computer software.

System specifications contain one or more such agents. The objects and actions of the global model must also be defined, and models of each agent provided. However, much variability is allowed in the completeness of specifying an agent model (P7). The model must define which actions the agent is allowed to perform, but it need not specify the information gathering processes used by the agent in deciding which actions to perform or their order. It may contain partial descriptions of these processes or merely constraints on the behavior of the agent.

If each agent is completely specified, the system can be simulated (P1); that is, the global model, either a particular instance or a symbolic version, can be advanced through successive stages. If the agents are incomplete, such simulated behavior can be accomplished only by having the user interactively inject agent actions into the sequence of actions being performed. As the completeness of agent description decreases, the amount of automatic analysis and checking which can be performed decreases correspondingly. However, partial descriptions are retained as consistency checks when supplanted by more complete descriptions or when agent behavior is provided by the user.

Implication 5: Global Data Base with Inference

The global model must be maintained in a data base which supports inference. The global model is a simulation of the environment in which the system operates. Normally such environments are quite rich and many of the relations between objects can be deduced (inferred) from other relations within the model. Since the specification can, and will, undergo much modification, the principle of locality and loose coupling (P8) requires that neither the use of information nor its method of derivation be explicitly determined in the specification. That is, any information requested from the global model should be available independent of whether it was directly produced by some action or could have been inferred from such directly produced data.

For each data item a choice must be made between explicitly representing that item in a data base and maintaining it as

actions are performed, or deriving (inferring) it when needed. Such choices are quite critical to the effective operation of a system, but they are implementation choices and have no place within the system specification (P1). Inference mechanisms provide a way of delaying these choices until implementation without sacrificing the operability of the specification (P6). The need to hide the distinction between explicit and implicit data through the use of inference implies that the global model is maintained in a data base accessible only through an interface which supports inference.

It should be noted that the "computation rules" of data flow languages¹¹⁻¹³ are a special case of inference rules. The advantage of specifying computation via such rules is that the control structure has been suppressed from the specification and these rules are invoked whenever necessary. This suppression of the control structure enables the user to specify the functional relations between objects (WHAT) without specifying when (HOW) to compute them (P1).

When appropriate, this method of specification should be heavily utilized. However, its applicability is limited to static situations in which the global model isn't changing (no actions are being applied). Rather, only the state of explicit knowledge about the global model is being altered as information is derived from other information.

A common specification mistake is the failure to differentiate the actual actions occurring in the global model, to which the agents respond and which they produce, and the "information actions" performed by an agent to produce required data. These latter actions are merely an implementation mechanism for hiding the distinction between explicit and implicit data, and hence have no place in the specification. Instead, only the functional basis (the inference rules) for such "information actions" should be specified.

Implication 6: Descriptive Reference

Since the global model is being maintained in a data base which obscures the distinction between explicit and implicit data (I5), references to data from the model must operate indirectly through some language processed by the data base rather than by direct access to the data itself. The use of a data request language with the support of an inference mechanism is one step in the direction of separating the specification of what data is required (functionality) from its method of access (implementation) (P1). A second step is the use of a fully associative relational data base.¹³ The final step is the capability to access data by describing its attributes—descriptive reference. Thus, a pattern is used to specify which relations the desired object(s) must have to other objects, which are prescribed, and which are optional. All of these conditions must be simultaneously satisfied for an object to match the pattern—the descriptive reference. The objects used to describe the desired object may themselves be descriptively described, and so on, so that very general descriptions can be composed. These descriptive references require a quite complex pattern match mechanism, but the specification is only concerned with functionality. A major portion of a systems implementation will, however, be concerned with simplifying

these data access mechanisms by proper choice of data structures and use of facilitating computations.

As a result of a successful descriptive reference, an association is made between the descriptor and a name called a placeholder. The placeholder can then be used elsewhere in the specification in place of the description as a shorthand for the object currently satisfying the descriptive reference or for the object which satisfied the descriptive reference at the time the association was formed (see Historical References below). It should be noted that this association between a reusable name (normally an object type) and a pattern to specify either the object currently or originally satisfying the pattern closely parallels the use of descriptive reference in natural language.

Implication 7: Historical References

The explanation of descriptive references above introduced the notion that the description might be used to reference either the object currently satisfying the pattern or the object which satisfied it at some previous time—the time at which the placeholder association was formed. This is a particularization of the general capability to obtain the object satisfying the pattern at an arbitrary earlier time.

The need for such a capability becomes clear when the alternative is investigated. Without such a capability, historical references such as “the location of the plane when first spotted” must be implemented by recognizing at the appropriate point (the time at which the plane was first spotted) that the object satisfying the descriptive reference (location of plane) must be saved so that it can later be used where required. This is a clear violation of both the principles of functional specification (P1) and loose coupling (P8).

The capability of satisfying descriptive references as of some arbitrary earlier point in time remedies these problems by merely specifying what data is desired (not how to obtain it) and by localizing the specification at the point of consumption (rather than creating an explicit coupling between the production and consumption points through a shared variable). Naturally, this capability implies some ability to specify earlier times. It should be clear that the only meaningful method of time specification is the specification’s own history; that is in terms of the sequence of actions performed on the global model. By reasoning similar to that motivating the need for descriptive references for objects (I6), so too are descriptive references required for the actions which mark the passage of time.

This coupling of descriptive references for both objects and actions provide the capability to examine (but not change) the entire history of a system. This includes the ability to examine any previous state of the system, to determine whether one state preceded another, or to use the historical time order to access a state (e.g., “the last plane launched before the storm”).

Implication 8: Elimination of Variables

The use of historical references (I7) means that a required object, the only type of “value” allowed (I3), can *always* be

recomputed even if the system state has been altered. In most languages, which have no historical reference, modification of the state forces the saving of the required value in some variable because it cannot be recomputed later. Here, since recomputation is always possible, there is a choice between saving the value (storage) and recomputation. By the principle of locality and loose coupling (P8), the choice must be universally in favor of recomputation. Otherwise, an explicit coupling is established between the consumption and the production through the shared (non-local) use of a variable.

Thus, values are always recomputed, as needed; they are never stored. This eliminates the need for variables. They serve no purpose other than holding saved values.

It should be noted that the use of placeholders (I6) represents a compromise with the complete elimination of variables. Placeholders are a type of variable, but they “hold” descriptive references rather than values and must be satisfied as of some time to yield a value (object). Thus they are like procedures in which the name is used as a shorthand for the definition, the definition must be applied to yield a value, and the association between name and definition is static for the lifetime of the name rather than being reassigned as with conventional variables. This “structured” use of placeholders is, we feel, warranted, even though it causes a named-based sharing (P8), because of the notational inconvenience which would otherwise result from recopying the reference. Furthermore, such recopying would itself violate the localization principle (P8). So it is quite clear that some compromise must be accommodated.

Implication 9: Constraint Capability

By the same reasoning which eliminated variables (I8) because their use would introduce explicit coupling between the producer and the consumer, the need for constraints is also established. Without a constraint statement (which would prohibit certain states from arising during the operation of the system), the constraints would have to be integrated (“compiled”) into the specification at all the appropriate places. Such integration (“compilation”) violates both the principles of locality (P8) and separation of function from implementation (P1).

Implication 10: Nondeterministic Constructs

For the constraints (I9) to be more than mere documentation of properties already guaranteed by the specification, they must actually constrain the set of allowable interpretations of the specification. Since the specification is operational (P6), the constraints proscribe those behaviors which would violate the constraints. Thus, the specification must contain nondeterministic constructs for which the choice rule is free except that no constraint may result. A key aspect of the IMPLEMENTATION of the specification is determining choice rules which guarantee that the constraints wouldn’t be violated.

For obvious reasons, this nondeterminism must exist in both the data and control spaces. The data nondeterministic

construct has already been introduced—descriptive references (I6). When more than one object can satisfy the reference, and one is desired, then a nondeterministic choice must be made. The control construct merely indicates that a nondeterministic choice must be made among the specified statements (e.g., “either launch another plane or allow a returning one to land”).

Implication 11: Result Specification

The nondeterministic constructs (I 10), in conjunction with constraint (I9), described above, provide a mechanism for describing desired behavior without specifying precisely the mechanism by which it should be achieved (P1): merely that some appropriate combination of choices for the nondeterministic constructs will result in the specified behavior without violating any constraints.

In a similar way, it should be possible to specify choices among alternative operations by the results desired or to be avoided (e.g., Achieve *S* by doing *X* or *Y* or *Z*). These required and/or proscribed results act like local constraints which must be satisfied nondeterministically by at least one specified method for achieving the desired state (to maintain the operability of the specification (P6)).

It should also be possible to use such result specifications to control the conditionality of some action (e.g., “launch another plane unless it would leave the ship vulnerable to attack”).

Such result specifications in which properties of the state resulting from performing some operation are used to determine whether to perform that operation are quite novel. Normally, such decisions are made by evaluating conditions existing in the current state. Here, through result specifications, the conditions are evaluated in the context existing after (hypothetically) performing the operation. This is simply a historical reference (I7) in which the specified time has not yet occurred and, like historical references, its need is justified by considering the alternative. Without such a capability, then the desired (or proscribed) resulting state must be described in terms of the current state of the model before the operation is applied. This translation of conditions across the application of an operation is highly dependent upon the exact nature of the operation, and is a type of “compilation” which violates both the principle of functionality (P1)—by specifying how to calculate the criteria (in the current state) rather than merely specifying the criteria (in the future state) to be used—and the principle of locality and loose coupling (P8)—by explicitly using the definition of the operation to determine the current state criteria.

Implication 12: Future Reference

This capability enables references to be made to objects that will satisfy a description as of some future time (e.g., “refuel all planes which will be launched today”). This capability can be thought of as the extension either of historical references into the future, or of result specifications to objects. Its justification is similarly motivated. Its absence would require determining the criteria expressed in the current state

for those objects which will satisfy the description as of the specified time, and would hence violate both the functionality (P1) and locality and loose coupling (P8) principles.

Again, it is an implementation, rather than specification, issue to determine effective mechanisms to efficiently calculate such references in the current state.

Implication 13: Demons

There are two separate reasons for including demons in a system specification language. The first is based on the relationship between the system being specified and its environment (P4). This environment is conceptualized as a set of agents which affect a global model (I4) by performing actions on the objects in that model. One or more of these agents constitute the system being specified. They, and the environment agents, must react to changes which occur in the global model. This can only be done by integrating the agents into a single control structure which activates each one at the appropriate time, by having each constantly poll the model for interesting changes, or by providing a demon capability which activates an agent whenever specified changes occur in the model. Since both integration and polling represent implementation techniques (P1) for achieving demon capability, and integration further violates the principle of locality (P8), the specification should be expressed directly in terms of demons.

The second reason for including demons in the specification language concerns the interactions between various parts of the system being specified. Like constraints, demons provide a method of localizing the response to some change in the global model, rather than distributing the response to all the places the change could have been initiated from (P8). Also, by localizing the response, protection is provided for future additions which might also initiate the change (P7). Thus demons provide a method of specifying a response *whenever* some change occurs, not just for those which are explicitly known.

Implication 14: Logical Aggregation

Descriptive references (I6), inference mechanism (I5), and a fully associative data base (I3) are required to separate the functional description of data items from the implementation mechanisms needed to access them (P1). These capabilities provide functional access to individual data objects. But process-oriented specifications (P2) also deal with collections or aggregations of objects which satisfy some common criteria. These aggregations are formed so that an operation, or sequence of operations, can be applied to each object in the aggregation or to the entire aggregation as a whole. Such aggregations are the basis for concise specification by expanding into a much larger set of individual actions to be applied to the individual objects of the aggregations, and correspond to the loop control structures of programming languages.

These aggregation constructs must satisfy all the requirements for separating functional description from access mechanisms (P1) described above. Hence, they should be com-

patible with the descriptive reference capability so that they can be used in conjunction with historical and/or future references. In addition, they must hide the implementation distinction (P1) between explicit aggregations (where each object belonging to the aggregation is explicitly represented), implicit ones (in which only the rule of membership is represented), and combinations thereof without sacrificing the operability of the specification (P6). This implies that all operations utilizing the members of an aggregation operate indirectly through some language processor so that implicit aggregations can be made explicit as objects belonging to the aggregation are needed. Since these same requirements exist for descriptive references themselves, the aggregation capability should exist not merely in a compatible form with descriptive references, but as an extension of that capability.

There are two detailed issues which unfortunately must be dealt with in the functional specification of aggregations. First, the operations performed on the elements of an aggregation may affect the membership of other objects in that aggregation. If so, then the specification must be completed by specifying whether the aggregation membership is static as of some specified time, or dynamic with additions and deletions allowed during its use. The second detailed issue which must be handled occurs when the order of selecting objects belonging to the aggregation affects the resulting state of the global model. If so, and no order has been specified, then the desired order must be added to the specification to retain its operability.

Implication 15: Alternative Constructs (Contexts)

The aggregation capability (I 14) provides a mechanism for functionally specifying a collection of objects and treating them similarly. But a capability is also required to treat the objects of an aggregation as mutually exclusive alternatives. Each of the alternatives must be separately investigated before a decision can be made as to which to select. During these investigations the exploration of the individual alternatives must not interfere with one another. Each exploration must be carried forth as if it were the only one being investigated so that actions performed in one exploration are not apparent in any other and constraints are applied only within an exploration (so that each exploration remains self-consistent but the explorations are not necessarily consistent with each other). Upon completion of the explorations it must be possible to compare the resulting states and determine which subset to retain (either the resulting state or the alternative which started the exploration may be retained).

It should be noted that this alternative construct capability is merely a generalization of the result specification capability (I 11) described earlier and is similarly motivated.

Implication 16: Analogous Specification

Often two or more processes are very similar to each other. In such cases, it is more convenient to specify one in terms of another by specifying the similarities and differences rather than repeating the common portions (which would violate

P8). More importantly, if, during maintenance, the definition of the base process should change, this change would automatically be reflected in all the analogously specified processes (in cases where this effect was not desired, a new exception clause could be added (to analogously specified processes for which the maintainer, guided by a simple maintenance tool, indicated that the effect should not be promulgated)). This capability directly supports the ability to make specifications more complete (P7) by localizing the base description (P8) and by explicitly maintaining the dependencies between process descriptions.

Implication 17: Normal-Case Specification

In support of the ability to deal with incomplete specifications (P7), it must be possible to specify the behavior of the process for the normal case and then augment that description with the behavior required in the various special cases which can arise. This capability is itself a special case of the general analogous specification capability (I 16) described above.

The important aspects of this capability are that each exception should be independently specified (P8) and that these alternatives are automatically organized (P1) so that the most specific applicable alternative is chosen in each case, and the normal case processing is performed only when none of the other alternatives are applicable.

Implication 18: Process Models (Scenarios)

Often only an incomplete model of an environmental agent (P4) exists so that only certain aspects of its behavior are known. It must be possible to specify the known aspects (P7), leaving the others open, while preserving the operability of the specification (P6).

One common form of incompleteness is knowledge of the possible actions which an agent can perform, but lack of information of the decision mechanism employed by the agent. This form of incompleteness can be easily modeled by "scenarios" which utilize nondeterminism mechanisms (I 10) to embed processing options into an expression (such as path-expressions) describing the range of the possible behaviors to be performed by the agent.

An analysis aid should be provided to determine whether the system being specified adequately responds to the range of possible behaviors specified for the environment agents.

CONCLUSION

Strong constraints have been placed on future specification languages by carefully considering design principles of "good" specifications which were themselves derived from the primary uses of specifications. These constraints have implied the need for an ultra-high-level language which combines the data base concept of a global model containing alternative viewpoints (hiding the distinction between explicit and implicit data) with the control structures (both asyn-

chronous demon structures and conventional branching and looping structures) of programming languages.

This combination obviates the need for conventional variables which are replaced by placeholders which retain access to specific portions of the global model. In addition, the need for several novel features such as the ability to access the global model as of any historical or future state and the ability to choose a course of action based upon the desirability (or lack thereof) of its results have been identified.

As mentioned in the Introduction, a language satisfying these constraints has neither been designed nor implemented; but work in this direction has begun. The SAFE¹⁴ project has an implemented language called AP2¹⁵ which provided the experience base from which the conclusions in this paper were derived. This existing language already satisfies half the constraints (Implications 1 through 6, 8, 9, and 13) and work is underway on including the rest, cleaning up and simplifying the existing features, and providing a habitable syntax. We have also begun implementation of a "smart" compiler for this (planned) language which would remove the need for much of the run-time support otherwise required. The purpose of this compiler is to make it feasible to run the specification for selected test cases, rather than to optimize it for production usage. This modest goal makes the implementation of the compiler not only feasible, but rather straightforward.

REFERENCES

1. Wirth, N. "The programming language Pascal", *Acta Informatica 1*, (1971).
2. London, R.L., J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek, "Proof Rules for the Programming Language Euclid," *Acta Informatica 10*, 1-26 (1978).
3. Balzer, Robert, Neil Goldman and David Wile, "On the Transformational Implementation Approach to Programming", *2nd International Conference on Software Engineering*, October 1976.
4. Codd, E.F., "Normalized Data Base Structure: A Brief Tutorial", *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control*, San Diego, California, 1971.
5. Hammer, Michael and Dennis McLeod, "The Semantic Data Model: A Modelling Mechanism for Data Base Applications. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Austin, Texas, May 1978.
6. Chen, P.P.S., "The Entity—Relationship Model: Toward a Unified View of Data," *ACM Transactions on Data Base Systems*, Volume 1, Number 1, pp. 9-36, March 1976.
7. Smith, J.M., and D.C.P. Smith, "Database Abstractions: Aggregation", *Communications of the ACM*, Volume 20, Number 6, pp. 405-413, June 1977.
8. Smith, J.M., and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems*, Volume 2, Number 2, pp. 105-133, June 1977.
9. Pirotte, A., "The Entity-Property-Association Model: An Information-Oriented Data Base System," *Technical Report*, M.B.L.E. Research Laboratory, Brussels, Belgium, 1977.
10. Quillian, M.R., "Semantic Memory," in *Semantic Information Processing*, M.I.T. Press, Cambridge, Mass., 1968, pp. 227-268.
11. Prywes, Noah S., "Automatic Generation of Computer Programs", in *Advances in Computers*, Volume 16, Academic Press, 1977.
12. Hammer, M.M., Howe W.G. and I. Wladawsky, "An Interactive Business Definition System," *SIGPLAN 9*.
13. Langefors, B., "Information System Design Computations Using Generalized Matrix Algebra," (*BIT*) 5(2).
14. Balzer, R., Neil Goldman and David Wile, "Informality in Program Specification," *Fifth International Joint Conference on Artificial Intelligence*, August 1977 and *IEEE Transactions on Software Engineering*, Volume SE-4, Number 2, March 1978; also *USC/Information Science Institute*, ISI-RR-77-59, April 1977.
15. Goldman, N., "AP 2 Pocket Guide", Draft copy, April 1978.

Modular documentation: a software development tool

by ROY E. ANDERSON

Hewlett Packard
Fort Collins, Colorado

ABSTRACT

This paper presents a scheme for documenting the design and implementation of a large software system. The scheme is presented in terms of a family of documents based on the decomposition of any system into specific levels of abstraction for the purpose of software development. It facilitates the use of structured design techniques, provides tangible objects for organizing manpower resources on the basis of the system's structure, gives management meaningful milestones with which to measure development progress, and results in a fully documented system when the implementation phase is complete.

INTRODUCTION

Much has been written in the literature concerning software design methodology, project management, and program verification techniques that can contribute to the success of a large software development project. However, the subject of adequate design and implementation documentation is frequently overlooked even though no one will dispute its importance or the problems associated with many of today's methods. One problem, typically, is that people end up documenting for the wrong reasons. An architect does not draw building plans to be filed away and never referenced by the builders; nor does he draw them after construction is finished. Instead, building plans are a product of the architect's intellectual activity and they serve to guide construction.

This paper expresses a philosophy and presents a scheme for software documentation that is similar to the "architect's" analogy. That is, documents are written, not for the purpose of providing "documentation" (although they do that coincidentally), but for the purpose of providing a system design to guide the system's "construction." The software project's documentation is simply a by-product of its design process.

In specifying this documentation scheme, care was taken to find the "middle of the road." Requiring too much documentation is counterproductive, especially when it involves repeating information. Too little documentation could result in a poor system design going undetected (until it is "too" late), or a loss of control of the project (manifested by slipped schedules, cost overruns, etc.).

This work is the author's refinement of various methods found to be effective in developing any kind of sizable software product, and has been successfully employed by the author (while at NCR Corporation) to produce a mini-computer operating system in 36 man-months. The operating system was completed without a schedule slip, and resulted in one detected "bug" after more than 1200 hours of use. While the scheme does not address the subject of customer-oriented documentation, effective design documentation provides the data necessary to produce those documents in an organized and timely manner.

SYSTEM ABSTRACTION

The human mind attacks the complexity of any problem by systematically dividing it into successively smaller parts, until each part is comprehensible by itself. When an understanding of how all the parts fit together is achieved, the problem has been mastered. Effectively, there are different levels of understanding, each corresponding to an abstraction of the detail contained in those levels (of understanding) existing below it.

For the purposes of this paper, five "formal" levels of abstraction are defined (although an arbitrary number may be conceived by a designer for the actual solution to any particular problem). They are formal in the sense that they serve as a definition of the components into which a system is decomposed for purposes of project management, system design, implementation, and documentation. However, they are not intended to dictate or restrict the choice of system structures, or the use of specific software design methodologies.

The following structure diagram (depicting part of a hypothetical software development system) shows these formal levels of abstraction. The lines connecting items do not represent control or data paths but simply depict structural composition. Each term is defined as follows:

- System. A collection of one or more major components—subsystems—that satisfy the functional requirements outlined by the "customer."
- Subsystem. A collection of one or more programs that together represent a major system component (e.g., operating system, compiler, etc.)

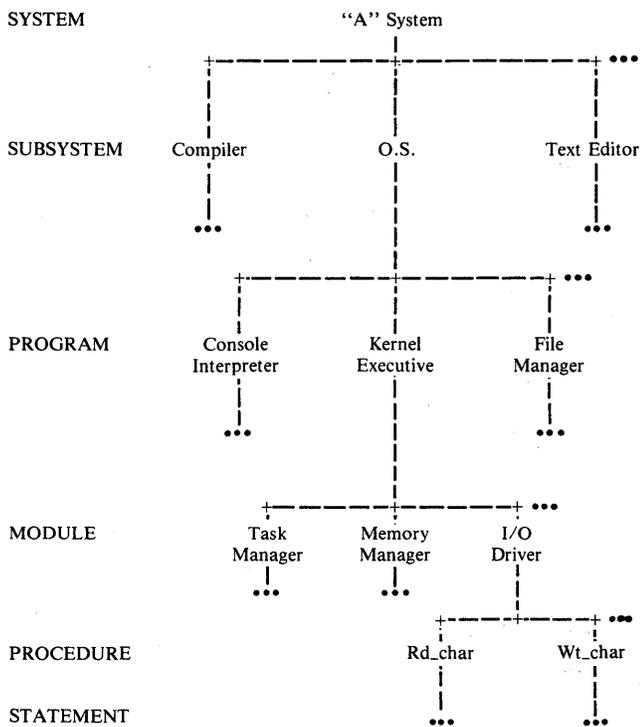


Figure 1—Levels of abstraction

- **Program.** A collection of one or more modules that together satisfy a logical function (e.g., a file manager or memory manager within an operating system).
- **Module.** A collection of one or more logically related procedures that may be compiled and controlled separately as a unit.
- **Procedure.** An identified portion of a module consisting of language statements that can be activated on demand.
- **Statement.** A collection of data element declarations and language statements (that specify actions to be carried out on data elements).

The actual names for these levels of abstraction should be consistent with the terminology used in a development group (or company) and the programming language used for implementation. The design of a component (at any level) is not constrained by the number of formally defined levels that exist below it. Furthermore, the number of items composing any one level (except the root) is unlimited, and normally indicates the magnitude of the project.

This structural framework lends itself well to structured design techniques as well as management methods oriented specifically to the characteristics of the product. Since written communications are essential in any design endeavor, a software documentation scheme represents a key ingredient in any successful management method or software methodology.

DOCUMENTATION SCHEME

One successful method of attacking the problem of system design is to take a "top down" approach, that is, proceeding

from the general to the specific. It would be beneficial, then, to identify a number of document types (or classes) that reflect this approach, and associate a particular view of the system's design with each document type. The levels of abstraction presented in Section 2 provide a convenient vehicle for association between a hierarchical family of documents and the system itself.

Table I identifies five document types and depicts their relationships to the levels of abstraction presented in Section 2. Notice that a distinction has been made between documentation of the design and of the implementation (the overlap that occurs between them is a natural result of the fact that the Module and Procedure levels of abstraction usually appear within a programming language). The following subsections explain the purpose and contents of each document type.

System Overview Specification (SOS)

This documentation identifies the major software products (Subsystems) that are to be developed or otherwise obtained. As such, its primary purpose is to specify the "design" of the first level of abstraction of the system so that activities can be identified and degrees of parallelism determined. The document should be constructed around the specified Subsystems, with sections providing general (or "high level") information concerning the purpose and objectives, acceptance criteria, design constraints, schedules, and so on, of each. The SOS will not only lay the foundation for the construction of the system, but it will serve as a global guide, or "index" to the system's organization and documentation.

Depending on the scope of the System, certain Subsystems may be acquired from sources outside the immediate project (or company). Sections dealing with such a Subsystem represent a "Design Requirements Specification" to its supplier. Other Subsystems may be defined as the result of considerable research into a subject (sometimes taking months or years)—these sections, in addition, contain or summarize (with references to appropriate reports), the results of this research.

Additional information that is of a system-wide nature, should be included in the SOS as the situation requires.

Subsystem Functional Specification (SFS)

The Functional Specification is crucial to the development of a Subsystem because it tells the designers what and what not to design. It is premature in this document to include a specific implementation method for developing the Subsystem. The specification's detail should be sufficient to verify that the objectives expressed in the System Overview Specification are obtainable. In order to ensure the best chances for success in the subsequent development of the Subsystem, the SFS must (a) describe all aspects visible to the user; (b) be reviewed by project members (and approved by management) before detailed design proceeds; and (c) be considered the only definition of functions and features to be implemented. The SFS contains sections dealing with the following topics (when applicable):

1. Subsystem Scope and Objectives
2. Requirements: Hardware; Software; Performance
3. Human Interfaces
4. Programming Interfaces

An accurate statement of objectives not only aids the designer in producing a Subsystem with the proper characteristics, but it gives management an opportunity to judge, early in the development cycle, if the designer's perception of the problem will provide an adequate solution to the Subsystem's requirements.

Similarly, the hardware and software requirements serve to define the environment the Subsystem must create or exist within. The minimum acceptable level of performance for the Subsystem (regarding such issues as execution speed, memory requirements, reliability, security, and fault tolerance) provides important data in planning for project resources as well as establishing a basis for design trade-offs that will inevitably be made.

When user interfaces exist for the Subsystem, they need to be specified both syntactically and semantically as accurately as possible, because they represent the most apparent facilities and features of the Subsystem. This section of the SFS contains the technical information necessary to produce a "User's Manual," and will serve as one during development and testing, providing an invaluable vehicle for intergroup communications during the project.

What is true for human interfaces is equally applicable to programming interfaces when a Subsystem will support user (program) calls for functional services.

By paying a high level of attention to detail when specifying Subsystem interfaces (which will make the SFS a valuable design aid), one can expect during subsequent design activity that some interfaces (or their parameters) may require modification to support unforeseen details or design solutions. For this reason "Change Procedures" (e.g., Metzger, 1973) are utilized to free the designer during production of the SFS (or any document) from concern for excluding a potential solution because of the way in which an interface was specified.

Subsystem Design Specification (SDS)

The SDS provides for decomposition of the Subsystem into its component Programs, serving a function analogous to that of the System Overview Specification. However, this document addresses itself to a greater level of design detail than was appropriate in the previous documents. The purpose of the SDS is to divide the Subsystem into manageable parts, specify the interfaces between them, and provide the design of any data structures relevant to the entire Subsystem. The SDS should be written in sufficient detail to verify that the requirements expressed in the Subsystem Functional Specification can be satisfied and that the design approach chosen is a sound one.

The contents always include the first three following sections, and the fourth when it is appropriate to the situation:

1. Subsystem Overview
2. Design Requirements

TABLE I—Document types and their relationship to levels of abstraction

Level of Abstraction	Document Type	
	Design	Implementation
System	1 System Overview Specification (SOS)	
Subsystem	2 Subsystem Functional Specification (SFS)	(None)
	3 Subsystem Design Specification (SDS)	
Program Module Procedure	4 Program Design Specification (PDS)	5 Program Source Specification
Statement	(None)	(Source Code Listing)

3. Program Identification: Functional Definitions; Program Interfaces
4. Global Data Structures

The first section is simply a brief summary of the Subsystem's purpose, provided as an introduction to the document for the sake of completeness. Similarly, the Design Requirements are a summarized form of those identified in the Subsystem Functional Specification.

The third section, Program Identification, represents the design activity during this stage of development. It explains the structural decomposition of the Subsystem into Programs. Each Program's function is defined, and its interfaces must be precisely specified syntactically, using the implementation programming language (if meaningful), and semantically using prose or a suitable graphic technique if it is well understood by the intended audience. If an interface coincides with one of the Subsystem's external interfaces presented in the Subsystem Functional Specification (SFS), its definition should be a reference to the SFS in order to avoid possible conflicting specifications resulting from future changes. It is important for this section to also contain a description of how the Programs in the Subsystem are related to each other and what interactions will exist between them.

When it is necessary to have data structures shared by two or more Programs, they are considered "global" to the Programs in the Subsystem, and are presented in the SDS since they represent a form of interfacing between the Programs that share them. It is important to specify the actions and responsibilities of the various Programs, with respect to the data structure, in as much detail as possible during writing of the SDS (unresolved details can always be updated as solutions become available in later stages of activity).

Following these guidelines for the preliminary design of a software system will set the stage properly for the detail design remaining to be done. It encourages the designer(s) to consider the global issues of the problem before becoming engrossed in the details of "local" design problems. It allows one to have a clear understanding of the problem before attempting to provide a solution—which has consistently resulted in software products that effectively satisfy the needs of their users.

Program Design Specification (PDS)

This document presents the details of the design, explained to the point that another designer (or the author himself) can understand how the functions of the Program are to be realized. A separate PDS is written for each Program specified in the Subsystem Design Specification (in the case of conceptually simple Programs, two or more could be combined into one document). Each PDS has sections dealing with the following topics:

1. Program Overview
2. Design Rationale
3. Module Identification: Module Function Definitions; Module Interfaces
4. Procedure Identification: Graphic Representation; Procedure Function Definitions; Procedure Interfaces
5. Data Structures
6. Important Algorithms

The first section presents a conceptual overview of the Program, serving as an introduction to the document. The Design Rationale explains why the Program's design was chosen over other possible designs, what assumptions were made in its selection, and depending on the nature of the design, gives a brief discussion of the technology involved. The depth of this section is a function of the complexity of the subject—generally, the more complicated a design is, the more explanation required to make it comprehensible to others. If research activity contributed to the design, the section summarizes the findings and identifies pertinent reference literature.

The Module Identification section is analogous to the Program Identification section of the Subsystem Design Specification and contains similar information. It provides further functional decomposition and an organizational approach toward the final solution. Each Module usually corresponds to a collection of source code, but this correspondence is secondary to the goal of providing an accurate identification of logical functions and their groupings.

Each Module identified in the last section is now decomposed into the principal Procedures that represent its functional components. The descriptions of these Procedures are the subject of the Procedure Identification section. It is usually valuable at this point to employ some graphical means of depicting the flow of control between the various Procedures composing a Module (e.g., block diagrams, schematic logic diagrams, HIPO charts, etc.¹). In addition to depicting control relationships, these graphs or diagrams can also provide a "map" to the Program's source code listings. This can be achieved by naming functional pieces of code (procedures and modules) in the diagram, and by depicting how one part of a listing (often large) is logically related to another. Each major Procedure represented in the Module's "procedure diagram" should be defined functionally, and its interfaces (i.e., its method of invocation) specified syntactically and semantically, including parameters.

Three levels of data structures are subject to description in the Data Structures section of the PDS (the Subsystem Design Specification contained data structures that were global at the Subsystem level): (1) Global within a Program (shared by

Modules); (2) Global within a Module (shared by Procedures); (3) Local to a Procedure (not shared). All data structures of a global nature are identified and described to the degree that is necessary for them to be clearly understood by others. Descriptions of local data structures are usually left up to the discretion of the author (based on their complexity, and the degree to which they can be described in the source code listings).

The section on Important Algorithms addresses those non-trivial algorithms that make up the Program. A narrative about the algorithm is often important to future readers who must maintain or modify the Program. It should answer such questions as: Why was this algorithm chosen over others? What assumptions were made about its use? and, How does it work? A "high level" representation of the algorithm in pseudo-code (i.e., a readable mixture of English and the programming language) should supplement the narrative discussion.

The PDS is finished when all of the design details have been presented in enough detail that source code can be written to implement the Program's design in a straightforward manner.

Program Source Specification (PSS)

This scheme relies on the various source code listings of each Program to provide documentation of the implementation aspects of the System. It is an accepted fact in most of the software industry that the classical methods of documenting "code" (e.g., flowcharts, structured flow diagrams, etc.) not only involved redundant activity, but also suffered from obsolescence soon after they were drawn. If program (code) changes are restricted to source language modifications, then the listings always reflect the current state of an implementation. Thus, it is natural to depend on them as the most accurate documentation of system implementation details.

This dependence creates a need for meaningful "Coding Standards" that incorporate the distinctive features of the implementation language(s) chosen for a system. The purpose of these standards is simply to maintain the consistency, both in documentation content and appearance, of program listings within a project. Numerous sources of information are available in the literature to aid in formulating such standards (e.g., Kernighan and Plauger²).

Documentation Scheme Summary

When a Subsystem is conceptually (functionally) simple, it may decompose into only one Program. Under these circumstances there is little to be gained by having separate documents for the SDS and PDS. There is an advantage, however, to retaining separation between objectives (SFS), design (SDS and PDS), and implementation (PSS).

The final result, in addition to the design itself, is a tree of documents that corresponds to the structure of the System. Each Subsystem is represented by a "family" of documents that contains: (1) a precise statement of the functional objectives (SFS); (2) a decomposition of the Subsystem's functions

Specification technique for parallel processing: process-data representation

by KEN HIROSE and KIYOSHI SEGAWA

Waseda University
Tokyo, Japan

NOBUO SAITO and NORIHISA DOI

Keio University
Yokohama, Japan

MASAHIRO HIRATA

University of Tsukuba
Ibaraki, Japan

TOSHIHARU YAMASAKI

Nippon Univac Kaisha, Ltd
Tokyo, Japan

and

MASAYUKI TAKATA

Tokyo University of Agriculture and Technology
Tokyo, Japan

ABSTRACT

This paper proposes a new specification technique called Process-Data Representation (PDR) which intends to describe precise and comprehensive specification for parallel processing.

The process representation consists of (a) the condition to start the actions of processes and (b) the execution ordering. The data representation specifies the constraints on the shared usage of particular data. We propose the forcing logic to describe these constraint conditions. A formula in the forcing logic can clearly specify the number of objects involved in some actions. The semantics of the formula is discussed in detail. We also give an operational model of the forcing logic and its effective implementation using semaphores. An example of the specification description for an operating system is given, and the verifications of its properties are discussed.

BASIC CONCEPT OF PROCESS-DATA REPRESENTATION

This paper proposes a new specification technique called Process-Data Representation (PDR)¹ which intends to describe precise and comprehensive specification for parallel processing through use of the two-way points of view: process

and data. A precise and formal framework for describing the specification of parallel programs would be inevitable for the verification and the automatic generation of such programs. This framework should be based on the concepts of the essential feature of parallel processing so that the logical inferences using a sound conceptualization can be carried out.

The process representation consists of (1) the specification of the condition to start the actions of processes, and (2) the specification of execution ordering of actions of processes. The latter includes the order in which a process operates on some data.

The data representation specifies the constraints on the shared use of particular data. It also includes the order of actions operating on particular data.

The specification description based on two-way points of view might include some amount of redundancy, but it would enhance the ease of reading and understanding the specification. It is not too much to say that the verification is easily carried out for the easily understood specification.

Specification Examples in PDR

Let us show the introductory examples of the PDR specification. The following notations are used to specify the number of the objects that participate in particular operations.

$\langle x_1, x_2, \dots, x_n \rangle_k \xrightarrow{P}$: At least k out of n objects should do the operation P ;

$[x_1, x_2, \dots, x_n]_k \xrightarrow{P}$: At most k out of n objects may do the operation P .

Note that this notation might be written on both sides of the arrow \rightarrow , and its meaning can easily be understood.

Example 1: The specification of the dining philosophers problem is described in PDR as follows:

$$[ph_1, ph_2, ph_3, ph_4, ph_5]_2 \rightarrow \langle f_1, f_2 \rangle_2, \langle f_2, f_3 \rangle_2, \langle f_3, f_4 \rangle_2, \langle f_4, f_5 \rangle_2, \langle f_5, f_1 \rangle_2]_2$$

and

$$[ph_1]_1 \rightarrow \langle f_1, f_2 \rangle_2 \ \& \ [ph_2]_1 \rightarrow \langle f_2, f_3 \rangle_2 \ \& \ [ph_3]_1 \rightarrow \langle f_3, f_4 \rangle_2 \ \& \ [ph_4]_1 \rightarrow \langle f_4, f_5 \rangle_2 \ \& \ [ph_5]_1 \rightarrow \langle f_5, f_1 \rangle_2$$

and

$$[ph_1, ph_2]_1 \rightarrow \langle f_2 \rangle_1 \ \& \ [ph_2, ph_3]_1 \rightarrow \langle f_3 \rangle_1 \ \& \ [ph_3, ph_4]_1 \rightarrow \langle f_4 \rangle_1 \ \& \ [ph_4, ph_5]_1 \rightarrow \langle f_5 \rangle_1 \ \& \ [ph_5, ph_1]_1 \rightarrow \langle f_1 \rangle_1$$

where phk ($k=1, \dots, 5$) represents the philosopher k , and fj ($j=1, \dots, 5$) represents the fork j .

Example 2: The specification of the rendezvous mechanism of the task declarations in Ada programming language²

```

task P is
  entry E(e);
  :
  accept E(e)
  do
    S
  end;
  :
end P;
task Qi is
  :
  E(di);
  :
end Qi;
for i=1, 2, 3, ..., n

```

is described in PDR as follows:

$$\langle P, [Q_1, \dots, Q_n]_1 \rangle_{S'} \rightarrow \langle P.e, [Q_1.d_1, \dots, Q_n.d_n]_1 \rangle_2$$

and

$$\langle P, Qi \rangle_2 \xrightarrow{S'} \langle P.e, Qi.di \rangle_2 \text{ for } i=1, 2, \dots, n,$$

where $S' = (\{P.e := Qi.di\}; S; \{Qi.di := P.e\};)$.

Specification Method in PDR

The fundamental model of parallel processing employed in PDR is considered to consist of a set of clusters, each of which is a set of processes logically related to each other. A list of the specifications of clusters gives a complete specification of a target parallel processing.

There are three classes of processes: standard, interruptive, and postlude. Each process has an activation condition (standard, postlude, and interruptive condition, respectively) which specifies when it starts the execution.

When the activation condition is evaluated and how many times a process is executed are summarized as follows:

	When	How many times
Standard	At the initiation of the cluster	Once at most
Interruptive	At any time*	Any number of times
Postlude	At any time	Exactly once

*The evaluation of the interruptive condition restarts after the end of its last execution.

A cluster operation terminates if all the activated standard and interruptive processes terminate and if all the postlude processes are activated and terminate.

A framework of the specification description for a parallel processing is as follows:

```

parallel processing
  global data and condition
  cluster path expression
  ...
  /* specification of cluster i */

cluster i;
  [process-path ..... end;]
  /* process specification section */
  ...
  s-process j; (or i-process j; or p-process j;);
  condition Bj;
  ref-path ..... end;
  ...
  exec-path ..... end;
  ...
  /* data specification section */
  ...
  data x : t;
  ref-path ..... end;
  ...
  exec-path ..... end;
  ...
  /* process data interaction specification section */
  ...
  specification using forcing logic

```

The terms *s-process*, *i-process*, and *p-process* stand for standard, interruptive, and postlude process, respectively. The path expression notation is used to specify the order of operations. The term *ref-path* gives the constraints for the

access to the object executed by other objects, while *exec-path* gives the order of the operations executed by that object which has this specification. Of course, these operations may be provided by other objects. The process data interactions are specified mainly by using formulas in the forcing logic.³

FORCING LOGIC

As the quantum logic reflects the properties of the quantum mechanics, so the formal logic for the specification description expresses the essential properties of the target system. It needs to have high readability so as to make the system easily understood. The first order logic would be suitable for formal treatment, but the higher order logic is needed if we want to have a simple and clear description of the real world.

Let us first define several fundamental concepts in the target system and embed them in the predicate logic. Since the introduction of many concepts might make it difficult to define the formal system, it is necessary to carefully introduce only a few concepts which maximize readability.

There may be several fundamental concepts, for example, those relating to the number of objects, forcing, prohibition, constraint, priority for some actions, and so forth. In this paper, we introduce the concept concerning the number of objects involved in some activities. This is described by the forcing logic, which is the first order logic with the operators [...] and <...>. As is shown below, this logic can express the forcing and prohibition constraints imposed on some activities. It is expected that these forcing operators and a certain kind of path expression⁴ make it quite easy to describe the specification of parallel processing.

Forcing Operators and Formulas in Forcing Logic

For a fixed natural number u , let R denote a set of all objects x_1, x_2, \dots, x_u existing in the target system.

Definition 2.1: For an integer u and k s.t. $0 \leq k \leq u$, $\langle \dots \rangle_k$ and $[\dots]_k$ are primitives to specify a family of subsets of R as follows:

$$\text{For } x_1, x_2, \dots, x_u \in R \\ \langle x_1, \dots, x_u \rangle_k = \{X \mid X \subseteq \{x_1, \dots, x_u\} \ \& \ \bar{X} \geq k\},$$

where \bar{X} denotes the cardinality of X .

$$\text{For } X_1, \dots, X_u \in \text{power set of } R \\ \langle X_1, \dots, X_u \rangle_k = \{\cup \{\text{Proj}_i(\xi) \mid i \in I\} \\ \mid \bar{I} \geq k \ \& \ I \subseteq \{1, \dots, u\} \ \& \ \xi \in \prod_{i \in I} X_i\}.$$

$$\text{For } x_1, x_2, \dots, x_u \in R \\ [x_1, \dots, x_u]_k = \{X \mid X \subseteq \{x_1, \dots, x_u\} \ \& \ \bar{X} \leq k\}.$$

$$\text{For } X_1, X_2, \dots, X_u \in \text{power set of } R \\ [X_1, \dots, X_u]_k = \{\cup \{\text{Proj}_i(\xi) : i \in I\} \\ \mid \bar{I} \leq k \ \& \ I \subseteq \{1, \dots, u\} \ \& \ \xi \in \prod_{i \in I} X_i\}.$$

For example, if $R = \{r_1, r_2, w\}$,

$$[\langle r_1, r_2 \rangle_1, w]_1 = [\{\{r_1\}, \{r_2\}, \{r_1, r_2\}\}, \{\{w\}\}]_1 \\ = \{\phi, \{r_1\}, \{r_2\}, \{r_1, r_2\}, \{w\}\}.$$

The bracket operators $\langle \dots \rangle$ and $[\dots]$ are called forcing operators, and they are represented by $\langle \dots \rangle$.

Now, we consider the formula

$$S \xrightarrow{P} T, \quad (1)$$

where S and T are a family of subsets of R , and P is in general a predicate over R (if it is a name of an operation, it denotes the predicate “ P is now being executed”).

The semantics of the above formula is given as follows:

$$\{(X, Y) \mid P(X; Y)\} \subseteq S \times T. \quad (2)$$

Specifically, we write

$$S \xrightarrow{P} T, \quad (3)$$

when

$$\{(X, Y) \mid P(X; Y)\} = S \times T. \quad (4)$$

Properties of Forcing Operators

Consider the case in which a family of subset S and/or T in the above formula is represented by a forcing operator $\langle \dots \rangle$.

$$\langle x_1, \dots, x_m \rangle_k \xrightarrow{P} Y$$

This means that for a given predicate $P(X; Y)$ and Y s.t. $\{X \mid P(X; Y)\} \neq \phi$,

$$\{X \mid P(X; Y)\} \subseteq \langle x_1, \dots, x_m \rangle_k \quad (5)$$

and

$$\exists X \in \langle x_1, \dots, x_m \rangle_k [\bar{X} = k \ \& \ P(X; Y)] \quad (6)$$

should hold. Moreover, k satisfying (5) and (6) is uniquely determined: k becomes as small as possible for $\langle x_1, \dots, x_m \rangle_k$, and it becomes as large as possible for $[x_1, \dots, x_m]_k$.

It is obvious that the following property is generally established:

If

$$\{X \mid P(X; Y)\} \subseteq \langle x_1, \dots, x_m \rangle_k,$$

then

$$\forall X \subseteq \{x_1, \dots, x_m\} [\bar{X} < k \supset \neg P(X; Y)]. \quad (7)$$

Therefore, if

$$\{X \mid P(X; Y)\} \subseteq \langle x_1, \dots, x_m \rangle_k$$

and

$$\exists X \in \langle x_1, \dots, x_m \rangle_k [\bar{X} = k \ \& \ P(X; Y)] \text{ hold,} \\ \langle x_1, \dots, x_m \rangle_k \xrightarrow{P} Y$$

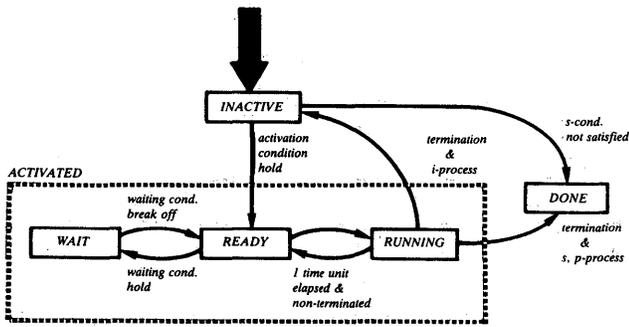


Figure 1—State transition diagram of process

means that “at least k objects out of $\{x_1, \dots, x_m\}$ should do the operation P to Y .”

Similarly, for $\llbracket \dots \rrbracket_k$, the following property is generally established:

If $\{X|P(X;Y)\} \subseteq \llbracket x_1, \dots, x_m \rrbracket_k$,
 then $\forall X \subseteq \llbracket x_1, \dots, x_m \rrbracket_k [\bar{X} > k \supset \rightarrow P(X;Y)]$. (8)

Therefore, if

$$\{X|P(X;Y)\} \subseteq \llbracket x_1, \dots, x_m \rrbracket_k$$

and

$$\exists X \in \llbracket x_1, \dots, x_m \rrbracket_k [\bar{X} = k \ \& \ P(X;Y)] \text{ hold,}$$

$$\llbracket x_1, \dots, x_m \rrbracket_k \xrightarrow{P} Y$$

means that “at most k objects out of $\{x_1, \dots, x_m\}$ may do the operation P to Y ,” i.e., “more than k objects may not do the operation P .”

$$X \xrightarrow{P} \langle y_1, \dots, y_n \rangle_k$$

This means that for a given X and a predicate $P(X;Y)$ s.t. $\{Y|P(X;Y)\} \neq \emptyset$

$$\{Y|P(X;Y)\} \subseteq \langle y_1, \dots, y_n \rangle_k \quad (9)$$

and

$$\exists Y \in \langle y_1, \dots, y_n \rangle_k [\bar{Y} = k \ \& \ P(X;Y)] \quad (10)$$

should hold. Moreover, k satisfying (9) and (10) is uniquely determined.

For the same reason discussed above,

$$X \xrightarrow{P} \llbracket y_1, \dots, y_n \rrbracket_k$$

means that “at least k objects out of $\llbracket y_1, \dots, y_n \rrbracket$ should have the operation P done by X ,” and

$$X \xrightarrow{P} \langle y_1, \dots, y_n \rangle_k$$

means that “at most k objects out of $\llbracket y_1, \dots, y_n \rrbracket$ may have the operation P done by X ,” i.e., “more than k objects may not have the operation P done by X .”

If the computer behavior is specified by the above-mentioned formula,

$$\langle x_1, \dots, x_m \rangle_k \xrightarrow{P} \langle y_1, \dots, y_n \rangle_j$$

represents that the constraint should always (i.e., at any time) hold.

EFFECTIVE IMPLEMENTATION OF PDR

Process State Transition

This section first discusses the operational model of PDR so that its meaning can be intuitively understood. A process in the PDR description is made to start when its activation condition holds. It also obeys the condition specified by formulas in forcing logic. This condition is called a forcing condition.

The state transition diagram of a process in the PDR description is given in Figure 1. The activation condition corresponds to a standard, interruptive, or postlude condition. The holding and breakoff of the waiting condition usually correspond to the breakoff and holding of the forcing condition.

The process scheduling in a cluster is summarized in the following procedure.

```

repeat
  if INACTIVE-queue is not empty, then
    remove processes of which activation condition hold,
    and enter them into READY-queue fi;
  if READY-queue is not empty, then
    i := remove the first process from READY-queue;
    allocate CPU to process i for one time-unit;
    if process i did not finish
      then put it in the end of READY-queue fi
  fi
forever
    
```

Implementation Schema for the Forcing Logic

Let us consider the implementation of a formula in the forcing logic through use of the semaphore system. The forcing operation can be implemented effectively as follows:

```



```

f:

if currentprocess in pset then $\langle V \rangle(s)$ fi ;

where *S*: counting semaphore,
 pset: a set of processes,
P, *V*: *P* and *V*-operation of the standard semaphore,
 $\langle P \rangle$, $\langle V \rangle$: inverse *P* and *V*-operation of the inverse semaphore.

$\langle P \rangle$ and $\langle V \rangle$ operations are defined as follows:

$\langle P \rangle$ (var *s*: semaphore') =
 begin
 s := *s* - 1;
 if *s* > 0 then block
 else if *s* = 0 then
 if there are blocked process on *s* then wakeup (all
 blocked process on *s*) fi;
 if there are suspended process on *s* then unsuspend (all
 suspended process on *s*) fi
 fi
 fi
 end
 $\langle V \rangle$ (var *s*: semaphore') =
 begin
 s := *s* + 1;
 if *s* > 0 then suspend (all processes which are in this critical
 section) fi
 end

The structure of the semaphore' is as follows:

```
type semaphore' =
  record
    counter: integer;
    blocked-queue: queue;
    suspended-queue: queue;
    in-critical-section-queue: queue
  end
```

Note that a process in the critical section is suspended if several processes leave the section, and the number of the processes in the critical section becomes less than that specified by the "at least" operator. Therefore, there are two waiting states (blocked and suspended) in the solution.

If the bracket operators [...] and $\langle \dots \rangle$ are nested, the above transformation can be applied from inside to outside.

For example,

```
 $\langle p1, \langle p2, p3 \rangle_1, p4 \rangle_2 \mathcal{G}$ :
  initialize {sem1 := 1; sem2 := 2;
             pset1 := [p2, p3];
             pset2 := [p1, p2, p3, p4];
  if currentprocess in pset 1 then  $\langle P \rangle$ (sem1) fi;
  if currentprocess in pset2 then  $\langle P \rangle$ (sem2) fi;
  g:
  if currentprocess in pset2 then  $\langle V \rangle$ (sem2) fi;
  if currentprocess in pset1 then  $\langle V \rangle$ (sem1) fi;
```

The bracket operators in the right-hand side of a forcing logic formula can also be implemented using the same technique. The details are omitted here.

SPECIFICATION AND VERIFICATION: AN EXAMPLE

Let us give an example of the specification description for the operating system SOLO⁵ and discuss the verification of several properties in this system.

Specification for SOLO Operating System

The SOLO operating system is hierarchically structured and designed through use of several types of basic abstract data: processes, monitors, and classes. In the specification we consider the following three levels of processes: (1) utility process level—including several utility routines (compiler, editor, file processing routines, etc.) executed on the job process "master," and device control routines (card, printer, disk, etc.) executed on the I/O process "producer" and "consumer"; (2) command process level—including the command interpreter routines (i.e., do, io) executed either on the job process "master" or on the I/O process "producer" and "consumer"; and (3) system process level—including all the processes originally defined in the SOLO system.

In the PDR specification, the initial process is of standard type, whereas all the other processes are of interruptive type.

All the objects defined as monitors and classes in SOLO are considered as data in the PDR specification. In the SOLO system, shared objects are controlled so as to be accessed mutually exclusively through use of several monitors (diskuse, typeuse, etc.) and their PDR specifications are simply described using formulas in the forcing logic.

The following is part of the specification description of the SOLO operating system for the situation in which one of the typical commands

```
copy(card,printer);
```

is given by a user.

"SOLO Operating System Specification Description by PDR"
 cluster SOLO;

Process Specification
Utility Process Level

```
i-process copy;
condition "command = 'copy' (called by do)"
exec-path
  lookup;writearg(inp);writearg(out);(readpage;
  writepage)*; readarg(inp);readarg(out)
end ;
i-process card;
condition "source = 'card' (called by input-io)"
exec-path (readline;write)*end ;
i-process printer;
condition "dest = 'printer' (called by output-io)"
exec-path (read*;writeline)* end ;
```

Command Process Level

```
i-process do;
condition called by master at the initialization;
exec-path
  (((accept)* + (display)*)*);
  lookup; run(command))*
end ;
```

```

i-process input-io;
condition called by producer at the initialization;
exec-path
  (readarg(inp);lookup; run(driver); writearg(inp))*
end ;
i-process output-io;
condition called by consumer at the initialization;
exec-path
  (readarg(out);lookup; run(driver);writearg(out))*
end ;

```

System Process Level

```

i-process master;
condition initialize by initial process
or system call by utility/command process;
ref-path
  initialization;
  (run + ... + readpage + writepage + readarg(inp) +
  readarg(out) + writearg(inp) + writearg(out) + ...)*
end ;
ref-path writearg(inp);readpage*;readarg(inp) end ;
ref-path writearg(out);writepage*; readarg(out) end ;
exec-path
  system call→
  (
  run→...
  readpage→inbuffer.read,
  writepage→outbuffer.write,
  readarg(inp)→inresponse.read,
  readarg(out)→outresponse.read,
  writearg(inp)→inrequest.write,
  writearg(out)→outrequest.write,
  ...
  ),
  initialization→
  initialize all local objects;
  "load and start 'do' "
end ;
i-process producer;
condition initialization by initial process
or system call by utility/command process;
ref-path
  initialization; (run + exit + ..... +
  writepage + readline + readarg + writearg + ...)*
end ;
ref-path readarg(inp);writepage*;writearg(inp) end ;
exec-path
  system call→
  (
  ...
  write→iostream.write,
  writepage→inbuffer.write,
  readarg→inrequest.read,
  writearg→inresponse.write,
  ...
  ),
  initialization→
  initialize all local objects;
  "load and start 'io' "
end ;

```

```

i-process consumer;
condition initialization by initial process or system call by
utility/command process;
ref-path readarg(out);readpage*;writearg(out) end ;
exec-path
  system call→...
  ...
end ;

```

Data Specification

```

systemprocess={master,producer,consumer,reader,
writer}

```

```

mainprocess={master,producer,consumer}

```

```

data inbuffer,outbuffer;pagebuffer;

```

```

ref-path write;read end ;

```

```

data inrequest,inresponse,outrequest,outresponse:
argbuffer;

```

```

ref-path write;read end ;

```

```

data master.instream(inbuffer),
master.outstream(outbuffer),
producer.iostream(inbuffer),
consumer.iostream(outbuffer):
charstream(buffer:pagebuffer) ;
ref-path (initread;read*)+(initwrite;write*) end ;
exec-path

```

```

  read→buffer.read,
  write→buffer.write
end ;

```

```

data master.file[1](catalog),...
: datafile(dcat:diskcatalog);

```

```

exec-path

```

```

  open→dcat.lookup;dfile.open,
  close→dfile.close,
  read→dfile.read
  write→dfile.write
end ;

```

```

[mainprocess]1 dfile.open,dfile.read,dfile.write→
<diskdevice>1

```

```

[systemprocess]1 read,write→<typedevice>1

```

```

[mainprocess]1 lookup→<diskdevice>1

```

Verification for SOLO Operating System

For the above-mentioned specification, the following properties of the system can be verified without using any formal system because of the specification's high readability.

Deadlock Freeness

A deadlock situation might occur in the environment where several processes lock shared objects. It is certain the deadlock will not occur if the process ordering defined by these locking requests does not make a cyclic chain. In the PDR specification, the mutually exclusive access to certain objects

inside certain procedures is described using formulas in forcing logic. Then, it is easy to find out there is no such process chain in the SOLO system by tracing the ref-path and exec-path expressions starting from the critical procedures. It is, of course, assumed that any I/O operation should be finished in a finite delay.

Consistency of the Specification

If one operation of an object is referred, this object then might execute (refer) several numbers of operations provided by any other objects. The objects defined in the SOLO operating system interact with each other following the *refer* (*execute*) and *referred* relationships described in the specification. It is therefore necessary to check the consistency of the overall interrelationships at the specification level by tracing the ref-path and the exec-path expressions.

Consistency of the Interactions via Buffers

The most complex and critical interactions in the SOLO system are caused by two pagebuffers (inbuffer, outbuffer) and four argbuffers (inrequest, inresponse, outrequest, outresponse). Consider a copy command mentioned above. The ref-path and the exec-path expressions relating to the pagebuffers and argbuffers are extracted from the specification to verify the consistency of the interactions among these objects.

Let us verify the consistency of the interactions caused in the course of passing data from "card" to "copy."

(i) the consistency among processes (master, producer) and buffers (inbuffer, inrequest, inresponse)

```
"master" ref-path writearg(inp);readpage*;readarg(inp)
end ;
"producer" ref-path readarg(inp);writepage*;
writearg(inp) end ;
"buffer" ref-path write;read end ;
```

The procedures writearg(inp), readarg(inp), and readpage in "master" are implemented by executing inrequest.write, inresponse.read and inbuffer.read, respectively, and readarg(inp), writearg(inp), and writepage in "producer" by executing inrequest.read, inresponse.write and inbuffer.write, respectively. They are consistent with the ref-path specification for buffers.

(ii) the consistency among copy, do, and master

```
"copy" exec-path writearg(inp);readpage*;readarg(inp)
end ;
"master" ref-path writearg(inp);readpage*;readarg(inp)
end ;
```

Since "do" activates "copy" by executing run procedure, it does not interfere with the interactions between the above two processes via buffers. Obviously, they are consistent.

(iii) the consistency among card, input-io, and producer

```
"card" exec-path write* end ;
"input-io" exec-path readarg(inp);writearg(inp) end ;
"producer" ref-path readarg(inp);writepage*;writearg
(inp) end ;
```

"card" is activated after "input-io" reads the input source argument "card" by executing readarg(inp), and the procedure write in "card" finally ends with buffer.write(i.e., writepage). Therefore, the interactions among these three processes are consistent.

The ref-paths of system level ("master" and "producer") are consistent with the ref-paths of "buffer's," and the exec-paths of utility and command level ("do," "copy," "input-io," and "card") are consistent with these ref-paths. Therefore, the total interactions caused in the course of passing data from "card" to "copy" using various buffers are verified to be consistent. The consistency among interactions caused in the course of passing data from "copy" to "printer" can also be verified in the same way.

CONCLUSION

In this paper, we have proposed a new technique for describing the specification of parallel processing and have demonstrated its usefulness with several examples. The high-level concepts and their description notations, such as forcing operators, describe the properties and the situations of the target system so clearly that the verification and understanding of the specification can be easily carried out.

More detailed discussion of the implementation will be necessary for further applications.

REFERENCES

1. Hirose, K., Saito, N., Doi, N., Segawa, K., Hirata, M., and Takata, M., "Process-Data Representation," *Proc. 3rd US-Japan Computer Conference*, Oct. 1978, pp. 225-230.
2. "Preliminary Ada Reference Manual," *SIGPLAN Notices*, Vol. 14, No. 6, 1979.
3. Hirose, K., Saito, N., Doi, N., et al., "Forcing Logic in Process-Data Representation," Technical Report KIIS-79-01, Institute of Information Science, Keio University, March 1979.
4. Campbell, R.H., and Habermann, A.N., "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science*, Vol. 16, Springer-Verlag, 1974.
5. Brinch Hansen, P., "The SOLO Operating System: Processes, Monitors and Classes," *Software—Practice and Experience*, Vol. 6, 1976, pp. 165-200.

A tiny portable language-independent macroprocessor and some applications

by ROBERT C. GAMMILL

North Dakota State University
Fargo, North Dakota

ABSTRACT

A tiny language-independent macroprocessor is described. It is easily implemented in most programming languages. A compact and portable implementation in ANSI FORTRAN is given. The simplicity of the implementation results from a user-specified escape character used to mark all macro calls. The result is a macro language not easily read by beginners. All primitive macro operations, including definition, are simple macro calls. The tiny macroprocessor has proved a powerful software tool in a number of applications. These include source text decompression, character set encoding for transmission, and as the basis for a linking loader. The compactness and portability of the processor make it useful in moving software from one machine to another. The simplicity of the processor makes it easy to extend or modify for new applications. The tiny macroprocessor is a simple yet useful addition to the software engineer's tool kit.

INTRODUCTION

The tiny macroprocessor was implemented as a bootstrap macroprocessor for use in moving a compressed source version of a larger macroprocessor called GPMX¹ from one computer to another. The tiny macroprocessor proved very useful in this application, allowing text compression approaching a factor of 5. The tiny macroprocessor was designed along the lines of GPM² and its offshoot GPMX. However, its internal operation has been simplified in the extreme. Where GPM and GPMX demand that certain characters of the alphabet be dedicated to the macro language, the tiny macroprocessor uses an extended alphabet through the introduction of a user-defined escape character. This allows the tiny macroprocessor to be used with any language or alphabet, without conflict.

Later the writer became interested in the experimental design of a powerful yet simple and compact linking loader for microcomputers. A desirable feature for a linking loader is the ability, when no address relocation or symbol linking is needed, to load a stream of bytes into sequential memory locations without excessive overhead. In other words, when loading absolute information, a linking loader should approach the efficiency of an absolute loader. When relocation or linking is needed, those operations should be flagged so as to be easily recognized in the surrounding absolute byte

stream. Clearly this requirement points to the tiny macroprocessor, whose language can be embedded within any alphabet, including one made up of absolute binary bytes. As a result, an experimental linking loader based on the tiny macroprocessor was constructed. This loader, which will be described in more detail later, is exceedingly small and simple. Its structure resembles a macroprocessor far more than a loader, but the primitive macro calls are functions relating to linking, external symbol dictionaries and relocation. This loader has the unique character that it is controlled by the object text which it loads, and that its functions can be combined through macro definitions to provide extended capabilities when those are needed. Perhaps most important is the fact that the loader is so simple that it can be maintained and modified easily, a feature rarely found in most loaders.

A third application of the tiny macroprocessor is in inter-computer communication. In a file transfer protocol between two minicomputers running UNIX,⁴ using the terminal interface on both machines, it is necessary to do character translation when sending binary files. So many characters are given special treatment in the UNIX terminal driver that some way must be found to avoid sending certain characters. The tiny macroprocessor satisfies the need, since it has capabilities to extend the alphabet by defining macros. A new version of the tiny macroprocessor, written in "C,"³ was produced, with primitive functions for defining the codes of dangerous characters numerically. It allows special macros to be defined for every character that would receive a functional interpretation by the terminal driver, and the macro definition and call to be transmitted through the drivers in place of the offending character. For example, control D (ASCII 4) is an EOT or end of transmission character. Transmitting this character on a line causes many modems and terminal drivers to immediately hang up the line. By simply defining that character to be represented by the macro \$D and transmitting that definition, all occurrences of EOT can subsequently be translated to \$D before transmission and macro expanded back to EOT at the receiving end.

DESCRIPTION

The tiny macroprocessor can be used with any language using any character set. This is possible because the tiny macroprocessor works by extension of the existing alphabet,

through use of a user-selected escape character. All other characters retain their original form. Assume that the alphabet is ASCII and the escape character is \$. The \$ sign is used to escape from the base language, with a single \$ in the original language now being represented by \$\$\$. Obviously the escape character should be chosen to be a character that occurs infrequently in the base language. Using this method, a single character of the base language becomes twice as long, and we gain up to 127 double characters (\$a, \$b, etc.), which are used for the macro language. This technique, although it produces a rather crude-looking macro language, guarantees that the macroprocessor can be used with *any* alphabet and language. This is extremely important in certain kinds of applications, especially those where intersystem character set differences are part of the problem.

Given this extended alphabet, the tiny macroprocessor uses certain of these new characters as primitive operations (macro calls). The most important primitive operation is macro definition. For example,

```
$0'abody'
```

This is a macro definition that defines \$a to be the text string "body." The \$0 indicates that a macro definition follows. The following character (chosen freely from the base alphabet) is a delimiter, which encloses the macro definition. The next character is the macro name, and the subsequent characters up to the next occurrence of the delimiter form the body of the macro. All macros have one-character names. The definition macro is the only primitive operation, other than macro calling, that is necessary to all applications. In addition, one other feature is supplied in all applications. One character is designated as a formal parameter character. When that character is encountered in a macro body, the next character from the input is substituted for it. The % sign will be used for this purpose here. Thus if \$0'ab%dy' is defined, \$ao produces "body" while \$aa produces "bady". Finally, if \$% is found in a macro body, the next character from the input is used as the macro name to be called (see Table II for examples).

APPLICATION I

The tiny macroprocessor was developed as a bootstrap processor to expand compressed FORTRAN source text for GPMX,¹ a much larger and more powerful macroprocessor, on new machines where it was being transported. The tiny macroprocessor allowed a 708-line source program, to be reduced to 166 compressed but readable lines. In this applica-

tion much of the task was to eliminate the cost of strings of blanks, and make full use of the card image source record. In addition to the basic macro definition primitive described already, nine additional primitive functions were added to generate strings of blanks and control input and output records. These are shown in Table I.

TABLE I—Primitive macro call results

CALL	yields	RESULT	or	SIDE EFFECT
\$1		none		read next input record
\$2		none		write next output record
\$3		3 blanks		none
.		.		.
\$9		9 blanks		none

The \$1 and \$2 macros are used to decouple input and output lines. The \$1 causes the elimination of the remainder of an input record or line. The \$2 causes the writing of an output record or line, i.e. it inserts an end-of-line in the output. As an example of the use of these macros, \$0'\$S\$2\$6' allows \$\$ to start a FORTRAN source line. Table II provides examples of the use of this macroprocessor. The last three examples in Table II should be examined closely, for they use the \$% construction. This allows a macro to be called whose name is specified in the call, not in the definition.

A very useful aspect of the form of text compression implemented by the tiny macroprocessor is that compressed source text can still be edited either on-line or on a keypunch. Because of the availability of the \$1 macro, a new source record can be added to the compressed text without necessity for the addition of exactly 72 characters to the output. The \$1 macro is also useful for adding comments to the compressed text that will not appear in the decompressed output. The source code for the tiny macroprocessor in portable ANSI FORTRAN is given in Figure 1.

APPLICATION II

Linking loaders for microcomputers have only recently begun to appear. One reason for this is that absolute assemblers and loaders are more compact and easier to write, but another is that object modules containing external symbols and relocatable addresses can be large and unwieldy for small microcomputer mass storage systems. Input to an absolute loader

TABLE II—Examples of tiny macroprocessor definitions, calls and results

Definition	Call	Yields
\$0,\$S\$2\$6,		
\$0.L(%(I),I = 1,%).	\$\$SLAN	(A(I),I = 1,N)
\$0*I\$SINTEGER *	\$IX,Y	INTEGER X,Y
\$0'W\$%WRITE(6,%%)\$L'	\$WS73AN	WRITE(6,73) (A(I),I = 1,N)
\$0'A\$2\$3%%'	\$WA2173AN	21 WRITE(6,73) (A(I),I = 1,N)
\$0'F\$%FORMAT('	\$FA73819)	73 FORMAT (819)

```

        INTEGER A,E,S,C,IN(72),ID(8),OUT(72),CH(14),ST(1000)
        READ (5,100) CH,E,S,C,N
20     IST=1
        1 READ (5,101) IN,ID
          WRITE(6,101) CH(4),IN,ID
          DO 16 L=1,72
            A=IN(L)
        2     IF(A.EQ.CH(1)) GO TO (3,11,8,9),IST
              GO TO (11,4,8,9),IST
        3     IST=IST+1
          GO TO 14
        4     DO 6 I=1,10
              IF(A.NE.CH(I+4)) GO TO 6
              IF(I.LT.4) GO TO (3,20,12),I
              DO 5 J=2,I
                N=N+1
        5     OUT(N)=CH(4)
              GO TO 13
        6     CONTINUE
          ST(S)=C
          S=S+1
          C=E+2
        7     IF(A.EQ.ST(C-1)) GO TO 13
          C=ST(C-2)+2
          IF(C) 77,77,7
        8     IST=4
          ISV=A
          A=E
          E=S
        9     IF(A.NE.ISV) GO TO 10
          IST=1
          A=CH(3)
10     ST(S)=A
          S=S+1
          GO TO 16
11     N=N+1
          OUT(N)=A
          IF(N.LT.72) GO TO 13
12     IF(N.GT.0) WRITE(7,101) (OUT(I),I=1,N)
          N=0
13     IST=1
14     IF(C.LE.0) GO TO 16
          IF(ST(C).NE.CH(3)) GO TO 15
          S=S-1
          C=ST(S)
          GO TO 14
15     A=ST(C)
          C=C+1
          IF(A.NE.CH(2)) GO TO 2
16     CONTINUE
          GO TO 1
17     WRITE(6,101) (CH(4),I=2,L),CH(1),A
100    FORMAT(1H1,2A1,A2,11A1,I2,3I1)
101    FORMAT(90A1)
        END
1$%;; 0123456789-7100 data card to initialize macroprocessor

```

Figure 1—FORTRAN source and data card for macroprocessor

TABLE III—Primitive functions for the macro loader

Call	Result
\$1	end of record mark for text strings and loader input
\$2	ac = next two bytes from the input
\$3	pcr = ac
\$4	rc = pcr
\$5	ac = ac + pcr
\$6	ac = ac + rc
\$7	memory (pcr) = ac; pcr = pcr + 2; 2 bytes stored
\$8	input contains text for name of external reference
\$9	input contains text for name of external definition (the present contents of the pcr is the value)

has little structure, but the input to a linking loader must have considerable structure. The tiny macroprocessor, with its language that can be embedded in any other language, seems an ideal basis for a linking loader where all structured information relating to symbols and relocation is marked by the escape character. Using this technique, a file of absolute bytes for loading directly into memory needs no structure, while symbols and relocation are all embedded through use of the macro language.

For this application the \$1 through \$9 primitive macros are replaced by a completely different set of primitives. In order to understand the new primitives, we shall describe an abstract loader machine for which the primitives are the operation codes. The executable code of the macro loader resides in upper memory, while object code is loaded into lower memory. Macro and external symbol definitions are stacked downward in memory from the loader. The macro loader has three registers, the ac or accumulator, the pcr or position counter register, and the rc or relocation counter. All these registers are the size of the machine address, assumed to be 16 bits or 2 bytes. The ac is a working register for intermediate results. The pcr keeps track of the present position in memory of the loading activity. Every time a byte is to be stored in the memory image, it is stored at the location specified by pcr, and pcr is incremented. The rc keeps track of the amount of relocation necessary for the present object module. Any relocatable address found in the module must have rc added to it before it is stored. The new primitives are shown in Table III.

The \$8 and \$9 macros need further explanation. They are called as \$8name\$1 or \$9other\$1. The \$1 terminates the string. When \$8 followed by the name of an external reference is encountered, the name is looked up in the table of external definitions. If it is there, the associated 16 bit quantity is stored in the memory at pcr. If it is not there, the new name is added to the table and a linked list is begun that will point through the memory at each location where this undefined name is referenced. When a \$9 macro is encountered, the name is looked up in the table also. If it is not there the name is added along with the present pcr value. If the name is already there, it is either multiply defined or there is a linked list of locations that must be filled in with the present value of the pcr.

Next we shall look at some examples. An absolute stream of bytes to be loaded beginning at location 0 will simply be those bytes terminated by a \$1. If those bytes need an origin at some point other than zero, that is accomplished by a \$2,

followed by two bytes specifying the origin, followed by a \$3. Hereafter we will use XX to represent a two byte numeric quantity. A block of uninitialized storage can be skipped by \$2XX\$5\$3. However, a much better way of handling such operations is through macro definitions. \$0 is still used to define a macro, but now \$1 is used to terminate the macro definition. Table IV gives a number of useful macro definitions, with a description of the purpose for each.

TABLE IV—Macro loader definitions and calls

Definition	Call	Purpose
\$0A\$2\$6\$7\$1	\$AXX	relocates address XX by rc.
\$0B\$2\$5\$3\$1	\$BXX	skips a block of XX bytes of memory.
	\$4	marks begin of relocatable module.
\$0P\$2\$3\$1	\$PXX	sets pcr to XX (ORG pseudo op.).
\$0R\$2\$5\$7\$1	\$RXX	deposits present address + XX.
\$0E\$P\$9\$1	\$EXXname\$1	equates external name to XX.

It should be emphasized that the macro loader is an experimental application. To use it would require the implementation of an associated assembler. It does appear that the approach has considerable promise. One problem could be that the macro language uses two bytes per symbol, and if relocatable addresses and external symbols are very common this could be expensive. The solution is to scan the object file to find out if there are any unused single byte codes. Any that are found could be used in place of the most common double character symbols used in the file. Such a method would require that a table of codes be the first portion of each object file.

APPLICATION III

Development of a file transfer mechanism between computers using a terminal-to-terminal link is a common problem among minicomputer users. Since the terminal drivers are designed to allow interactive input from users, a number of characters are treated specially. For example, in UNIX⁴ the # character is used to erase the preceding character, and the @ character to delete the characters preceding it in the line. In addition, the back slash character is used as an escape and control D (EOT) is used as an end of file indication from the terminal (causing modem hang up). Such problems are annoying when transmitting text files, but become serious when the goal is to transmit a storage image of arbitrary bytes. One solution is to do a translation from the 8-bit byte, sending only six bits with each transmitted ASCII character, and adding octal 40 (space) in order to produce only printable characters. However, some of the printable characters must still be translated (e.g. #, @ and back slash in UNIX).

The tiny macro processor presents another solution: to experimentally identify all the characters which cause problems, and substitute macros for all of them. Of course, if seven bit ASCII characters are being transmitted, only seven bits per byte can be sent out in each character. In addition, for those characters which require a macro, two characters must be sent. However, the set of characters which must be avoided due to special interpretation by the terminal driver is usually fairly small, meaning that the use of the macros need not be

```

#define LSTAK 1000
int st[LSTAK]; char ch[] {"%01%="};
main(argc, argv) int argc; char *argv[];
{
    register int i, c;
    if(argc > 1) for(i=0; i<5&&(c=argv[1][i]); i++) ch[i]=c;
    expand(0, 0);
}
expand(p, a) int p, a; /* p is defn ptr, a is input ptr */
{
    int x, d; register int c, i;
    for(x=p; c=getch(&a); )
        if(c == ch[0]) { /* macro call mark found */
            if((c=getch(&a))==ch[3] && a) c=getchar();
            for(i=0; c!=ch[i] && i<3; i++);
            switch(i) {
                case 0: putchar(c); break; /* $ */
                case 1: push(x, &p); x=p; /* 0 */
                    d=getch(&a); push(getch(&a), &p);
                    if(d == ch[4]) push(value(a), &p);
                    else while((c=getch(&a))!=d) push(c, &p);
                    push(0, &p); break;
                case 2: while(getchar() != '\n'); break; /* 1 */
                default: for(i=x; i && st[i] != c; i=st[i-1]);
                    if(i==0)
                        printf("\n%c%c undefined\n", ch[0], c);
                    else { expand(x, i+1); break; }
            }
        }
        else if(a && c == ch[3]) putchar(getchar()); /* % */
        else if(a !& c != '\n') putchar(c);
}
setch(x) int *x; { return(( *x ? st[( *x )++] : getch() )); }
push(a, b) int a, *b; /* push a on stack at *b */
{
    if( *b < LSTAK ) st[( *b )++] = a;
    else { printf("\nStack overflow, quit.\n"); exit(); }
}
value(a) int a; /* collect a 3 digit octal value */
{
    register int i, n, c;
    for(i=3, n=0; i; i--)
        if((c=getch(&a)) >= '0' && c <= '7') n = 8*n + c - '0';
        else printf("Illegal character %c in value.\n", c);
    return(n);
}

```

Figure 2—Tiny macroprocessor in C for UNIX

very frequent. A special advantage enjoyed by the macroprocessor solution is that the definition of the macro that is to be substituted for the problem character can be transmitted to the receiving computer at the beginning of the actual transmission and need not be agreed upon before the transmission starts. However, the macro language characters must be agreed upon previous to transmission. To support character translation we modify the tiny macroprocessor to allow definitions that specify a character in terms of its octal code. Examples are:

```
$0 = b007    $0 = e004    $0 = s043    $0 = a100
```

This new form of macro definition uses the unique character

= in the delimiter position to specify that the one-character macro name will be followed by exactly three octal digits specifying the code for the single character which makes up the macro body. As a result, \$b generates the ASCII bell character, \$e generates EOT, \$s generates #, and \$a generates @. It is interesting to note that the macroprocessor can also be used to provide the terminal user with the capability to generate any character code from the terminal.

The primitive macros \$2 through \$9 have been removed from this version of the tiny macroprocessor. The capabilities are still available though. For example, the definition \$0 = 2012 (which defines \$2 to be the new line character in UNIX) and definitions like \$0'3____' provide the same

abilities. Source code for a recursive version of the tiny macro processor written in the C language³ is provided in Figure 2.

One additional point should be made about character translation. Electronic data transmission is rapidly increasing in volume. Most of this transmission is in fixed length codes. Many coding techniques, e.g., Huffman coding, exist that allow considerable compression of character codes, but generate a bit stream which cannot easily be sent via fixed length codes. However, the method outlined here for substitution of macros for undesirable characters could remove that problem (depending on how many characters are undesirable). Such methods could increase the effective throughput of a transmission line, and totally through software methods. The latter point is important when the transmission hardware is under the control of others, as it often is, who have little to gain from increased transmission efficiency. With increasing processing power in the hands of users of transmission facilities, such end-to-end translation schemes could prove an important tool.

SUMMARY

A tiny macroprocessor and three of its applications have been described. The processor is a simple tool, which is easy to modify for specific purposes. Macroprocessors are an important tool for the software engineer and it is hoped that this tiny one will provide fruitful ideas for further uses. The expanding use of microcomputers, and the ever increasing number of interfaces across which communication must occur, leads to greater need for understanding and use of such simple tools.

REFERENCES

1. Gammill, R.C., "GPMX—A Portable General Purpose Macro Processor Adapted for Preprocessing FORTRAN," *Proceedings of NCC 76, AFIPS Press*, 1976, pp. 927-933.
2. Strachey, C., "A General Purpose Macrogenerator," *Computer Journal*, 8, 3, October 1965, pp. 225-241.
3. Kernighan, B.W. and Ritchie, D.C. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.
4. Ritchie, D.M., and Thompson, K.L., "The UNIX Time-sharing System," *CACM*, 17, 7, July 1974, pp. 365-375.

CAPACITY AND PERFORMANCE ANALYSIS

Finite queueing approximation techniques for analysis of computer systems

by DIMITRIS A. PROTOPAPAS

ITT/Advanced Technology Center
Shelton, Connecticut

ABSTRACT

Computer systems with finite request sourcing have been analyzed in the past using infinite source queues or an M/M/1/N queue, both of which may lead to large errors. This paper develops approximations to the response times of the M/G/1/N and GI/G/1/N queues, allowing application of these more realistic models in performance analyses of computer systems.

INTRODUCTION

In a finite source queueing model requests for service are generated by a finite population of input sources. Such a system is schematically shown in Figure 1, where each member of the calling population N alternates between an *operational state* (i.e., being outside the queue), and a *service state* (i.e., waiting to or being serviced). Each member i of the calling population is characterized by an operational time distribution with mean $1/\lambda_i$ ($i = 1, 2, \dots, N$) and a service time distribution with mean $1/\mu_i$ ($i = 1, 2, \dots, N$).

When operational and service times are exponentially distributed with identical statistics for all members of the population, and First-Come-First-Serve (FCFS) service discipline, the model is known as the classical machine interference (CMI) or machine repair model.¹ Analytic expressions for performance measures of the CMI model (M/M/1/N) are found in most queueing theory books.^{1,2,3,4} Alternatively performance measures can be approximated from closed form expressions which are derived using the asymptotic properties of the CMI model.⁵

The M/G/1/N queueing system can be analyzed only when an analytic expression for the service time distribution is known. However, even in this case, the computational load increases very rapidly with N and requires resorting to numerical computation. When the service time distribution is non-analytic or operational times are non-exponential (i.e. GI/G/1/N queue), no analytic or approximation results are known to exist.

This paper derives an approximation to the mean response time of the M/G/1/N queueing system, which does not require an analytic service time distribution. The approximation developed for M/G/1/N queues is subsequently generalized for application to GI/G/1/N queueing systems. Examples showing

the applicability of finite source queueing models in the analysis of computer systems are given. Finally, in the course of analyzing a disk subsystem, we demonstrate the magnitude of errors introduced, when an infinite source model is used in the analysis of a system with finite request sourcing.

M/G/1/N QUEUE

The mathematical analysis of the M/G/1/N queueing system (i.e., exponential operational times, general service times, single server, N request sources) is very involved and can be found in Jaiswal,⁶ and Takacs.⁷ Jaiswal arrives at an expression for the mean number L of customers waiting in the system, in terms of the probability p_o of the server being idle, assuming servicing on a FCFS basis:

$$L = N - (1 - p_o)/(\lambda/\mu) \quad (1)$$

The probability the server is idle, is given in the same reference to be

$$p_o = \left[1 + N(\lambda/\mu) \sum_{i=0}^{N-1} \binom{N-1}{i} C_i \right]^{-1} \quad (2)$$

where

$$C_i = \prod_{j=1}^i \left\{ \frac{\phi(i\lambda)}{[1 - \phi(i\lambda)]} \right\} \quad (3)$$

$(i = 1, 2, \dots, N-1)$

$$C_0 = 1$$

with

$$\phi(s) = \int_0^{\infty} e^{-st} dF(t) \quad (4)$$

being the Laplace-Stieltjes (LST) transform of the service time distribution $F(t)$.

Takacs derives an expression for the queue waiting time W_N in terms of a different parameter. However, taking into account that the composite arrival rate $\bar{\lambda}$ at the queue is

$$\bar{\lambda} = (N - L)\lambda \quad (5)$$

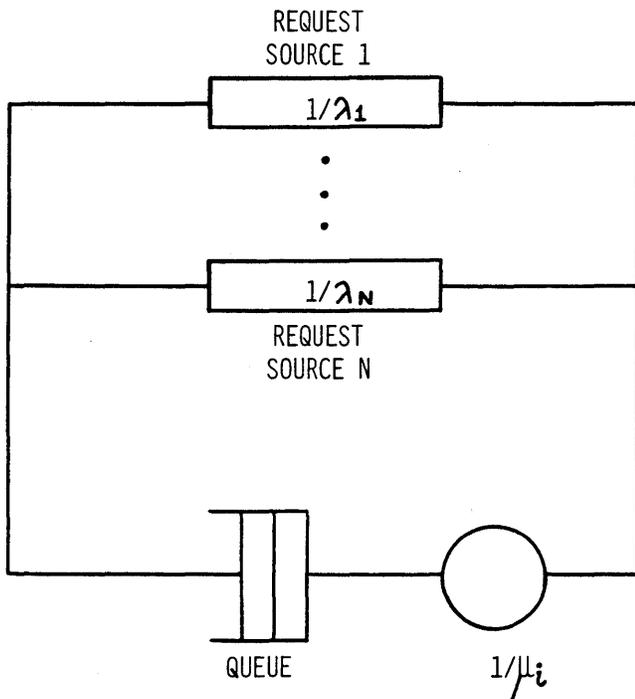


Figure 1—Finite source queueing system

and applying Little's formula⁸ we can prove that the results obtained in both references are equivalent, arriving at

$$W_{RN} = N / [\mu(1 - \rho_o)] - 1/\lambda \quad (6)$$

where W_{RN} is the mean system response time ($W_{RN} = W_N + 1/\mu$).

It is easily seen that

$$1 - \rho_o = \rho \quad (7)$$

is the equivalent composite utilization factor of the server, while

$$u = \lambda/\mu \quad (8)$$

represents the utilization of the server by one request source. Substitution into Equation (1) yields

$$L = N - \rho/u \quad (9)$$

a surprisingly simple expression.

Thus, the computational task in determining performance measures of the M/G/1/N model amounts to that of calculating the utilization factor (or equivalently the probability ρ_o that the server is idle) of the server. This, in turn, appears to necessitate knowledge of an analytic expression for the service time distribution $F(t)$, as seen from Equation (2). In addition, for a large N it is necessary to resort to numerical computations because of the magnitude of the computational task.

COMPARISON WITH THE M/G/1 QUEUE

Application of the M/G/1/N queueing model in the analysis of computer systems has been very limited⁹ to date. However, the corresponding infinite source model, i.e. M/G/1, has been used very extensively by researchers, with the implied assumption that the number of request sources is large enough to allow approximation by an infinite source queue. Such an assumption, though, would require more than 150 request sources in many cases, which is hard to justify even for some multiprogrammed systems.¹⁰

The queue waiting time for the M/G/1 queue is given by the well known P-K formula, found in most queueing theory texts including Gross:⁸

$$W = [\rho(1 + C_s^2)]/[2\mu(1 - \rho)] \quad (10)$$

where, μ is the service rate, ρ the server utilization factor, and C_s^2 the squared coefficient of variation of service times, defined from

$$C_s^2 = \sigma_s^2/(1/\mu)^2 \quad (11)$$

in terms of the variance σ_s^2 of service times.

Buzen and Goldberg¹⁰ tabulate relative response errors,

$$E = (W_R - W_{RN})/W_{RN} \quad (12)$$

where, $W_R = W + 1/\mu$, and $W_{RN} = W_N + 1/\mu$ (W_N being the queue waiting time of the M/G/1/N system). In their numerical computations they assume $\mu = 1$, determining E in terms of ρ and N for the following analytic service distributions:

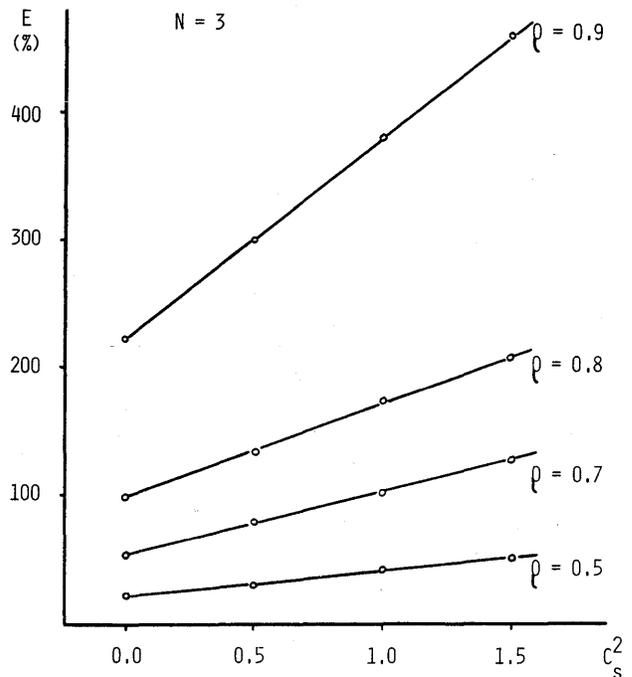


Figure 2—Relative response error vs. C_s^2 for $N = 3$

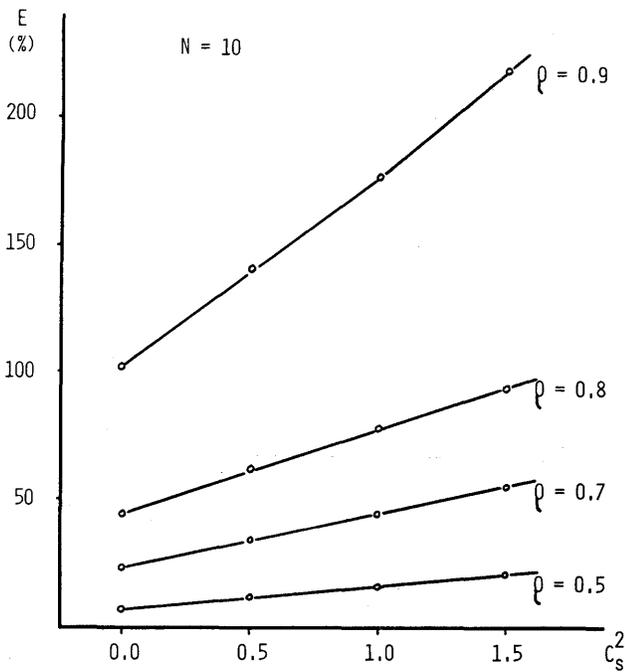


Figure 3—Relative response error vs. C_s^2 for $N = 10$

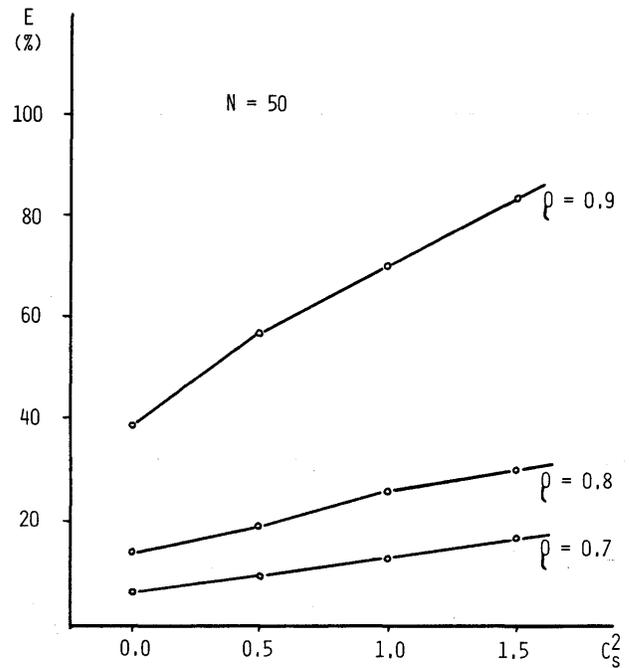


Figure 4—Relative response error vs. C_s^2 for $N = 50$

1. Constant ($C_s^2 = 0$)
2. Erlang - 2 ($C_s^2 = 0.5$)
3. exponential ($C_s^2 = 1$)
4. hyperexponential with $C_s^2 = 1.5$

Considering Equations (6), (10), and (12) we observe that, if we normalize the RHS of the expression for E , μ and λ are eliminated and E is expressed only in terms of ρ and u . Hence, E is independent of μ , making Buzen's numerical results more valuable.

From Equation (12) we have

$$W_{RN} = W_R / (1 + E) \tag{13}$$

and further

$$W_N = (\mu W - E) / [\mu(1 + E)] \tag{14}$$

However, even for the tabulated cases we have no means of relating μ/λ to ρ through a closed form expression.

We now use the numerical results published in Buzen and Goldberg¹⁰ to gain insight into the variability of the relative response error E , introduced when a simple M/G/1 model is used to approximate the finite source M/G/1/N queue. Figures 2 to 4 show plots of E as a function of the squared coefficient of variation C_s^2 , using the number of request sources N , and the equivalent utilization factor of the server ρ , as parameters. As seen, these graphs suggest a linear relationship between E and C_s^2 (for N, ρ , constant). Figure 5 depicts a strong dependency of E on N , especially for relatively small values of the latter. Thus, for instance, in the hyperexponential case ($C_s^2 = 1.5$), for $N = 20$ and $\rho = 0.9$ the response error is about 150%. However, we observe that (for the same case) even for $N = 95$, relative response errors of the order of 50% are intro-

duced. On the other hand, for $\rho = 0.5$ a relative response error of 10% is reached with only 10 and 20 independent request sources, for $C_s^2 = 0$ and $C_s^2 = 1.5$, respectively. Figure 6 shows that E varies exponentially with the utilization factor ρ of the server.

Thus, care must be exercised when applying the M/G/1 model to ensure that the number of independent request sources is sufficiently large. The latter depends on the distri-

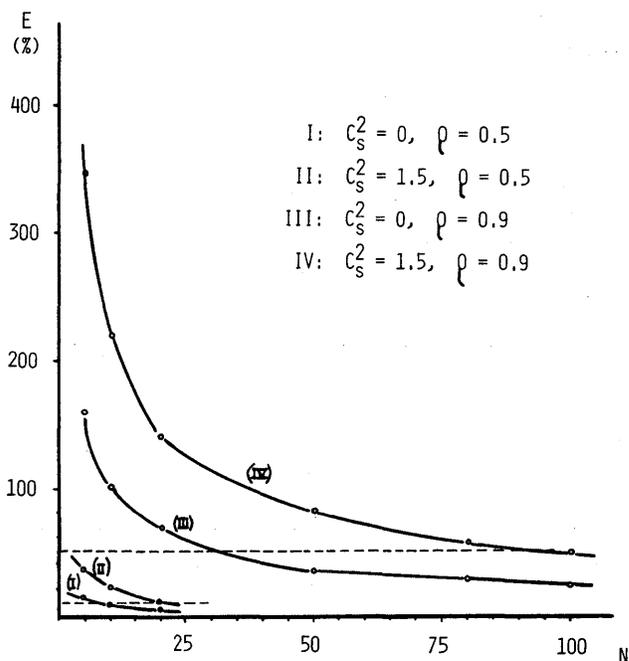
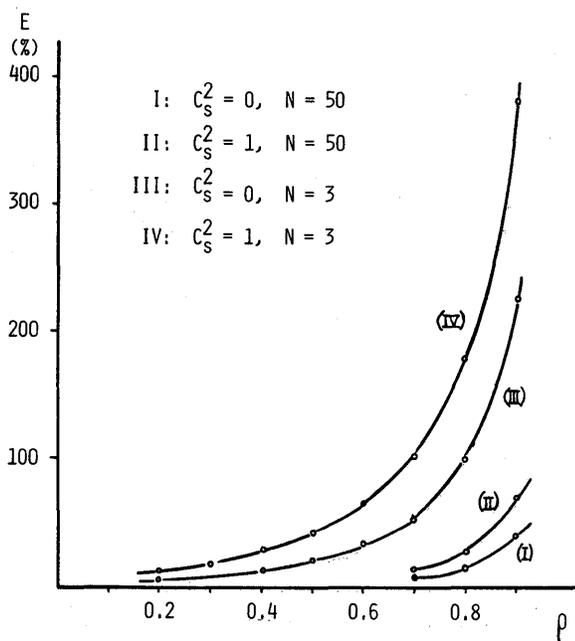


Figure 5—Plot of E vs. N

Figure 6—Plot of E vs. ρ

bution of the service times and the utilization factor of the server. Deliberate application of M/G/1 can introduce large errors which lead to gross overestimation of response times, especially at high loads.¹⁰

The two queueing models were compared on the basis of equal utilization factors. Obviously, the increased performance of the finite model is due to its "self-regulating" property, in the sense that as the number of waiting customers increases, the rate of arrivals decreases, hence preventing a long waiting line.

APPROXIMATION METHODS

From Figures 2 to 4 and from the values of E tabulated by Buzen and Goldberg,¹⁰ we observe that (for ρ fixed) the relative response error E_x in the exponential case is about twice the error E_D of the deterministic case, i.e. $E_x = 2E_D$. Similarly we observe for the errors E_E in the Erlang - 2 case that $E_E = (3/2)E_D$. Expressing both E_D and E_E in terms of E_x we have $E_D = (1/2)E_x$ and $E_E = (3/4)E_x$, which suggest the general relationship

$$E_G = E_x(1 + C_s^2)/2 \quad (15)$$

where E_G is the relative response error of a general service time distribution with squared coefficient of variation C_s^2 .

In order to investigate the accuracy of the proposed approximation we use Equation (15) to calculate E_D , E_E , and E_H (hyperexponential case), in terms of E_x . We then compare the calculated values with the corresponding "exact" values found in Buzen and Goldberg, and tabulate the errors introduced by the approximation formula in Table I.

It is seen from Table I that results obtained through Equation (15) do not deviate by more than 14% from exact ones.

Using Equation (13), it is determined that the maximum relative error on W_{RN} occurs for deterministic service and amounts to an overestimation of W_{RN} by only 10%. It is noted that this represents a worst case, since N is only 3, the utilization factor is 0.9, and the service distribution is deterministic. Therefore, this approximation may be used to provide reasonably accurate predictions to response times of the M/G/1/N queueing system, when:

1. The service time distribution is non-analytic.
2. An analytic expression for the service distribution is available, but N is comparatively large, making an exact analysis very involved computationally.

Alternatively, Equation (15) may be used to provide directly the error relative to the corresponding infinite source M/G/1 queue, in order to decide if the particular M/G/1/N model can be approximated satisfactorily by an M/G/1 one.

Substitution of Equation (12) into (15) yields

$$\frac{(W_{M/G/1} - W_{M/G/1/N})/W_{M/G/1/N}}{(1 + C_s^2)(W_{M/M/1} - W_{M/M/1/N})/2W_{M/M/1/N}} = \quad (16)$$

where W represents response times. Thus, determination of $W_{M/G/1/N}$ reduces to determining the response time of the respective M/M/1/N queueing system. Computation of the latter is comparatively straightforward, or it may be taken from extensive published tables.¹¹

Now, considering that in an M/G/1/N queue the squared coefficient of variation of arrival times C_a^2 is 1, Equation (15) may be generalized for GI/G/1/N queues to

$$E_G = E_x(C_a^2 + C_s^2)/2 \quad (17)$$

or

$$\frac{(W_{GI/G/1} - W_{GI/G/1/N})/W_{GI/G/1/N}}{\times (W_{M/M/1} - W_{M/M/1/N})/2W_{M/M/1/N}} = (C_a^2 + C_s^2) \quad (18)$$

Equation (18) allows approximation of performance measures of finite source queueing systems when operational times are independent and generally distributed. An analytic expression for the distribution of the operational times is not required; only the respective squared coefficient of variation need be known. As seen, prior to determining $W_{GI/G/1/N}$ the response time of the corresponding infinite source generalized queue (GI/G/1) must be calculated. Although the latter is not analytically tractable,⁸ reasonably accurate and computationally simple approximation methods exist for its analysis.¹²

The factor $(C_a^2 + C_s^2)/2$ is also known to appear in approximation formulas for GI/G/1, and GI/G/c queues. Interestingly, the approximation formulas holding for both classes of queues (i.e. finite source and infinite source), have the same form, except that for infinite source queues they involve queue lengths,¹² while in finite source queues they involve relative response errors E .

Finally, from Equations (6), (7), (10), (12) we obtain for the M/G/1/N queue after some algebraic manipulations

$$\mu/\lambda = N/\rho - 1/(1 + E) - [\rho(1 + C_s^2)]/[2(1 - \rho)(1 + E)] \quad (19)$$

TABLE I—Percent accuracy of the proposed approximation for deterministic, Erlang-2, and hyperexponential service distributions

N	ρ	Percent Error		
		E_D	E_E	E_H
3	0.5	3.3	4.0	2.3
3	0.7	6.8	4.3	0.8
3	0.8	11.3	1.6	1.4
3	0.9	10.8	5.6	3.1
10	0.5	1.3	1.6	0.5
10	0.7	5.6	2.7	0.2
10	0.8	9.1	3.8	5.7
10	0.9	13.7	5.4	0.4
50	0.8	6.5	3.2	7.3
50	0.9	10.7	8.2	4.7

which given an assumed value of ρ allows determination of the corresponding ratio μ/λ of the request source. It can be proven that Equation (1), and hence Equation (6), hold under more general conditions² than those implied in the M/G/1/N queue. Specifically, these two Equations hold also for GI/G/1/N queues. Thus, using again Equations (6), (7), (10), and the chosen approximation formula for $W_{GI/G/1}$, we can arrive at a relation of the type $\mu/\lambda = f(\rho, E)$ for the GI/G/1/N queueing model. If, for example, we choose the approximation formula¹²

$$W_{GI/G/1} = \rho(C_a^2 + C_s^2)/[2\mu(1 - \rho)]$$

we arrive at an Equation similar to (19) except for the parenthesized term $1 + C_s^2$ which is now $C_a^2 + C_s^2$.

APPLICATIONS

Infinite source models are easier to analyze (than finite source ones), and they are used very extensively in the analysis of computer systems. However, these models are inappropriate when:

1. The system under study is by structure a finite source one.
2. The assumption of infinite sourcing is not realistic.

Multi-microprocessor and multi-microcomputer systems are representative examples of case 1. Microprocessors/microcomputers sharing resources enter "wait states" while waiting for any resource to become "ready"; no other requests are issued by a microprocessor/microcomputer until the current request is serviced. In common-bus systems the bus itself constitutes such a shared resource.⁹ Hence, finiteness of bus requesting is an inherent characteristic of real microprocessors/microcomputers.

Case 2 concerns systems where request sourcing can be unlimited, but the actual number of request sources in a particular application may not be large enough to justify use of an infinite source model. In a small timesharing system, for example, the number of independent sources issuing disk requests is comparatively small.

Buzen and Goldberg¹⁰ were the first to caution against deliberate use of infinite source models, and point out that use of such models, when the number of request sources is not large enough, may lead to large errors. In the following we demonstrate the impact of the finiteness of request sources in the particular case of a single-drive moving-head disk subsystem.

Analysis of a Disk Subsystem

Disk models in the literature assume an infinite number of independent request sources.^{13,14,15,16,17,18} Here we assume that disk access requests are generated by N independent request sources on a cyclic basis, and that all requestors have exponentially distributed operational times and the same mean operational times. These, plus the assumption of a FCFS service discipline allow modeling of the disk using an M/G/1/N queueing system (exponential arrivals and FCFS service are also commonly used assumptions in infinite source disk models).

The access time T of a disk consists of a seek time, a latency time, and a data transfer time.¹⁵ Expressions for the mean seek time and the variance of seek times in terms of the drive parameters a , b , and the number of tracks L per disk surface, are reported by Chang.¹⁵ Latency time statistics are determined, in terms of the rotational speed r of the drive and the number of sectors/track K , from expressions derived by Fuller.¹⁶ Assuming typically $a = 8.33$ msec, $b = 0.125$ msec/

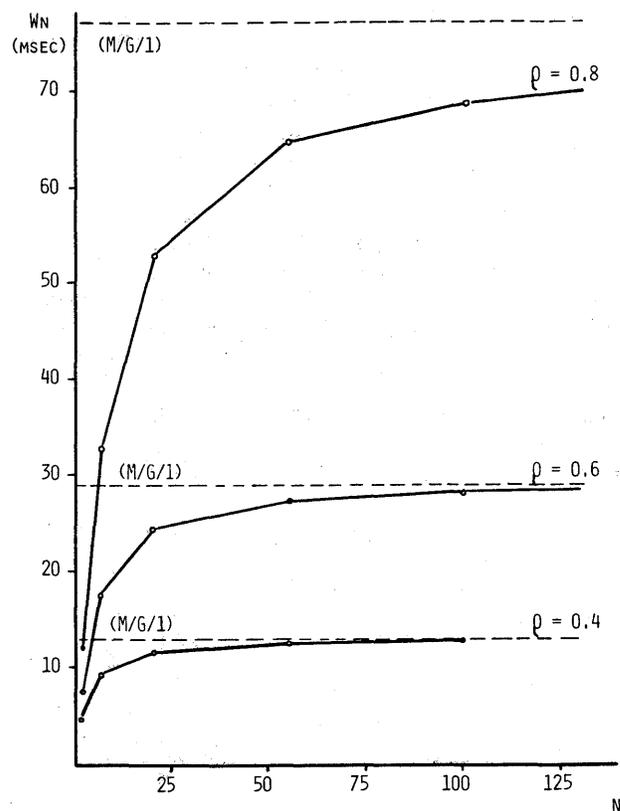


Figure 7— W_N vs. N in a disk subsystem

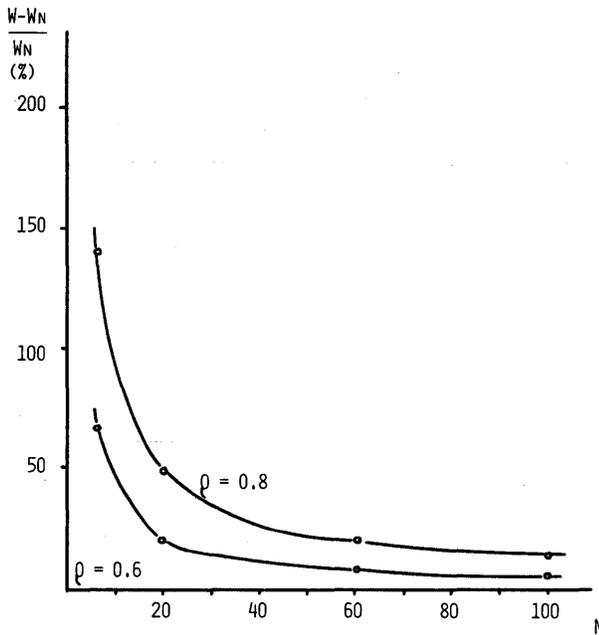


Figure 8—Relative queue waiting time errors vs. N in a disk subsystem

track, $L = 400$ tracks/surface, $r = 3600$ RPM, $K = 32$, data transfers at 1MB/sec, and exponentially distributed data block lengths with a mean of 512 bytes, we calculate $T = 34$ msec and $\sigma^2 = 163.25$ msec², where σ^2 is the variance of disk access times. Hence, the squared coefficient of variation of access times (i.e., service times in the M/G/1/N queue) is $C_s^2 = 0.14$. The disk case study represents a typical example, where the squared coefficient of variation of service times can be determined, while the respective distribution function is unknown.

The plots of queue waiting time W_N in terms of N , shown in Figure 7, indicate that the queue waiting times predicted by the infinite source model (M/G/1) are very pessimistic compared to those predicted by the correct M/G/1/N model, unless the number of request sources N is large. As seen in Figure 7, the functions $W_N = f(N)$ for the M/G/1/N queueing model approach asymptotically those corresponding to the M/G/1 model (dashed lines). For $N = 20$ and $\rho = 0.8$ M/G/1 predicts a queue waiting time of about 78 msec while the actual is only about 52 msec (i.e. 50% error in queue waiting time prediction or 30% error in response time prediction). Percent errors in the predictions of the M/G/1 queueing system are shown in Figure 8. Finally, Figure 9 shows the number of request sources N_E required to achieve a 10% error or less, (when using the M/G/1 model to predict queue waiting times) in terms of ρ . As seen, while for $\rho = 0.4$ about 20 request sources will suffice, for $\rho = 0.8$ the requirement is 160 sources. This requirement may be met in large timesharing systems; in other computer systems, in general, N is considerably smaller. Therefore for high utilizations, in particular, infinite source disk queuing models may lead to very pessimistic results, when the actual number of request sources is not sufficiently large. The latter is decided on the basis of the respective relative response error E , which is determined from Equation (15) or (17).

CONCLUSION

Response times of finite source queues are approximated in terms of the response times of the respective infinite source queues and the tractable M/M/1/N queue. These approximations (a) simplify the analysis of the M/G/1/N queue for large values of N when the service distribution is known, (b) allow analysis of the M/G/1/N queueing system when service times are non-analytic, and (c) provide means of analyzing generalized GI/G/1/N queues on the basis of the respective square coefficients of variation. Finite source queueing models are very important in analyzing multi-microprocessor/multi-microcomputer systems, in which request sourcing is finite by structure. In addition, such models are more appropriate (than infinite source ones) when an infinite request source assumption is not realistic, although the system may not be characterized by finite sourcing structurally.

REFERENCES

1. Hillier, F., and G. Lieberman. *Operations Research* (2nd ed.). San Francisco: Holden Day, 1974.
2. Kobayashi, H. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*. Reading, (Mass): Addison-Wesley, 1978.
3. Allen, A. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. New York: Academic Press, 1978.
4. Kleinrock, L. *Queueing Systems, Volume I: Theory*. New York: Wiley, 1976.
5. Ferdinand, A. "An Analysis of the Machine Interference Model." *IBM Systems Journal*, 1971, 2, pp. 129-142.
6. Jaiswal, N. *Priority Queues*. New York: Academic Press, 1968.
7. Takacs, L. *Introduction to the Theory of Queues*. Oxford: Oxford University Press, 1962.
8. Gross, D., and C. Harris. *Fundamentals of Queueing Theory*. New York: Wiley, 1974.
9. Protopapas, D., and E.J. Smith. "Modeling and Analysis of Single- and Multiple-Bus Multi-Microcomputer Systems." *COMPCON Proceedings of the IEEE*, Fall 1980, pp. 471-478.

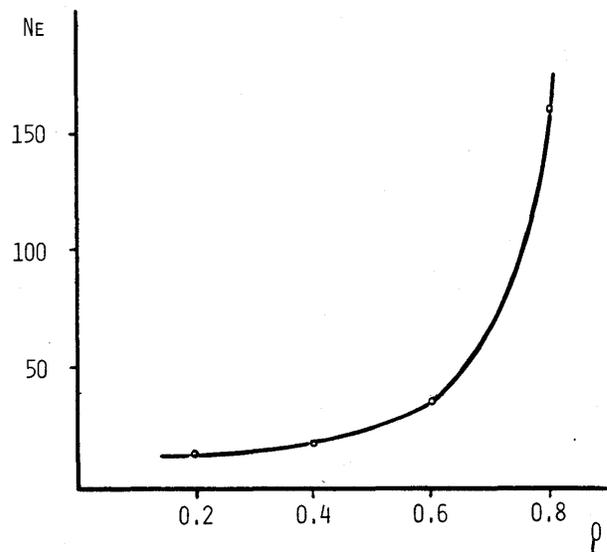


Figure 9—Required N to reduce relative error to 10% vs. ρ in a disk subsystem

10. Buzen, J., and P. Goldberg. "Guidelines for the Use of Infinite Source Queueing Models in the Analysis of Computer System Performance." *AFIPS Proceedings of the National Computer Conference, 1974*, pp. 371-374.
11. Peck, L.G., and R.N. Hazelwood. *Finite Queueing Tables*. New York: Wiley, 1958.
12. Protopapas, D. *Multi-microprocessor/Multi-microcomputer Architectures: Their Modeling and Analysis*. Ph.D. Dissertation, Polytechnic Institute of New York, May 1980.
13. Abate, J., H. Dubner, and S.B. Weinberg. "Queueing Analysis of the IBM 2314 Disk Storage Facility." *Journal of the Association for Computing Machinery*, 15 (1968), 4, pp. 577-589.
14. Abate, J., and H. Dubner. "Optimizing the Performance of a Drum-Like Storage." *IEEE Transactions in Computers*, C-18 (1969), 11, pp. 992-996.
15. Chang, J., and S. Gorenstein. "A Disk File System Shared by Several Computers in a Teleprocessing Environment." *Proceedings PIB MRI Symposium on Computer Communication Networks and Teletraffic*. Brooklyn: PIB Press, 1972.
16. Fuller, S., and F. Baskett. "An Analysis of Drum Storage Units." *Journal of the Association for Computing Machinery*, 22 (1975), 1, pp. 83-105.
17. Teorey, T., and T. Pinkerton. "A Comparative Analysis of Disk Scheduling Policies." *Communications of the Association for Computing Machinery*, 15 (1972), 3, pp. 177-184.
18. Wilhelm, N. "A General Model for the Performance of Disk Systems." *Journal of the Association for Computing Machinery*, 24 (1977), 1, pp. 14-31.

Throughput-response measurements in a distributed CAD/CAM processing network

by J.R. RAO and W.L. HANNA

McDonnell Douglas Automation Company
St. Louis, Missouri

ABSTRACT

A methodology for monitoring response and throughput of a distributed graphics CAD/CAM processing network is presented. The hardware and software components of distributed processing include intelligent graphics terminals, remote minicomputers, high speed communication processing, host processor, and auxiliary storage devices. Software functions consist of loading and executing CAD/CAM applications such as computer aided design, computer aided quality assurance, etc.

The response and throughput are measured in terms of completed graphics functions, which may be as trivial as generating a point and as complex as retrieving a drawing from a disk file. The graphics system response performance has been a critical consideration for its status as a cost-effective design tool. The data gathering facility discussed in this article has been used quite effectively in production for evaluating the impact of hardware and software changes. As of December 1980, the distributed graphics system at the McDonnell Douglas Corporation-St. Louis, consists of approximately 70 3-D graphics terminals, 11 remote processing systems and 5 front-end processors. A sample illustration of the measured data is presented in this article.

INTRODUCTION

The distributed graphics system (DGS) at McDonnell Douglas Corporation (MDC) has been in production for CAD/CAM applications since 1978. The number of satellite remote processing centers, PDP 11/70s sharing graphics processing workload with the IBM 370/3033 host, is now at 11 in St. Louis alone. In 1980 the remotes altogether supported about 70 Evans & Sutherland (E&S) graphic terminals and about 10 hard copy plotter terminals. The remote systems interface with the host system via 9.6KB-56KB high speed communication lines and the PDP 11/34 front-end processors. Figure 1a illustrates the distributed processing concept and Figure 1b gives an overall DGS network configuration.

The intended purpose of DGS is to: (1) improve CAD/CAM user productivity and (2) distribute application-dependent processing, (3) reduce the cost of graphics pro-

cessing on the host, (4) provide state-of-the art hardware and software graphics technology. As a result, several levels of processing are introduced, making it a "top-down" system network configuration; the network operation gets complex when a graphics-user-oriented software needs to be implemented on philosophically different vendor-supplied hardware components. No matter how complex these issues may be, the graphics system performance and the user productivity are the key factors in determining the acceptability of the system. Through the phased-in development approach we have managed to isolate and resolve individual problems as they are encountered. The DGS performance monitoring is being addressed at different levels. In an earlier publication,¹ we have presented a methodology for measuring graphics-application software host execution characteristics for individual modules and subroutine functions. In this article, we further extend real-time monitoring capabilities to measure network response and throughput.

In an interactive system environment, the user's relative productivity depends mainly on the system response performance. Time will not be spent covering arguments such as the impact of user think time on the system utilization and performance.

In this paper we will discuss the tools developed for measuring: (1) host interaction response, (2) graphics user workload, and (3) the impact of altering hardware and software configurations. A methodology for interactive computer service measurements has been proposed in terms of 5 categories:^{2,3,4} (1) time-based measures, (2) measures of length or volume, (3) multiplicities and frequencies, (4) rate-based measures, and (5) ratios. The response and throughput measurement concepts presented here for graphics networks encompass the descriptions of categories 1 & 2.

The monitoring tools operate in display and background data collection modes. The display mode is used in real-time support of the remotes. The background program collects data for response and throughput performance trend evaluations. Management has been very interested in knowing the DGS productivity and response when hardware and software enhancements are made. Availability of reports had to be made very precise and timely. From the data gathered over a period of stable system operation, a correlation seems to exist between response and throughput. Since different CAD/CAM applications put varying amounts of cpu and I/O loads

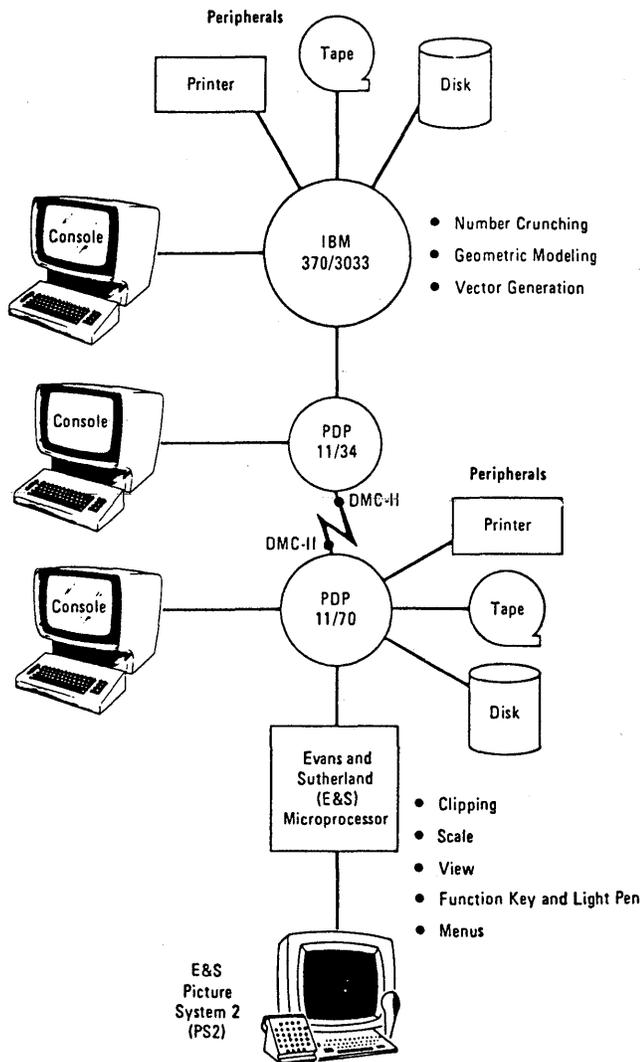


Figure 1a—Distributed Graphic System (DGS)

on the system, a constant effort is made through scheduling to maintain a balanced mix of users.

The present DGS configuration at the McDonnell Douglas Corporation—St. Louis, has the capability to provide 24-hour support for about 70 CAD/CAM graphics users. A similar organization exists for the Los Angeles, California, MDC facilities.

In this paper we will discuss how response-data gathering tools are used in determining the impact of hardware and software configuration changes on DGS response and throughput.

DISTRIBUTED GRAPHICS PROCESSING SYSTEM

The DGS hardware configuration is based on four steps of processing: (1) picture generation, (2) remote processing, (3) communication processing, and (4) host and direct access storage device (DASD) processing. The picture generation function features include picture clipping for display, CRT

electron beam positioning and display buffer refresh control. These are accomplished with the aid of hardware components such as geometry and text picture processor, picture system memory, picture display generator, and display devices.⁵

The remote processing is implemented on a PDP 11/70. The processing functions here include execution and coordination of graphics subroutine tasks initiated by a number of picture generation systems (Evans & Sutherland intelligent graphics terminals). Other processing supports provided by the remote system are local hard copy production, remote-host message communication interface software, local engineering application processing, and performance monitoring.

The message communication processing is done mainly through the high speed bitserial full-duplex communication using DEC's DMC11 controller pairs. The DMC11s provide for message synchronization, header and message formatting, error checking, and retransmission control. The message multiplexing for serial-parallel conversions between DMC11 and the host takes place by way of a front end PDP 11/34 and DEC DX11 hardware. On the remote side, message multiplexing is done between DMC11 and PDP 11/70 through communication's ancillary control programs (ACP) software. The details of ACP are presented in a paper by Veck.⁶

The graphics interactions requiring high speed processing capability, such as those shown in Figure 1a, go to the host. These also include interactions requiring access to different graphics application and functional modules that need to be loaded from the DASD. The DASD capabilities are also required for saving and retrieving of the drawing models. The CAD/CAM DASD processing includes use of a high speed electronic drum and a number of IBM 3330 and 3350 disk units.

Basically there are two types of graphics interactions (that require processing) included under distributed processing: (1) on remote system only, and (2) on remote and host. Since the remote systems provide a dedicated service to the graphics terminals, the interactions of the remote-system-only type are assumed to take constant processing times (at this point we have no remote application disk processing). The DGS response measurements discussed in this article will refer to the interactions of the second type.

The host interactions pass through two phases of communication processing in addition to the remote, host, and DASD processing. These phases are: (1) message SEND-message ACKNOWLEDGE (NOT-ACKNOWLEDGE) and (2) message SEND-message RECEIVE (see Figure 2). The message SEND-ACKNOWLEDGE is completed between a remote PDP 11/70 and the front-end PDP 11/34 processor. The standard message communication procedure adopted includes use of DEC's DDCMP byte-oriented protocol. The DMC11s facilitating high speed 9.6KB–56KB line synchronous transmission provides for send/receive message and data integrity.^{7,8} The data or text transfer between the remote and the host takes place during message SEND-message RECEIVE phase. A typical graphics interaction such as retrieving a drawing from the DASD may involve several byte receives depending on the size of the drawing for a single message SEND (or transmit). An interaction for saving a drawing on the disk (equivalent to a single graphics activity) may require a number of byte transmits.

The DGS system is always in one of two operational states: (1) user and (2) system. State 1 is referred as the user think-time period and state 2 is used to determine system response. The graphics user interaction response is defined as

$$\text{User Response} = (t_{PS2} + t_{RMt}) + t_{DMC} + t_{FRD} + t_{Host} + t_{DASD} \quad (1)$$

Where

- t_{PS2} = PS-2 processing time,
- t_{RMt} = Remote PDP 11/70 Processing time

t_{DMC} = DMC11-DMC11 high-speed processing time (includes error checking and retransmission)

t_{FRD} = Front-end processing time

t_{HOST} = Host processing time

t_{DASD} = DASD Processing time, a function of number of disk or drum accesses.

Considering t_{PS2} and t_{RMt} as constants for a host-bound graphics interaction, the host response is defined as

$$\text{Host Response} = t_{DMC} + t_{FRD} + t_{HOST} + t_{DASD} \quad (2)$$

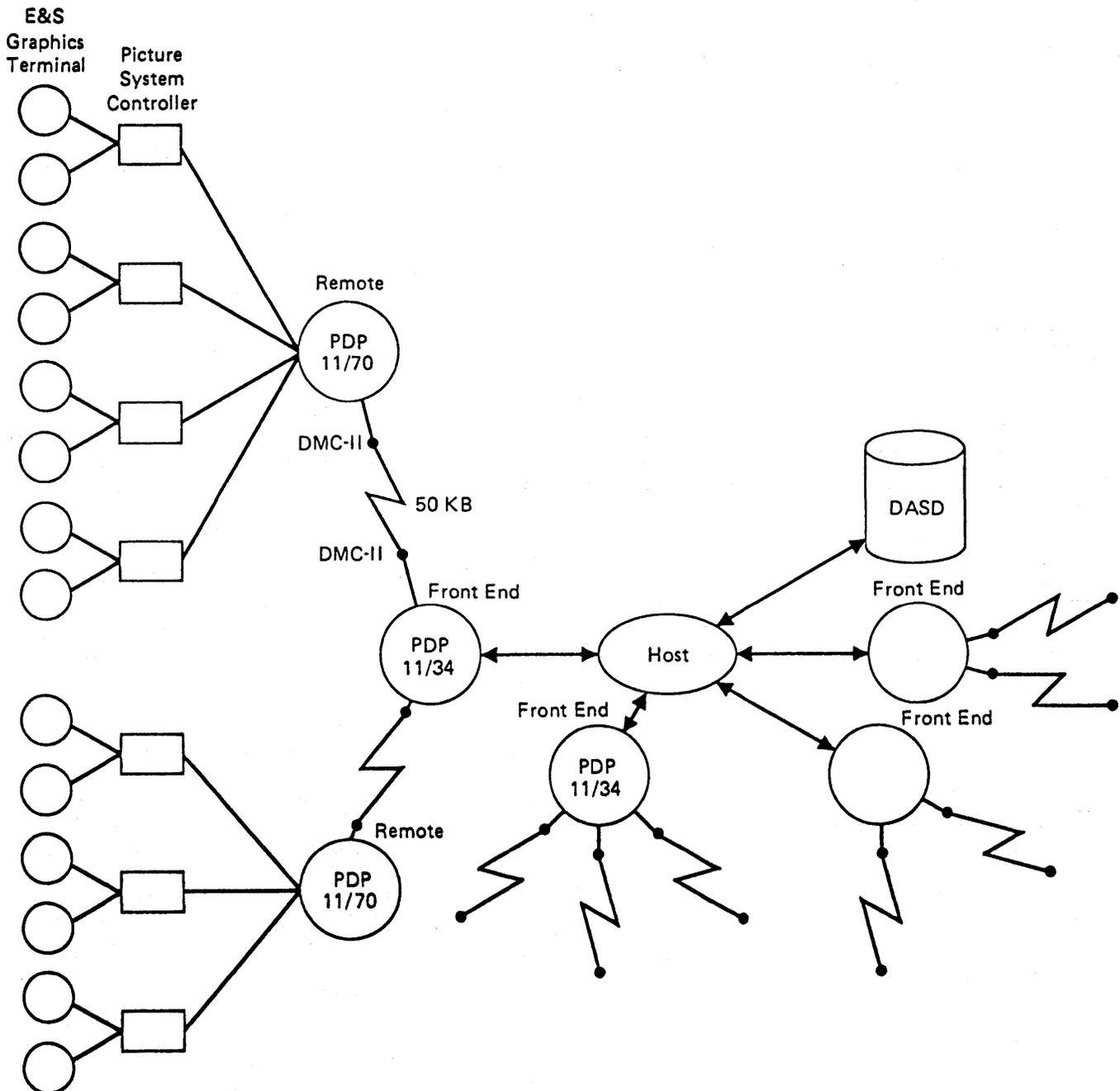


Figure 1b—Distributed Graphic System network configuration

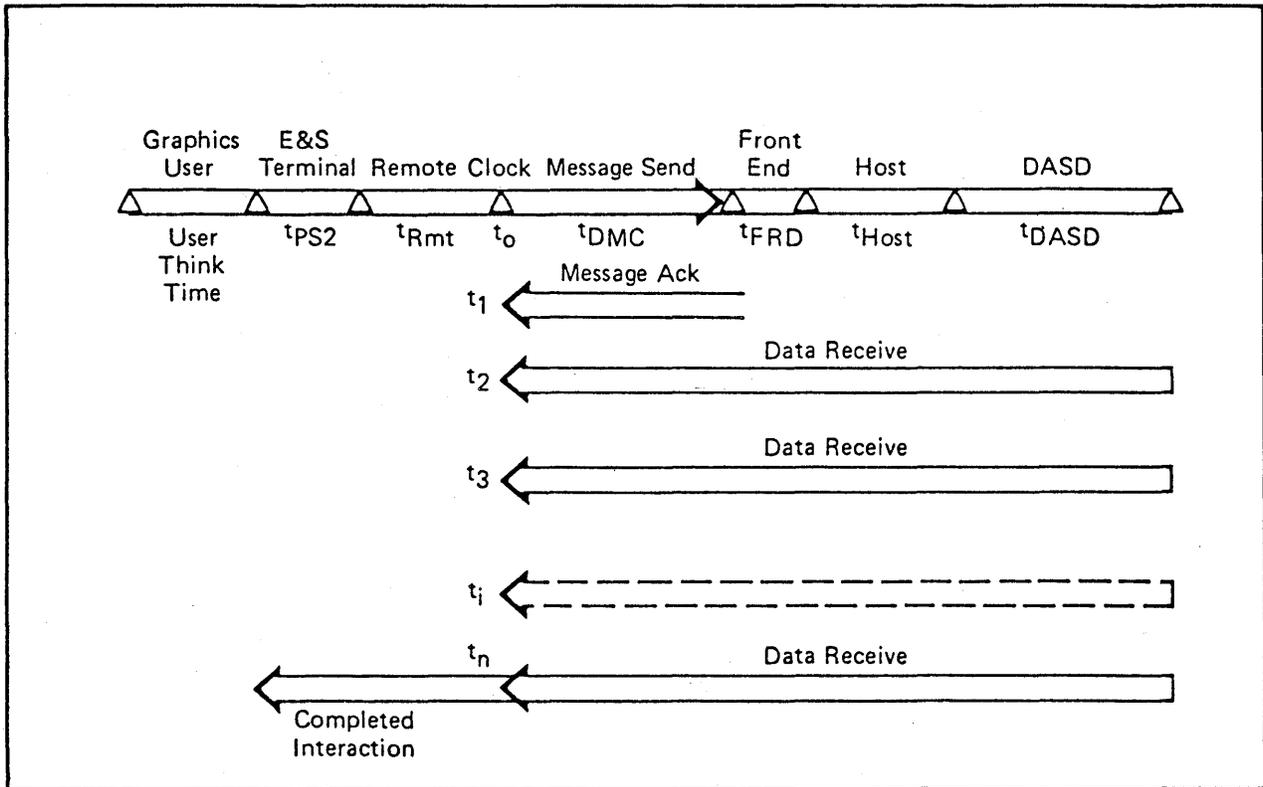


Figure 2—Send-Ack-Receive concepts

As shown in Figure 2, the difference in clock measurements gives the elapsed time. The graphics transaction could be of any size from a trivial one to a large one involving a number of receives and transmits. In a trivial interaction, such as generating a geometric point on the graphics terminal, there may be a single data receive for a successful SEND-ACK completion.

By definition,

$$\text{Remote/front-end communication time } (t_c) = (T_1 - T_0) \quad (3)$$

$$\text{Host response time } (t_H) = (T_n - T_0) \quad (4)$$

The t_c and t_H are measured as clock ticks and converted to seconds for convenient representation. Next we will discuss the software tools used for collecting and analyzing the data.

RESPONSE DATA GATHERING

The DGS network-response evaluation programs discussed in this paper operate in two modes, (1) display and (2) background data gathering. The display mode is used by the system support operator for real-time monitoring of user activity on geographically distributed remotes. This feature is invoked by using telephone dial-up capability into each remote. The display program, when active, maps to a common statistics area in the application at the specified sample intervals and displays updated interaction information. The communication interface routines collect communication statistics between the remote and the host systems for all the active application

tasks using graphics. The display program calculates running averages indicating the response trends of poor, average and good over the total session. The display data is used to identify and resolve user, remote, communications, or host problems. When the remote production support operator notices user problems, he initiates appropriate system corrective-action procedures. (See Figure 3 for display output format and Appendix A for field descriptions.) In Figure 3, eight active users are shown on Remote system #8. The average interaction time over the previous 60 seconds for all users (2.2 seconds), being less than the averages taken over the beginning-of-session periods (2.8 seconds), indicates a good response trend.

The background data collection task is kept active all the time on all the remotes; its execution command is made part of remote system booting sequence. This program writes out statistics records to a disk file at fixed intervals and whenever there is an asynchronous event such as logon or logoff. The asynchronous events are generated by the response monitor software of the application (see Figure 4). The background program attends to these asynchronous events on a FIFO basis with priority to logoff (user return to job scheduling) events. In Figure 4, two functional flow charts are presented.

The response monitor code is made part of the communication interface, which interfaces graphics application software and the remote system communication software. The response monitor recognizing the logoff, logon, system abend, and system cancellation, sets up event flags as they arise, and sends a 15-word data packet to the background program. The data packet consists of application task name,

DGS RESPONSE MONITOR

SAMPLING INTERVAL 15 SECONDS

Host Channel Address 281
DGs Remote #8

Date 1 Oct-80
Time 12:50:47

TUBE #	SESS STRT TIME	WAIT MODE	CURR ELAP TIME	PREV FNCT WAIT	LAST MIN WAIT	AVER SESS WAIT	# HOST INST	# XMTS TO HOST	# BYTES TRANS MITTED	# RCVS FROM HOST	# BYTES RECVD	RMT COMM TIME
1	12:18	U(R)	135	0.17	0.3	2.6	136	160	2106	356	117632	0.05
2	12:13	H(R)	0	0.28	0.4	2.2	5	19	296	50	25438	0.05
3	12:48	H(R)	0	21.55	7.0	4.3	45	60	5412	230	114072	0.07
4	12:42	U(R)	0	0.22	0.3	2.3	115	132	10400	329	135366	0.06
5	12:42	U(R)	30	0.25	1.2	5.1	49	49	2414	138	73334	0.12
6	12:48	U(R)	15	12.32	0.6	3.5	29	51	4792	171	73608	0.08
7	12:44	U(R)	45	19.73	28.5	7.4	29	29	1422	88	45360	0.15
8	12:44	U(R)	0	19.35	1.8	3.9	34	44	5908	134	39424	0.08

Avg. 2.2 Avg. 2.8
Figure 3—DGS real-time monitor display output

Response Trend—Good

common block address, and other mapping information. This response monitor program also develops host and remote/communication statistics histograms. The background program maps to the area pointed to by the common block address in data packet for response and other statistics. This program uses receive-data-packet directives for processing data packets passed on to it by the response monitor. The

response monitoring software has been efficiently coded in order to impose minimum overhead on the resource-limited remote systems.

Each statistics record written by the background program is 248 bytes long (256 bytes with overhead, exactly one-half block size), and consists of data on application task name, remote ID, terminal ID, etc., as shown in Appendix B. The

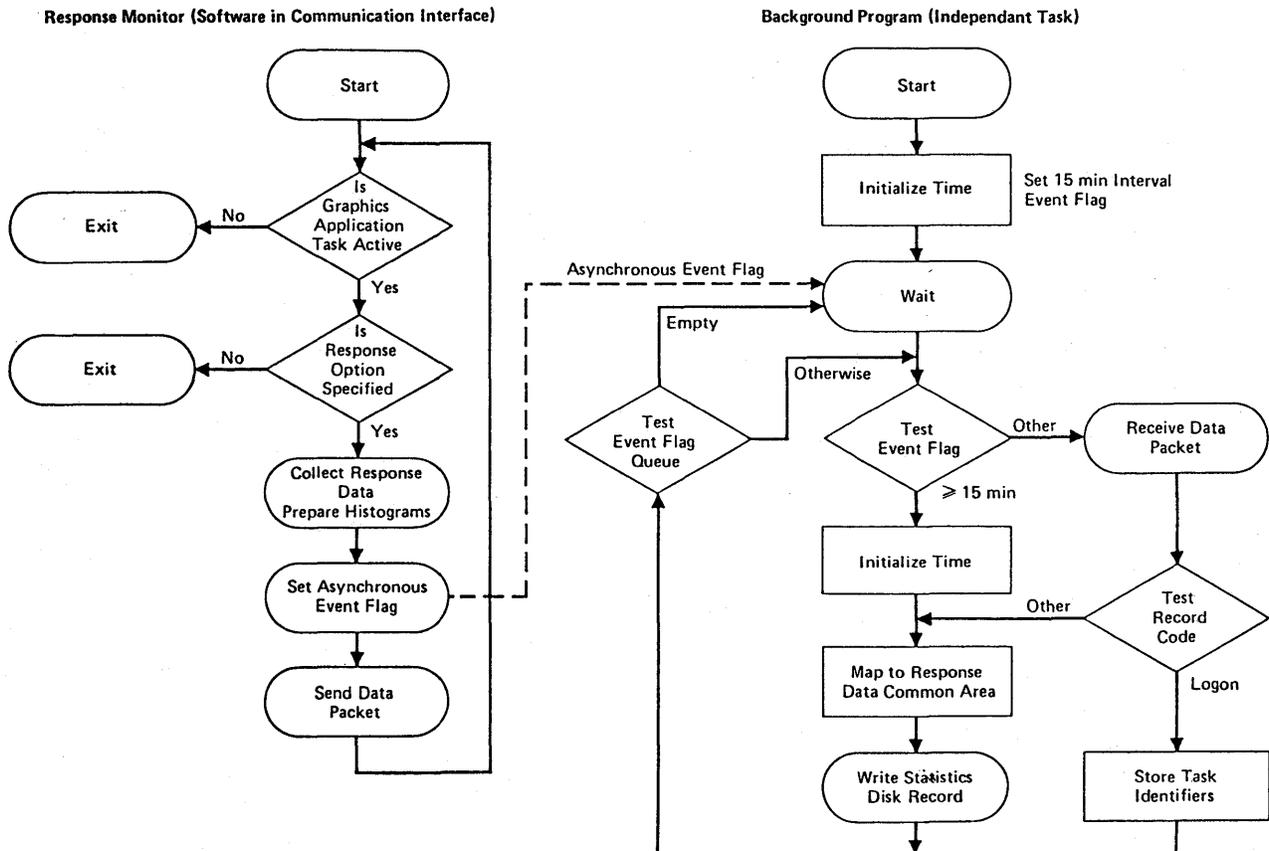


Figure 4—Overview of response data gathering software

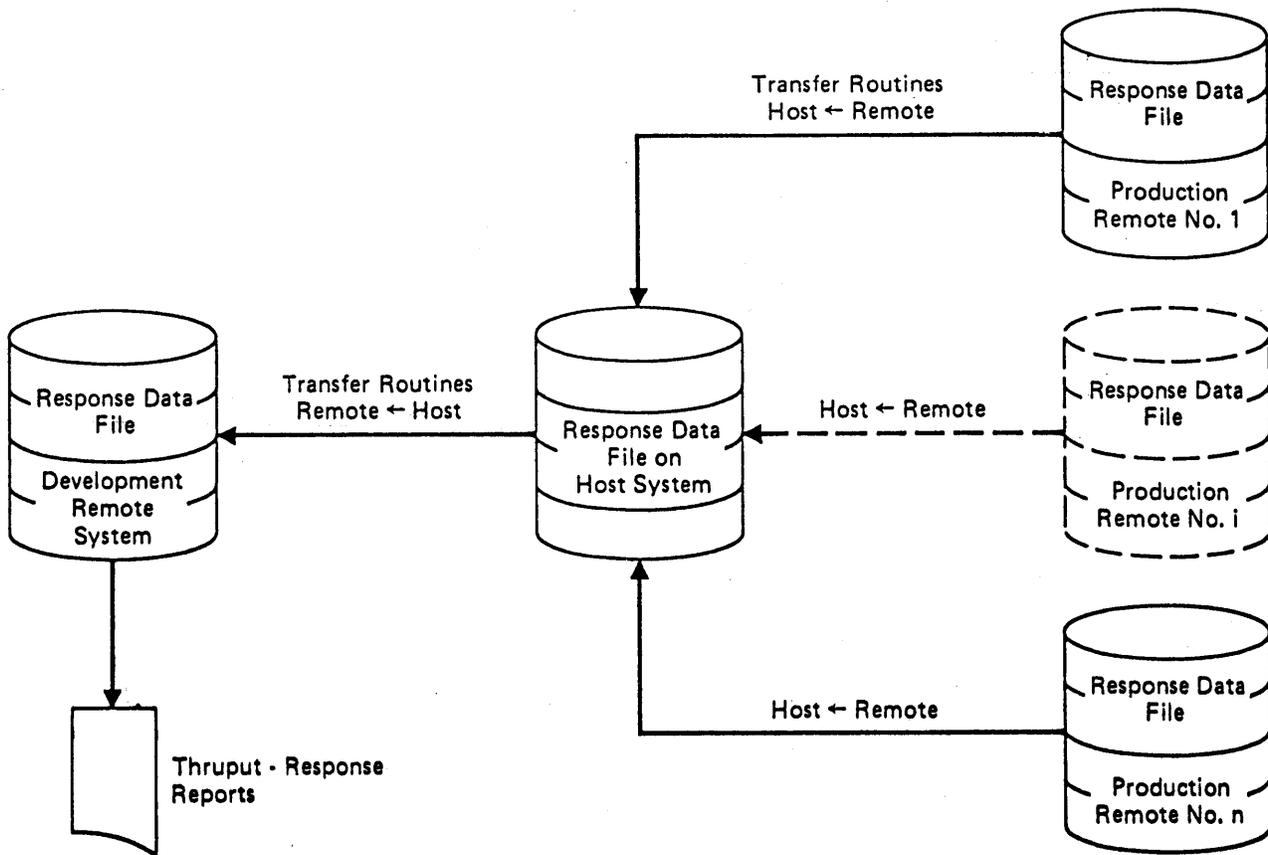


Figure 5—Response data transfer facility

record-type codes indicate graphics session cancellations, session aborts, logoffs, multiple logons without logoff, etc. Based on these codes, the record summaries are prepared for response and throughput reporting.

REPORT GENERATION CAPABILITIES

The geographically distributed remotes normally do not have report generation capability. This is done mainly to reduce the remote system overhead; besides, the application users are often little interested in network management information.

A general capability of transferring data from remotes to a central location is developed, as illustrated in Figure 5. The response data files from each of the production remote disks are transferred to appropriately named files on the host (Prod 1, Prod 2, . . .) and from there the data is transferred to the production support remote for report generation. The backup procedures include keeping the response data for at least two weeks. Every day about 600 248-byte binary data records are transferred from each remote; the system tags 8 bytes of overhead to each record, making 256 bytes total transfer. We have automated most of the transferring and data sorting functions in order to minimize human intervention, errors, and processing requirements.

To accommodate various requirements, three levels of reports are generated:

1. DGS Individual Remote Management Report,
2. DGS Response Cumulative Management Report, and
3. DGS Individual Remote Detailed Report.

Some details of level 2 reports are discussed in this paper. The other two types of reports are used mainly for detailed analysis of DGS network operation. Due to a combination of hardware and software maintenance error, the date, remote system ID, LDL, etc. were recorded incorrectly. A program "Detail" is used to rectify some of these problems.

The type 2 management report consists of cumulative summary for all the specified remotes. (See Figure 6 and Appendix C for explanations.) The maximum number of active users is determined, based on simultaneous graphics users on the DGS network. The response time histogram indicates the number of user interactions completed in each time interval and their relative percentages. The graphics throughput per tube-minute is calculated based on the number of host instances (user interactions) completed over the user session tube-minutes. As shown in Figure 6, over a 12-hour production period, 77% of the interactions were completed within one second. About 3.1% of the interactions took longer than 10 seconds. Interactions taking longer than 10 seconds are normally activities such as filing or retrieving a drawing from DASD. On the average, slow response is indicated by the instance histogram skewed to the right. For example, observe the data for hours 1300–1500 in the figure. The throughput

between 1300-1400 was about 5 interactions/tube-minute, with a maximum of 37 users, and the average response of 2.11 seconds. During 1400-1500 the average response improved to 1.77, and the throughput is about 6, with the maximum number of users remaining at 37. In the next section we will further illustrate how such data is used in evaluating the impact of changes in hardware and software configurations.

EVALUATION PROCEDURE

The response data has been gathered on the DGS network for more than a year. During this period we have gone through many hardware and software changes. The development and expansion of the DGS network has been scheduled so that the productivity impact on users will be minimal. The host system also supports other workloads in addition to the DGS system load.

The results discussed in this article are only a sample representation of response reports' development as a barometer of DGS activities. For instance, the three major phases DGS went through over the last year were: (1) changing the host system from IBM 370/168 to IBM 370/3033; (2) implementation of computer-aided design and drafting (CADD) software on the two different host configurations; and (3) implementation of Large Model Access (LMA) software on IBM 370/3033. The processing speeds of the two host configurations are different. The 3033s offer higher cpu processing capabilities. The CADD and LMA software concepts are also different. Under LMA, drawing sizes approximately 20 times larger than CADD's can be created and saved, offering extremely powerful design capabilities. LMA also provides geometry group compress/expand features that were absent in CADD software. For LMA, drawing file/retrieve software has been considerably modified to keep track of common entities

necessary for building a working file for a large drawing. The interaction byte transfer requirements between remote station and host are also higher for LMA.

A summary of response and throughput for the three major configuration changes are presented in Figures 7 and 8, based on the data collected during their production use. Changing host configurations resulted in a considerable improvement in response, though implementing LMA software slowed response down slightly. The response seems to be much more balanced throughout the day with the IBM 370/3033 hardware configuration.

The DGS throughput has improved with the new host configuration. However, the degree of improvement is not as great as that shown by the response improvements, indicating that the return from response improvements in the form of throughput will be marginal beyond a certain level of response service.

For example (Readings from Figures 7 and 8):

	CADD on IBM 370/168		LMA on IBM 370/3033		% improvement	
	Avg. Time response	Avg. throughput	Avg. response	Avg. throughput	Avg. response	Avg. throughput
0800-1200	3.06	5.225	1.585	5.91	48%	13%

At the best response service level, the user think time becomes a determining factor of productivity and graphics design throughput.

We are continuing work in this area of improving CAD/CAM productivity and evaluating it. With the rapid changes in CAD/CAM technology, the configuration discussed in this paper may be altered in the next six months.

MANAGEMENT REPORT

DAILY DGS RESPONSE

REMOTE SYSTEMS (8): 1 3 4 5 6 8 10 11

APPLICATION: LMATSK DATE: 9/29/80

HR. OF DAY	NO. OF SESS INIT.	NO. OF SESS TERM.	MAX ACTIVE USERS	TOTAL TUBE MIN.	NO. OF HOST INSTANCES	AVE. HOST RESPONSE	% HOST SESS TIME	RESPONSE TIME/INSTANCE HISTOGRAM (RESP. TIME INTERVALS (SEC), NO., REL. %)									
								0 - <= 1.	1 - <= 3.	3 - <= 5.	5 - <= 10.	> 10.					
6-7	8	0	8	168	1119	0.60	6.69	1017	90.9	68	6.1	10	0.9	13	1.2	11	1.0
7-8	37	10	23	952	5742	1.04	10.49	4899	85.3	524	9.1	121	2.1	78	1.4	120	2.1
8-9	19	11	28	1680	8266	1.95	16.02	6212	75.2	1256	15.2	293	3.5	203	2.5	302	3.7
9-10	17	16	35	2092	10522	2.24	18.75	7427	70.6	1877	17.8	512	4.9	335	3.2	371	3.5
10-11	14	13	36	2160	9848	2.81	21.34	6365	64.6	1955	19.9	646	6.6	439	4.5	443	4.5
11-12	8	20	37	1873	9673	1.94	16.68	7375	76.2	1357	14.0	358	3.7	274	2.8	309	3.2
12-13	27	15	26	1560	9280	1.19	11.82	7862	84.7	870	9.4	213	2.3	118	1.3	217	2.3
13-14	18	19	37	2154	10709	2.11	17.47	7825	73.1	1852	17.3	417	3.9	273	2.5	342	3.2
14-15	12	20	37	1792	10142	1.77	16.72	7853	77.4	1433	14.1	319	3.1	213	2.1	324	3.2
15-16	21	32	21	1042	6063	1.22	11.82	5023	82.8	621	10.2	139	2.3	112	1.8	168	2.8
16-17	14	14	15	897	5260	0.93	9.06	4465	84.9	528	10.0	86	1.6	81	1.5	100	1.9
17-18	12	10	10	600	4192	1.09	12.71	3542	84.5	429	10.2	69	1.6	57	1.4	95	2.3
							AVG. 1.78										
							%15.58										
TOTAL	207	180		17250	90816			69865	12770	3183	2196	2802					
AT ANY INSTANT MAX USERS ON SYSTEM: 37							TOTAL %	76.9	14.1	3.5	2.4	3.1					

Figure 6—DGS response cumulative management report

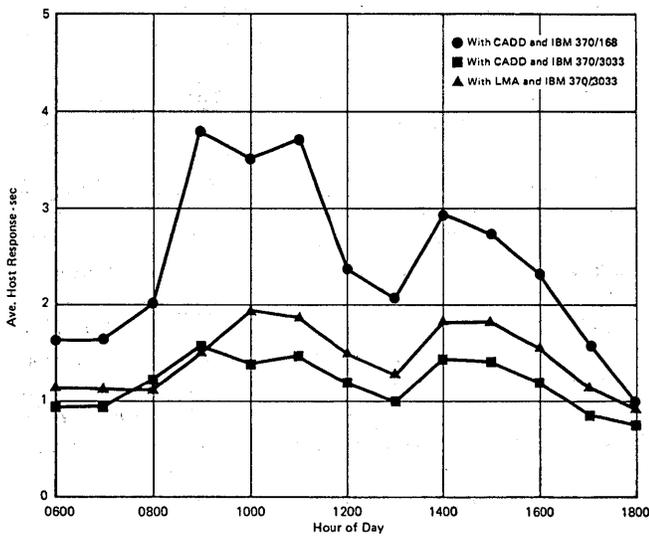


Figure 7—Average host response pattern

ACKNOWLEDGMENTS

We acknowledge the considerable effort put in by E. Ackerman, F. Dawson and B. Durham in coding and testing the software. We further extend our appreciation to the National Computer Conference's referees for reviewing the paper.

APPENDIX A: DESCRIPTION OF FIELDS IN REALTIME DISPLAY (see Figure 3)

DGS Remote #—Remote system being monitored
 Sampling Interval #—specified interval in seconds for updating display data

Tube # — Graphics terminal ID

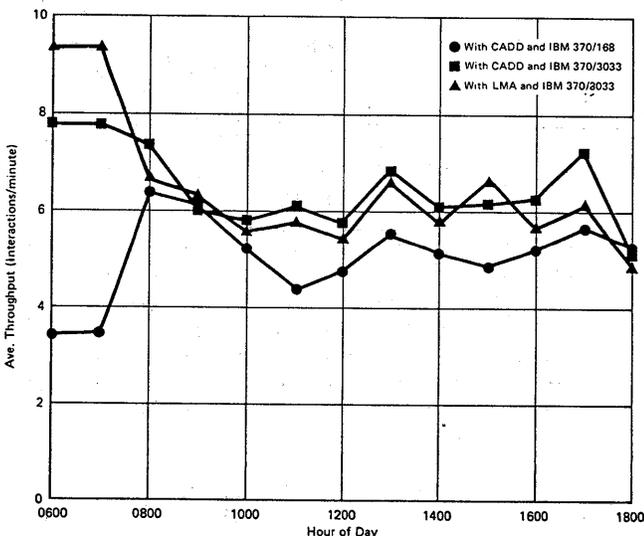


Figure 8—Average host throughput pattern

SESS STRT TIME — Graphics user session start-time (hour:minutes)

WAIT MODE — Indicates graphics activity mode by A(B), where
 A = H if the activity is on the host side
 A = U if the activity is on the remote or user side
 B = R if user is in receive mode
 B = S if previous activity was send from remote
 B = O if in wait for open communication to host
 B = W if hung on host side

CURR ELAP TIME — Elapsed time in seconds in the current wait mode

IMME PRE WAIT — Elapsed time in seconds in completing the immediately previous function

LAST MIN WAIT — Average elapsed time in seconds over all functions done during last 1 minute

AVG SESS WAIT — Average elapsed time in seconds over all functions done since the beginning of the user graphics session

I N S — Number of host interactions completed

XMTS — Total number of transmits from remote to host

BYTS XMTD — Total number of bytes transmitted from remote to host

RECP — Total number of receives from host to remote

BYTS RECV — Total number of bytes received from host to remote

REMT COMM TIME — Elapsed time in seconds on remote side, as shown in Figure 3

APPENDIX B: STATISTICS RECORD LAYOUT

Field	Length (in bytes)	Description
1	4	Number of transmits to host
2	4	Number of bytes transmitted to host
3	4	Number of receives from host

4	4	Number of bytes received from host	No. of SESS TERM	- Number of user proper logoffs (i.e., proper return to job scheduling)
5	4	CAD/CAM Application task name	MAX	- Maximum number of active graphics users at any instant during an hour on the remote
6	4	Total host interaction time (in ticks)	ACTIVE USERS	
7	2	Remote system ID		
8	2	Logical data link	TOTAL SESS	- Total number of tube minutes for all logged-on users during an hour
9	2	Graphics terminal ID	TUBE MIN	
10	2	Record code		
11	2	Total number of host interactions	NO. OF HOST INSTANCES	- Total number of interactions completed by all logged-on users during an hour
12	2	Average remote-front end communication time		
13	48	Remote—front end communication time histogram data	AVG HOST RESPONSE	- Average time (seconds) to complete a host interaction
14	2	Remote—front end communication minimum time	% HOST SESS TIME	- Total host time of all the active users during an hour percentaged over the users' tube-connect time
15	2	Remote—front end communication maximum time		
16	94	Host interaction time histogram data		The host response time is broken into 5 histogram intervals. Each column consists of the number of host instances completed within the time interval and the relative percentage.
17	2	Host interaction maximum time		
18	3	Date		
19	3	Time of day		
20	3	User logon time		
21	3	User logoff time		
22	1	Number of times file open failures		
23	51	(Reserved for future use)		

APPENDIX C: DESCRIPTIONS OF FIELDS IN DGS-RESPONSE CUMULATIVE MANAGEMENT REPORT (see Figure 6)

Hr. of Day - Hourly interval of the day

No. of SESS INIT - Number of user logons on DGS using graphics application task

REFERENCES

1. Rao, J. R., W. F. Winters, and L. D. Schmidt. "Performance Evaluation of a Test Distributed Graphics System." *IEEE Computer Society's Third International Computer Software and Applications Conference (COMPSAC)*, November 1979, pp. 512-518.
2. Abrams, M. D., and S. Treu. "A Methodology for Interactive Computer Service Measurements." *Communications of the ACM*, (Vol. 20, No. 12), December 1977, pp. 936-944.
3. Abrams, M. D., I. W. Cotton, S. W. Watkins, R. Rosenthal, and D. E. Rippey. "The NBS Network Measurement System." *IEEE TR. on Communications*, (Vol. COM-25 No. 10), October 1977, pp. 1189-1198.
4. Mamrak, S. A., and M. D. Abrams. "A Taxonomy for Valid Test Workload Generation." *Computer IEEE Publ.*, December 1979, pp. 60-65.
5. *The Picture System 2 User's Manual*, Evans & Sutherland Computer Corp., Salt Lake City, Utah, 1976.
6. Veck, F. R., "User Written Ancilliary Control Processors." To be presented to 1980 Fall DECUS, U.S. Symposium, November 4-7, 1980, San Diego, California.
7. Loveland, R. A. and C. W. Stein. "How DECNET's Communications Software Works." *Data Communications*, January 1979, pp. 49-65.
8. *Terminals and Communications Handbook*, Digital Equipment Corporation, Maynard, Mass., 1979.

DATABASE SYSTEMS

Effective inference control mechanisms for securing statistical databases*

by VANGALUR S. ALAGAR, BERNARD BLANCHARD, and DAVID GLASER

Department of Computer Science
Concordia University

ABSTRACT

A database that provides statistical summaries for the purpose of research, planning, and decision making must remain rich, functionally useful, and protected from fraudulent usage. Unless restrictions are placed on the types of queries and responses, protecting an individual's information in the database is impossible. Some effective inference control mechanisms that make it extremely hard for any user to control and compromise a database are discussed. For each method it is argued why the method is effective, supported by some test results.

INTRODUCTION

A database with the main function of providing statistical summaries required in planning, decision making, and research will be called a statistical database. Medical information in hospitals, personal information held in centralized data banks, and sociological information collected, screened, and preserved by the Census Bureau are typical instances of statistical databases.

There are broadly two classes of users of a statistical database, of which one class of users will have authorization to read, write, and update data and the other class of users will have "read-only" access to data. The latter class of users may require extensive statistics on subsets of the database. The security problem considered here arises when unlimited "read-only" access is granted to the users seeking statistical summaries.

Any statistical database, regardless of its content and function, must remain rich and useful. Thus the data must be guarded against damage and fraudulent exposure. Protection mechanisms and encryption schemes³ are appropriate to enforce such safeguards. Recent studies^{6,8,12,18} have shown that there is a real and great threat of invasion of privacy through user inference. It is intuitively obvious that unrestricted and intensive on-line dialogue will lead to the isolation and exposure of any sensitive information in the database. The surprising and rather unexpected conclusion of this recent research

is that such inference will lead to fast compromise, even under restricted dialogues, and that the cost of compromise is little. Many questions are not satisfactorily resolved, however, and hence remain open for investigation. These are:

1. The minimum amount of restrictions (inference controls) that retain the integrity of the database and the usefulness (or accuracy) of the statistical summaries and yet guarantee security of data in the sense that no sequence of answers to user queries can be correlated to isolate and identify an individual's information in the database.
2. Proving that a strategy of response is secure, easy to implement, and adaptable to statistical databases of all sizes.

In this paper three strategies are discussed, of which one is old with a new approach and two are new. It is argued why making a statistical database secure against user inference is inherently difficult; however, we do not offer any rigorous proof for the complexity issue. In the following sections we give formal definitions, discuss complexity, review the recent research, and summarize our proposals. Detailed discussions on the proposals together with analyses of each will appear elsewhere.

BASIC CONCEPTS ON DATABASE MODEL AND SECURITY

We conceive of a statistical database as a single relation over a fixed number of attributes. A record R has k attributes and is defined to be a k -tuple (r_1, \dots, r_k) where r_i , the value in the i th field (or attribute), is derived from a domain D_i , $1 \leq i \leq k$. Some of the attributes are qualitative in nature and others are quantitative, and among them they may or may not uniquely identify a record. All the records are forced to have the same attributes and the same number of attributes. For example, the qualitative attributes can be *sex*, *status*, *age*, with their corresponding domains—male, female; professor, student, administrator; 20, 30, 40, 45. The quantitative domains usually derive values from real or integer numbers.

In general a subset of the attribute values will identify a subset, possibly empty, of records. One particular attribute, if

* This research is supported by the National Sciences and Research Council of Canada grant no. A3552.

retained as part of record occurrences, may uniquely identify a record. The existence of such a field may be a matter of fact to the system; however, when it is masked from the user's view and hence not made available for retrieval purposes, only subsets of attributes are to be used for retrieval. Depending on what the user is permitted or not permitted to see, one can classify the user queries.

A query will have two parts, that is, a qualification part and a target part. The qualification part specifies the extent of information the user has, and the target part indicates the extent of information the user demands from the system.

The complexity of a query depends on the number of attributes and the mode of specification of the value of each attribute. Since the qualification part in a query is a predicate expression, all predicate calculus operations can be extended. Moreover one or more values of one or more attributes may be related by boolean operators, the class of intersection queries. The response to such a query is in general a quantification of the subset of the database. This quantification is based on the specification in the target part. If the qualification part of a query is allowed to specify one or more keys, we call the query a "key-specified query;" if the qualification part of a query specifies a predicate calculus expression (characteristic formula), we call it an "attribute-specified query."

The decision to include or exclude a query type for any retrieval purpose is dictated more by the user environment, rather than by the inherent model. For example, in small databases it may be necessary to permit just key-specified queries, whereas in large databases only attribute-specified queries seem more natural, although each record may be uniquely identified by a single attribute.

Within each broad category of query types, one can further classify queries depending on the nature of the statistics permitted to be drawn out. A "count query" is an attribute-specified query that requests the number of records satisfying a predicate expression. A "sum (average, median) query" may be either attribute-specified or key-specified. In either case the query will specify a particular quantitative attribute. If the query is attribute specified, it requests the sum, average, or median of the data values occurring in the specified quantitative fields of records that satisfy the predicate expression specified. If the query is key specified, it requests the sum, average, or median of the data values in the specified quantitative fields of records that are uniquely identified by the keys given in the query.

The system must have a strategy of responding to each query of each type; the strategy will remain the same over each query type. We can define several levels of compromise, varying directly with the extent of inferred information, through a sequence of permissible queries under a strategy.

In key-based queries the existential problem of a record does not exist. However, the existential problem of the value in a quantitative attribute is very much existent. In attribute-based queries both types of problems exist. Let us start with the hypothesis that a subset of attributes do exist to uniquely identify a record. Under this hypothesis a user with attribute-based queries might infer one more qualification part or quantification part of a record. (He or she already knows a portion of a record.) We call this *partial local compromise*. When all

of the qualification and quantification parts have been inferred, the existential problem has been solved (totally inferred). We say that this is *total local compromise*. If through a sequence of queries, a user gathers a subset of records and achieves partial local compromise for each, we say that there is *weak global compromise*. We say the database is *strongly globally compromised* if every record is locally totally compromised. A strategy consists of a method of response to a predetermined set of queries within a particular query type. The effectiveness of a strategy should be measured in terms of its prevention of compromisability at one of the levels described as well as its ease of implementation.

COMPLEXITY OF DATABASE SECURITY

A statistical database system usually must admit only queries belonging to specific query types. Moreover suitable restrictions must be set on the number of keys to be specified in key-specified queries and the number of attributes specified in attribute-specified queries. The first restriction is necessary¹² but can be shown to be not sufficient for securing confidentiality in the database. The restriction on the number of attributes is suggested due to the explosion of the query set size. By query set we mean the set of permissible queries (not in the sense defined by Denning).⁵ For example, let us ignore *or* and *not* and consider only *and* as an allowed boolean operator in queries. Let there be k qualification attributes. Let j_1, j_2, \dots, j_k denote the cardinalities of the domains; that is, attribute A_i can specify any one or more of the j_i values or do not care to specify. Then for any query A_i can specify $j_i + 1$ values. The number of queries in the query set that contains queries specifying at most r out of k attributes is

$$\sum_{m=0}^r a_m, a_m = \sum \delta_{i_1} \delta_{i_2} \dots \delta_{i_m}$$

$$i \leq i_1 < i_2 \dots < i_m \dots < i_m \leq k$$

Example 1

Consider four qualification attributes A_1, A_2, A_3, A_4 , whose value sets are $\{v_{11} v_{12}\}$ for A_1 , $\{v_{21} v_{22} v_{23} v_{24}\}$ for A_2 , $\{v_{31} v_{32} v_{33} v_{34}\}$ for A_3 , and $\{v_{41} v_{42} v_{43}\}$ for A_4 . Assume each record has one value from each attribute. There are 96 records. However, the number of possible queries of the allowed type is

$$300 = \sum_0^4 a_m,$$

where

$$a_0 = 1, a_1 = 13, a_2 = b_2, a_3 = 128, \text{ and } a_4 = 96.$$

Even when we ignore the other boolean operators and quantification parts of attributes, we end up with a large number of queries. We are bound to increase the number of queries by admitting *or* and *not*. Thus permitting just one query type enables a user to construct a large number of queries within that type; the size of this set is much larger than the size of the database. If we imagine these queries spread suitably in an overlapping set of layers over a sequential record placement, then clearly the query set is a maximal covering for the records, and it is only necessary to identify a

set of layers that intersect pairwise at individual records to isolate and identify them. Thus the sheer size, availability, and unrestricted browsing gives the ability to compromise, thus making security a difficult task. We argue, therefore, that the type and number of attributes must be restricted without adversely affecting the statistical summaries. Most of the previous research was concerned only with limiting the size of response and proving that this strategy can be easily subverted.

Given a file F and a set of queries Q , does there exist a strategy S_p that safeguards the database to a maximum degree in the following sense: Let S_B be an opposing strategy that tries to infer through S_p . If it can be shown that finding an S_B that compromises in none of the levels described by us is impossible, then S_p is an effective strategy. It may be very hard or impossible to find such an S_p for a given Q . However, if it can be shown that finding an S_B for a local total compromise is hard, then the chances of global partial compromise would be low and global total compromise can be ruled out.

It was DeMillo, Dobkin, and Lipton⁴ who first pointed out the underlying principles of combinatorial inference and database security. Their observations lead to the conclusion that proving database security for a strategy may be very hard. However, if a strategy S_p can be built on a strong set of axioms similar to those in the theory of complexity, it may just be possible to prove that finding an S_B is hard, thereby establishing that S_p is safe until it is proven to be otherwise. There may just be an element of hazard in such an approach when complete security is to be virtually assured.

COMPROMISE VERSUS SECURITY

We will briefly review some of the recent research reported in the area of security of statistical databases. Most of the past research has been devoted to methods that systematically aim at breaking the security, rather than methods to effectively safeguard a statistical database. The notable exceptions to this are Yu and Chin²⁴, Denning¹¹, and Beck¹.

We can classify the studied methods on the basis of the inference controls, which are the strategies defending against compromise, as follows:

1. Set controls on the size of response; that is, when attribute-based count queries are asked, true count is given only when the count is neither small nor large.
2. Set controls on the overlaps of queries; that is, in key-specified queries a minimum amount of overlap between queries is enforced; in attribute-specified queries a minimum amount of overlap in response must be set.
3. Distort the responses to queries; that is, random perturbations are applied to true responses.
4. Give response by sampling from the database; that is, in response to attribute-specified queries, a random sample selects a subset of the true response and the result is computed on this random sample.

Kam and Ullman¹⁶ modeled a database as a collection of records with a key of k bits uniquely identifying each record. Every allowable query is a string over $(0,1,*)$, and clearly every p query (that has $k-p$ *s in it) will have a response of 2^{k-p} records. Thus any sum would be in a subgroup of 2^{k-p}

values. They have given necessary and sufficient conditions of compromisability. Such a model is too restrictive, however, to be realistic.

Chin² overcame the restriction by letting fewer than 2^k records in the database when each record in the database requires k bits for identification. This data model would permit attribute-specified queries, which can be transferred to a k -bit string. Thus the query type is a hybrid set. Any subset of attribute values will be transferred to an s -bit key. Thus the response to a count query that specifies attributes is a subset of the database. Chin's² strategy is to give responses if the count is ≥ 2 , that is, the exact count and the exact sum; however, if the count < 2 , no response is given. The very fact that no response is given will make the user uneasy and suspect in the usefulness of the database. Chin gives necessary and sufficient conditions for compromisability.

We find that even extending Chin's results for $m=3$ is not easy. However, we have the result: if $N \geq \frac{3}{4}(2^k)+1$, the database can be compromised when the response level is ≥ 3 . Consider the following databases:

D ₁	D ₂	D ₃	D ₄
000	000	000	001
001	001	010	010
010	011	100	100
110	110	101	101
111	111	111	110

Consider the set of queries:

$$Q = (q_1 = 1.., q_2 = .1., q_3 = ..1)$$

and the response to $COUNT [q_i : D_j] =$ the number of records satisfying q_i in database $D_j = 3, i = 1, 2, 3, j = 1, 2, 3, 4$. Since all D_j return the same count for each $q_i \in Q$, it is impossible to determine using q_i s with which database we are dealing. Even if we know the existence of one record (even up to four), we cannot find with certainty the remaining keys. In our case, however, one can systematically isolate and then regroup bit strings so that all four databases are constructed. In general for any k and response level m , there exists $f_m(k)$ such that if $N < f_m(k)$ and the strategy is to respond only if the response $\geq m$, there is no compromise. The existence of trackers and the restrictive nature of this database model make all these results unattractive.

A tracker⁸ is a "characteristic formula" or predicate expression built on the qualitative attributes. This formula can be padded to any query to force a response from the system. The tracker itself can be posed as a query and will be responded to because the tracker is built to circumvent any restrictions imposed on the response level. Not only do trackers exist, but they are also easy and inexpensive to construct⁹. Thus no amount of controls on the size of response will secure a database under attribute-specified queries.

When attribute-specified queries are to be permitted, the only alternate scheme is to set controls on the type of response (and not on the size). One can modify the true answer by introducing an element of uncertainty. The type and extent of uncertainty ultimately depends on the query type, whereas

the effectiveness of such a strategy must be tested to prove total randomization.

One approach is known as pseudo-random rounding; that is, the measure of uncertainty is a function of the attributes characterizing the record so that the same query returns with the same response. Such methods can be effective most of the time, although if symmetric errors (rounding up/rounding down) are introduced the uncertainty can be removed through standard statistical techniques.

The second approach is to apply queries to a random subfile of the database. This is known as random sampling. (See the recent paper of Denning¹¹. It has been shown that random sampling is reasonably effective in preventing compromise and is inexpensive to implement.

An effective method of preventing a user from compromising a record by a sequence of queries is to partition the entire database into a number of groups of small size²⁴. The response to a query is based on the groups to which the characteristics refer and not to individual records in the groups. The effectiveness of this strategy critically depends on the size and nature of each group. Ill-formed and large groups will affect the statistics. Although Yu and Chin²⁴ proposed this four years ago, not enough results have been reported on its performance. The algorithm as reported by Yu and Chin is both inefficient and ineffective. Our recent study shows that there are faster methods of partitioning the result into nicer groups, which will hence be effective. (See the following section of this paper for details.)

Another strategy that has been studied is to control the extent of overlap in key-specified queries or the extent of overlap in response to attribute-specified queries. In the case of key-specified queries, two restrictions are imposed. (1) The number of keys per query is the same. (2) Any two queries should not overlap in more than a predetermined number of positions.

The results reported by Dobkin, Jones, and Lipton¹² confirm that when exact answers are given to queries, even when we restrict the overlap, it is relatively easy to compromise in the sense that only a few queries are required. If we can keep track of this sequence of queries, however, and forbid one or more of them, compromise may be avoided. But it is extremely difficult and may be impossible to monitor the queries of users for at least two reasons: (1) A user can query at various times correlating the results at the end. (2) Several people can collaborate and query at various times, correlating their final responses.

Schwartz, Denning, and Denning²³ studied linear queries further. Their strategy has been to give a weighted sum of values corresponding to the keys specified in an average or sum-seeking query. If k keys are permitted in a query, they associate k weights that are independent of the actual keys that may be specified in a query. The immediate consequence is that the responses to queries in which the same set of keys appear in different order will be different. We believe this is too misleading and very simplistic. It seems that an appropriate way would be to associate weights with keys so that responses to queries, regardless of the order of specification of keys, would be the same at all times; but then the problem of fixing the weights, as we shall discuss in the next section, is not that easy.

INFERENCE CONTROL PROPOSALS

There are three methods that can effectively control inference and prevent compromise. The first method discusses a protection strategy when key-specified linear queries are permitted in the database. We will show how a maximum subset of such queries can be given true responses in a very stringent environment; we will show why random linear combinations of values is the only solution to enhance security when certain conditions are relaxed. To preserve the accuracy of statistics and to assure security, the random weights must be associated with keys and these weights must be kept secret. The other two methods are applicable to attribute-specified queries. One discusses a partitioning strategy in a different way than Yu and Chin did; the other method proposes that in response to count queries, an interval containing the true count be given and in response to sum-seeking queries, the true value be given. We will comment on the effectiveness of these methods.

Key-Specified Linear Queries

The results of Dobkin, Jones and Lipton¹² confirm beyond doubt that any database that permits key-specified queries cannot afford to return true values to queries, since in such a strategy only a small number of queries are required to compromise the database. Not all forms of lying can ensure security⁴. The main aim of this section is to demonstrate strategies in each of which the error in any response can be controlled and kept to a predetermined level, whereas the error in any inferred value will be unpredictable.

Queries involve exactly t keys, $t > 2$, and we permit average seeking queries. We will determine $\alpha_1, \dots, \alpha_N, \alpha_i$ for the i th record (hence for the i th key) once and for all. The response to a query that specifies keys k_1, \dots, k_t is

$$R(q) = (\alpha_1 v_1 + \dots + \alpha_t v_t) / (\alpha_1 + \dots + \alpha_t),$$

where v_i is the value associated with the i th record.

The following algorithm determines the α s:

Algorithm K1

1. Order the records such that $v_1 < v_2 < \dots < v_N$.
2. Divide the database into $k + 1$ groups, as evenly as possible. Let G_1, G_2, \dots, G_{k+1} be the groups.
3. Find a group representative for each group. Let $\bar{v}_1, \bar{v}_2, \dots, \bar{v}_{k+1}$ be the representatives.
4. Determine $\alpha_{G_1}, \alpha_{G_2}, \dots, \alpha_{G_{k+1}}$, using the group representatives (see the discussion later).
5. Determine individual α s for each record as follows:
do the following steps for $i := 1$ to $k + 1$
for each record j in group G_i do

$$\begin{aligned} \alpha_j &\leftarrow \alpha_{G_i} - d_j / s, \\ \text{where } d_j &\leftarrow v_j - \bar{v}_i \\ s &\leftarrow \sqrt{\sum d_j^2} \end{aligned}$$

To complete the algorithm, it remains only to explain Step 4. Given a set of $k + 1$ values, we wish to determine the α s, one for each value, such that: (1) For any query q that actually

requests the average of any k of these $k + 1$ values, the response $R(q)$ differs from the true value $T(q)$ (true average) by at most a predetermined limit. And (2) the individual values that may be inferred by solving a system of linear equations will differ considerably from the true individual values.

To optimize the error in the inferred values, the region of optimization is the convex polytope determined by the set of linear inequalities imposed by Condition 1. It is well-known that the extreme value of a function in this convex region must be attained only on the boundary, that is, must be at one of the vertices of the polytope determined by the linear inequalities. One may simply choose any one vertex or the vertex that minimizes the maximum error so that the choice of these α s assures a maximum error in the inferred values.

We comment here that the set of α s cannot be inferred unless one knows the strategy and exhaustively examines all choices of α s corresponding to the boundary of the polytope. Thus even if the strategy is known, it is hard to know the group α s. Moreover, in Step 5 of Algorithm K1, the individual α s can be computed as a function of the true values. Thus the α s can be compromised if and only if the v s are compromised. That is, it is as hard to compromise the v s as it is hard to compromise α s. In turn, this is much harder than finding the group α s. Since the number of faces and vertices of a convex polytope increases with the number of independent linear inequalities, it is reasonably hard to find the set of α s chosen by this strategy.

There is one drawback in this strategy. For queries involving records in the same group, we have no control on the error introduced in Step 6 of Algorithm K1. The next strategy is an attempt to remedy this situation.

The response to a query that specifies keys k_1, \dots, k_t is

$$R(q) = (\alpha_1 v_1 + \dots + \alpha_t v_t) / (\alpha_1 + \dots + \alpha_t),$$

where v_i is the value associated with the i th record. However, the α s here are determined dynamically; that is, the α s are computed for each query and thus are not fixed. The main deviation of Algorithm K2, below, from Algorithm K1 is in the way the groups are formed and not in how the α s are computed once group representatives are chosen. Assume $t = 2$ for the following description; the generalization for $t > 2$ is easy.

Algorithm K2

1. Sort the records such that $v_1 < v_2 < \dots < v_N$.
2. Initialize the group boundaries. Set $m = 1$, $mm = N$ (all records are in one group)
3. Read a query $q(k_i, k_j)$ (this requests the average of values v_i and v_j associated with k_i and k_j).
4. Divide database G into three groups.
5. If v_i and v_j are in the same group G then do:
 - a. set $m =$ lower boundary of G
 - b. set $mm =$ upper boundary of G .
 Else execute Step 7.
6. If the size of group G is less than 3, extend one of its boundaries by 1. Execute from Step 4.
7. (Now we have $t + 1 = 3$ groups such that v_i and v_j are not in the same group.)
Do Steps 3 to 5 of Algorithm K1.

If the group representatives are suitably chosen in this method, then we have complete control over the extent of error introduced in responses, and our test results reveal that most of the inferred values have error greater than the maximum error allowed in the query responses. We defer discussions on complete error analysis, since it may be out of place here.

An alternate strategy to successfully eliminate compromise through linear queries will be outlined later. The main thrust in this strategy is to identify a smallest subset of queries without which compromise is impossible. We will call this subset a *forbidden query set*. If a user's query is a member of this forbidden query set, a perturbation (either to the true result or by means of random convex linear combinations) will be introduced in the response. However, if a query is not a member of the forbidden query set, the true answer is given. To assure security the query type must be restricted to contain queries of size t (t keys per query) and exactly $t - 1$ elements are common between any two queries. We will not discuss here the algorithm that forms a forbidden query set but will explain the strategy through an example.

Example 2

Let $N = 7$ (the number of records)
 $t = 3$ (number of keys per query)

$$Q = \{q \mid |q| = 3\}. |Q| = \binom{7}{3} = 35.$$

Without loss of generality let us denote the keys of records by $\{1, 2, \dots, 7\}$ and the associated values $\{v_1, \dots, v_7\}$. There are 15 queries each specifying the first key 1. If we remove from this set the five queries

$$\{(1\ 2\ 3), (1\ 2\ 7), (1\ 3\ 6), (1\ 4\ 5), (1\ 6\ 7)\},$$

we can not solve for the value v_1 by asking the 10 other queries. In general consider the set

$$F = \{(1\ 2\ 3), (1\ 2\ 7), (1\ 3\ 6), (1\ 4\ 5), \\ (1\ 6\ 7), (2\ 3\ 6), (2\ 4\ 5), (2\ 4\ 6), \\ (2\ 5\ 6), (3\ 4\ 5), (3\ 4\ 7), (3\ 5\ 7) \\ (4\ 5\ 6)\}$$

If these 13 queries are forbidden from Q , the database cannot be compromised under the restrictions on permissible queries.

In general for any N and t , if we allow only queries that overlap in exactly $t - 1$ places, a forbidden query set of minimal size can be chosen and the strategy would be not to answer a query if it is a member of this set. For moderately large values of N and even small values of t , the size of a minimal forbidden query set will be very large. Thus the decision to totally disallow queries from this forbidden set will adversely affect the usefulness of the database. An attractive proposition would be to give true responses for every query not in the forbidden query set and to lie for every query in the forbidden query set. The following comments summarize the complexity and effectiveness of this approach:

1. The proportion $|F|/|Q|$ is at least $1/t + 1$. The lower bound remains independent of N ; the size of the data-

base and the lower bound is achievable for several values of N and t .

2. Finding a set F of minimal size is hard.
3. If a set F is found for which $|F|$ is 10% to 15% above the lower bound, the number of instances of lying is close to the minimum.
4. By controlling the extent of lying, it is possible to predict the error in the inferred value of any individual record.
5. Since minimal F is hard to find and there may exist several F , it is difficult to identify and distinguish queries on which a lie is given from those on which true response is given.
6. If we relax the restriction on the overlap to "at most $t - 1$," then our strategy fails to protect the database. Under this modified restriction we can show that the only strategy that protects a database is lying on every query.

Attribute-Specified Queries

Random sampling¹¹ and partitioning²⁴ are effective methods that prevent a user from isolating an individual's information. In random sampling the response to a query is computed from a random subset of the records in the database. When the database is large, the summaries obtained this way may be statistically significant; in small or medium size databases, however, the statistics may not be significant. Although partitioning a database into several small groups and restricting the response to groups effectively secures the database, it has been generally believed that partitioning fragments the database to the extent that it may become functionally useless. Partitioning methods have not been tested thoroughly enough, however, to be disregarded as useless.

In this section we study two strategies. One of them is partitioning and the other is a form of approximate response to queries.

An attribute-specified query may require two types of statistics: the count and the value. When the qualitative part of a query specifies a predicate P and a quantitative attribute v , the required response is either the number of records (count) for which P is true or the sum of the values in the v field of these records, depending on the specification in the target part of the query.

Range response to count queries

Our strategy is to give a range, instead of the true count, for count queries and exact sums for value queries. By giving a range that includes the true count, we are not misleading the user. Since exact values are given to queries that request sums of data values, the richness and usefulness of statistical summaries are maintained.

For any predicate P , let X_p denote the subset of records such that for every record in X_p , P is true. Let $n_p = |X_p|$. We consider a sequence of non-overlapping intervals of fixed size s each, that is,

$$(0, s - 1), (s, 2s - 1), (2s, 3s - 1), \dots$$

Formally stated, our strategy of response is as follows:

$$\text{Count}(P) = (a, b), a \leq n_p \leq b, a = (i - 1)s, b = is - 1.$$

$$\text{Sum}(P, V) = \begin{cases} \text{undefined if } a = 0, b = s - 1 \\ \sum v_i, i \in X_p \text{ otherwise} \end{cases}$$

where v_i is the value of the data field v in the i th record.

Thus for all queries the exact interval containing the exact count is given. There are two major reasons why Sum is undefined when $0 \leq n_p \leq s - 1$. The first reason is that the user can easily infer and force a local compromise. For example, let the response to $\text{Sum}(P, V)$ be given as T when $0 \leq n_p \leq s - 1$. If $T = 0$ then it readily follows that $n_p = 0$, and if $T > 0$ then n_p lies in the range $(1, s - 1)$. The other reason is that if a user knows from sources external to the database that P uniquely identifies an individual, a response to Sum query will reveal the data value corresponding to the individual and then there is local total compromise.

Under this strategy it can be shown that no individual's information can be identified with certainty. There exists a small element of hazard that might lead to total local compromise; such an instance arises only when a clever sequence of querying and manipulation of responses lead to the reduction of a particular range to a single point. Our investigation shows that in the majority of cases a range cannot be reduced; in a small number of cases, a range can be reduced to a single point. We will simply state a theorem without proof on the reducibility of an interval to a point.

Theorem 1 Let each attribute derive values from a domain of size t , and all these t values are equally likely to be specified in a query. Let the count of this query belong to the interval $[(i - 1)s, is - 1]$. Let $P(i, t)$ denote the probability that this interval can be reduced. Then

$$P(i, t) = \frac{x}{x + y}, \text{ where}$$

$$x = \binom{i - 1}{t - 1} \binom{s}{t}$$

$$y = \binom{t + s - 1}{t} \left[\binom{t + i - 1}{t} - \binom{i}{t} \right]$$

and $\binom{c}{d}$ is the binomial coefficient.

Since $P(i, t) = 0$ if $i < t$ or $s < t$, it follows that no reduction is possible in these cases. Reduction to a single point is possible only when $t = s$, and in all other cases a range can be reduced by at most $t - 1$. For example, if $s = 3$, $t = 3$, and $i = 5$, the probability of reduction is .0234.

No general tracker can possibly be constructed to reduce more ranges than are already reducible to points. The number of cases in which reduction in range is possible decreases as the number of values per attribute increases. Moreover when the number of values per attribute is large, the amount of work required per one reduction is also large. Thus in large databases reductions to single points would be really rare, and hence the level of security is high.

Partition and protect

Our next strategy can be best described as Partition and Protect. Although Yu and Chin²⁴ first proposed this strategy, the method that they suggested for partitioning seems to produce ill-defined groups. Let us first review their method and then propose our modifications.

Assume that there are k attributes per record, and the number of different values of the i th attribute from a domain is d_i . Construct a k -dimensional matrix of size d_1, d_2, \dots, d_k . Any record R with values (r_1, \dots, r_k) can be mapped onto some cell c_j in the matrix, such that the coordinates of c_j match the k -tuple describing the record R . After all records have been mapped in this fashion, the cells with their weights (or frequencies equal to the number of records in the cells) are a partition of the database, though perhaps not an acceptable partition. If $t > 0$ is a threshold, Yu and Chin require reorganization of this primitive partitioning into an acceptable partitioning wherein each non-empty cell has at least t records.

The cells that are empty can be ignored; those non-empty cells that have fewer than t records must be merged somehow with neighboring cells to create an acceptable partition. The merging procedure suggested (but not implemented) by Yu and Chin is to merge a non-empty cell having fewer than t records with one of its neighboring cells by combining all the cells with those two adjacent values in that domain. This procedure is repeated until the cell has at least t records. To fix ideas, let us consider an example in two dimensions; that is, there are just two attributes per record. Let z_1, \dots, z_m and y_1, \dots, y_n denote the set of values in the domains from which the fields derive their values. Thus we start with a two dimensional matrix B of size nm (there are nm cells). Let $B(i, j)$ contain all records (x, y) such that $x = z_i$ and $y = y_j$. If this region is non-empty and has fewer than t records, Yu and Chin's algorithm merges this region with its neighbors in the arbitrary order $B(i, j + 1)$, $B(i + 1, j)$, $B(i, j - 1)$, and $B(i - 1, j)$, until the resulting region will have at least t records. Suppose $B(i, j)$ is merged with $B(i, j + 1)$ and $B(i + 1, j)$, then the new domains will be $D_1 = \{z_1, \dots, z_i U z_{i+1}, \dots, z_m\}$ and $D_2 = \{y_1, \dots, y_j U y_{j+1}, \dots, y_n\}$. See Figure 1.

When carried to completion, this method makes many unnecessary combinations of cells, resulting in partitions wherein the number of non-empty cells will be small and the size of

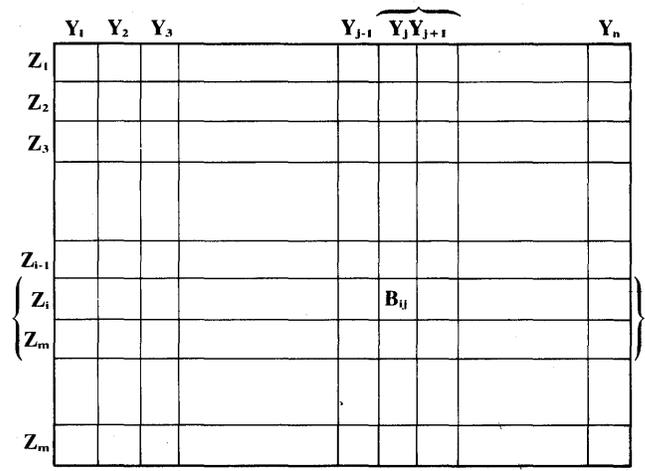


Figure 1—Merging B_{ij} with its neighbors $B_{i,j+1}$ and $B_{i+1,j}$

each cell will be much greater than t . As a consequence the database would have virtually collapsed in its usefulness. See Figure 2.

We have designed an algorithm that partitions a database into groups so that each group has at least t records. If the database is mapped onto a k -dimensional matrix, our algorithm obtains a disjoint set of rectangular regions (rectangular parallelepipeds for $k > 2$) that cover all the non-empty cells of the matrix, such that: (1) Each region has at least t records. (2) The number of rectangular regions obtained is maximum. And (3) the difference between the area (volume) covered by the non-empty cells in the original configuration and the sum of the areas (volumes) of the covering rectangles is minimum. See Table I for selected test results.

In general, obtaining an exact solution subject to the above requirements is hard, in the sense that any algorithm obtaining such an optimal covering will invariably be required to do an exponential amount of work proportional to the input size. What we have designed is an "approximate algorithm" that produces a "nearly optimal" partitioning in time $O(x^2 \log_2 x)$ for most of the input and in time $O(x^3)$ for some rare kind of input database in which the distribution of values in the do-

TABLE I—Selected test results of our partitioning method

d_1	d_2	N	t	x	N/t	Nx	$N/d_1 d_2$	% area covered	$X/N/t$
5	5	25	3	7	7	3.57	1.0	72	100
8	8	25	3	6	8	4.17	.391	59.4	75
10	10	25	3	7	8	3.57	.25	53	87
5	5	50	5	8	8	6.25	2.0	96	100
10	5	50	5	9	9	5.56	1.0	90	100
10	15	50	5	8	10	6.25	.33	76.7	80
20	30	50	5	8	10	6.25	.083	70	80
8	8	100	5	15	19	6.67	1.562	89.1	78.9
10	15	100	5	16	20	6.25	.667	83.3	80
24	24	100	5	17	20	5.88	.174	63.9	85
15	15	500	5	80	100	6.25	2.22	94.7	80
50	50	500	5	84	100	5.95	.2	68.3	84

Figure 2—Merging a sample database by Yu and Chin's method. (Arrows indicate which cells are being merged. Brackets indicate which values are being merged. Each record is represented by an x. For this example, $k = 2$, $t = 2$, $N = 15$, $d_1 = 4$ and $d_2 = 5$.)

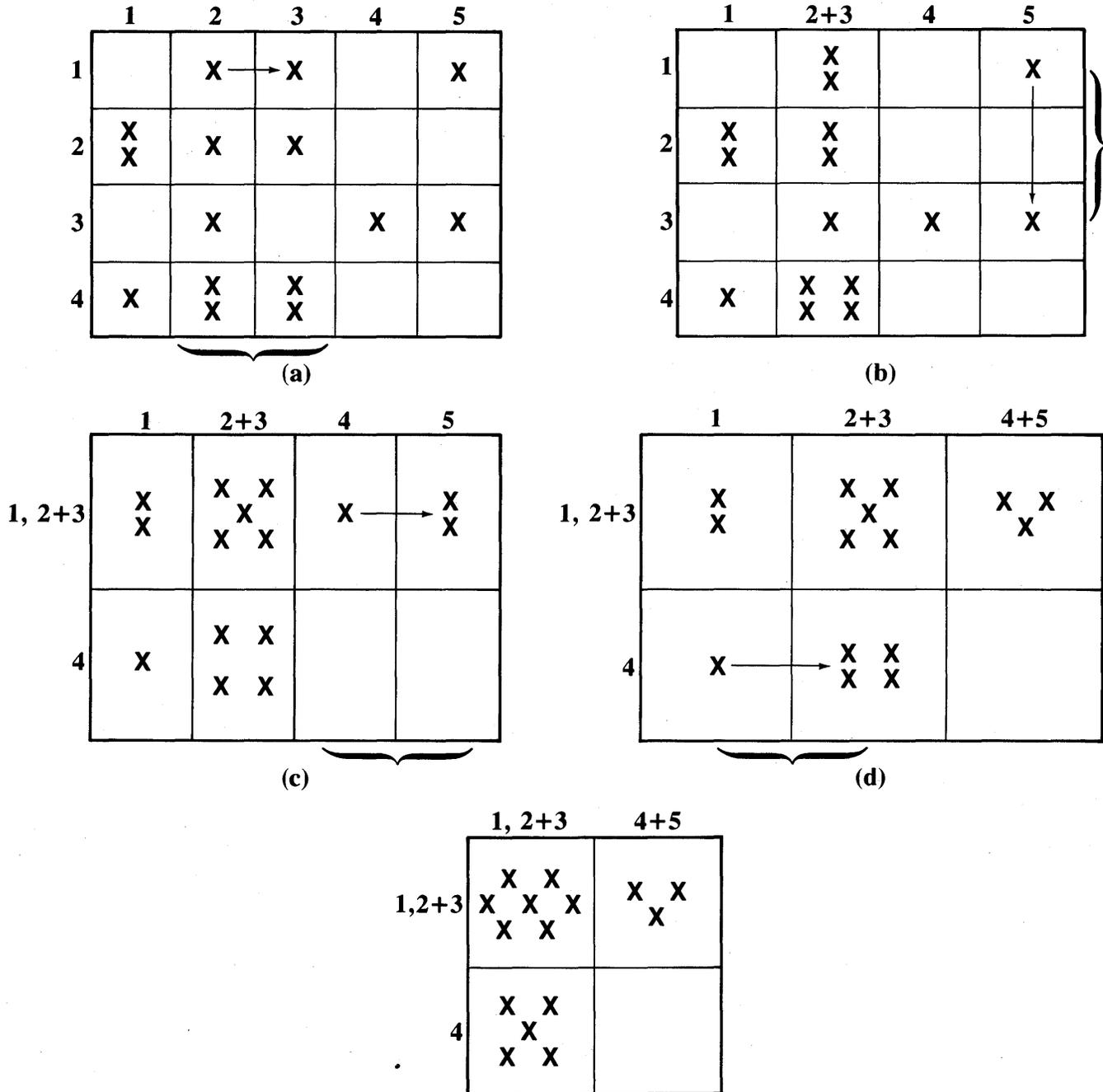


TABLE II—Comparison of the performance of three methods

Method	d_1	d_2	N	t	x	N/x	N/d_1d_2	$\frac{\% \text{area covered}}{d_1d_2}$	$x/N/t$
Yu and Chin	6	6	25	3	5	5.0	.694	91.7	62.5
Alagar, Blanchard, and Glazer	6	6	25	3	7	3.57	.694	75	87.5
Optimal	6	6	25	3	8	3.125	.694	63.9	100

Figure 3—Comparison of three methods for partitioning a sample database.
 (The thick lines indicate the boundaries of the partition.)

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

a. Unpartitioned database

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6		X X		X		

b. Yu and Chin's method

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

c. Our method

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

d. Optimal partitioning

(The thick lines indicate the boundaries of the partition.)

mains of records is concentrated, where x is the number of groups formed.

The algorithm and its analysis are too detailed to be described here. See Figure 3 and Table II, which compares Yu and Chin's method, our algorithm, and an optimal algorithm on one input. Our preliminary results show that our algorithm does indeed produce nearly optimal partitioning almost all of the time. The groups that are produced by our method seem more homogeneous, and hence we expect the statistics produced to be more relevant. One of the important features of our algorithm is that the time complexity remains independent of k , the dimensionality of the matrix (the number of attributes), and is thus superior to Yu and Chin's method both in speed and performance. It is needless to say that compromise is impossible if the strategy of response is to base statistics on groups and not individual records relevant to a query. This strategy is appropriate only for static databases. In fact, any scheme modeled on a matrix type is not suited for dynamically changing databases.

CONCLUDING REMARKS

We have discussed effective methods to prevent compromise in a statistical database. In each method we have pointed out the reasons why the method should prove effective. We have also commented on the complexity of any opposing strategy that aims at a level of compromise in a database guarded by a strategy; the effectiveness of the guarding strategy is directly related to the complexity of creating an opposing strategy. We have also indicated how and why the statistical summaries produced by these methods are not unduly affected by our complex strategies.

REFERENCES

1. Beck, L.L. A security mechanism for statistical databases. *ACM Trans. Database Syst.* 5,3 (Sept. 1980), 316-338.
2. Chin, F.Y. Security in statistical databases for queries with small counts. *ACM Trans. Database Syst.* 3,1 (March 1978), 92-104.
3. Davida, G.I., et al. Database security. *IEEE Trans. Softw. Eng.* SE-4,6 (Nov. 1978), 531-553.
4. DeMillo, R.A., Dobkin, D., and Lipton, R.J. Even data bases that lie can be compromised. *IEEE Trans. Softw. Eng.* SE-4,2 (Jan. 1978), 73-75.
5. Denning, D.E. A review of research on statistical database security. In *Foundations of Secure Computation*, R.A. DeMillo et al. Eds. Academic, New York, 1978.
6. Denning, D.E. Are statistical data bases secure? *Proc. AFIPS 1978 NCC*, Vol 47, AFIPS Press, Arlington, Va., pp. 525-530.
7. Denning, D.E. and Denning, P.J. Data security. *Comput. Surv.* 11,3 (Sept. 1979), 227-249.
8. Denning, D.E., Denning P.J., and Schwartz, M.D. The tracker: A threat to statistical database security. *ACM Trans. database Syst.* 4,1 (March 1979), 76-96.
9. Denning, D.E., and Schlorer, J. A fast procedure for finding a tracker in a statistical database. *ACM Trans. Database Syst.* 5,1 (March 1980), 88-102.
10. Denning, D.E. Complexity results relating to statistical confidentiality. *Computer Science and Statistics: 12th Ann. Symp. Interface*, Waterloo, Canada, May 1979, pp. 252-256.
11. Denning, D.E. Secure statistical databases with random sample queries. *ACM Trans. Database Syst.* 5,3 (Sept. 1980) 291-315.
12. Dobkin, D., Jones, A.K., and Lipton, R.J. Secure databases: Protection against user influence. *ACM Trans. Database Syst.* 4,1 (March 1979) 97-106.
13. Fellegi, I.P., and Phillips, J.L. Statistical confidentiality: Some theory and applications to data dissemination. *Ann. Econ. Soc. Meas.* 3,2 (April 1974), 399-409.
14. Haq, M.I. On safeguarding statistical disclosure by giving approximate answers to queries. *Int. Computing Symp.*, 1977, pp. 491-495.
15. Hoffman, L.J., and Miller, W.F. Getting a personal dossier from a statistical data bank. *Datamation* 16,5 (May 1970), 74-75.
16. Kam, J.B. and Ullman, J.D. A model of statistical databases and their security. *ACM Trans. Database Syst.* 2,1 (March 1977), 1-10.
17. Nargundkar, M.S., and Saveland, W. Random rounding to prevent statistical disclosure. *Proc. Am. Stat. Assoc., Soc. Stat. Sect.* (1972), 382-385.
18. Reiss, S.B. Medians and database security. In *Foundations of Secure Computation*, R.A. DeMillo et al., Eds. Academic, New York, 1978.
19. Schlorer, J. Identification and retrieval of personal records from a statistical data bank. *Methods Inform. Med.* 14, 1 (Jan. 1975), 7-13.
20. Schlorer, J. Disclosure from statistical databases: Quantitative aspects of trackers. *Inst. Medizinische Statistik und Dokumentation, Univ. Giessen, Giessen, W. Germany, Mar. 1979. To appear in ACM Trans. Database Syst.*
21. Schlorer, J. Statistical database security: Some recent results. *Inst. Medizinische Statistik und Dokumentation, Univ. Giessen, Giessen, W. Germany, 1979. Presented at Medical Informatics, Berlin, 1979.*
22. Schwartz, M.D., Denning, D.E., and Denning, P.J. Securing data bases under linear queries. *Proc. IFIP Congress 77, North-Holland, Amsterdam, 1977*, pp. 395-398.
23. Schwartz, M.D., Denning, D.E., and Denning, P.J. Linear queries in statistical databases. *ACM Trans. Database Syst.* 4,2 (June 1979), 156-167.
24. Yu, C.T., and Chin, F.Y. A study on the protection of statistical data bases. *ACM SIGMOD Int. Conf. Management of Data, 1977*, pp. 169-181.

Using partitioned databases for statistical data analysis*

by RUVEN BROOKS

*University of Texas Medical Branch
Galveston, Texas*

MEERA BLATTNER

*University of California, Davis, and
Lawrence Livermore National Laboratory
Livermore, California*

ZDZISLAW PAWLAK

*Polish Academy of Sciences
Warsaw, Poland*

and

EAMON BARRETT

*Jaycor Corporation
Washington, D.C.*

INTRODUCTION

The statistical analysis of scientific data is a process that can be viewed as consisting of three fundamental phases. First, the observations are recorded. Next, they are encoded into a numeric form suitable for statistical analysis. Finally, the calculations are performed for the particular type of analysis needed for the design of the study. This ordering is, however, only conceptual; in most real studies, the three phases interact and are overlapped. Thus, it may be the case that a preliminary analysis run indicates that a more refined coding scheme is needed or that the coding process reveals deficiencies in the data collection.

A consequence of this interaction is that the data analysis calculations are done repeatedly over time using new coding schemes, new variables, and new selections of grouping of the cases. We propose the structure of a data organization that can result in substantial savings in the amount of calculation needed across these repeated calculations.

OPERATIONS NEEDED IN STATISTICAL ANALYSIS

Case Structure

For most statistical analyses, the data is structured in terms of a row vector consisting of the information relating to a particular observational unit. Typical observational units

might include a single individual in a psychological study, a single experimental animal in a medical study, or a single household in a sociological survey. The information recorded in each case can be a mixture of classification variables which indicate to which experimental group the particular case belongs and measurement variables which are the values of measurements for that particular case. (Note that it is possible for some variables to serve both functions.) The set of cases for any particular study or experiment can then be viewed as a rectangular matrix with each case being a row and all the values across cases for a given variable occurring in a single column.

User Operations

In preparing data for statistical analysis and in the course of performing the analyses, users of statistical packages typically perform the following operations:

1. Addition and deletions of cases (row addition and deletion).
2. Updating of variable values within individual cases.
3. Addition and deletion of variables across cases (column addition and deletion).
4. Variablewise transformations, such as multiplying a variable by a constant, adding two variables together to create a new one, etc. Such transformations may also involve conditional operations.
5. Value replacement (recoding) in which all occurrences

*This research was supported in part by NSF Grant No. MCS 77-02470.

of specific values for a particular variable are replaced by new values. This may involve collapsing several values into a single value.

6. Case selection on the basis of predicates applied to variable values. This operation may be used to designate subsets for particular analyses or for the application of variable transformations and recodings.

Statistical Operations

While the variety of possible statistical analyses is extremely large, nearly all of the commonly used analyses begin with a common set of calculations. If X is an $m \times k$ data matrix in which rows are cases and columns are variables, then these calculations can be specified as follows:

1. On the basis of values of the classifier variables, partition the rows of the matrix into sets which are not necessarily disjoint. Let n be the number of rows in a set.
2. For each set, form the following sums:
 - a. For each of the j variables (columns), sum x_i , $i = 1 \dots n$, where n is the number of rows in the partition.
 - b. For all pairs of variables j and k , sum $x_i * x_j$, $i = 1, \dots, n$. (Note that $j = k$ is included.)

These sums form the basis for further calculations in analysis of variance, analysis of covariance, multiple regression analysis, discriminant analysis, factor analysis, partial correlation, canonical correlation, and multivariate analysis. They are also useful for the calculation of descriptive statistics, such as mean and standard deviation.

In conventional statistical systems such as SPSS¹ and BMD², these quantities are calculated *de novo* for each analysis, an extremely costly process if the amount of data is large. The data organization we propose here is intended to save this cost by updating and preserving these quantities across user manipulations.

BASIS FOR THE PROPOSED ORGANIZATION

Partitioned Databases

The organization is based on the work of Z. Pawlak and colleagues (Marek and Pawlak³, Lipski and Marek⁴, Lipski⁵) on partitioned databases. In their terminology, a database consists of a set of objects which have attributes. If our objects are patients participating in a research study, then the patients have certain attributes of interest for data analysis purposes. Typical attributes might include sex, age group, weight, blood pressure, amount of cigarettes smoked, region of the country, etc. A case for analysis purposes then consists of the values of these attributes for some particular object.

In a partitioned database, those attributes which will be used to identify and select cases from the database are designated as *selection* attributes. Typically, though not necessarily, selection attributes have a small number of possible values and are referred to as "nominal" or "categorical" variables.

The values of selection attributes are considered to be divided into disjoint descriptors so, for example, sex is divided into male and female, age is divided into ranges of years, etc. Each object must have a value for each selection attribute, though certain values might be used to indicate "unknown" or "data missing." The attributes will be referred to by capital (subscripted or unsubscripted) letters and the descriptors will be referred to by small letters. The selection attributes are ordered in some way so that the list of attributes is A_1, A_2, \dots, A_n . Since each attribute has a list of descriptors, a, b, c, \dots, m (we need not have the same number of descriptors for each attribute), the description of an object may be made by a vector (a_1, a_2, \dots, a_n) where a_i is a descriptor in the set of attributes of A_i . We will ordinarily use the designation $a_1 a_2 \dots a_n$ instead of (a_1, a_2, \dots, a_n) . The A_1, A_2, \dots, A_n are a coordinate system for class descriptors. Observe that each object falls into one and only one class, hence the name *partitioned database*.

An example will illustrate this notation. Suppose that the following selection attributes have the following values:

```
sex
  a. male
  b. female
age range
  a. 20-30
  b. 31-40
  c. 41-50
  d. over 50
region
  a. north
  b. east
  c. south
  d. west
```

The class of all women 41-50 years old and living in the north would be designated by *bca*; there would be a total of 32 possible classes.

The proposed data organization is based on specifying selection attributes which define groupings of cases that are important for analysis purposes and storing with the classes that these attributes define the quantities described earlier. Performing a statistical analysis would then require forming a query which specified the classes that were required, retrieving the summary quantities stored with the class, and performing the necessary statistical calculations on these quantities. For example, if sex and age group had been specified as the only selection attributes, and an analysis of variance were to be performed on blood pressure data for cells defined by sex and age group, $\sum(x)$ and $\sum(x^2)$ would be retrieved for each of the eight classes of cases that sex and age group define. To collapse across these classes, the sums for the individual classes need only to be added together.

A Query Language for Partitioned Databases

In order to specify which classes of objects are to be used for a statistical analysis, a query language is needed. A formal syntax for a formal language for partitioned databases was thoroughly and rigorously described by Lipski and Marek⁴. The query language described in this section is not the same

one described by Lipski⁵, however, much of our reasoning follows his proofs closely. Our *query language* is called Q.

Definition: An *alphabet* for Q is:

1. The set of lower case Latin letters, L.
2. The set of lower case Latin letters with a superscripted c, L^c.
3. The set of lower case Latin letters with superscripts g, L^g.
4. The set of lower case Latin letters with superscripts 1, L¹.
5. The symbol, @.

Definition: If the *descriptors* D(B) of attribute B are (a,b,c,d), the *complement* of x, x^c, are those descriptors, D(B) - {x}. The set of descriptors *greater than* x, x^g, are those elements of D(B) that appear after x in the linear ordering of the descriptors (for convenience, we will assume that the descriptors of an attribute always begin with a and continue sequentially through the alphabet) and the set of descriptors *less than* x, x¹, are those elements of D(B) that come before x. @ is a "don't care" symbol, used to indicate that an attribute is not of interest for a particular query.

Definition: A *simple term* is a concatenation of n symbols from Q, where n is the number of attributes. A *constant simple term* is a concatenation of n symbols from L while a *variable simple term* contains an occurrence of a symbol not in L. Note that a simple constant term is also a class description as long as t = a₁ a₂ . . . a_n and each a_i is in the set of descriptors of A_i.

A *term* is defined recursively as:

1. A simple term.
2. If t and s are terms, then t, t + s, t*s, t→s are terms. If we wish we may define a set of formulas over Q that use the equality sign as well as the above Boolean operations and introduce the symbols, TRUE and FALSE. We will omit this part of the query language from the current discussion, since formulas do not increase the complexity of the implementation procedures as presented here. The *value of a term* v(t) is as follows:
 1. If t = @@@@...@, then v(t) is the set of all objects X. If t = e, then v(t) = φ.
 2. The value of @@...@a@...@, where a is in the ith position, is the set of all objects that have descriptor a of A_i. The value of @@...@x^c@...@ is the set of all objects that have y in the A_i attribute, where y is in the complement of x and x is a descriptor of the ith attribute. Similarly with @@...@x^g@...@ and @@...@x¹@...@.
 3. If t is a simple term and t = a₁ a₂...a_n, then v(t) is the set of objects v(a₁ @...@)*v(@a₂ @...@)*...*v(@@...@a_n).
 4. v(-t) = X - v(t), where X is the set of all objects in the data base, v(t + s) = v(t) + v(s), where + is union, v(t*s) = v(t)*v(s) where * is set intersection, and v(t→s) = (X - v(t)) + v(s).

Definition: t = s if v(t) = v(s).

Definition: Term t is in *normal form* if t = t₁ + t₂ + . . . + t_n and each t_i, 0 ≤ i ≤ k, is simple.

Theorem 1. Let t be a term. Then there is a term s such that t = s and s is in normal form.

The proof is straightforward and omitted.

The result we obtain from these definitions and Theorem 1 is that if we wish to identify the classes that have class descriptors satisfying a query, then we may put the query in normal form and find the set of class descriptions for each simple variable term separately.

STORAGE ORGANIZATION

We now describe a storage organization that will permit storage of data for statistical analysis as a partitioned database, and that will allow efficient implementation of the user data-manipulation operations described earlier while preserving the advantages of a partitioned database for the statistical analyses themselves.

Data Structures

For each equivalence class defined by a set of values of the selection attributes, a block of storage would be allocated to contain the following items:

1. The summary quantities described in *Statistical Operations*;
2. Head and tail pointers to a chain of blocks containing the values of the variables for each case.

In order to minimize overhead, users creating a new database would be asked to estimate the maximum number of variables they are likely to require. This estimate would be used to determine the size of both the class header blocks and the data blocks. (Suitable utilities could be provided to rebuild the database, should the estimate prove grossly in error.)

Retrieval

The major problem in retrieval from a partitioned database is that the number of possible classes may be very large; if there are 10 selection attributes with only 5 values each, 5¹⁰ classes are possible. There are two circumstances which make it likely that, at any point in time, a substantial number of the classes will be empty. The first is that in the initial stages of the research, when the researcher is still making decisions about appropriate coding schemes, the number of cases actually entered may be quite small, even though the number of classes needed to store them may be quite large. Second, there may be functional dependencies within the data that insure that some classes remain empty; for example, if everyone who smokes has at least one heart attack, then the class for smokers who have not had a heart attack will always remain empty.

In order to avoid storage wastage, it would be desirable to avoid creating class headers for non-existent classes. Hence,

the problem becomes one of searching a sparse space of class identifiers.

The solution we propose is to treat the search for class headers as a substring search problem. The nonempty class descriptors are formed into a string sequentially. By sequentially, we mean one class descriptor follows another without separator characters. The order in which class descriptors are placed in this string is the same as the order of the class headers. This string will be referred to as the data stream.

Let f be a comparison function where:

1. $f(a,b) = 0$ if $a = b$ and 1 if $a \neq b$.
2. $f(@,a) = 0$ for all a .
3. $f(a^c,b) = 0$ if $a \neq b$ and 1 if $a = b$.
4. $f(a^g,b) = 0$ if $b > a$ and 1 otherwise.
5. $f(a^l,b) = 1$ if $b < a$ and 1 otherwise.

We may assume that when a query is entered, it may be analyzed and simplified and changed to normal form. The query is now of the form where each term is of the form, $t_1 + t_2 + \dots + t_n$, and each t_i is a simple term. A pattern is query in normal form where the terms are concatenated in the following order: $p = t_1 t_2^R t_3 t_4^R \dots t_n$ or t_k^R if k is even and t_k^R is the reversal of t_k . The data stream is $d_1 d_2 \dots d_n$, where each d_i is a class descriptor.

First, we shall take the case where p has only one simple term. Then the pattern p is passed over d_1 and compared symbol by symbol using the comparison function. The comparison continues until the comparison function registers a 1 or all n symbols in the pattern have been compared. If the comparison function registers a 0 for all the symbols in the pattern, then we know d_1 is in the value set of the query. The pattern then goes on to d_2 . If a 1 has been registered, then the pattern goes directly to d_3 . The pattern match continues this way until the entire data stream has been matched.

If the pattern is composed on more than one simple term, then after t_1 is compared, the pattern and data flow are reversed and d_1 is compared to t_2^l , and so on until the entire pattern is compared to d_1 . A similar procedure is used for $d_2 \dots d_m$. A tally is kept as the data stream advances so that if the d_i class description is being examined, the tally contains an i . The system can then locate the i th class header to retrieve the desired information.

If each term is t descriptors long, there are c classes in the database, and there are q terms in the query, the algorithm will have a worst case performance of $O(q * t * c * t)$. While this bound will be large, particularly if the number of descriptors is large, we note several advantages to this approach over other methods of locating items in a sparse space. First, since the information is stored in a highly compact form, without any space needed for pointers, it will usually be possible to contain the entire data stream in primary memory; hence, the time required for a search is kept to a minimum. Second, it is easy to add new classes by simply adding their descriptions to the end of the data stream; deletions can be handled by replacing a description by a symbol not in the alphabet so that a match never occurs. Finally, since the pattern matching problem already occurs in a wide variety of applications, it is

a likely candidate for hardware enhancement, such as the use of VLSI components.

User Operations

In a previous section we specified operations that users of statistical analysis systems desired to perform on their data before doing the statistical analyses. We now describe how these operations would be performed using the structures proposed here.

Operations that do not create new classes

Adding cases. If a case to be added falls into an existing equivalence class, then addition of a case will involve (1) locating the class header via the pattern match mechanism just described, (2) updating the summary information in the header by adding the quantities from the new case, and (3) storing the case values in a data block. This last step could be accomplished most rapidly by using the tail pointer stored in the header.

Deleting a case. If a case to be deleted was not the only case in the class, it could be deleted by the following steps: (1) Locate the class header. (2) Subtract out the case quantities to update the summary information. (3) Search the chain of data blocks to locate the case. (4) Mark the case as deleted. (A special variable or bit in each case might be reserved for this purpose.) If desired, the spaces for deleted cases might be chained into a free storage list.

Adding a non-selector variable. If the preallocation scheme described earlier is used, then values for the new variable would be placed in the next available slot in all of the cases. (We assume that each case has the same number of variables even though some of the variables may have missing values.) The problem that this presents is how to locate a particular case for which a value is to be added. If each case has stored as a variable an identification number or case number, then it is easy to specify which case is desired. Finding it, however, could, conceivably, require a search of the entire file. A more tractable scheme is to require that the user specify values of selector variables for the case. Search would then be confined to a single equivalence class.

Deleting a variable. This would require a pass through all the data blocks and changing the value of the variable to some distinguished value. Alternatively, if a table is maintained which links names of variables to their locations within a case, the entry for the variable could be removed.

Transforming a non-selector variable, including recoding it. This operation would, in general, also require a pass through all data blocks. Note that if all the blocks for all classes are stored in the same file, this can be done by reading the blocks sequentially as if they were not linked. If the transformation is needed for only one analysis and is not to be saved afterward and if the transformation only involves operations with a constant, the summary information in the header block is sufficient to calculate the quantities needed for statistical analysis of the new variable.

Operations that create new classes

Operations that do not change the length of class descriptions

Adding cases that create new classes. This will occur when the combination of selection attributes in a new case has not occurred previously. When this happens, (1) a new class header is set up and the class description for the new class is added to the end of the search string, and (2) a new data block is allocated and the values of the new case are placed in it.

Deleting the only case in a class. This can be accomplished by (1) replacing the class description in the search string with symbols not in the alphabet so that a match against it always fails and (2) returning the data block to a free storage list.

Operations that change the length of class descriptors

All of the following operations will require rewriting the entire search string. Note, however, that even for the 100,000 class example, this could be accomplished in less than 50 milliseconds on the slowest commercially available processors.

Adding a new selector variable. This includes situations in which an existing non-selection variable is declared to be a selection variable, as well as those situations in which a new variable is created either by user entry or by transformations on existing variables. The steps involved are: (1) Recopy the search string to permit extending the length of a class. Pad each descriptor with a dummy, constant attribute. (2) For each existing class, examine the value of the new variable for the first case in that class. Use this value to replace the dummy attribute for that class. If the remainder of the cases in that class have the same value, no further work is needed. Otherwise, delete each case from its former class and create a new class for it. Repeat this process for all existing classes.

CONCLUSION

The problem of performing a statistical analysis on an entire database is equivalent to computing a function over that entire database. Addressing or indexing schemes that aid in the localization of particular entries are not of much help in such situations; indeed, if all access to the database must be through these schemes, their use can be slower than sequential processing of an unordered file. The scheme described here relies, instead, on computing and updating that function on the data as it is entered into the database. This notion is similar to the concept of "alerters" in relational databases (Buneman & Clemons⁶). Since most statistical analyses are based on a common set of initial computations, this means that if these quantities are computed at the time of entry they are available for all subsequent analyses. The use of this type of partitioned architecture for the database, thus, offers considerable advantage over simple, sequential organizations or conventional indexing structures.

REFERENCES

1. Nie, N.H.; Hull, C.H.; Jenkins, J.G.; Stenbrenner, K.; and Bent, D.H.: *Statistical Package for the Social Sciences*, New York: McGraw-Hill Book Company, 1975.
2. Dixon, W. J. (Series Ed.) *Biomedical Computer Programs P-Series*. 1977, Berkeley, Ca.: University of California Press.
3. Marek W. and Pawlak, Z. Information storage and retrieval systems: Mathematical Foundations. *Theoret. Computer Science*, 1976, 1, 331-354.
4. Lipski, W. and Marek, W. On information storage and retrieval systems. In *Mathematical Foundations of Computer Science*, A. Mazurkiewicz and Z. Pawlak (Eds.), 1977, Banach Center Publications, Vol 2, Warsaw Poland: Polish Scientific Publishers, 215-259.
5. Lipski, W. On semantic issues connected with incomplete information databases. *A.C.M. Transactions on Database Systems*, 1979, 4(3), 272-296.
6. Buneman, O. Peter and Clemons, E. K. Efficiently monitoring relational databases. *A.C.M. Transactions on Database Systems*, 1979, 4(3), 368-382.
7. Wong, E. and Chiang, T. C. Canonical structure in attribute based file organization. *Comm. A.C.M.*, 1971, 14, 593-597.

Development of an automatic sleep EEG analysis and staging system

by M. W. VANNIER

University of Kansas Medical Center
Kansas City, Kansas
and *Washington University School of Medicine*
St. Louis, Missouri

E. OTHMER and S. OTHMER

University of Kansas Medical Center
Kansas City, Kansas

P. FISHMAN

University of Kansas Medical Center
Kansas City, Kansas
and *Massachusetts Institute of Technology*
Lexington, Massachusetts

INTRODUCTION

The application of computers to sleep staging has not replaced manual classification techniques because of the simplifying assumptions employed in the design of most systems. These assumptions result in significant deviations from the results of automated analysis when compared to manual techniques applied to the broad spectrum of sleep electrophysiologic signals that may be observed.²

We undertook the development of a minicomputer (PDP-11)-based software system for the accurate classification of sleep EEG and sleep stage analysis⁹ based on the Rechtschaffen and Kales standard.⁵ This system is intended to obviate the need for manual analysis in sleep staging.

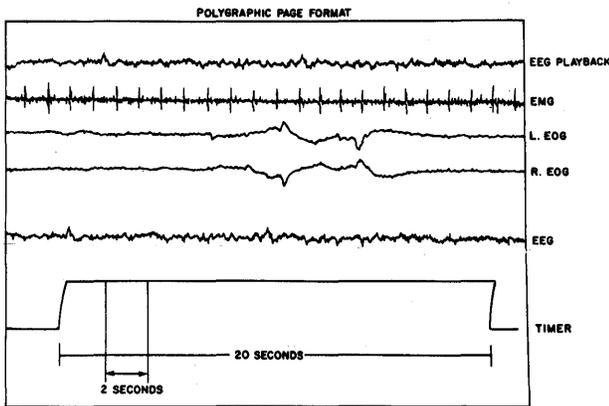
The system described here, which uses statistical and deterministic pattern recognition algorithms developed to permit accurate and sensitive detection of the characteristic structural elements that comprise sleep, is a powerful tool for sleep research. The system, requiring a relatively modest hardware environment, offers remarkable flexibility. The user may specify the parameters that define the specific structural features of each sleep EEG transient to accommodate personal bias in the results or to accommodate individual differences between subjects. Careful attention has been paid to the synchronization of the analysis result and the original sleep record. The user may request a detailed analysis of the data record relating to one specific segment of the analysis procedure. Excerpts may be taken from a long sleep record and be stored in secondary storage for subsequent analysis. These techniques have greatly facilitated the debugging of the system, allowing us to concentrate on difficult passages selected from multiple records.

In standard sleep laboratory practice^{1,5,6} the analysis of sleep stages from an all-night recording of 6 to 8 hours is carried out by manual scoring of a polygraphic chart record (Figure 1), applying standard criteria.⁵ The human observer evaluates the record one page at a time, where a page consists of a 20-second plot of the electroencephalogram (EEG), electrooculogram (EOG), and electromyogram (EMG). The observer is required to classify the stage of sleep as awake, movement (artifact), light sleep (Stage 1) to deep sleep (Stage 4), or a rapid-eye-movement (REM) episode.

By using the automated system described in this paper, a polygraphic chart record of the electrophysiologic activity is produced as it would be in manual analysis. During the same period, the signals are recorded on digital magnetic tape by the computer, and optionally on analog tape as well (Figure 2). After the magnetic tape is rewound, an off-line analysis program is run that will classify the predominant EEG pattern for each two-second epoch. These two-second results are stored in an intermediate disk file, and after the first pass through the data is complete, the sleep-staging routine will make a final-stage determination for each page of the record (Figure 3).

KUMC SLEEP SYSTEM

The data acquisition phase acquires five analog signals (Table I) and stores them on digital magnetic tape. These data are stored as ADC units with calibration information, so that direct computation of the measured potential in microvolts is possible using linear interpolation. The availability of the raw data in absolute units is essential, since the standard criteria⁵ require decisions based on these units.



Each polygraphic page contains 20 seconds of data that is analyzed in 10 equal 2-second segments. Five signals, including EEG, left EOG, right EOG, EMG and timing, are recorded by the computer. The EEG playback signal originates in the analog tape recorder, and is used for quality control.

Figure 1—Sample polygraphic page format

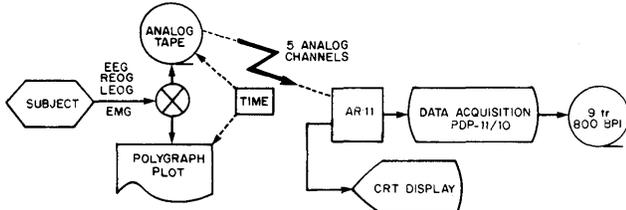


Figure 2—Data acquisition scheme used in sleep signal (processing involving sampling of 5 analog signals by the PDP-11)

When the acquisition phase is complete, the pattern recognition phase may be initiated. This is implemented in an overlay structure (Figure 4) with phases dedicated to (1) EEG: background EEG pattern analysis and refinement; (2) TRANSIENT: the detection of EEG transients such as sleep spindles and K complexes; (3) MISC. ANALOG: adjunct signal analysis, including detection of eye movements from the EOG and muscle tone from the EMG; (4) SUMMARY: preparation of a page summary from 10 individual two-second epochs with refinement of the background analysis and formation of a preliminary stage score, and (5) OUTPUT: the creation of a pattern listing and intermediate file with scores used in the final-stage analysis, and optionally a debugging file containing detailed information on each step used in the decision process applied to the raw data for one aspect of the feature extraction procedure.

The hardware requirements for the system are modest; the present hardware configuration at the University of Kansas Medical Center Sleep Laboratory is shown in Figure 5. The system is implemented in Fortran IV under the RT-11 operating system.

PATTERN RECOGNITION

Feature extraction procedures have been implemented to identify the EEG background activity (Table II) and to detect

the presence of EEG transients, spindles, and K complexes, which occur sporadically and are important in sleep stage identification. Determination of muscle tone level from the EMG and detection of eye movements from the EOG are included in the system.

The determination of EEG characteristics is performed for every two-second interval in the record, where 500 samples are available for each determination. Using a linear discriminant analysis function determined from teaching samples that had been classified earlier,⁹ the EEG background activity is assigned to one of 15 categories (Table II). This discriminant analysis procedure is augmented by deterministic criteria to refine and improve the accuracy of these elements. A detailed exposition of the operation of these routines is available elsewhere,^{4,9} and in general the EEG background activity is estimated using a bisector analysis applied to the turning points in the raw EEG.⁴ As shown in Figure 6A, the line segments joining the turning points in the raw EEG are bi-sected, and their average amplitude and frequency are computed. This procedure correlates closely with manual analysis, and validation of these procedures including accuracy and sensitivity results have been described previously.⁴

For improvement of the accuracy of delta EEG activity, stringent amplitude criteria must be applied when using standard analysis techniques⁵ to distinguish theta and delta rhythms. As shown in Figure 7, we measure the percentage of each epoch in which the EEG amplitude exceeds 75 microvolts. If this measure is greater than a specified minimum, the epoch is termed delta, otherwise theta.

Time synchronization of the analog tape recording, polygraphic plot, and digital computer was accomplished using a

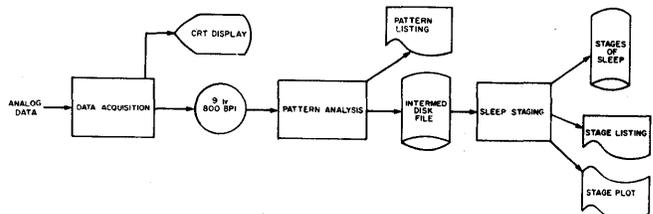


Figure 3—Sleep EEG analysis and staging system block diagram

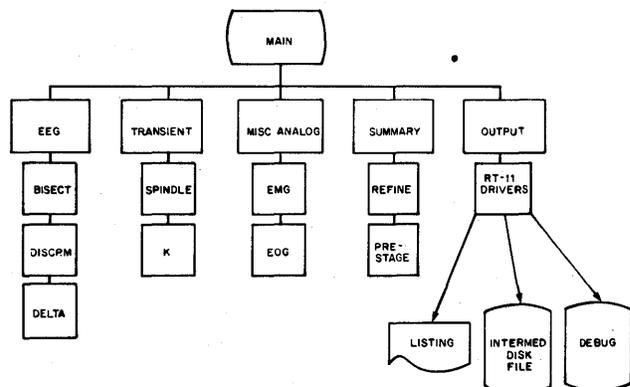


Figure 4—Overlaid software system block diagram

TABLE I—Input signals

Type	Sampling Rate
EEG	250 Hz
EMG	25 Hz
R. EOG	25 Hz
L. EOG	25 Hz
Timing	250 Hz

TABLE II—EEG types recognized

Abbreviation	Definition
AH	High alpha
AL	Low alpha
AM	Alpha w/ muscle artifact
B	Beta
TL	Low theta
TH	High theta
TR	Rhythmic theta
DL	Low delta
DH	High delta
M	Muscle (movement) artifact
X	Artifact
DF	Delta, based on frequency
<	Theta, based on amplitude
>	Delta, based on amplitude
F	Awake, non-alpha

simple electrical device⁷ that produced a measurable voltage transition at each 20-second interval.

Sleep spindles are defined as rhythmic bursts of 13–18 Hz activity having a minimum duration of one-half second and lasting no longer than two seconds. The presence of sleep spindles is detected by inspection of the turning points in the raw EEG data to determine the average frequency within a window of one-half second. If the frequency and amplitude criteria are met within a region of appropriate background EEG activity, a spindle is identified (Figure 6B).

K complexes are sharp biphasic waves consisting of a negative peak immediately followed by a positive deflection. A simple application of slope criteria to each transition above a given threshold is used to identify the presence of K complexes (Figure 6C).

The EMG envelope width is taken as a measure of muscle tone. The EMG envelope width is determined by undersampling the EMG signal, sorting the 50 samples from each two seconds in ascending order and selecting the thirty-fourth largest value. Selection of the sixty-eighth percentile (thirty-fourth of 50 values) results in a relatively artifact-free estimate of envelope width.⁸

Eye movements occurring during sleep are essential components of REM periods, and computer processing of the EOG is directed at the detection of these movements. Deviations of the EOG in either the left or right channel beyond a specified threshold above baseline are inspected to determine the presence of a rapid eye movement.^{4,8} The duration of the

deviation above threshold and the presence of corresponding activity in the other eye channel must meet specified criteria for an eye movement to be considered REM (Figure 6D).

The system supports the formation of an external secondary data file containing important or difficult regions of the raw data record for repeat analysis at a later time. Analysis may be started at any point in the data record by specifying the starting page number. The raw data may be plotted in the same format as the polygraphic chart record to assure absolute registration of the time signals. Extensive debugging information is available concerning each important decision in the classification of EEG background, EEG transients, EOG, and EMG. These data are written to a disk file, which may be printed or summarized as required. The debugging data is hierarchical, and the operator may be selective regarding the level at which decision data on the classification algorithms are retained.

Reports of the intermediate two-second epoch scores may be printed, and stage summaries and plots (Figure 8) are also available.

CONCLUSION

A comprehensive system for the analysis of the stages of sleep has been developed for the PDP-11 minicomputer. The sys-

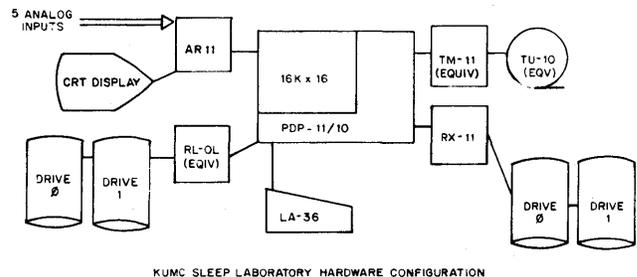


Figure 5—University of Kansas Medical Center Sleep Laboratory hardware configuration

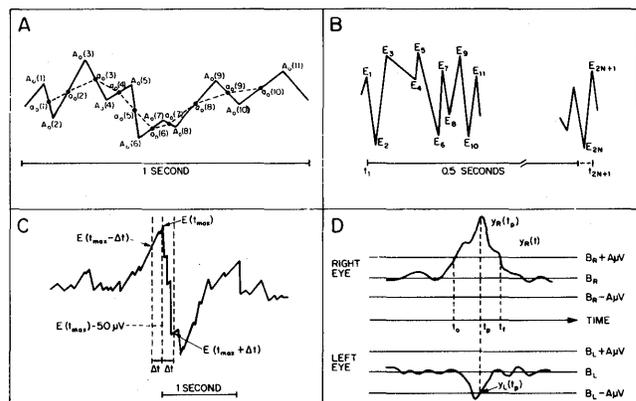


Figure 6—Pattern recognition algorithms. A. EEG bisector analysis. B. Sleep spindle detection. C. K-complex detection. D. Rapid eye movement (REM) detection.

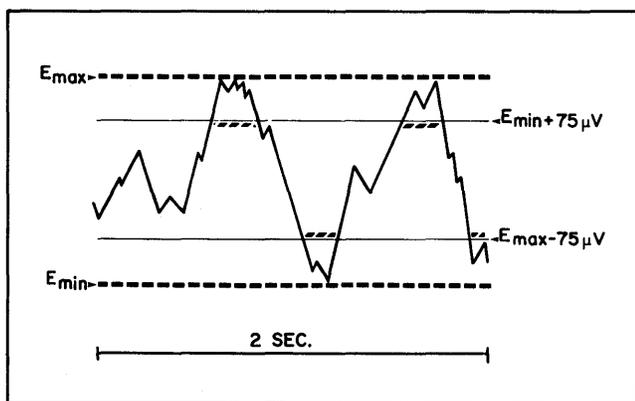


Figure 7—Delta algorithm

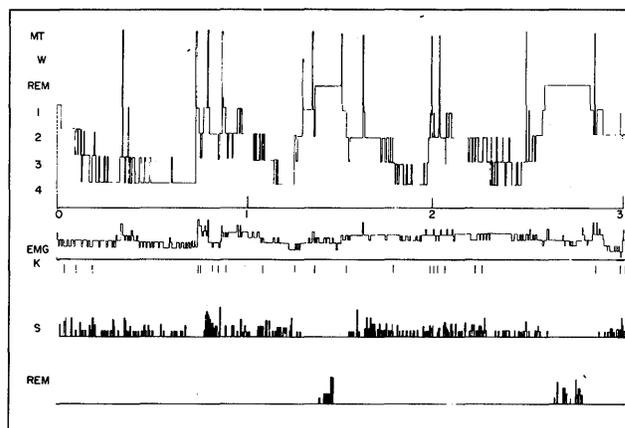


Figure 8—Plot of the stages of sleep

tem consists of data acquisition, pattern analysis, and stage decision modules. Exceptional flexibility in the operation of the system is a major feature of the system, where key parameters in the decision processes may be interactively modified by the operator to match personal bias or adjust for individual differences between subjects.

REFERENCES

1. Handbook of Electroencephalography and Clinical Neurophysiology, Volume 7: Physiological Correlates of EEG, Part A: EEG and Sleep, 1975, 95 pp.
2. Smith, J.R. "Computers in Sleep Research", *CRC Crit. Rev. in Bioengr.*, 3 (1978), pp. 93-148.
3. Cox, J.R., F.M. Nolle, and R.M. Arthur. "Digital Analysis of the Electroencephalogram, the Blood Pressure Wave, and the Electrocardiogram." *Proc. IEEE*, 60 (1972), pp. 1137-1164.
4. Othmer, E., M. Vannier, S. Othmer, and P. Fishman. "Pattern Recognition in Sleep Research." *Proc. Vth Intl. Conf. on Pattern Recog.*, December 1-4, 1980, Miami Beach, Florida.
5. Rechtschaffen, E.A., and A. Kales. "A Manual of Standardized Terminology, Techniques and Scoring System for Sleep Stages of Human Subjects." Public Health Service. Washington, D.C.: U.S. Government Printing Office, 1968.
6. Karacan, I., W.C. Orr, T. Roth, et al. "Establishment and Implementation of Standardized Sleep Laboratory Data Collection and Scoring Procedures." *Psychophysiology*, 15 (1978), pp. 173-179.
7. Othmer, E., S.C. Othmer, M.W. Vannier, P.M. Fishman, and W. Holland. "A Simple Time Synchronization Device for Polygraph, Analogue Tape and Digital Computer as Used in Sleep Research." *J. Biomed. Eng.*, 1 (1979), pp. 127-128.
8. Othmer, E., S.C. Othmer, P.M. Fishman, and M. Vannier. "Electromyogram Processing for Sleep Research." *Int. J. Bio-Med. Comput.*, 11 (1979), pp. 33-39.
9. Fishman, P.M., and E. Othmer. "An Algorithmic Description of the SLEEP Program for the Analysis of Sleep Polygraphic Records." Washington Univ. Biomed. Comput. Lab Monograph 282, St. Louis, Missouri, June 1976.

Embedding an information system within a generalized network environment*

by DARRELL L. WARD

North Texas State University
Denton, Texas

ABSTRACT

This paper introduces a generalized network information facility called HYPERTEXT, which is currently implemented on a TANDEM-16 loosely-coupled, high-availability multi-processor minicomputer. The overall structure of HYPERTEXT is described as well as several important user functions available within HYPERTEXT. An actual example of embedding a complete information system (a clinical data management system) within the HYPERTEXT facility is developed, as well as the underlying motivations for such an incorporation. The inclusion of peripheral aspects of an information system, such as user and system documentation, is presented in the HYPERTEXT context, along with some components that HYPERTEXT supports very well and that are typically not considered part of a large information system.

INTRODUCTION

Information systems have typically evolved as unique software packages with the usual complement of user and system documentation necessary for using the system. This generally consists of user documentation to assist the end user of the system as well as system documentation to aid the system maintainer. Recently, several approaches have been developed to aid developers in the design, implementation, and documentation of large computer systems.^{1,2,3,4} The motivation underlying such systems is to insure an organized and consistent approach during the development cycle. However, upon completion of an information system, the documentation typically is distributed to various users of the system, requiring, in many instances, massive efforts when that documentation must be revised. One objective of this paper is to introduce a total environment providing the developer facilities for program development and documentation as well as providing the end user with an environment representing the totality of information needed to use the end system effectively.

HYPERTEXT^{5,6,7,8,9,10} is a facility designed to provide a rich informational structure within which information and as-

sociations among information can be represented and traversed easily and flexibly. The HYPERTEXT informational environment will be introduced as well as the functions available to the user of such a facility. HYPERTEXT, as it is currently implemented on the TANDEM-16 minicomputer system at The University of Texas Health Science Center at Dallas, will be introduced in the next section of this paper. The hardware/software features of the TANDEM-16 have been presented elsewhere and will not be developed in this paper; however, it is appropriate to point out that the TANDEM design criteria include reliability and responsiveness in a transaction environment.^{11,12} The section "An Information System Within HYPERTEXT" will demonstrate how a complete information system (a clinical data management system [CDMS]) may be represented in the HYPERTEXT informational environment. The advantages of such an arrangement to the developer, maintainer, and end user will be presented in that section.

THE HYPERTEXT FACILITY

The description of the HYPERTEXT facility will be presented in two parts as follows:

1. Development of the informational environment
2. Development of the operations applicable to the information environment

HYPERTEXT INFORMATIONAL ENVIRONMENT

The HYPERTEXT system logically consists of two major components, the information environment and the HYPERTEXT monitor. The HYPERTEXT monitor is responsible for creating and maintaining the informational environment as well as interpreting and accomplishing requests of the HYPERTEXT user. The logical view of the HYPERTEXT environment and its relationship to the rest of the world is illustrated in Figure 1.

In introducing the environment presented by HYPERTEXT, it is first appropriate to present some basic definitions. A HYPERTEXT page is an entity designed to contain a finite amount of information. Each page will have a unique number

* This work was supported in part by USPHS Grant P50A20543.

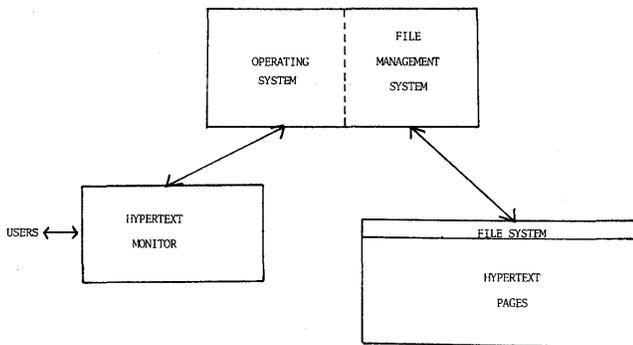


Figure 1—The hypertext environment

assigned to it by the HYPERTEXT monitor and may contain two types of information:

1. Textual information
2. HYPERTEXT language commands
 - a. Commands that are interpreted as encountered
 - b. Commands that must be specifically invoked by the user in order to be interpreted

The page is the basic unit of information in the HYPERTEXT system; and, as such, pages are objects of operations. A page is typically configured to correspond to the output screen of a CRT type of terminal. Although any teletype-compatible terminal may be used for most of the operations to be developed later, the HYPERTEXT system is best used on a page mode terminal with local intelligence to support most basic editing functions. A user, upon entry to the HYPERTEXT system, will be presented one unique entry page associated with that particular user and chosen because of the user's sign-on account number. This entry page of each user represents the first node of a network structure of information pages available for visitation during a HYPERTEXT session. A visit to a page is defined to be the invocation of a page by the HYPERTEXT monitor. When one visits a page, two activities are possible:

1. HYPERTEXT commands on that page, if any, are interpreted by the monitor.
2. Textual information on that page, if any, is presented to the user at a terminal.

Pages will be further categorized depending on the type of functions occurring as the page is visited. A page is called a

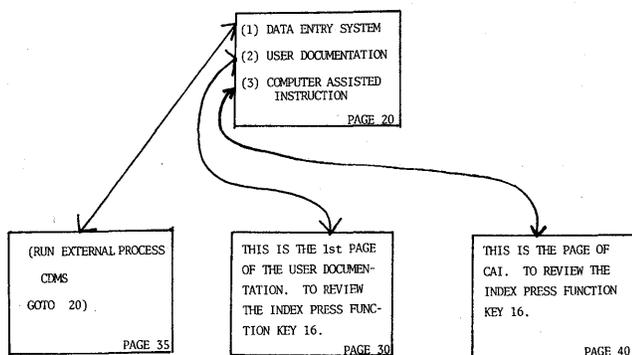


Figure 2—Example hypertext pages and relationships

passthrough page if when it is invoked it contains a HYPERTEXT command that automatically invokes another page. On the other hand, a page is called a presentation page if no other pages are automatically invoked when that page is visited. Whenever a presentation page is invoked, the user is placed in user command mode so that user operations may be interpreted by the HYPERTEXT monitor at that point in time. Command mode is apparent to the user as the text on the page is presented, followed by a line at the bottom of the screen as follows:

ENTER:

The user may now enter a single HYPERTEXT command. Some of these commands will be illustrated in the section "HYPERTEXT Operations."

A page A is said to be directly linked to Page B if a HYPERTEXT command is embedded within Page A, permitting the user to invoke Page B by explicitly using the HYPERTEXT monitor. Any page validated for public access may be visited by merely requesting the HYPERTEXT monitor to invoke that page (the page must be referenced by its page number).

Figure 2 illustrates the above concepts. The type of structure represented is a generalized network structure, since any page is accessible from any other page. Page 20 is the entry page, designated by the HYPERTEXT monitor to be presented at the beginning of a HYPERTEXT session. Page 20, a presentation page, is directly linked to Pages 30, 35 and 40, since the choices (1), (2), and (3), represent embedded commands pointing to other pages. Page 35 is a passthrough page, as it contains a command, (GOTO 20), causing Page 20 eventually to be reinvoked. In this particular example, Page 35 will invoke a process capable of being executed by the hardware of the system, and upon completion of execution Page 20 will be reinvoked. When the user visits each of the presentation pages—Pages 20, 30, 40—the HYPERTEXT system presents the textual information, interprets and executes any HYPERTEXT commands embedded on that page, and immediately places the user in command mode.

HYPertext OPERATIONS

This section introduces some of the language elements facilitating use of the information environment. This is not designed to be an exhaustive development, nor is it designed to be syntactically precise. Rather, it is intended to provide the background for the presentation of the section "An Information System Within HYPERTEXT." The following represents operations or functions that users may wish to accomplish within an informational system such as HYPERTEXT:

1. Create and maintain pages of information.
2. Establish relationships among the pages making up the informational environment.
3. Traverse the information network in a flexible manner.
4. Invoke application programs (systems) outside the HYPERTEXT environment under control of the HYPERTEXT monitor.

```
(*2900)  PATCHING IN YOUR TERMINAL

(*3502)  LOGGING IN

(*510)   DATA ENTRY SCREENS

(*7500)  DATA DICTIONARY

                                           PAGE 2510
```

Figure 3a—A page as it appears in edit mode

```
(1)  PATCHING IN YOUR TERMINAL

(2)  LOGGING IN

(3)  DATA ENTRY SCREENS

(4)  DATA DICTIONARY

ENTER: 
      ^
      |
CURSOR POSITION

                                           PAGE 2510
```

Figure 3b—Page in command mode form

The remainder of this section will develop the first three of the above items; the section “An Information System Within HYPERTEXT” will address the invocation of application programs within the HYPERTEXT environment.

In order to create a new page, one merely responds with “NEW PAGE” while in HYPERTEXT command mode—e.g.,

ENTER: NEW PAGE

Immediately, a blank page is presented and the user is placed in edit mode in order to place information on that page. Any existing page may be edited by using the “EDIT” command while on that page in presentation mode—e.g.,

ENTER: EDIT

In both of the above situations, the user is in local mode, capable of designing or altering the page to suit the application. The edit session is terminated by depressing a special function key (Function Key 1 in this implementation) signifying that the page is to be stored for subsequent access within the HYPERTEXT system.

Establishing and maintaining relationships among pages requires the use of page links. A page link, a facility for directly invoking another page within HYPERTEXT, is either explicit or implicit. An explicit page link is a pointer placed within a page that subsequently requires the user to merely make a menulike selection to invoke that page. Figure 3a shows a page in edit mode with several explicit links showing. When the page in Figure 3a is actually invoked by the user it will appear as shown in Figure 3b, with the explicit links transformed to menu selection items. Thus pages are linked explicitly by implanting explicit links on the parent pages.

One implicit link may be implanted per page. An implicit link is merely a method for indicating a next page. For example, if a serial set of information is needed, these pages can be nicely linked together via the implicit link, and the user repeatedly depresses the next function key (Function Key 16 in this implementation) to review all pages representing the information. Figure 4a shows three pages in edit mode form with the implicit links shown; Figure 4b illustrates how those pages would appear when invoked.

Traversing the network is done using several function keys and/or traversal commands. The traversal commands are

1. GOTO <page number>
2. NEXT
3. BACK <n> PAGES
4. BACK <n> LANDMARKS
5. BACK <n> TURNS

The first two commands have previously been informally addressed; thus we will address the BACK commands in their respective order.

HYPERTEXT “remembers” the pages that have been visited in a session (up to 200) and the BACK <n> pages allows the user to revisit pages. This is extremely important in reviewing previous information as well as in reinvoking menu pages, as illustrated in Figure 2. We clearly could have replaced the command (GOTO 20) with the command (BACK 1 PAGE) and accomplished the same function.

As a user traverses pages, there may be pages that are special, in the sense that they need to be revisited. In such a situation the user can mark that page with a LANDMARK. This is accomplished by depressing the LANDMARK func-

DATA FROM RELATION NAMES		DATA FROM RELATION NAMES		DATA FROM RELATION NAMES	
NAME	LICENSE NO.	NAME	LICENSE NO.	NAME	LICENSE NO.
Jones, B.	52532146	James, L.	87123478	Ross, R.	17214300
Sims, K.	00043217	Bone, O.	19163214	Doe, J.	84326458
Wilson, R.	53217421	:	:	:	:
:	:	Smith, J.	39241798	Ward, D.	00001234
Harris, J.	00214381	(continued on next page)	:	:	:
(continued on next page)	:	(**10)	:	END OF RELATION	:
(**8110)	PAGE 200		PAGE 8110		PAGE 10

Figure 4a—Implicit links shown in edit mode

DATA FROM RELATION NAMES		DATA FROM RELATION NAMES		DATA FROM RELATION NAMES	
NAME	LICENSE NO.	NAME	LICENSE NO.	NAME	LICENSE NO.
Jones, B.	52532146	James, L.	87123478	Ross, R.	17214300
Sims, K.	00043217	Bone, O.	19163214	Doe, J.	84326458
Wilson, R.	53217421	:	:	:	:
:	:	Smith, J.	39241798	Ward, D.	00001234
Harris, J.	00214381	(continued on next page)	:	:	:
(continued on next page)	:	(continued on next page)	:	END OF RELATION	:
ENTER:	PAGE 200	ENTER:	PAGE 8110	ENTER:	PAGE 10

Figure 4b—Traversing implicit links via NEXT key (Function Key 16)

VETERINARY PATHOLOGY SYSTEM

- | | |
|-----|-------------------------------------|
| (1) | DATA ENTRY, UPDATE, PATIENT INQUIRY |
| (2) | RETRIEVAL AND REPORTING |
| (3) | AD HOC RETRIEVAL |
| (4) | USER DOCUMENTATION |
| (5) | COMPUTER AIDED INSTRUCTION |

Figure 5—An example entry page

tion key or typing in LANDMARK when in user command mode. Then, when needed, the page may be recalled by merely depressing the BACK 1 LANDMARK function key or issuing the command BACK <n> LANDMARKS.

Any page containing branches or multiple explicit paths is designated as a TURN page. Users may return to TURN pages by using the function keys or BACK <n> TURNS, analogous to the LANDMARK operation. This function may be of particular importance if an index of information items is available and the user is in the midst of traversing an area of information. With this function one can immediately return to the index and select another area for traversal.

AN INFORMATION SYSTEM WITHIN HYPERTEXT

A generalized clinical data management system (CDMS)^{13,14} has been developed to support the research activities of the clinical environment at the University of Texas Health Science Center at Dallas. The embedding of this complete system

COMPUTER ASSISTED INSTRUCTION

- | | |
|-----|------------------|
| (1) | CAI - HYPERTEXT |
| (2) | CAI - CDMS |
| (3) | CAI - STATISTICS |
| (4) | CAI - ENFORM |

Figure 6—CAI index page

within the HYPERTEXT environment will be illustrated as well as underlying motivations. At this point the system consists of two major software components:

1. A data entry/update component responsible for all alterations to the data of the system
2. A retrieval and report writing system for extraction of information from the system

After passing the HYPERTEXT security check and successfully entering the HYPERTEXT environment, the CDMS user is presented the entry level page. This entry level page is shown in Figure 5. The menu selection of data entry/update will cause the invocation of a process external to the HYPERTEXT informational environment, this case being the process that supports the entry or update of patient information. A passthrough page is created to accomplish the following:

1. Invoke the process that supports the data entry and update functions.
2. Upon return to the HYPERTEXT environment, an explicit command, BACK 1 PAGE, will immediately re-present the menu page to the user.

The two commands accomplishing the above on the passthrough page are

```
(CALL EXTERNAL PROCESS CDMS
BACK 1 PAGE)
```

The effect of the above is to invoke the process CDMS with the suspension of the HYPERTEXT monitor process. Upon termination of the CDMS process, the HYPERTEXT monitor will resume with the interpretation of BACK 1 PAGE and immediately present the presentation page containing the menu.

Thus the user is presented an environment of HYPERTEXT pages with selection menus and flexible traversal options with the capability of selecting a page that can effectively invoke a portion or all of an information system. The second menu option provided to the user of CDMS presents the end user with a set of predefined retrieval operations, each identified by an explicit link to a set retrieval procedure file. The third menu option (AD-HOC RETRIEVALS) will invoke a page initiating the execution of the TANDEM-16 relational retrieval facility called ENFORM.¹⁵

Within the ENFORM environment the user can interact with a relational database directing output to the terminal, external files, or line printer; creating procedure files of commands for future use; and performing the entire spectrum of operations associated with a retrieval facility. Thus the selection of the ENFORM menu item provides the user with a complete retrieval system complemented by the availability of the system text editor. Again, upon completion of the ENFORM session, the user will return to the page that invoked ENFORM, and the HYPERTEXT monitor will interpret additional commands within that page. Typically, the user will be directed back to the original selection menu.

Reviewing additional user options from the top page, one easily sees that user documentation is available in an on-line mode for traversing in the flexible manner described above. Additionally, the CDMS information system has incorporated a Computer-Assisted-Instruction (CAI) component for users wishing to train personnel on the structure and use of the total

information facility. HYPERTEXT provides a rich structure enhancing the on-line training of personnel within the facility itself; this is a significant feature of the approach.

The motivations behind the embedding of the CDMS project within the HYPERTEXT information environment are as follows:

1. The end user is provided with an open-ended and flexible interface to the information system.
2. The end user is provided with documentation and training tools within the informational environment.
3. The system implementer can use the HYPERTEXT environment as a structured tool assisting the implementation process.
4. The system maintainer is capable of dynamically altering various components of the system to satisfy changing user objectives over the life of the system.

Most of this presentation constitutes the basis for the motivations supporting the end user. As one can clearly see, there definitely are benefits to both the user and maintainer with respect to the management of an information system or systems. For example, the maintainer can develop a CAI module for instruction on the use of an information system and make that module available via appropriate links to all the information systems of that particular mode. This has been accomplished in CDMS and provides an excellent training tool for all users as well as residing in an environment permitting effective maintenance of the CAI modules. Figure 6 illustrates the CAI table of contents when that selection is made.

SUMMARY

The generalized network environment HYPERTEXT and its current implementation on the TANDEM-16 minicomputer system at the University of Texas Health Science Center at Dallas has been developed. The clinical data management system (CDMS) has been used as an example of embedding an information system within this informational environment. During the process of developing the above topics the advantages that the end user accrued from this approach were presented. Finally, some advantages from a management approach were developed illustrating the richness and feasibility of using the environment for the totality of the life of the information system.

There are currently the following production systems that operate in this environment:

1. A funded five-year study of the effects of various treatments on kidney stones
2. A funded five-year study of ischemic heart disease
3. A funded five-year study of the effects of hypertension
4. A veterinary pathology research application
5. A neuropathology research application
6. An internal terminal location and status database application

Each of the above applications has complete HYPERTEXT access to the user documentation, the CDMS data entry system, and predefined retrieval requests; ad hoc relational retrieval facility (ENFORM); and complete CAI sessions on ENFORM, HYPERTEXT, and the functions of CDMS. User experiences to date have been very supportive of the interface provided. Extensive evaluation of the positive impact of this environment on the user population is currently under review; but clearly their experiences, as well as those of the system maintainers, are positive.

Future work can certainly be directed to the continued investigations of tools that will enhance the interface associated with the development and use of information systems. The environment described herein, it is hoped, can serve as a model for additional work to provide useful, user-oriented tools enhancing the decision-making processes that emanate from information systems.

REFERENCES

1. Software Technology Company. "SADT The Softect Approach to System Development." The Software Technology Company, Jan. 1976.
2. U.S. Department of Defense. Automated Data Systems Documentation Standards Manual. Manual 4120.17M, Dec. 1972.
3. IBM Corporation. HIPO—A Design Aid and Documentation Technique. Order No. GC201851, IBM Corporation, Data Processing Division, White Plains, New York 10504.
4. Meyers, G. *Software Reliability*. Wiley Interscience, John Wiley and Sons, New York, 1976.
5. Nelson, T.H. "A File Structure for the Complex, the Changing and the Indeterminate," *Proceedings*, ACM 20th National Conference, 84-100.
6. Nelson, T.H. "No More Teacher's Dirty Looks." *Computer Decisions*, Sept. 1970.
7. Nelson, T.H. "A Conceptual Framework for Man-Machine Everything." *Proceedings*, National Computer Conference, 1973.
8. Nelson, T.H. "The hypertext." *Proceedings*, World Documentation Federation, 1965.
9. Carmody, S., W. Gross, T.H. Nelson, D. Rice, and A. van Dam. "A Hypertext Editing System for the 360." In Faiman and Neivergelt (eds.), *Pertinent Concepts in Computer Graphics*. Urbana, Illinois: University of Illinois Press, 1969, pp. 291-330.
10. Ward, D., and S. Bush. "HYPERTEXT—a General Purpose Educational Computer Tool." Available through the Medical Computing Resources Center as a technical report.
11. Bartlett, J.F. "A Nonstop Operating System." *Proceedings*, 11th Hawaii International Conference on the System Sciences, Honolulu, January 1978.
12. Katzman, J.A. "A Fault Tolerant Computing System." *Proceedings*, 11th Hawaii International Conference on the System Sciences, Honolulu, January 1978.
13. Ward, D., D. Mishelevich, C. Pak, and A. Sheehan. "A Relational Clinical Data Management System Supporting Urclithiasis Research." *Proceedings*, 3rd Annual Computer Applications in Medical Care, Washington, D.C., 1979, pp. 314-318.
14. Rothenberg, L., D. Ward, and D. Mishelevich. "The Logical Structure and Use of a Relational Clinical Data Management System." *Proceedings*, 13th Annual Hawaii International Conference on System Sciences, January 1980.
15. *TANDEM-16 Enform Reference Manual*. Tandem Computers Incorporated, 1979.

The design of the Clinical and Research Information System for Psychiatry

by RUVEN BROOKS

University of Texas Medical Branch
Galveston, Texas

ABSTRACT

The Clinical and Research Information System for Psychiatry (CRISP) is a general mental health information system for clinical, research, and administrative functions. From a software perspective, CRISP is designed to solve three problems: First, by attaching procedures to databases, it is designed to permit dynamic addition of new database formats and organizations without the need to restructure existing information. Second, CRISP uses a message-passing architecture with security checks on each message so that each user can be given a different set of access rights on each individual patient. Finally, the process attachment combined with message architecture makes it easier to distribute CRISP across a network. The network can, in turn, be used to provide smooth storage migration across different secondary storage devices.

INTRODUCTION

The Clinical and Research Information System for Psychiatry is intended to be a general mental health information system¹ for use in inpatient, outpatient, and community mental health settings. It will incorporate a variety of clinical, evaluation and assessment, and administrative functions, including basic demographic and admissions data, mental status, progress notes, treatment plans, direct patient services, and billing. From a computing perspective, the system is intended to be a paperless medical record in which interactive access to the computer system replaces paper charts. This paper describes the software design intended to support construction of the system.

Software Design Criteria

The software design for this system is intended to meet five major design criteria:

1. The database should permit equally easy access to all the information on a given patient or to any particular type of information across all patients.
2. The database should efficiently accommodate the frequent, continuous addition of new types of information

without modification to existing programs or existing files.

3. The database should support the design of a security system that could be used to restrict access to selected groups of patients and to selected information about each patient.
4. The database should assume the existence of a hierarchy of storage devices with widely varying access times and should provide for graceful migration of the least recently used data to media with longer access times.
5. The database should be distributable across a network of processors in such a way as to minimize communications traffic and to maximize reliability.

Elaboration

1. One of the fundamental assumptions behind the design of the entire information system is that there will be no inherent distinction between clinical and research data and that the same information that is used by a clinician to make decisions about treating a patient will also be used for research. There will, however, be some differences in the use of this information. From the clinician's viewpoint, the stress will be on the individual patient. Since the problems, history, and treatment course of patients in the database will be extremely varied, it will be the case that only the barest minimum of kinds of information—perhaps only name and birthdate—can be expected to be uniformly present for all patients. In using the database, therefore, the clinician's first task will be to find out what kinds of data are available on the particular patient and then to retrieve the particular data relevant to a clinical decision.

From the researcher's viewpoint, the stress will be on groups of patients on whom a common body of measurements is available. In using the database a frequent task will be to locate all patients who meet certain criteria. These criteria may be stated in terms of some particular measurement or classification, as in "retrieve the records of all schizophrenics" or in terms of the information available on them, as in "retrieve all patients on whom a SADS score is available." The database should accommodate this type of retrieval as well as the kind needed by the clinician.

2. An implicit assumption in most database designs is that

the number of different kinds of data schemas (or record formats) will be relatively small and static, with additions being made to the schemas only at the time of systems maintenance and reorganization. This assumption is unduly restrictive in a patient record system. Clinical recording devices continuously evolve over time; therefore information recording and retrieval requirements are constantly changing. Researchers will frequently want to be able to add new measurement devices in response to new research objectives. A goal in the design of the database will be to permit the continuous addition of new kinds of information with their associated schemas, without the need for recompilation or reorganization of the entire database. (A likely consequence of this property is that very few items will be common to all patients. If there are 10 versions of the initial admissions data, but the data collection is done only once per patient, then only 10% of the patients can be expected to have a database entry for a particular version of the intake data.)

3. There are many situations in which a user's access should be restricted to selected information about a patient. Further, the restrictions may vary from patient to patient. The database system should contain tools for implementing such a security system.

4. Judging from the size of current paper medical record systems, it seems that the data on even a small number of patients may consume large quantities of storage. Consequently, the system must provide for migration of information onto offline storage. This is done conventionally by marking patients as inactive and moving all the data on them to secondary storage. If patients are hospitalized or treated for extensive periods of time, however, a great deal of rarely needed data will accumulate on them. Thus, it may happen that data that are currently still being used for research are moved to archival storage because the patient is not currently being treated while valuable space is wasted on unused data on current patients. To avoid this problem, the system should be designed so that just the least used portions of a record can be moved to archival storage. Additionally, as is desirable in an archiving system, when data are moved offline, pointers to them should be left in the online storage.

5. As processors and storage become cheaper, multiple-processor installations are inevitable. Both for reliability and to reduce communications costs, the database should be distributable among processors. The design ought to assume a reasonable degree of repeated locality of access: if a datum has been accessed by one processor, there is an increased likelihood that the processor will access it again.

DESIGN OVERVIEW

To meet these criteria, the design for the CRISP system is based on a collection of asynchronous processes that communicate by means of messages. Data structures and databases are attached to particular processes and are only accessed via messages to them. An example will illustrate: The system maintains a master index of all patients that includes fundamental demographic information such as address. The process that controls this information is called the master patient index guardian. To retrieve information or update it, the user

runs an entry or display program at his terminal. This program does not read or write the master patient index directly; instead, for each access or update, the entry program sends a message containing the relevant information to the master patient index guardian. The guardian then accesses the file to retrieve the necessary information to perform the requested operation.

Processes fall into one of three classes:

1. *Access control processes.* These processes are responsible for controlling user access to various portions of the database. Their data structures contain access rights and pointers to the locations of information.
2. *Terminal Interaction Processes.* These processes handle terminal interaction with users of the system. They are responsible for formatting and displaying information for the user and for reading user commands and input. They do not directly access any of the databases except via messages to other processes.
3. *Database read/write processes.* These processes handle the actual storage and retrieval of data for users when authorized to do so by the access control processes. They send data to users or retrieve it from them by messages to the terminal interaction processes.

Access Control Process

The system has three basic access control processes. The master patient index guardian is responsible for the fundamental identification of the patient on whom information is being requested; it makes use of a data structure that contains basic identifying information, such as name and birthdate. The document header guardian is responsible for determining what kinds of additional information are available on that patient and for starting the appropriate process to read or write that information. The user security checker is responsible for determining whether users have the right to perform a particular operation; it is used by both the master patient index guardian and the document header guardian.

Terminal Interaction Processes (TIPS)

In every user interaction with CRISP, communication to and from a user's terminal is performed by terminal interaction processes. None of these processes accesses files directly; instead, all their accesses are made via the access control processes and the reader/writer processes. Depending on requirements, terminal interaction processes may access multiple types of information from multiple data bases.

In order to permit users to write or modify their own terminal interaction processes without potentially compromising system security, it is assumed that no security checking is done in the terminal interaction processes themselves and that all security checking is done by the access control processes. Since a terminal interaction process may accidentally or intentionally alter patient or user identification between messages, every message received from a terminal interaction process must be verified, even if the data contained in it were previously sent to the terminal interaction process by another process.

Since different documents may be stored by using different database systems, and since Terminal Interaction Processes are not permitted to access documents directly, a set of processes are needed to actually perform the database accesses. In CRISP, reader/writer processes perform this function. In order to provide security, however, these processes are not started by the terminal interaction processes directly. Instead, they are spawned by the document header guardian in response to user requests to access a particular database for a particular type-of information. They then communicate via messages with the user program to perform the needed accesses or updates.

There may be multiple reader/writer processes for any particular database if the database must be accessed in different formats or via different access strategies. In particular, different databases may be constructed using different database management systems; thus, it is possible to use CRISP with systems such as AMBASE, TOTAL, or ADABAS. Additionally, the same terminal interaction process may communicate with multiple reader/writer processes. Tables available to the document header guardian are used to decide which process to start in order to satisfy a particular request.

RELATIONSHIP BETWEEN THE DESIGN AND THE CRITERIA

The structure that has been described here effectively packages together data bases with processes that access them. This has two consequences that are useful in meeting the design criteria. First, since the linkage between patients and databases is itself a database, it permits researchers to easily select patients with entries in particular databases without the need to actually access the databases themselves. Second, by establishing new reader/writer processes every time a new type of

information needs to be stored, existing databases do not have to be modified by the addition of new fields or linkages.

In this structure, before a terminal interaction program can begin communicating with a reader/writer process, it must first communicate with the master patient index guardian and the document header guardian to locate the needed information. These access control processes can, therefore, check each interaction both as to the patient and as to the type of information being accessed.

This design is also useful in providing for appropriate storage migration. Individual documents can be migrated without the need to move other data relating to a patient; and, since the patient-database linkage occupies only a small amount of storage, the linkage can be left in place to indicate the existence of the information. When an attempt is made to read or write the document, the document header guardian indicates to the reader/writer process the information's location in off-line storage. Depending on the user's desires, the request may be canceled or provision can be made for retrieval of the document.

A final useful property of the design is that the decomposition into processes provides a useful tool for distributing the system across a network of processors, with different processes located on different processors. To minimize network traffic, one might, for example, locate databases on the processors with terminals most likely to use them. If one of these processors went down, only that database would be unavailable to users of other processors.

REFERENCES

1. Hedlund, J., B. Vieweg, D. Cho., R. Evenson, C. Hickman, R. Holland, S. Vogt, C. Wolf, and J. Wood. *Mental Health Information Systems: A State of the Art Report*. Health Services Research Center/Health-Care Technology Center, University of Missouri, Columbia, 1979.

A concurrency control algorithm in a distributed environment

by PAUL DECITRE

Centre de Recherche Cii-Honeywell Bull
Grenoble, France

ABSTRACT

As a continuation of the POLYPHEME¹ study, the Cii-Honeywell Bull research center has launched a project on co-operating transactional systems with particular attention paid to distributed concurrency control and commitment.

Following the presentation of the application-driven approach being taken, the distributed concurrency control algorithm is described as an improvement of the proposal made by Rosenkrantz, Stearns and Lewis.¹⁴ Salient technical features such as deadlock prevention, wrong aborts, parallel execution, and relation between concurrency control and commitment are detailed. Then the main choices are justified, and the rejected techniques criticized.

INTRODUCTION

Concurrency control is the technique used in multi-user data management systems to make each user feel as if he were alone on the system. The fact that several users use the same data concurrently may cause wrong results to be delivered to the user or wrong data to be stored in the database. These kinds of inconsistency have been widely studied in the literature.⁸

To solve this problem in a centralized system, the working session of each user is chopped into units of work called transactions (or commitment units). All along a transaction execution, the system avoids interferences with the other running transactions in such a way that the results delivered to users and the changes made to the database are equivalent to a serial execution of the same transactions. This property is called "serializability."^{4,18} To enforce the serializability of transaction executions, the locking technique is commonly used.

When a transaction wants to use a piece of data, the system locks it until the transaction has been completed. A transaction that wants to use data already locked must wait for the completion of the transaction owning the lock. To improve the parallelism between transactions two types of locks are introduced. Exclusive access locks are used when a change may occur on the data, shared access locks when no change is anticipated on the data during the transaction. A sharable lock means that a transaction denies, until its end, another

transaction the right to make changes to the accessed data. Several transactions may simultaneously access data as long as they acquire a shared access lock. This locking technique introduces deadlocks in several steps as soon as a transaction acquires resources. If deadlock detection is no longer a problem in centralized systems, distributed deadlock detection mechanisms still pose implementation problems.^{12,13} The main difficulties lie in the delay and in message traffic caused by deadlock detection messages in the case of wait. Recent research in centralized or distributed resource allocation proposes a technique based upon time ordering to ensure concurrency control. This technique avoids deadlocks instead of detecting them. The main idea is to define an a priori ordering of transactions using timestamps (or circulating tickets), then to try to force the ordering of (conflicting) transaction executions to be the same as that of the timestamp. Many propositions of that kind can be found in the literature.^{9, 11, 14, 18}

If there are many propositions for distributed concurrency control algorithms it is still hard to find a good overview of these algorithms and of their best environment. We find classifications and descriptions,^{3,17} but it is difficult to guess for each distributed application scenario which distributed concurrency control algorithm should be chosen. Furthermore, we find few simulations and almost no performance indication on distributed applications using concurrency control.

THE SCOT APPROACH

A major goal of the SCOT project is to select the best solutions to fit the requirements of a given distributed application and to measure on a functional prototype the corresponding behavior. So the first step of the project has been a survey to select an "interesting" distributed application, "interesting" meaning realistic, truly distributed on several computers, representative of a large class of management applications, and of reasonable complexity. We have chosen a distributed banking system. To remain coherent with the real world, we decided to work in close cooperation with a French bank and to select in common one of their computerized applications in which distributed concurrency control would be needed.

At the same time we sought among the proposed distributed concurrency control algorithms the ones that could best fit the requirements of that particular distributed application.

We now are at the end of the second step, which consists of both the specification of distributed concurrency control and the specification of the distributed banking application. The next step for SCOT will be the validation of our algorithms by proving techniques² (using PETRI nets), leading to the implementation of the SCOT system on a set of Level 6 connected through the Transpac network under the DSA protocols. This implementation will be accompanied by a simulation of the rejected distributed concurrency control algorithms.

SOME DEFINITIONS

A "local transaction" is a sequential process running an algorithm (user's code) under the control of a transactional system.

A local transaction can

- compute,
- access data,
- cooperate with other local transactions using the cooperation rules given below, and
- exchange messages with an end user.

A "global transaction" is the result of the cooperation of several local transactions. The cooperation rules are described within the SCOT protocol. These cooperation rules are of one of the three following types:

1. The launching of a remote "son" transaction by another local transaction, providing that these two transactions run for the same global transaction.
2. The inter-transaction synchronization based upon message exchange.
3. The synchronization of the local transaction ends in order to provide the atomic execution of the global transaction (commitment phase).

The "superior" transaction is the initial transaction (the first local transaction of the global transaction), and is connected to the end user. (This rule is justified in the SCOT research report.¹⁶) The transactional system that is in charge of the superior transaction controls the global transaction commitment.

A local transaction is said to be "inferior" if it has been launched by another transaction (father transaction). An inferior transaction can in turn launch another inferior transaction without reporting to its parent. An inferior transaction is subordinate to the superior's decisions in the commitment phase.

A BRIEF DESCRIPTION OF THE SCOT CONCURRENCY CONTROL ALGORITHM

This paper focuses on the concurrency control aspect of the SCOT protocols. The complete description with distributed execution, commitment and error recovery can be found in Scot Group Research Reports no.s 8 and 9.^{16, 17}

The SCOT concurrency control algorithm is derived from the Rosenkrantz et al. proposition,¹⁴ which we suggest improving in the following ways:

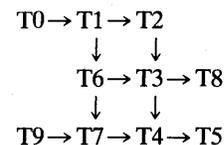
- links with a commitment phase,
- parallel execution of local transactions,
- two kinds of locks (shared and exclusive),
- decrease of wrong aborts number in case of deadlock prevention,
- lessening of the impact of an abort.

A global transaction acquires its resources dynamically as they are claimed by the user's programs representing local transactions. The resources are controlled by locks in the shared memory of each transactional system. These locks are of two types: shared access locks and exclusive access locks. The proper type of lock is obtained through the access primitive to the database. For example, a read may be done with an exclusive lock because an update is anticipated by the programmer. A lock is set until the end of the commitment phase. The lockable resources are local to the transactional system so that the memory lock technique can be used. On the other hand, the fact that locks cannot be set at once on several entities located on independent sites, combined with a dynamic (step by step) resource allocation, may lead to distributed deadlocks.

A deadlock is caused by a chain of waiting global transactions T_i as follows

$$T_0 \rightarrow T_1 \dots T_n \rightarrow T_0$$

We note $T_i \rightarrow T_j$ when T_i waits for a resource owned by T_j . The transaction T_i waits for the transaction T_j if one (or more) of its cooperating local transactions requests a resource owned by one of the cooperating local transactions of T_j . A "chain" of waiting transactions is a subset of connected arrows in a general wait-for graph, as shown in the example below.



One global transaction T_i can wait for several other global transactions in two ways:

1. One of the local transactions of T_i waits for an exclusive access to a resource owned in a shared access mode by several local transactions.
2. Several cooperating local transactions of T_i are simultaneously waiting for other global transactions at different sites. This case is introduced by parallel execution possibility inside a global transaction.

No conflicts can occur between two local transactions of the same global transaction. This kind of multiple Wait is impossible in the Rosenkrantz et al. algorithm because the locks are all exclusive and the parallelism is not allowed inside a global transaction. Rosenkrantz et al. propose to avoid deadlock by

an algorithm using timestamps. Each global transaction is given a unique timestamp in the system. Each local transaction of a global transaction inherits the global transaction timestamp. The timestamps are created in such a way that two global transactions are always related by a strict order according to their timestamps. This order is "close to" a real time order of the events occurring in the distributed system. The timestamps are generated by the Lamport method;¹⁰ the current value of local time is added to each message exchanged between transactional systems.

We have chosen the "WOUND-WAIT" technique: The transaction entering in conflicts with another transaction must "WAIT" for the end of the other if it is younger and "WOUND" it if it is older. The WOUND (a WOUND message broadcast to all the cooperating local transactions of the wounded global transaction) will kill the concurrent transaction only if the concurrent transaction waits for a resource.

The Rosenkrantz algorithm, allowing that a younger transaction waits on an older, allows the building of chains as follows:

$$T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_i$$

with the following relation order on the T_i timestamps

$$TS_0 > TS_1 > TS_2 > \dots > TS_i$$

assuming that $TS_0 \dots TS_i$ are the timestamps of the transactions $T_1 \dots T_i$.

This relation order of the transaction timestamps, being strict, implies that two transactions in the chain are always different.

The WOUND broadcast allows the last transaction of the chain to wait on a transaction T_{i+1} younger than T_i (with the relation $TS_i < TS_{i+1}$). We denote a wait with WOUND by \rightarrow . The Rosenkrantz algorithm allows the chains

$$T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_i \rightarrow T_{i+1}$$

with $TS_0 > TS_1 > \dots > TS_i$ and $TS_i < TS_{i+1}$

Although $T_0 \dots T_{i+1}$ may be on different sites, the control of the chain construction is done without intersite communication except for the Wound broadcast. No portions of the "wait-for graph" are exchanged. The elements $(\rightarrow T)$ and $(\rightarrow T \rightarrow)$ are licit, while the elements $(\rightarrow T \rightarrow)$ and $(\rightarrow T \rightarrow)$ are forbidden.

We propose as an improvement the construction of chains with the elements $(\rightarrow T \rightarrow)$ in order to permit the following:

$$T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_i \rightarrow T_{i+1} \rightarrow \dots \rightarrow T_j$$

with $TS_0 > TS_1 > \dots > TS_i$ and $TS_i < TS_{i+1} < \dots < TS_j$

The elements $(\rightarrow T \rightarrow)$ are still forbidden and this constraint is sufficient to avoid the construction of a loop.

We also propose to transform the Roll-back in case of deadlock prevention into a partial rollback. To break the possible deadlock, the local transaction that owns the resource at the origin of the Wound must rollback after releasing its resources so that the older transaction waiting the younger's resources can get them and run. The rollbacked local transaction, requesting the same resources again, will probably wait for the

end of the older transaction responsible for its wound and rollback. To force the Wounded local transaction to rollback, the site that detects the conflict with Wait will send a rollback command to the "wounder" site" (see the section "Abort Impacts").

TECHNICAL POINTS

Link with the Commit Phase

In a transactional system, concurrent access control is not the only source of inconsistency. The main ones are hardware or software failures in the middle of transaction execution. These failures may leave the database in a transient state, usually invisible. To cope with this difficulty, transactional systems propose the commitment technique, whose main idea is now briefly described.

After an error, the recovery process validates all the updates if a transaction has completed before error; otherwise it undoes the updates. To detect the completion of transaction updates, the transactional system writes a record in a journal at the beginning and after the end of each transaction. When an error occurs, a recovery process analyzes the journal and executes the undo operations for all the transactions started but not completed.

The extension of this technique to the distributed environment is one of the goals of the SCOT project. The complete description of the SCOT commitment protocol may be found in SCOT Group Technical Report #9.¹⁷

The technique used in SCOT to commit a global transaction is to subordinate the commitment of the inferior local transactions to the commitment of the superior transaction.

1. Each inferior warns the superior of its end by sending an END message.
2. When all the END messages are received by the superior it broadcasts a PRE-COMMIT message to all the inferiors.
3. Each inferior writes on a journal a PRE-COMMIT record and sends to the superior an acknowledgment of pre-commit.
4. When all these acknowledgments are received by the superior, it commits (writes a COMMIT record in its journal, resources release). Then it broadcasts a COMMIT message to the inferiors.
5. On the reception of these COMMIT messages, inferiors write a COMMIT record in their journal and release the resources.

The time interval between the writing of PRE-COMMIT record and the writing of COMMIT record is called the "grey zone." When a system crashes, the recovery process detects the local transactions in grey zone and either commits the local transaction if the superior has committed during the crash, or undoes the updates in the other case. To avoid a "domino effect," the resources owned by local transactions in grey zone must be kept locked until the recovery process receives a commitment status from the superior (or from a cooperating local transaction).

In a complete system, which SCOT claims to be, concurrency control and error recovery must be studied simultaneously, because they are closely related. Concurrency control deals with resources allocation and release, while error recovery deals with resources release after a failure. Both techniques provide the atomicity property of transactions (nobody can see transient states).

The resource release must respect simultaneously the conditions required by concurrency control and error recovery.

- The concurrency control condition⁵ is that a "well-formed" transaction begins the resource release after the end of the acquisition of all its resources (no resource can be requested after the first resource release). An easy way to satisfy this decreasing condition is to release resources at the end of the transaction. This condition implies that for each local transaction the resource release begins when the local transaction knows that all the cooperating local transactions are terminated. For an "inferior" transaction this point is the beginning of the grey zone (arrival of the pre-commit message). For a superior transaction this point is the commitment point.
- Error recovery management implies that updated data remain inaccessible until all the updates are successfully completed. That is to say, no lock release for updated data is allowed before the end of the commitment. The locks on updated data must be maintained until the end of the grey zone to avoid a domino effect in case of abort. For an inferior transaction, this point is the end of the grey zone (reception of the commit message). For a superior transaction this point is the commitment point.

The merging of concurrency control and error recovery conditions is straightforward. For an inferior transaction, unmodified data can be released at the beginning of the grey zone. Locks on updated data must be released at the end of the grey zone. For a superior transaction, the resource release of all accessed data is done at the commitment point.

The early unmodified data release improves the data availability and lessens the impact of long waits in grey zone.

Parallel Execution and Concurrency Control

In Rosenkrantz's proposal no parallelism is allowed between cooperating local transactions. This restriction, adopted for the sake of presentation clarity, does not simplify the concurrency control protocol. Furthermore, it implies some artificial complexity in the writing of global transactions performing a long dialogue between two sites.

Partial knowledge of sibling

Parallel execution means that a local transaction can start two or more "son" transactions. The first son and its own sons are unaware of the existence of brothers. This lack of total knowledge of the sibling imposes the forwarding of Wound or Abort messages. When a transactional system wants to broadcast a Wound or Abort message to all the sites running those

local transactions implied in a same global transaction, it broadcasts this message to the known family of the local transaction and inserts the list of addressed sites in the Wound or Abort message. Once received at one of the addressed sites, the message is forwarded to all the sons known at this site but without the list of addressed sites. The address list in the message is increased at each forwarding step with new addressed sites. As a local transaction has only one father, this forwarding transmission is able to reach all the family without double messages. This forwarding technique is slightly more complex than the forwarding technique of the Rosenkrantz's algorithm, where a Wound or Abort message is always forwarded to the unique son or to the father. Nevertheless, broadcasting the Wound or Abort directly to the known list seems to be more efficient in most simple transaction scenarios. The delay implied by this Abort or Wound transmission is shorter.

Several loci of control

Parallel execution means that local transactions are allowed to run simultaneously on several sites. The problem raised by this simultaneity is with the consistency of the decisions taken at each locus. The commitment protocol ensures that if one of the sites decides to abort while another has terminated and is waiting for the commit, both sites will eventually abort the same global transaction. In the same way, one site can decide to enter a wait state while another site decides to abort the same global transaction. The message forwarding technique ensures that this Abort will eventually reach the waiting local transaction.

The low complexity implied by parallel execution is amply compensated by the gains that can be expected in terms of response time or global transaction design effort.

Wrong Abort Due to Deadlock Prevention

A criticism often addressed to the deadlock prevention technique is the possibility of wrong aborts (to prevent a deadlock, a transaction may be aborted without being in a real wait loop). One of the ways to lessen the wrong aborts rate is to set a timeout before the final abort of a wounded transaction, so giving more chances to the wounded transaction to complete. This improvement, simple to implement, will be tested in SCOT. The difficulty of this timeout technique is, as usual, the choice of the timeout value.

However, the wrong abort is a real problem only if its frequency is high. Preliminary simulations of concurrency control systems show that the Wound rate is $\frac{1}{3}$ of the conflict rate in the case of the wound-wait technique. This ratio is due to the fact that it is the younger transaction that waits for the older's end. A second interesting characteristic of wrong aborts is that they occur just after the aborting conflict. There is no delay introduced by a cyclic deadlock detection algorithm. This quick reaction implies a quick restart and so a low impact on the response time. One of the SCOT goals is to offer significant inputs to simulation models allowing a serious comparison of those two techniques.

Parallel versus Sequential Execution of Cooperating Local Transactions

Parallelism has been chosen, as said before, because of the simplicity of this powerful technique. It should be interesting in some cases to use parallelism, although in the SCOT application it is not a requirement. We can imagine global transactions sending the same questions to many sites and waiting for a few replies. In this case the parallelism is a must.

Deadlock Detection versus Deadlock Prevention

Deadlock detection in distributed environment has been studied in various papers.^{7, 12, 13, 19} The most recent proposal is Obermack's algorithm, which provides significant optimization that, using timestamps, lessens the number of deadlock management messages. This algorithm is interesting for having very few wrong aborts and a relatively low traffic overhead due to deadlock detection as compared with what is usually encountered in distributed deadlock detection. We consider it as a good algorithm, combining and improving previous distributed deadlock detection proposals. Nevertheless, some criticisms may be made:

1. It is designed and proven for migrating transactions, with one locus of control at a time; consequently, parallel execution inside a global transaction is impossible.
2. Deadlock detection is made at time intervals whose determination is as difficult as that of the choice of a time-out value.
3. The propagation of "wait-for" information is not broadcast to all the cooperating local transactions concerned by a conflict. The propagation is made by forwarding the "strings" all along the possible cycles. This step-by-step propagation causes the detection delay to increase with the number of elements in a cycle.
4. The wrong deadlock cycles are still possible in a simple version of the algorithm but can be avoided either by waiting some extra periods (to see whether the deadlock cycle persists) or by sending an invitation to abort all along the suspected cycle. Again, both solutions increase the detection delay.
5. The choice of the victim is not described in the IBM research report and needs to be closely studied. In fact, the SCOT Wound-Wait technique aborts the younger transaction, giving priority to the older ones. This policy improves the throughput of the system, since transactions close to their end will survive a conflict while the younger ones, with few resources acquired, will be rolled back. The second advantage of this policy is that it resequences the transactions according to the timestamps order, so diminishing the Wound rate. The same policy may be adapted to the Obermack's algorithm, giving it similar benefits. A side effect of this policy is to eliminate the tricky case where two sites, detecting the same cycle simultaneously, choose two different victims.
6. The partial rollback is not possible, because the site name of the local transaction owning the wait-for resource is absent from the deadlock detection messages.

Each deadlock detector knows only the global transactions names.

One of the SCOT goals is to offer significant inputs to simulation models to allow a serious comparison of the deadlock detection and deadlock prevention techniques.

Our choice for deadlock prevention was justified mainly by the simplicity of the distributed algorithm and the apparent low overhead in messages for deadlock management. The new propositions in deadlock detection should improve the chances of this technique and will confirm the need for serious comparison based on real application measurements.

CONCLUSION

One advantage of the concurrency control in SCOT, as described above, is that it integrates most advanced techniques with a give and take acceptable in an important class of applications. The strong points of the algorithm are the dynamic resource allocation, the low message overhead caused by concurrency control, the parallelism within global transactions, the absence of starvation, and the imbrication with the distributed commitment. The main drawback as compared with deadlock detection technique is the existence of wrong aborts. The performance of the choices will hopefully be confirmed by simulation results and prototype measurements, which are under way.

REFERENCES

1. Adiba, M., J.M. Andrade, P. Decitre, F. Fernandez, Nguyen Gia Toan, "An Experience in Distributed Database System Design and Implementation," *International Symposium on Distributed Databases*. (Paris: North-Holland, 1980), pp. 67-84.
2. Azéma, P., B. Berthomieux, P. Decitre, "The Design and Validation by Petri Nets of a Mechanism for the Invocation of Remote Servers," IFIP Congress, Melbourne, Oct. 1980.
3. Bernstein, Philip A., and Nathan Goodman, "Fundamental Algorithms for Concurrency Control in Distributed Database Systems." Technical Report CCA-80-05, Computer Corporation of America. (Cambridge: Feb. 1980).
4. Bernstein, Philip A., D.M. Shipman, W.S. Wong, "Formal Aspect of Serializability in Database Concurrency Control," *IEEE Transactions on Software Engineering*, Vol. SE.5 (1979) #3.
5. Eswaran, K.P., J. Gray, R.A. Lorie, I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System." Research Report, IBM Research Laboratory, San Jose. November 1976.
6. Gardarin, Georges, and Brigitte Piot, "Detection and Frequency Evaluation of Concurrency Conflicts." *1st European Conf. on Parallel and Distributed Processing*, Toulouse, France, February 1978, pp. 236-261.
7. Gligor, Virgil D., and Susan H. Shattuck, "On Deadlock Detection in Distributed Systems." University of Maryland Computer Science Technical Report 837, December 1979.
8. Gray, J., "Notes on Database Operating Systems." Research Report, IBM Research Laboratory, San Jose, February 1978.
9. Herman, D., and J.P. Verjus, "An Algorithm for Maintaining the Consistency of Multiple Copies," *First International Conference on Distributed Computing Systems*, Huntsville, Alabama, October 1979.
10. Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Vol. 21 #7 (July 1978).
11. Le Lann, Gérard, "Algorithms for Distributed Data-sharing Systems Which Use Tickets," *3rd Berkeley Workshop on Distributed Data Management and Computer Networks*. San Francisco, August 1978, p. 259.
12. Menasce, Daniel, and Richard R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *3rd Berkeley Workshop on Distributed*

-
- Data Management and Computer Networks*. San Francisco, August 1978, pp. 215-232.
13. Obermack, Ron, "Global Deadlock Detection Algorithm." Research Report, IBM Research Laboratory, San Jose, June 1980.
 14. Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis, "A System Level Concurrency Control for Distributed Database Systems," *2nd Berkeley Workshop on Distributed Data Management Computer Networks*. Univ. of California at Berkeley, May 1977, pp. 132-145.
 15. Wilms, Paul. "An Overview of Update Algorithms in Distributed Databases. Formalization with Nutt's Evaluation Nets." SCOT Research Report #7, Centre de recherche CII-Honeywell-Bull, Grenoble, France.
 16. SCOT group. "SCOT, Présentation générale des mécanismes." SCOT Research Report #8, Centre de recherche CII-Honeywell-Bull, Grenoble, France.
 17. SCOT group. "Description du protocole SCOT." Scot Research Report #9, Centre de Recherche CII-Honeywell-Bull, Grenoble, France.
 18. SDD1 Group. *Technical Reports*. Computer Corporation of America, Cambridge, Mass., 1980.
 19. Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering*, Vol. SE 5 (May 1979) #3.

An alternative approach to distributed database updating

by RICHARD J. GREENE

Sperry Univac
Blue Bell, Pennsylvania

ABSTRACT

This paper presents a new updating approach for a fully redundant DDB operating in a transaction-based environment. Existing approaches utilize either a single, migrating controlling process (DBMS) or multiple controlling processes to access the DDB. Updating approaches based on a single, migrating DBMS generally exhibit simplicity of design and lock-based concurrency control. Unfortunately, access to the DDB is usually single-thread. Approaches based on multiple controlling DBMSs generally exhibit complex design and concurrency control but do permit multi-thread access to the DDB. The approach proposed in this paper, called "Cooperative Multi-Thread" (CMT), is based on a single, migrating DBMS and lock-based concurrency control and yet permits multi-thread access of the DDB. The basis of this approach is the time-driven, cooperative scheduling of control migration. Contrast this with the request-driven, competitive scheduling of control migration of the existing approaches in the genre. The advantages of the proposed approach are an economical communications structure, high throughput, flexibility, and predictable performance. This paper presents the technical aspects of CMT approach as well as a quantitative and qualitative appraisal of it.

INTRODUCTION

The introduction of distributed database (DDB) systems into the marketplace seems imminent. Two major conditions support this belief: the technological feasibility of DDB systems and the compatibility between the characteristics of DDB systems and the characteristics of many end-user operational environments. Within the computer industry, two factors contribute to the technological feasibility of DDB systems. First, technological advancements have increased the performance of communications media and decreased the cost of computer and computer-related hardware. One result of this trend has been a proliferation of online, centralized database systems accessed via remote terminal interfaces. In short, the hardware vehicle for DDB systems exists. Second, while applying technological solutions to the information processing needs of industry and government, the computer industry has broadened and deepened its understanding of the needs of these

sectors. Increasingly, end-user needs are driving technological solutions. The similarity between the characteristics of many operational environments and DDB systems suggest a natural matching of information processing problems to solution techniques. For example, the availability of accurate and timely information is often a requisite in meeting organizational goals. Toward this end, online, centralized database systems have played a key role. Nevertheless, access patterns to the database may reflect the geographically and functionally distributed nature of a user's operational environment. In other words, centralization implies that the logical information needs of users and the physical point of data use are also centralized even though this may not be the case. The cost of communication, the degradation of response times due to communication bottlenecks, and the catastrophic impact of hardware/software failure suggest that database users might be better served if the database were reorganized by physically fragmenting it, replicating portions if necessary, and locating the fragments closer to their points of use. In short, the idea is to evolve a distributed database.

A DDB system may offer the user several important advantages over a centralized database system. Fragmentation of the logical database can increase database security and simplify the control of database growth. On the other hand, the physical proximity of the database portion to its point of use reduces communications costs and decreases response times. An added advantage is the decreased impact of failure. In short, DDB systems offer Sperry-Univac an additional technique for achieving availability, reliability, and maintainability in software.

Nevertheless, obtaining the advantages of a DDB system may not be a simple matter. A tenuous balance must be preserved between the abstract model of the DDB system and its realization, while complex interactions among system components confound design decisions. The choice of updating approach can serve as a basis for structuring a DDB system because all subsystem architectures should support it and potential performance depends on it. The general lack of practical experience regarding the implications of the updating approach to the overall system is diminishing with emergence of prototype DDB systems. Such experiences, coupled with increased technical expertise, should provide a firm foundation for the future of DDB systems.

Currently, much attention is being devoted to the devel-

opment of new updating approaches. Yet the relationship between the approach and the user's operational environment is vague.

The purpose of this paper is twofold: to provide a general framework for evolving DDB updating approaches and to present a new approach designed for a transaction-based operational environment.

The remainder of this paper is organized in three parts. First, relevant concepts and terminology are introduced. Next, the new algorithm, Cooperative Multi-Thread (CMT), is described, analyzed, and evaluated. Finally, the conclusion reviews the main points of the paper and proposes future areas of research.

CONCEPTS AND TERMINOLOGY

For the purpose of describing DDB systems, a concise working vocabulary needs to be introduced. A more complete treatment of distributed computing concepts and terminology may be found in LeLann.²

The basic conceptual building block of the DDB system is the process (task). Informally, a process is a computation that can execute concurrently with other processes. Both user processes ("programs") and system processes ("software") consume system resources such as memory space, processor time, peripherals, and information. Processes communicate with each other only via messages. Processes may be grouped to form specific functions such as database management. To illustrate, a user process requesting database access communicates indirectly with the database management function whose process set communicates in turn with each other to perform the desired function.

The notion of distributed control distinguishes the DDB system from the centralized database system. A centralized database system has a unique, omniscient controlling process within the database management function which insures that all processes accessing the logical database receive consistent and identical views of the global database state. A true DDB system has no unique controlling process although there is a single, logical database.

Figure 1 depicts the general framework for designing distributed database updating approaches.

A full discussion of the tree diagram, which is beyond the scope of this paper, can be obtained in Greene.³ Nevertheless, the abbreviated version should clarify the major design decisions addressed by any updating approach. The purpose of the general framework is to provide a conceptual scheme for designing DDB updating approaches. Once derived, the approach is refined into an algorithm which provides the actual details of sharing the DDB.

Essentially, database utilization is viewed as resource sharing. Level one of the tree diagram indicates the key design decisions of an updating approach. The successive levels represent the components of each key decision while the leaf nodes represent the actual choices for each decision. In short, the major design decisions pose four questions:

1. What is the prime performance goal of the approach?
2. How is the resource (granule) defined?

3. What are the constraints on allocation?
4. When are resources allocated and reclaimed?

Finally, the two basic methods used to distribute the logical database, replication and partitioning, distinguish DDB systems. Replication implies that the logical database is distributed by physical redundancy while partitioning implies the opposite. Actually, both types of DDB systems are extremes. Nevertheless, the replication case applies in one form or another to most DDB systems and is the more general case. For these reasons, this paper assumes the fully replicated case.

THE COOPERATIVE MULTI-THREAD ALGORITHM (CMT)

Overview

The major design goals of the CMT Algorithm are simplicity, high throughput, and predictable performance. To effect these, control of the DDB is passed cooperatively in a time-shared fashion from DBMS to DBMS. This time-driven, cooperative migration of DDB control distinguishes CMT in its genre.

Basically, the CMT Algorithm operates as follows. A single DBMS controls DDB access for a specified time quantum during which it services requests in a multi-thread mode and maintains an "update list" reflecting its modifications to the DDB state such as granule allocation and granule modification. At quantum expiration, the controlling DBMS passes both the control token and the update list to its predetermined successor.

The allocation policy of the CMT Algorithm is to allocate all of a user process' required granules in advance. This simplifies concurrency control and deadlock prevention.

The CMT Algorithm makes several assumptions regarding its supporting environment. First, each DBMS should be kept busy by a pool of waiting requests. Second, inter-DBMS communication is based on a logical ring network structure.

In sum, time-driven cooperative control migration is the basis of the CMT Algorithm. Simple in design and capable of high throughput via multi-thread mode, the CMT Algorithm also offers predictable performance: each of n DBMS' may allow DDB accesses every $(n-1)q$ seconds where q is the quantum length.

Description

The CMT Algorithm must perform two related functions: cooperative control migration and DDB state migration. The description of the four steps along with two supportive tables which make up the algorithm follows below.

Two tables, the control and update tables, provide key information needed by the algorithm to perform the two major functions. Resident at each DBMS is the control table specifying the order of control migration (control cycle); it contains integer tuples in the form (i,p,s) where i is a unique DBMS identifier and p and s are the predecessor and successor of DBMS i in the control cycle. In short, DBMS i receives

The Components of an Updating Approach

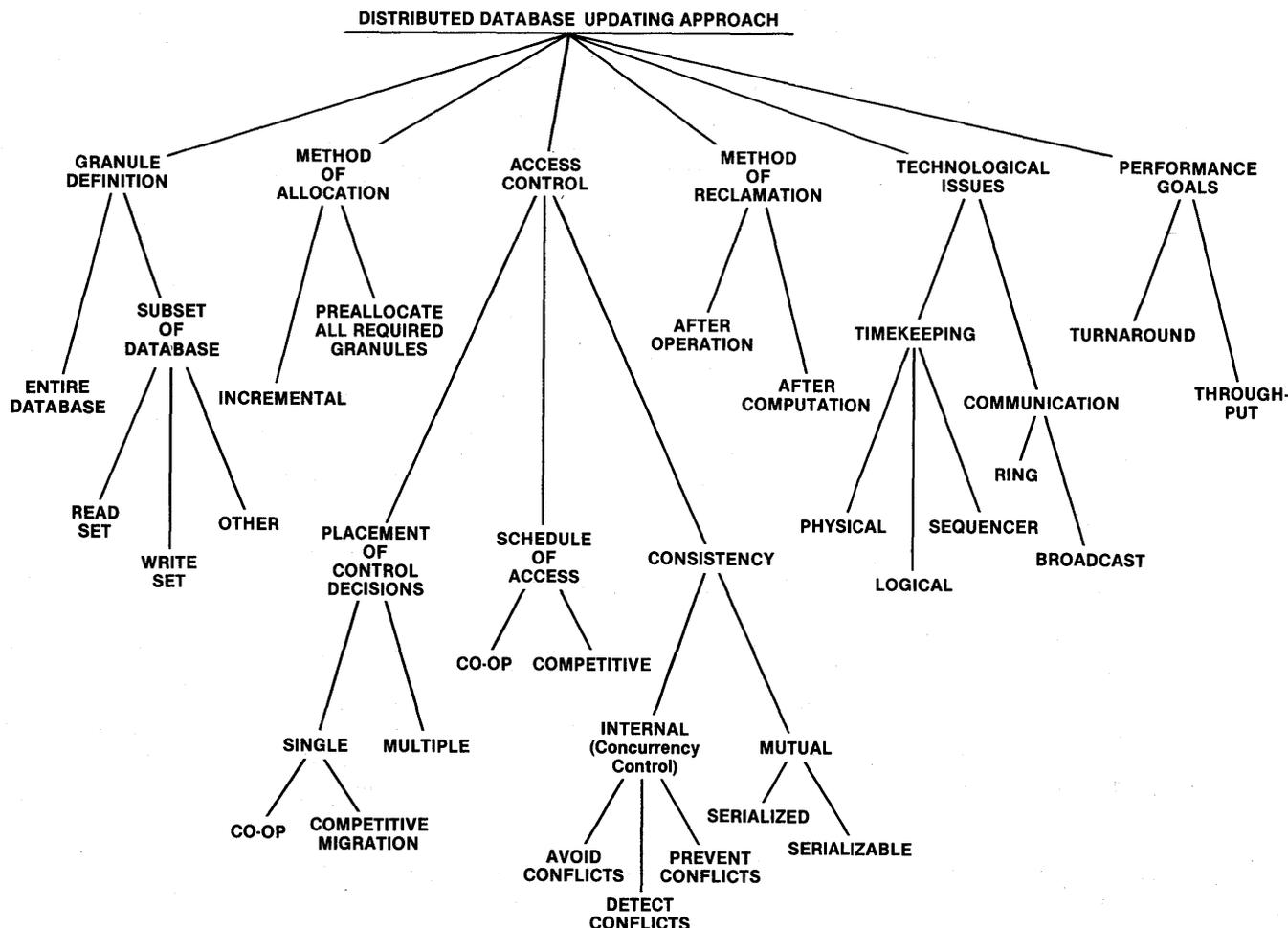


Figure 1—The components of an updating approach

control from DBMS p and passes control to DBMS s . The control table is initialized at system generation time and may be modified by the communication subsystem as contingencies arise. The second table, the update table, indicates the current state of the DDB. This state logically consists of granule locking and modification information. To represent this, the update table consists of two parts, a sublist and a modification sublist, which pass with the control token to a DBMS.

The CMT Algorithm proceeds in four steps: waiting, accounting, user, and termination. These steps implement the ingress and egress of control as well as maintain a current, consistent DDB.

The waiting step is passive. During this step, user requests for DDB access must wait, read operations waiting until the user step and write operations waiting until the accounting step.

The accounting step which prepares the DDB for the user step commences with the advent of the control token and the update table. The controlling DBMS appends the waiting write operations to the modification sublist and applies it to

the replication. This action creates the current state of the granule values of the DDB. Next, the DBMS applies the lock sublist to the replication. This action creates the current state of granule allocation. When these two actions are completed, the controlling DBMS exclusively owns the current logical DDB. Finally, the update table is itself updated; each entry in the update table is associated with an integer sub-cycle number. When a controlling DBMS applies an entry, it increments the entry's sub-cycle number by one. If the sub-cycle number equals the number of DBMSs in the DDB system, the entry has been noted globally and can be removed from the update table. Each DBMS must maintain a list of unresolved user processes whose granule locks span a control cycle. When control passes to the DBMS, these locks are re-initiated.

The user step is the most active step. The controlling DBMS operates in a multi-thread mode and reflects all new allocations and modifications in the update table. Due to the fixed order of control migration, an implementation may require an enhancement to the allocation policy. Basically, it is possible that a user process can be starved by its predecessors of its needed granules. To remedy this, a locking queue of DBMS

identifiers can be associated with the locking sublist. Then a DBMS may allocate a granule only if the granule is unlocked and that DBMS has claimed it. This strengthening of the allocation policy may not be necessary if the granule size is small.

The termination step commences at quantum expiration. At this time, the DBMS initiates the migration of control and DDB state by presenting its successor to the communication subsystem.

In sum, the description of the CMT Algorithm is intended as a guide to further refinement. The flexibility of the algorithm allows several variations which will be mentioned in the conclusion.

Analysis of the CMT Algorithm is based on storage and communications requirements. These two measurements provide a common basis for the comparison of qualitatively different approaches.

The control and update tables compose the key storage requirements. Each DBMS in a DDB system of ' n DBMS' requires a control table of size $3n$ which remains relatively fixed. The size of this table varies slightly as the DDB system adds or deletes a DBMS. The update table's size is more volatile. Although its size may vary at any given time, it is expected that for a fixed request arrival rate and time quantum, the size of the update table will stabilize.

Due to cooperative control migration, the CMT Algorithm exhibits an apparent economy in the number of required inter-DBMS messages: Each lock and modification is circulated $(n-1)$ times, resulting in a minimum of $2n-2$ messages per update. User processes requiring more than one control cycle to complete must recirculate their locks, adding to the minimum figure.

Evaluation of the CMT Algorithm

Where the analysis is quantitative, the evaluation is qualitative. The purpose of the evaluation is to suggest subsystem architectures which promote the performance of the CMT Algorithm and to elucidate several critical performance areas of the algorithm. Following this, a supportive operational environment will be suggested.

The architectures of the nodes, database, and communication subsystems strongly influence the performance of the CMT Algorithm. Node architecture alludes to the organization of local functionality consisting of communications, data processing, and data management functions. Node throughput can be enhanced via parallelism and overlap by functional and geographical distribution at the node level. In actuality, node architecture would consist of a front-end processor for communications processing, a multi-processor for applications processing, and a back-end processor for data management processing. Note that a multi-processor would enhance throughput at the processor level. Database architecture, too, can enhance throughput by supporting rapid access.

A judicious choice of the database model and access methods as well as frequent tuning can all aid in this effort. Finally the architecture of the communication subsystem can enhance throughput if the speed and bandwidth of the communications medium is chosen with the expected communications load in

mind. However, this is no trivial matter; the choice of a suitable bandwidth depends on the average size of the update list, which in turn depends both on the frequency of request arrivals and the quantum length. Adequate information needs to be available to guide this choice. Analytic methods and simulation can also provide a basis for a sound decision. Fortunately, the quantum length can be adjusted to increase performance and compensate for errors in judgment. On the technological side, high speed, very high bandwidth, long distance communications links using microwave radio links, and satellite links expand the performance potential for the CMT Algorithm by providing an efficient means for large data block transmission.⁴

A brief examination of several critical performance areas completes the qualitative analysis. The CMT Algorithm is evaluated below with respect to mutual consistency, deadlock, robustness, equalization of access opportunities, and throughput/turnaround considerations.

Mutual consistency, the "agreement" among the replications at time t denoted $c(t)$, is a customary measure of the "quality" of a DDB system.⁵ Since there is only one logical database, all physical copies of it should, theoretically, be identical.

However, due to non-instantaneous communication media and different methods of message passing, complete agreement among the replications is seldom the case. The measurement of mutual consistency is the number of granules of one replication which differ with their counterparts in all replications. Thus, $c(t) \geq (N+1)/2$ for the CMT Algorithm where $N = \max\{n\}$, n being the number of granules differing from their counterparts in the most recent replication of the DDB. Based on this, the CMT Algorithm provides relatively weak mutual consistency since $c(t) \rightarrow 0$ only if all updating actions cease for a full control cycle. However, this should cause little concern as a user may access only the most recent version of the DDB.

The CMT Algorithm prevents deadlock by forcing users to claim all required granules prior to execution. A complete granule inventory can be stored as part of the program information block either by user declaration or by system software designed for this purpose. A DBMS will allocate a granule only if all required granules are available.

The CMT Algorithm simplifies robustness and recovery in several ways. Failed hosts can be bypassed by modifying the control table. Predecessor update tables aid recovery by furnishing the most recent legal replication. A more elaborate description of robustness and recovery procedures of the CMT Algorithm may be found in Greene.³

The CMT Algorithm equalizes access opportunities at the DBMS level rather than the user process level. By time-sharing the DDB at the DBMS level, the allocation of time increments within a quantum is under local control, responding to local needs. This adds to the overall flexibility of the algorithm.

The topic of throughput has been mentioned; however, the potential of acceptable turnaround exists also. A host can execute a user process regardless of the current location of DDB control because the lock list ensures mutual exclusion of access at a global level.

Finally, the topic of DDB system evolution reveals the true

flexibility of time-driven, cooperative control migration. As the DDB system changes its workloads and/or nodes are added or deleted, appropriate adjustments can be made to the quantum length to stabilize resource utilization and performance.

At this point, the characteristics of an hospitable operating environment for the CMT Algorithm can be suggested. First, frequent and consistent demand sufficient to justify cooperative control migration should exist at each node. This supports the assumption made by cooperative control migration: each DBMS has waiting users. This also aids in stabilizing the size of the update table. Second, to prevent granule locks from being re-initialized, the execution time of user processes should fall within a control cycle. Third, the operating environment should be distributed both functionally and geographically. In other words, the users are performing the same functions in different physical locations but potentially requiring the same data. This characteristic suits the concept of time-sharing a DDB. A transaction-based DDB system might easily exemplify the above characteristics.

CONCLUSION

The emergence of DDB systems in the marketplace is imminent. In order to efficiently serve the target user population, particular attention needs to be given to the updating methodology, which must meet performance expectations within a user's operational environment. Although all current updating approaches "work," each is based on different properties of a DDB system, depicted in the general framework, and offers the user qualitatively different choices. The compatibility of the updating approach with the user's operational environment can be a critical factor in the overall success of the DDB system. An updating approach is compatible with an operational environment only insofar as its fundamental as-

pects approximate the fundamental characteristics of its operational environment.

The design of the CMT Algorithm is oriented to a transaction-based operational environment and a performance goal of high throughput. These two characteristics guide the design of the approach and culminate in an algorithm with a time-driven, cooperative control structure.

The future offers several challenges for future research in DDB systems. Within the CMT Approach, the determination of quantum length and node scheduling seem to be productive areas of future research. For example, if the quantum length were dynamically adjusted, perhaps by heuristic methods, what would be the effect on performance? Similarly, what other node scheduling methods besides round-robin apply to the CMT Approach? Since control migration is cooperative, it is suspected that control table would be constructed using artificial intelligence techniques. With regard to DDB systems in general, simulation studies designed to match updating approaches with specific types of operational environments might provide the necessary basis for intelligent DDB system design.

REFERENCES

1. Bernstein, Phillip A. et al., "Analysis of Serializability in SDD-1: A System for Distributed Databases," *IEEE Transactions on Software Engineering*, IEEE, NY, NY, 1978.
2. LeLann, G., "An Analysis of Different Approaches to Distributed Computing," *Proceedings of the First International Conference on Distributed Computing Systems*, IEEE, NY, NY, 1979.
3. Greene, Richard J., "Updating Approaches for Distributed Databases," Thesis, University of Alabama in Huntsville, 1980.
4. Wilkes, Maurice V., "The Impact of Wide Band Local Area Communication Systems on Distributed Computing," 1979.
5. *Centralized and Distributed Data Base Systems*, Edited by Wesley W. Chu, and Peter P. Chen, IEEE, NY, NY, 1979.

Multibase—integrating heterogeneous distributed database systems*

by JOHN MILES SMITH, PHILIP A. BERNSTEIN, UMESHWAR DAYAL, NATHAN GOODMAN, TERRY LANDERS, KEN W. T. LIN, and EUGENE WONG

Computer Corporation of America
Cambridge, Massachusetts

ABSTRACT

Multibase is a software system for integrating access to pre-existing, heterogeneous, distributed databases. The system suppresses differences of DBMS, language, and data models among the databases and provides users with a unified global schema and a single high-level query language. Autonomy for updating is retained with the local databases. The architecture of Multibase does not require any changes to local databases or DBMSs. There are three principal research goals of the project. The first goal is to develop appropriate language constructs for accessing and integrating heterogeneous databases. The second goal is to discover effective global and local optimization techniques. The final goal is to design methods for handling incompatible data representations and inconsistent data. Currently the project is in the first year of a planned three year effort. This paper describes the basic architecture of Multibase and identifies some of the avenues to be taken in subsequent research.

1. INTRODUCTION

What is Multibase?

The database approach to data processing requires that all of the data relevant to an enterprise be stored in an integrated database. By "integrated," we mean that a single schema (i.e., database description) describes the entire database, that all accesses to the database are expressed relative to that schema, and that such accesses are processed against a single (logical) copy of the database. Unfortunately, in the real world many databases are not integrated. Often, the data relevant to an enterprise is implemented by many indepen-

dent databases, each with its own schema. Such databases are nonintegrated. Furthermore, these databases may be managed by different database management systems (DBMS), perhaps on different hardware. In this case, in addition to being nonintegrated the databases are distributed and heterogeneous. Thus, the real world of nonintegrated, heterogeneous, distributed databases differs greatly from the more ideal world of an integrated database.

Nonintegrated, heterogeneous, distributed databases arise for several reasons. First, many of these databases were created before the benefits of integrated databases were well understood. In those days, total integration was not a principal database design goal. Second, the lack of a central database administrator for some enterprises has made it difficult for independent organizations within an enterprise to produce an integrated database suitable for all of them. Third, the large size of many data processing applications has made distribution a necessity, simply to handle the volume of work. Since integrated distributed DBMSs have not been available, it has been necessary to implement applications on different machines. Since different applications often have different performance and functionality requirements, different DBMSs were often selected to run on these machines to meet these different requirements. Many data processing organizations have experienced these problems, so there are many nonintegrated, heterogeneous, distributed databases in the world.

A principal problem in using databases of this type is that of integrated retrieval. In such databases, each independent database has its own schema, expressed in its own data model, and can be accessed only by its own retrieval language. Since different databases in general have different schemata, different data models, and different retrieval languages, many difficulties arise in formulating and implementing retrieval requests (called queries) that require data from more than one database. These difficulties include the following: resolving incompatibilities between the databases, such as differences of data types and conflicting schema names; resolving inconsistencies between copies of the same information stored in different databases; and transforming a query expressed in the user's language into a set of queries expressed in the many different languages supported by the different sites. Implementing such a query usually consumes months of program-

* This research was jointly supported by the Defense Advanced Research Projects Agency of the Department of Defense and the Naval Electronic Systems Command and was monitored by the Naval Electronic Systems Command under Contract No. N00039-80-C-0402. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Naval Electronic Systems Command or the U.S. Government.

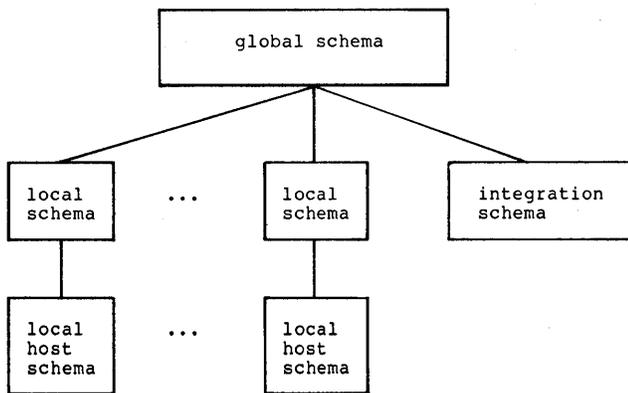


Figure 1—Schema architecture

ming time, making it a very expensive activity. Sometimes, the necessary effort is so great that implementing the query is not feasible at all.

Multibase is a software system that helps integrate non-integrated, heterogeneous, distributed databases. Its main goal is to present the illusion of an integrated database to users without requiring that the database be physically integrated. It accomplishes this by allowing users to view the database through a single global schema and by allowing them to access the data using a high level query language. Queries posed in this language are entirely processed by Multibase as if the database were integrated, homogeneous, and non-distributed. Multibase uses the Functional Data Model¹ to define the global schema, and the language DAPLEX¹ as the high level query language.

Implementation Objectives

There are many approaches to the design of the Multibase system. In deciding which approach to choose, we begin with the following design objectives.

1. **Generality:** we do not want to design an application-specific Multibase system. Instead, we want to provide powerful generalized tools that can be used to integrate various database systems for various applications with a minimum of programming effort.
2. **Extendability:** we want a design that allows expansion of functionality without major modification. There are areas in the Multibase design where substantial research effort is still required, so we must be able to add additional features to the Multibase system as we learn more about the problems.
3. **Compatibility:** we want a design that does not render existing software invalid, because such software represents a very large investment. Thus, we must leave the existing interface to the local DBMS intact.

The proposed architecture of the Multibase system consists of two basic components: a schema design aid and a run-time query processing subsystem. The schema design aid provides tools to the "integrated" database designer to design the glob-

al schema and to define a mapping from the local databases to the global schema. The run-time query processing subsystem then uses the mapping definition to translate global queries into local queries, ensuring that the local queries are executed correctly and efficiently by local DBMSs. The schema design aid is discussed first.

Schema Architecture

The Multibase architecture has three levels of schemata, a global schema (GS) at the top level, an integration schema (IS) and one local schema (LS) per local database at the middle level, and one local host schema (LHS) per local database at the bottom level. These components and their inter-relationships are depicted in Figure 1.

The local host schemata are the original existing schemata defined in local data models and used by the local DBMSs. For example, they can be relational, file, or CODASYL schemata. Each of these LHSs is translated into a local schema (LS) defined in the Functional Data Model. By expressing the LSs in a single data model, higher levels of the system need not be concerned with data model differences among the local DBMSs. In addition, there is an integration schema that describes a database containing information needed for integrating databases. For example, suppose one database records the speed of ships in miles per hour, while the other records it in kilometers per hour. To integrate these two databases, we need information about the mapping between these two scales. This information is stored in the integration database.

The LSs and IS are mapped, via a view mapping, into the global schema (GS). The GS allows users to pose queries against what appears to be a homogeneous and integrated database. Roughly speaking, the LHS to LS mapping provides homogeneity and the LS and IS to GS mapping provides integration. The schema design aid provides tools to the database designer to define LSs, the GS, and the mapping among them and the LHSs.

Query Processing Architecture

The architecture of the run-time query processing subsystem consists of the Multibase software and local DBMSs.

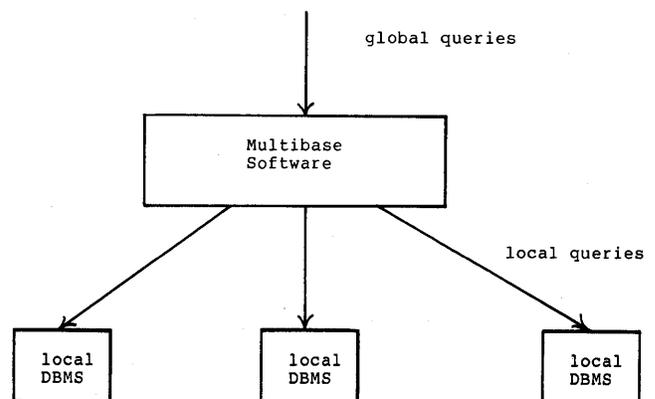


Figure 2—Run-time query processing subsystem

These components and their interrelationships are depicted in Figure 2. The users submit queries over the global schema (called global queries) to the Multibase software, which translates them into subqueries over local schemata (called local queries). These local queries are then sent to local DBMSs to be executed.

Since the global queries are posed against the global schema without any knowledge of the distribution of the data and the availability of "fast access paths," the Multibase software must optimize queries so they can be executed efficiently. In addition, the translation process must also be correct; that is, the local queries must retrieve exactly the information that the original global query requests.

Meeting the Objectives

The proposed architecture meets the objective of generality. The only component of the Multibase system that is customized for the application is the global schema and its mapping definition to the local schemata. The only component of Multibase that is customized for the local DBMSs is the interface software that allows Multibase to communicate with the heterogeneous DBMSs in a single language. These are only small components of the Multibase system. Thus, most of Multibase is neither application-specific nor DBMS-specific. Multibase also meets the objective of compatibility, because local databases are not modified; therefore, existing application programs can still access local databases through local DBMSs. And as the details of the architecture are discussed in later sections, it will become clear that the objective of extendability is also met.

Project Status

The Multibase project is a three-year effort. Within the first two years, the research problems in the system design will be resolved and evaluated, using a "breadboard" implementation of the system. In the final year, a revised design will be developed and implemented in ADA. The ADA version will be made available for experimental testing within the Navy "Command and Control" environment.

It is anticipated that the major research problems are

1. basic architecture of the system,
2. global and local optimization, and
3. handling incompatible data.

At the time of this writing, an architecture has been designed that supports a restricted version of DAPLEX with reasonable efficiency and that can be tailored to handle certain kinds of data incompatibility. This basic architecture is currently being implemented as a breadboard system. Subsequently, research will be devoted to removing the restrictions on DAPLEX and investigating algorithms for processing incompatible data. The breadboard system will then be enhanced to include the new capabilities. This paper describes the basic architecture developed to date.

Organization

The architecture of the Multibase system is expanded in more detail in Section 2. The process of mapping each LHS to a LS and merging LSs into a GS is discussed in Section 3. Section 3 also discusses the problem of data incompatibility and inconsistency. The method by which user queries are translated into efficient local queries is discussed in Section 4. Section 5 is a summary.

2. QUERY PROCESSING ARCHITECTURE

The architecture of the Multibase run-time subsystem consists of

1. a query translator,
2. a query processor,
3. a local database interface (LDI) for each local DBMS, and
4. local DBMSs.

A global query references entity types and functions defined in the global schema. Before it can be processed, it must be translated by the query translator into a query referencing only entity types and functions defined in the local schemata. In other words, the query translator translates a global query over the global schema into a global query over the disjoint union of local schemata. The query processor decomposes the global query over the disjoint union of local schemata into individual local queries over local schemata. The query processor also does query optimization and coordinates the execution of local queries. The LDI translates local queries received from the query processor into queries expressed in the local DML and translates the results of the local queries into a format expected by the query processor. These components and their interrelationships are depicted in Figure 3.

The User Interface

The global schema is expressed in the functional data model.¹ In this data model, a schema is composed of *entity types* and *functions* between entity types. Each entity type contains a set of entities, so functions map entities into entities. Functions can be *single-valued* or *multi-valued*, and can be *partially defined* or *totally defined*.

The functional data model was selected because it embodies the main structures of both the flat file data models, such as the relational model, and the link structured data models, such as CODASYL. Entity types correspond roughly to relations in the relational model or record types in the CODASYL model. Functions correspond to owner-coupled sets in the CODASYL model.

The query language that we use with the functional data model is called DAPLEX. DAPLEX is a high level language that operates on data in the functional data model and is designed to be especially easy to use by end users.

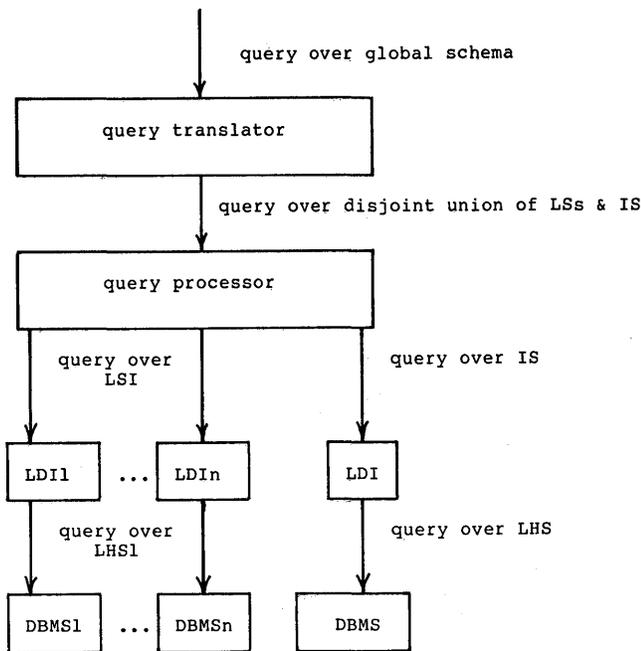


Figure 3—Run-time query processing subsystem

Query Translator

The query translator receives global queries expressed in DAPLEX over the GS and translates them into queries expressed in an internal language over the disjoint union of LSs and IS.

To perform the translation, the query translator must use the mapping that defines how entity types and functions of the GS are constituted from the entity types and functions of the LS and the IS. The query translator uses these mapping definitions to substitute global entity types and global functions in the global query by their mapping definitions. The substitution results in a query containing only entity types and functions of the LSs and the IS. Therefore references by the global query to entities in the GS are now expressed as references to the actual entities at particular sites that implement the global GS. Any extra data needed from the integration database to resolve incompatibilities among LSs is now explicitly referenced in the translated query.

The query produced by the query translator only references data in the LS and the IS. Thus, we can imagine that this query is posed against a database state that is the disjoint union of the LSs together with the IS. This disjoint union is a homogeneous and centralized view of the distributed heterogeneous database.

The language used for defining the mapping between schemata must be compatible with the global DML. Otherwise, it would be awkward to translate the query from the GS to LSs and IS using conventional query modification techniques. (Query modification composes the given query, which is a function from GS states to answer states, with the mapping from LS and IS states to GS states, to produce a query from LS and IS states to answer states.²) Therefore, we propose to

use the same language DAPLEX as both the query and mapping language. The process of constructing the global schema from the local schemata is discussed in Section 3.

Query Processor

The query processor translates a query over the disjoint union of LSs and IS into a *query processing strategy*. This strategy includes the following: a set of queries, each of which is posed against exactly one LS or the IS; a set of "move" operations to ship the results of these queries between the local DBMSs and the query processor; and a set of queries that is executed locally by the query processor to integrate the results of the LS and IS queries. The main goal of this translation is to minimize the total cost of evaluating the query, where cost is measured by local processing time and communication volume.

A query processing strategy is produced in two steps. First, the query is translated into an internal representation called a *query graph*. Using this representation, the query processor isolates those subqueries of the given query (which are essentially subgraphs of the query graph) that can be entirely evaluated at one local DBMS. Thus, the result of the first step is the set of single-site subqueries of the given query.

The second step is to combine the single-site queries with move operations and local queries issued by the query processor. Move operations serve two purposes. First, they are used to gather the results of the single-site queries back to the query processor. These results can be integrated by the query processor by executing a query local to itself. The integrated results may be the answer to the query, in which case they are returned to the user. Second, they may be used as input to other single-site queries. In this case, a move operation is issued to ship the data to the local DBMS that needs it. The method by which single-site queries, move operations, and queries local to the query processor are sequenced to produce a correct and efficient strategy is discussed in Section 4.

Local Database Interface (LDI)

Local queries posed against the LSs are sent by the query processor to the LDIs in an internal format. The LDI translates these local queries into programs in the local DML and programming language over the local host schema (LHS). This translation is optimized to minimize the processing time of the translated query. When the local DBMS uses a high level (i.e., set-at-a-time) language, such as DAPLEX, this translation is fairly direct. However, when the local DBMS uses a low level (i.e., record-at-a-time) language, such as CODASYL DML embedded in COBOL, this translation may be quite complex and may require nontrivial optimization. Translation methods for a file system and CODASYL language are described in Section 4.

To do the translation, the LDI must have information about how entity types and functions in the LS are mapped to objects in the LHS. These mappings are defined using the rules discussed below.

3. SCHEMA INTEGRATION ARCHITECTURE

“Schema Integration” is the process of defining a global schema and its mapping from the existing local schemata. The general architecture of this design process is discussed in this section.

There is one local host schema (LHS) for each local database. Each LHS can be expressed in a relational, CODASYL, or a file language. To merge these LHSs we must convert them into a common data model first. Otherwise, we would be mixing relations from a relational model with record types and set types from a CODASYL model. Thus the first step of schema integration is to translate LHSs into Local Schemata (LS) defined in the Functional Data Model of DAPLEX.

The second step is to merge LSs into a GS. To do this, an integration schema which defines an integration database is often needed. An integration database contains: information about mapping between different scales used by different LSs for the same entity type; statistical information about imprecise data; and other information needed for reconciling inconsistency between copies of the same data stored in different databases. The integration schema and LSs are then used to define a global schema.

The overall architecture of schema integration consists of

- a) a global schema,
- b) a mapping language,
- c) local schemata (LS) and an integration schema (IS),
- d) a mechanized local-to-host schema translator, and
- e) local host schemata (LHS) and local DBMSs.

These components and their interrelationships are depicted in Figure 4. The local host schemata are translated into local schemata by the mechanized local host schema translator, and local schemata and the IS are mapped into the GS by using the mapping language facility.

Mapping between LHS and LS

Since an LHS can be defined in the relational, CODASYL, or file model, how an LHS is mapped into an LS depends on the data model used.

CODASYL model

If an LHS is defined in the CODASYL model, then it consists of record types and set types. The functional data model consists of entity types and functions on entity types. So, to map the LHS into an LS one simply maps record types and set types into entity types and functions respectively.

The concept of record type in the CODASYL model is very similar to that of entity type in the functional data model. A record in the CODASYL model has a record ID, and one or several attributes. The record ID uniquely identifies the record, and the attributes describe properties of the record. Similarly, in the functional data model, an entity is an object of interest, and the functions defined on the entity return values that describe the properties of the entity. Therefore, a

record type corresponds to an entity type, and the attributes of the record type correspond to functions defined on the entity type.

If an attribute of a record type is a key (in CODASYL terminology, a key is the data item(s) declared “NO DUPLICATE ALLOWED”) then the corresponding function must be a totally defined one-to-one mapping. If the attribute is a repeating group (declared to have multiple occurrences in a CODASYL model), then the corresponding function is a set-valued function.

A set type in the CODASYL model is a mapping between an owner record type and one or several member record types. A set type maps an owner record to a set of member records, or, conversely, a set type maps a member record to a unique owner record. Therefore, a set type resembles a function that maps an owner entity to a set of member entities, or, conversely, maps a member entity to a unique owner entity.

In a CODASYL model, a set type implies not only certain semantic information but also the existence of access paths. For example a set type “work-in” between “department” and “employee” record types implies that the employees owned by a department work in that department. But it also implies that there is an access path from a department record to the employee records owned by that department and another access path from each employee record to its own department record. Since the LSs will be used for query optimization, we

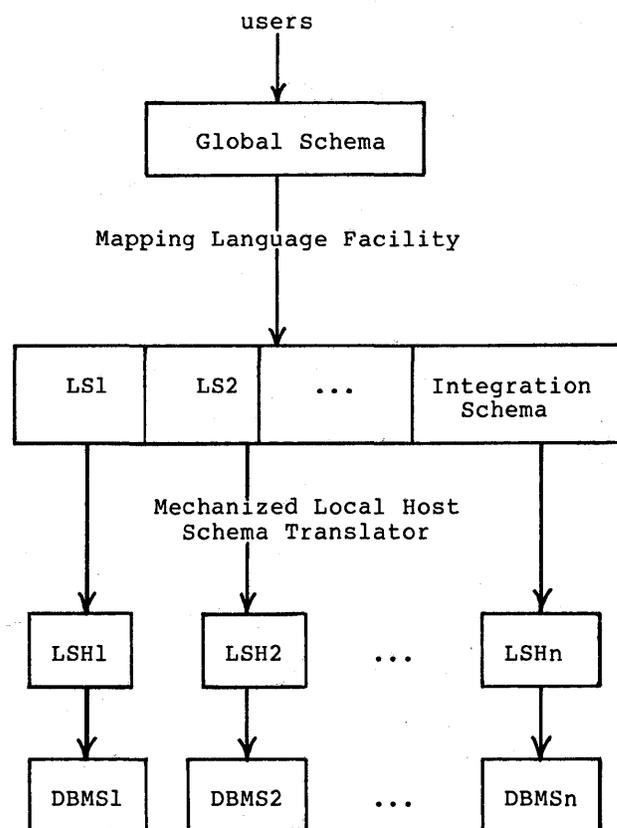


Figure 4—Schema integration architecture

must capture all this access path information in the LSs. Therefore, for each set type in an LHS, not only a set-valued function from the owner entity type to the member entity type, but also a single-valued function from each of the member entity types to the owner entity type must be defined in the corresponding LS.

In a CODASYL model, a record type can be declared to have a "LOCATION MODE CALC USING KEY." This means that an index file is created for the key, and the record type is directly accessible through the indexed key. Therefore, for each record type with "CALC KEY" in the LHS, a system set function of which the domain is the key value and the range is the entity type (corresponding to the record type) must be defined in the LS. This system set function will be used only for query processing optimization. It is not visible to the database designer. Therefore, it cannot be incorporated into the global schema. This restriction is imposed to preserve the data independence of the global schema.

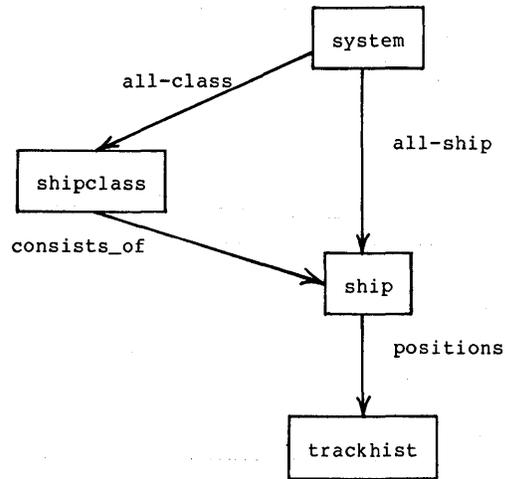
For example, the CODASYL schema shown in Figure 5 is translated into the schema in the functional data model shown in Figure 6. In Figure 6, the inverse of a function F is denoted by "F-inv."

Relational model

A relational database schema consists of a set of relation definitions. To translate a relational LHS to a functional LS we essentially map each relation to an entity type. A tuple of a relation in a relational model is similar to an entity in a functional data model. A tuple is uniquely identified by its primary key and has one or more attributes, just as an entity has one or more functional values. Therefore, to map a relational model LHS into a functional data model LS, for each relation in the LHS an entity type is defined in the LS, and for each attribute of the relation a function is defined on the corresponding entity type. The range of the function is the domain of the attribute. If the attribute is a primary key, then the function must be totally defined and one-to-one. If it is a candidate key, then the function can be partially defined, but it must still be one-to-one. In any case, due to the relational format, the function must be single-valued, not set-valued. For example, the relational LHS shown in Figure 7 is translated into the functional data model LS shown in Figure 8.

File model

A file model consists of record files and indexed fields (keys) in those files. A record file consists of a set of records of the same type, which is similar to the concept of record type in the CODASYL model or a relation in the relational model. To map a file LHS to a functional data model LS, for each record file in LHS a corresponding entity type must be defined in the LS, and for each field of the record file a function must be defined on the entity type. Since a key supports an access path to the record file, for each key of a record file, a system function must be defined whose domain is the key field's entity type and whose range is the entity type corresponding



Shipclass Record		Trackhist Record	
*classname	char(24)	** DTG	char(10)
length	char(6)	speed	char(3)
draft	char(2)	latitude	char(5)
beam	char(3)	longitude	char(6)
displacement	char(5)	course	char(3)
endurance	char(3)		

* primary key
** key within a set

Ship Record	
* UIC	char(6)
VCN	char(5)
name	char(26)
type	char(4)
flag	char(2)
owner	char(2)
hull	char(4)

Figure 5—A CODASYL schema

to the record file. This system function is not visible to the database designer; it is used only for query optimization.

Integration of LSs

To integrate LSs into a global schema, the database designer designs an integration schema that defines an integration database. He then designs a global schema and defines it in terms of the LSs and the Integration Schema by using the view support facility.

An integration database contains information needed for merging entity types and their functions. For example, two entity types, E1 and E2, from two schemata are shown in Figure 9. These two entity types represent information about ships. There are two functions defined on each entity type; one function returns the ship-id of a ship and the other returns the ship-class of the ship. The ship-class of E1 and E2 are coded differently. A sample of entities and their functional values are also shown in Figure 9. To merge E1 and E2 into a single entity type, a uniform code must be defined, and the two existing codes must be mapped to the new code. Definitions of the new code and the mapping function are shown in Figure 10, and a sample of the function is shown in Figure 11.

```

type shipclass is entity
  classname : string(1..24);
  length    : string(1..6);
  draft     : string(1..2);
  beam      : string(1..3);
  displacement : string(1..5);
  endurance : string(1..3);
  consists-of : set of ship;
end entity;

type ship is entity
  UIC : string(1..6);
  VCN : string(1..5);
  name : string(1..26);
  type : string(1..4);
  flag : string(1..2);
  owner : string(1..2);
  hull : string(1..4);
  positions : set of trackhist;
  consists-of_inv : shipclass;
end entity;

type system is entity
  all-class : set of shipclass;
  all-ship : set of ship;
end entity;

type trackhist is entity
  DTG : string(1..10);
  speed : string(1..3);
  latitude : string(1..5);
  longitude : string(1..6);
  course : string(1..3);
  positions_inv : ship;
end entity;

```

Figure 6—A schema in the functional data model

The definitions of the new code and the function are stored in the integration database. A global schema defined on the two local schemata and the integration schema is shown in Figure 12.

As the discussion above indicates, integration of local schemata which are not disjoint involves two activities: merging of entity types and merging of their functions. These activities are discussed in the next section. Two special problems relating to schema integration, the creation of new entity types,

and the integration of incompatible data, are discussed in subsequent sections.

Merging Entity Types and Functions

To merge two entity types, say E1 and E2 in Figure 9, into an entity type, say E in Figure 12, the database designer must first determine whether the set of entities of type E1 is disjoint from the set of entities of type E2. If E1 and E2 are disjoint, then E is simply the union of E1 and E2. If E1 and E2 are not disjoint, then the condition under which two entities from E1 and E2 respectively are identical must be specified. To specify the condition under which entities are identical, entities of E1 and E2 must be able to be identified by their attributes. Therefore, for each entity type to be merged, a function or combination of functions of the entity type must be a primary key. Two entities from two entity types being merged can then

```

type platform is entity
  VesselName : string(1..26);
  class      : string(1..25);
  type       : string(1..6);
  hull       : string(1..6);
  flag       : string(1..2);
  category   : string(1..4);
  PIF        : string(1..4);
  NOSICID    : string(1..8);
  IRCS       : string(1..8);
end entity;

type position is entity
  PIF : string(1..4);
  NOSICID : string(1..8);
  DTG : string(1..10);
  latitude : string(1..5);
  longitude : string(1..6);
  bearing : string(1..3);
  course : string(1..3);
  speed : string(1..3);
end entity;

```

Figure 8—A schema for the functional data model

```

Relation Platform
  VesselName char(26)
  class      char(25)
  type       char(6)
  hull       char(6)
  flag       char(2)
  category   char(4)
  * { PIF char(4)
      NOSICID char(8)
      IRCS char(8)

Relation Position
  * { PIF char(4)
      NOSICID char(8)
      DTG char(10)
      latitude char(5)
      longitude char(6)
      bearing char(3)
      course char(3)
      speed char(3)

```

* primary key

Figure 7—A relational model

```

type E1 is entity
  shipid1 : integer;
  class1 : code1;
end entity;

type E2 is entity
  shipid2 : integer;
  class2 : code2;
end entity;

-----
E1  shipid1  class1
-----
e11  1212    c1
-----
e12  1240    c3
-----
e13  2341    c5
-----

E2  shipid2  class2
-----
e21  3440    d2
-----
e22  3651    d3
-----
e23  4411    d4
-----

```

Figure 9—Local schemata

```

type code is entity
end entity;

Define a new function
  f : (code1 union code2) -> code.

```

Figure 10—Integration database

Sample of function f

code1,code2	c1	c2	c3	c4	c5	d1	d2	d3	d4
code	1	2	3	4	5	6	7	8	9

Figure 11—Sample of function f

```

type E is entity
  shipid : integer;
  class : code;
end entity;

```

Figure 12—Global schema

be specified as identical if and only if they have identical primary key values.

In Figure 13, entity types E1 and E2 (which are assumed to overlap), are merged into an entity type E. The syntax used is a subset of DAPLEX. Notice that "shipid1" and "shipid2" are assumed to be primary keys of E1 and E2 respectively. Further, it is assumed that an E1 entity and an E2 entity are identical if and only if they have the same primary key values.

Creation of a New Entity Type and its Functions

Merging two entity types into a single entity type is a special case of creating a new entity type. Essentially, a new entity type may be created which is a combination of the existing entity types. However, this combination does not create new objects in the database. Rather, it simply presents many existing objects of different types as objects of a single type to the global schema users. Properties of the new global entities are simply those that previously existed in the local schemata.

However, in some cases, a database designer may want to design a more sophisticated global schema in which new (virtual) objects derive their properties (attributes) from many dissimilar existing objects. An example is used to illustrate this process, and general principles can be drawn from the example.

```

type E is entity
  shipid : integer;
  class : code;
end entity;

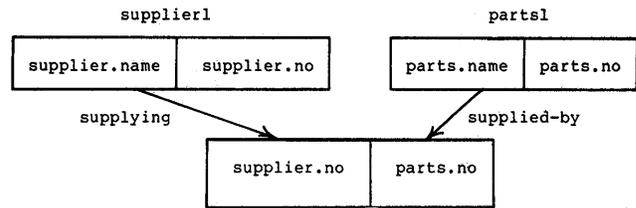
for each x in E1 where not (shipid1(x) isin
                           shipid2(E2))
loop
  create new E(shipid => shipid1(x)
               class => f(class1(x)));
end loop;

for each x in E2
loop
  create new E(shipid => shipid2(x),
               class => f(class2(x)));
end loop;

```

Figure 13—The mapping definition of entity type E

Local Schema 1:



```

type supplier1 is entity
  sname : string(30);
  sno : integer;
  supplying : set of supply1;
end entity;

```

```

type parts1 is entity
  pname : string(15);
  pno : integer;
  supplied-by : set of supply1;
end entity;

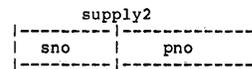
```

```

type supply1 is entity
  sno : integer;
  pno : integer;
end entity;

```

Local schema 2:



```

type supply2 is entity
  sno : integer;
  pno : integer;
end entity;

```

Figure 14—Two local schemata

Suppose a global schema with two entity types, "supplier" and "parts," is to be designed from two local schemata shown in Figure 14. The global schema must capture all the information contained in both schemata. Notice that in the second schema, "supplier" and "parts" entities do not exist, but their existence is implied by the presence of supplier numbers and part numbers: "sno" and "pno." To capture this information, virtual "supplier" and "parts" entities corresponding to those "sno" and "pno" must be created in the global schema. A definition of the global schema is shown in Figure 15. Notice that in the definition primary keys "supplier.no" and "parts.no" are used to map the new entities to existing entities in the first schema and the implied entities in the second schema.

Data Incompatibility

Several sources of data incompatibility are discussed in this section. The objective of the discussion is to show how the proposed architecture allows us to incorporate our present understanding of incompatible data into Multibase. The details of solutions to the problem are to be fully investigated later in the project.

Some sources of data imprecision are:

1. *Scale difference.* For example, in one database four values (cold, cool, warm, hot) are used to classify climates

```

type supplier is entity
  sno      : integer;
  supplying : set of parts;
end entity;

type parts is entity
  name: string(15);
  no  : integer;
end entity;

for each x in (sno(supplier1) union sno(supplier2))
loop
  create supplier (sno => x);
end loop;

for each y in (pno(parts1) union pno(parts2))
loop
  create parts (pno => y);
end loop;

for each s in supplier loop
supplying(s) := (p in parts where (for some y1 in supply1:
sno(s) = sno(y1) and pno(p)= pno(y1)) or
(for some y2 in supply2 :
sno(s) = sno(y2) and pno(p) = pno(y2)));
end loop;

```

Figure 15—A global schema

of cities, while in another database the average temperatures in Fahrenheit may be recorded.

2. *Level of Abstraction.* For example, in one database “labor cost” and “material cost” may be recorded separately, while in another they are combined into “total cost.” Another example is recording an employee’s “average salary” instead of his or her “salary history” for the previous five years.
3. *Inconsistency Among Copies of the Same Information.* Certain information about an entity may appear in several databases, and the values may be different due to timing, errors, obsolescence, etc.

There are many other sources of data incompatibility. Data incompatibility must be resolved if different databases are to be integrated. The architecture of schema integration developed previously can be extended to handle the problem.

Let E1 and E2 be two entity types, and f1 and f2 be functions defined on E1 and E2 respectively. If E1 and E2 have been merged into an entity type E, then f1 and f2 can be merged into the function f defined on E as follows,

$$\begin{aligned}
 f(e) &= T1(f1(e)) && \text{if } e \text{ in } E1 - (E1 \text{ intersect } E2) \\
 &T2(f2(e)) && \text{if } e \text{ in } E2 - (E1 \text{ intersect } E2) \\
 &g(f1(e), f2(e)) && \text{if } e \text{ in } (E1 \text{ intersect } E2)
 \end{aligned}$$

The transformations T1 and T2 are typically used to map the ranges of f1 and f2 into a common range as discussed in the section “Merging Entity Types and Functions.” On the other hand, the function g is used to reconcile any inconsistencies between the values of f1 and f2 over the same entity. Typically, g will involve accessing data described in the integration schema.

For example, in Figure 16, the entity types E4 and E5 are merged into the entity type E6 by using functions IS2 and IS3 of the integration database. In the figure, the data values of the entities and functions are shown in tabular form. In this example, T1 and T2 transform the climate of cities from two

different scales, (cold, cool, warm, hot) and Fahrenheit, into a unified scale (temperature range, probability) by combining E4 with IS2 and E5 with IS3. The function g could return all the (temperature range, probability) pairs from the two databases without any further processing, as is shown in Figure 16.

Alternatively, g could use some statistical technique to process sets of (Temp range, probability) pairs, and return a simpler but descriptive summary of those pairs. For example, the function g could return the average value and the standard deviation of the distribution represented by these pairs; it can make statistical estimation and return a confidence interval; or it can do time series analysis and return information about the spectral function.

The above examples are merely illustrative of potential data integration problems and their solutions. More complete approaches to the problem will be fully investigated later in the project.

4. RUN-TIME QUERY PROCESSING SUBSYSTEM

Overall Architecture

Now we will show how the schema mappings developed during schema integration are utilized to drive query processing over the global schema. As we explained in Section 2, the run-time subsystem consists of a query translator and a query processor. Here we will expand these two components in further detail.

A “Global Database Manager” (GDM) is that part of the Multibase System which consists of the query translator, and the query processor. A query over the global schema is normally sent to the nearest site that has a Global Database Manager (GDM). There may be one or more GDMs in a Multibase system. A GDM stores a copy of global schema,

E4 (of LS1)		IS2 (of integration database)		
city1	climate	climate	range of temp	probability
Boston	cold	cold	0 - 20 F	20%
Norfolk	cool	cold	20 - 40 F	40%
Dallas	warm	cold	40 - 60 F	25%
Miami	hot	cold	60 - 80 F	10%
...	...	cold	80 - 100F	5%
		cool	0 - 20 F	10%
		cool	20 - 40 F	20%
	

E5 (of LS2)		IS3 (of integration database)		
city2	mean temp	mean temp	range of temp	probability
Denver	52 F	52 F	0 - 20 F	20%
Chicago	54 F	52 F	20 - 40 F	35%
Los Ang	75 F	52 F	40 - 60 F	30%
...

E6 (of global schema)		
city	temp range	probability
Boston	0 - 20 F	20%
Boston	20 - 40 F	40%
...

Figure 16—Example of data incompatibility

local schemata, integration schema, and the mapping definitions among them. It uses this information to parse, translate, and decompose queries over the global schema into local queries over local schemata, and coordinates execution of the local queries. The structure of a GDM and its interface with local DBMSs is shown in Figure 17.

A query expressed in DAPLEX over the global schema is first parsed by the parser and a parse tree is generated. Components of the parse tree, which are entities and functions of the global schema, are then replaced by their corresponding definitions, which are expressed in terms of the local schemata LSs. The result is a parse tree consisting of entities and functions of the local schemata. The parser is part of the query translator.

The parse tree is then simplified to eliminate the inefficient boolean components. For example, the boolean expression $(a > 5) \text{ or } (a < 20)$ is reduced to "true," and $(a > 5) \text{ and } (a < 2)$ is reduced to "false." The query simplifier is also part of the query translator.

The parse tree is then decomposed by the decomposer into subtrees. Each subtree represents a local query referencing only entities and functions of a single local schema.

The "ACCESS PLANNER" transforms the local queries into "data movement" and "local processing" steps. Depending on the memory size and processing power of each individual site, and the capacity of the communication channels, the "ACCESS PLANNER" may move data and distribute the computing load among several sites, or it may move

data to a central site which has large memory and computing power and do most of the processing there. In doing this planning, the "ACCESS PLANNER" tries to produce steps which minimize the cost of processing the query. The meaning of "cost" depends on the individual systems being integrated. It may mean the amount of data moved between sites, or the amount of processing time.

The execution of the access plan is coordinated by the "EXECUTION STRATEGIST." It sequences the steps of the access plan and it makes sure that the data needed by a step are there before the step is initiated.

The "EXECUTION STRATEGIST" communicates with local DBMSs through the Local Database Interface (LDI). The LDIs receive "data move" and "local processing" steps from the "EXECUTION STRATEGIST," translate these steps into programs in the local query language or Data Manipulation Language (DML), or call local routines to process these steps, and translate the results of these steps into the format expected by the "EXECUTION STRATEGIST." The LDI may reside in a GDM if the local site does not have enough memory or cpu power; otherwise it resides with the individual local DBMS at the local site.

The query processor to be described in this section is oriented towards the initial breadboard system. It is designed to handle restricted versions of the user interface language and view mapping language with reasonable efficiency. Subsequent research is needed to extend the query processor to efficiently handle the unrestricted languages.

Within the "Query Processor," the database is modelled as a collection of *entity types* and *links*. A link L from entity type R to entity type S is a function from entities of S to entities of R; S is called the *owner* entity type and R is called the *member* entity type relative to L. We assume that if L links R to S, then L, R, and S are all stored at the same site. We also assume that there is a database schema describing the entity types and links of the database.

We will sketch the Multibase query processing strategy in three steps. First, we define the set of queries that can be posed. Second, we define the set of basic operations that Multibase is capable of executing. Third, we describe how to translate a query into a sequence of basic operations that solve the query. Finally, we describe how to translate a local query posed over a CODASYL local host schema into a program in a low level Data Manipulation Language.

Queries

A *query* consists of a target list and a qualification. A *target list* consists of a set of *function terms* of the form $A(R)$ where R is an entity type and A is a non-link function of R. A *qualification* is a conjunction of selection clauses, join clauses, and link clauses. A *selection clause* is a formula of the form $(A(R) \text{ op } k)$ where $A(R)$ is a function term, op is one of $\{=, \leq, <, >, \geq, \neq\}$ and k is a constant. A *join clause* is a formula of the form $(A(R) = B(S))$ where $A(R)$ and $B(S)$ are function terms. A *link clause* is a formula of the form $(L(R) = S)$ where L is a link from R to S.

Let r and s be entities in R and S respectively. We say that

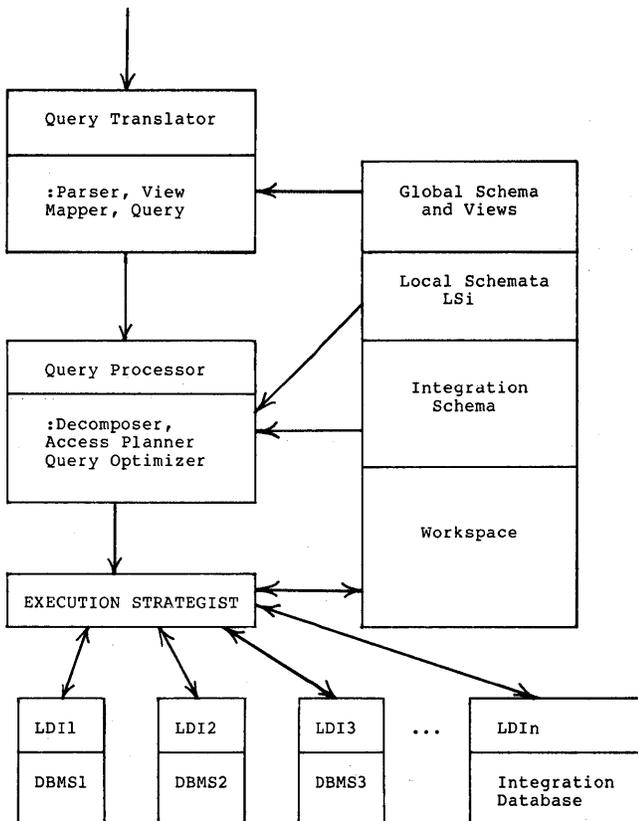


Figure 17—Run time query processing subsystem

r satisfies the selection clause $(A(R) \text{ op } k)$ if the A -value of r is op -related to k (i.e., $(A(r) \text{ op } k)$). We say that r and s satisfy the join clause $(A(R) = B(S))$ if the A -value of r equals the B -value of s (i.e., $A(r) = B(s)$). And we say that r and s satisfy the link clause $L(R) = S$ if L connects r and s (i.e., $L(r) = s$).

Let R_1, \dots, R_n be the entity types referenced by qualification q , and let r_1, \dots, r_n be entities in R_1, \dots, R_n respectively. We say that r_1, \dots, r_n satisfy the qualification q if r_1, \dots, r_n satisfy all of the clauses of q .

Let Q be a query consisting of target list $T = ((A_{j1}(R_{i1}), \dots, A_{jm}(R_{im})))$ and qualification q . Let R_1, \dots, R_n be the entity types referenced in T and q . The answer to Q is the set of all tuples of the form $((A_{j1}(r_{i1}), \dots, A_{jm}(r_{im})))$ such that r_1, \dots, r_n are in R_1, \dots, R_n (respectively) and r_1, \dots, r_n satisfy q . Given a database R_1, \dots, R_n and a query Q , our goal is to compute the answer to Q efficiently.

The subset of DAPLEX that we have just described makes the following simplifications:

1. Set expressions in range predicates and qualifications have been "flattened out," and quantifiers eliminated. This allows us to utilize existing view algorithms for relational databases. Further research will be devoted to handling the novel aspects of view processing in the DAPLEX functional model.
2. The type-subtype hierarchy is not explicitly handled. This hierarchy will be useful in the schema integration step. However, the mechanics of interpreting queries against the hierarchy require further research.

A query graph $QG(N, E)$ is an undirected labelled graph that represents a query Q . The nodes, N , of QG are the entity types referenced in Q . Each node is labelled by the entity type name of the node, the non-link functions of the entity type that appear in the target list, and the selection clauses of Q 's qualification that reference the entity type. The edge set E of QG contains one edge (R, S) for each join clause or link clause that references R and S . Each edge is labelled by its corresponding clause(s).

A query is called *natural* if (a) join clauses are of the form $(A(R) = A(S))$, that is, the functions referenced in both terms of a join clause have the same name; and (b) if A is a non-link function of two entity types R and S , then $A(R)$ and $A(S)$ are "connected" by a sequence of join clauses. There is a simple and efficient algorithm that, given a database description and a query Q , renames the functions of the entity types where necessary to produce an equivalent natural query Q' ; Q and Q' are equivalent in the sense that they produce the same answer for any database state (up to the renaming of fields). We will therefore assume, without the loss of generality, that our queries are natural. Given that we deal only with natural queries, the edge labels corresponding to join clauses are unnecessary. Also target lists need only contain function names, instead of function terms.

Given a join clause $(A(R) = A(S))$ and a selection clause $(A(R) \text{ op } k)$, we can deduce that $(S(A) \text{ op } k)$. We assume that the qualification of each query is augmented by all clauses that can be deduced in this way. A simple and efficient transitive closure algorithm is sufficient for performing such deductions.

Basic operations

There are three types of sites in the breadboard Multibase: File, CODASYL, and GDM. Each type of site is capable of executing a different set of basic operations. This section describes these basic operations.

1. *File Select*. If record type R is stored at a File site S , then the only operation that can be applied to R at S is a selection of the form

$$R[(A_1 = k_1) \text{ and } (A_2 = k_2) \text{ and } \dots \text{ and } (A_n = k_n)].$$

The result of the selection is a record type consisting of the set of all records r in R such that $r[A_i] = k_i$ for $i = 1, \dots, n$; this result is always transmitted to the GDM.

2. *File Semijoin*. In principle, File select can be generalized into File semijoin by performing selections iteratively. Let R be a File file and S a GDM file, and suppose A_1, \dots, A_n are fields of R and S . Then the semijoin of R by S on A_1, \dots, A_n , denoted $R[A_1, \dots, A_n]S$, equals $\{r \text{ in } R \mid (\text{there exist } s \text{ in } S) (r.A_1 = s.A_1 \dots r.A_n = s.A_n)\}$.

This can be computed by the following program.

```
Result: = 0;
for each s in S
loop
  k1: = s.A1, ...; kn: = s.An;
  Result: = Result  $\cup$  R[(A1 = k1) ...
    (An = kn)];
end loop;
```

In practice, this operation may place an unacceptable load on the File system and hence may not be usable.

3. *CODASYL tree queries*. The basic operation that can be performed at a CODASYL site S is to solve a natural tree query (defined below), returning the result to the GDM. A *natural tree query* Q at site S has two properties: (1) All record types referenced in Q must be stored at S . (2) Let Q' be Q minus its join clauses (i.e., all clauses of Q' are selections or links), and let QG' be the query graph of Q' ; then QG' must be a tree.

To solve a tree query Q using CODASYL DML, one essentially expands the cartesian product of the record types referenced by Q and evaluates the qualification on each element of the cartesian product. We describe how this cartesian product can be systematically generated in the section "Processing CODASYL Tree Queries."

4. *CODASYL Tree Semijoins*. The preceding operation can be generalized into a semijoin-like operation. Let Q be a CODASYL tree query and S a GDM record type, and suppose A_1, \dots, A_n are fields of S and fields of record types of Q . Let Q' have the same qualification as Q , and the target list augmented by A_1, \dots, A_n . Finally, let R' be the result of Q' . The semijoin of Q by S on A_1, \dots, A_n , denoted $Q < A_1, \dots, A_n$, equals

$$\{r' \text{ in } R' \mid (\text{there exist } s \text{ in } S) (r'.A_2 = s.A_2) \dots (r'.A_n = s.A_n)\}.$$

This can be computed as follows. Suppose A_1, \dots, A_n are fields of R_1, \dots, R_n respectively where R_1, \dots, R_n are record types of Q . (R_1, \dots, R_n need not be distinct.) Augment the qualification of Q' by adding the clauses $(R_1.A_1 = k_1) \dots (R_n.A_n = k_n)$. And execute the following program.

```
Result: = 0;
for each s in S loop
  k1: = s.A1; ...; kn: = s.An;
  Result: = Result  $\cup$  Q';
end loop;
```

5. *GDM Queries.* The GDM can process any natural query Q provided (1) all entity types referenced in Q are stored at the GDM, and (2) Q contains no link clauses. Suppose Q references entity types R_1, \dots, R_n . Q is processed by constructing a request to the local DBMS (the Datacomputer for the initial breadboard system) of the form:

```
for each r1 in R1 where (selection clauses on R1)
for each r2 in R2 where (selection clauses on R2)
  and (join clauses on R1 and R2)
.
.
.
for each rn in Rn where (selection clauses on Rn)
  and (join clauses on R1 and Rn)
  and (join clauses on R2 and Rn)
.
.
.
and (join clauses on Rn-1 and Rn).
print (target list).
```

It is important that the "for" statements be in a "reasonable" order for performance reasons. Optimization techniques developed by Wong for the SDD-1 DM³ are directly applicable.

Query Decomposition

To solve a query Q , we must decompose it into a sequence of basic operations. Our basic strategy is to find subqueries of Q that can be entirely solved at File and CODASYL sites, move the results of these subqueries to the GDM, and solve the remainder of the query at the GDM.

To follow this strategy, we must isolate File and CODASYL subqueries of Q . File subqueries are easy to find. We simply find entity types in Q that are stored at File sites. For each such entity type R , we produce a subquery consisting of the selection clauses on R .

Let QG be the query graph of Q . To find CODASYL subqueries, we begin by deleting from QG all entity types not stored at a CODASYL site and all join clauses. Each connected component of the resulting graph includes entity types and links that are stored at the same site, because no link can connect two entity types stored at different sites (c.f., the section on "Overall Architecture"). If a connected component is a tree, then it corresponds to a tree query and can be solved by the CODASYL site. If it has a cycle, then it must

be further decomposed into two or more tree queries. (In the breadboard version of Multibase, we will only handle queries whose CODASYL subqueries are tree queries; if some CODASYL subquery is cyclic, the query cannot be processed.)

Having extracted the File and CODASYL subqueries, we must now choose an order for these subqueries to be executed. As a first-cut solution, we propose to solve all File and CODASYL subqueries before processing the results of any of these subqueries at the GDM. This strategy will be an especially poor performer if a File or CODASYL subquery has no selection clauses. For such cases, we recommend use of File and CODASYL semijoin operations, so that the results of some subqueries can be used to reduce the cost of other subqueries. However, this tactic brings us into the realm of new query optimization algorithms and will require further research.

Processing CODASYL Tree Queries

Let Q be a CODASYL tree query and QG its tree. The following algorithm compiles Q into a program that solves Q . The program contains statements of the form:

1. *for r in set(s) loop ... end loop* ; where S owns R via set ;
2. $r := \text{set_inv}(s)$; where R owns S via set. Note that set_inv is the inverse function of set and is always a function.

Algorithm

1. Do a pre-order traversal of QG . The result is a list of the nodes of QG . Call this list P .
2. Let R and S be nodes of QG ; with R the parent of S .

Cases

 - R is the root of QG ; replace " R " by "*for r in R loop*" in P .
 - R owns S : replace " S " by "*for s in set(r)*" in P .
 - S owns R : replace " S " by " $s := \text{set_inv}(r)$ " in P .
3. Push loop independent assignments up as high as possible.
4. Add an "output (target list)" statement, add selections, and joins as high as possible, tack on enough *ends* to balance the *fors*.

As an example let QG be the query graph of Figure 18.

1. Preorder traversal: R, S, T, U, V .
2. *for r in R loop*
 for s in L1(r) loop
 $t := L2_inv(r)$
 for u in L3(t) loop
 $v := L4_inv(t)$
3. Push up T and V ; add an output statement; add *ends* to balance the *fors*.
 for r in R loop
 $t := L2_inv(r)$;
 $v := L4_inv(r)$
 for s in L1(r) loop

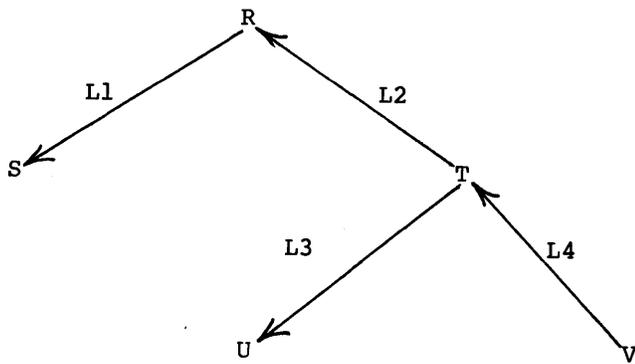


Figure 18—A query graph

```

for u in L3(t) loop
  output (target list);
end loop;
end loop;
end loop;

```

5. SUMMARY

This report describes the architecture of the Multibase system. Details of the components of the architecture to be

implemented in the initial breadboard version are also described. Although additional research is required to fill in the details of optimization and incompatible data handling, the architecture already contains several innovative ideas in integrating distributed heterogeneous databases. These include the following:

1. the idea of using an integration database to resolve data incompatibility;
2. the idea of using a mapping language to uniformly define the global schema in terms of the local schemata and the integration schema; and
3. the idea of using query modification and query graph decomposition to transform a global query into local queries and queries over the integration database.

REFERENCES

1. Shipman, D., "The Functional Data Model and the Data Language DAPLEX", *SIGMOD 79*, Boston, MA, 1979.
2. Stonebraker, M.R.: "Implementation of Integrity Constraints and Views by Query Modifications." *Proc. ACM-SIGMOD Conf.*, San Jose, CA, 1975, pp. 65-78.
3. Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," *1977 Berkeley Workshop on Distributed Data Management and Computer Networks*, Univ. of CA, Berkeley, CA, May 1977.

Architecture of a distributed database information resource

by JAMES R. SWAGER
Honeywell, Inc.
McLean, Virginia

ABSTRACT

In today's world, it is becoming quite clear that database management systems form a very important element in a "Distributed Information Resource" environment; however, there are other components, and the purpose of this paper is to present all of the components in a unified system.

Creation of a distributed information resource implies that a number of hardware and software components are to be designed and integrated into a controlled environment. These components include one or more database management systems, a user language interface, a data/dictionary/directory catalog, a transaction controller, and a data input/output control module.

The inclusion of automated office facilities: word/text processing, electronic mail/message, computer-assisted inquiry, etc., in such a distributed processing environment increases the requirements for careful planning and control.¹

This paper describes the various system components and demonstrates the integration of them with the International Organization for Standards (ISO)² communications architecture and a data storage and retrieval architecture (DSRA). The seven layers of the DSRA are defined and certain functions are assigned to each. In addition, the possible placement of the DSRA is presented in two different scenarios.

INTRODUCTION

Distributed database management systems are one component of a number of components in an enterprise. I have written this paper to provide a better understanding of the enterprise environment that is referred to as a "Distributed Information Resource."

As the definition of distributed information resource becomes more clear and work stations themselves become more distinct, more and more systems will be developed that take into account the requirements of the user. Whether the user wants transparency, data sharing, data transfer, process transfer, or a combination of strategic, managerial and operational reporting, he/she should take into account the following environmental constraints:

- data communications,
- data storage and retrieval,
- meta data,

- user language support,
- process and resource management,
- information representation,
- system management,
- integrity, and
- security

Vendors address these and other issues in different perspectives. The challenge arises when a user's needs evolve so as to require the integration of the "communications" with cooperative processing of the multiple vendor's software and hardware.

This paper addresses the current considerations of the Database Interface with Distributed Systems Task Group³ for integrating these components and requirements.

FUNCTIONAL FRAMEWORK

Creation of a distributed information resource requires that a number of hardware and software components be designed and integrated into a controlled environment. The required features for this environment, whether remote or centralized, could be categorized as: (1) communications from the user or processor to the data; (2) tools for the user to manipulate the data; (3) methodology for storage and retrieval of the data.

For the category of communications, the form of communications architecture should follow the ISO seven layer architecture. These layers are physical, link, network, transport, session, presentation and application. They are portrayed on the left-hand side of Figure 1. In addition, you will notice that the communications device, such as a terminal, is connected to the physical layer.

The tools for manipulation of the data include host languages, queries, report generators and data dictionaries. These are represented by the middle section of Figure 1.

The storage and retrieval of data involves database management systems and access methods that handle the physical and logical data storage process. These are represented by the right-hand side of Figure 1.

System Components

By breaking the functional framework (Figure 1) into three categories of system components, the seven components in

Figure 2 can be derived. The Transaction Controller (TR) performs the data communications functions. The User Language Interface (ULI) and the Data Dictionary/Directory/Catalog (DD/D/C) are the tools for data manipulation functions. The Data Base Management Systems (DBMS), Data Base (DB) and the External Data Storage (non-DBMS) provide the methodology for data storage/retrieval. The Data Input/Output Controller (DIOC) acts as a traffic manager for data that is retrieved and stored within the node, regardless of whether the request is external or internal to the node. Thus, you can comprehend that the DIOC is considered the most crucial element because of its function to link the other elements into a unified system of components.

The definition and functions of each component are given below.

Transaction controller

The Transaction Controller (TR) controls the transfer of information between nodes of a network (data communications) and functions as the focal point for all transactions and data in and out of a system.

Its functions are

1. Network control—
 - polling
 - message switching
 - status maintenance of nodes/devices
 - graceful degradation
 - TR error control
2. Schedule control—
 - journalizing (TR)
 - security (level 1) (User/node ID [from DD/D])
 - message translation
 - Scheduling (criteria)—priority, response required, and resource availability and allocation, both for data and hardware
 - TR statistical collection
 - TR recovery
3. Message control—
 - TR process control modules
 - user status reporting

Data input/output controller

The Data Input/Output Controller (DIOC) manages and controls the exchange of information among all components of the environment, and the flow of data to and from the data base.

DIOC functions are

1. DML generation—
 - multiple DBMS
 - multiple nodes-sites
 - sub-schema translations
2. Validation—
 - Level 1—based on message content
 - Level 3—results of computation (prior to output or DB update)

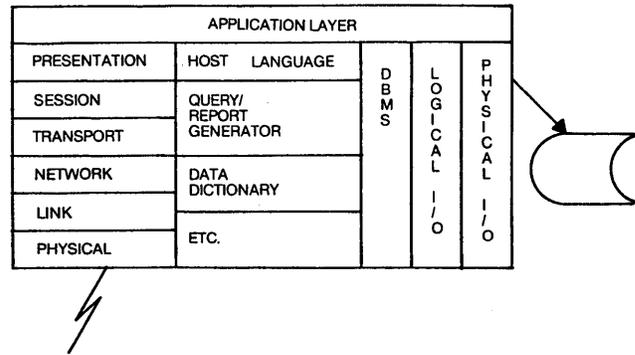


Figure 1—External communications and data storage

3. Statistical collection—
 - access to data base
 - requirement at multiple nodes
4. Output volume control—
 - ULI
5. Security (level 2)—
 - data access
 - function
6. Error translation

User language interface

The User Language Interface (ULI) is a set of problem-oriented languages which provide users with access to data. ULI functions are

1. n host language(s)
2. Query/multiple criteria—
 - natural language
 - relational
3. Report program generators

Data dictionary/directory/catalog

The Data Dictionary/Directory/Catalog (DD/D/C) is an organized, integrated repository of data describing the entire processing environment. The catalog is a subset of the data dictionary, limited to general data describing the location and contents of data at remote locations. Its functions are

1. Meta data-representing—
 - process
 - system
 - program
 - module

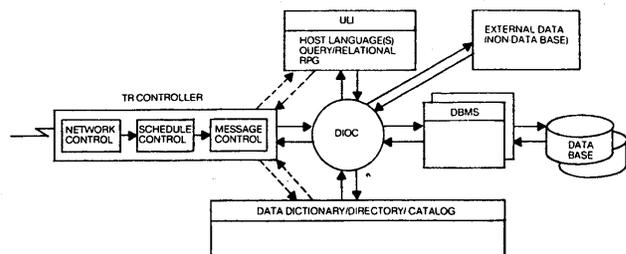


Figure 2—System components

- user profiles
 - security and tables: access and function rules
 - reports
 - transactions
 - TR and data journals
 - system(s) architecture
 - status
 - characteristics
 - network protocols
 - data: validation tables
2. Access and performance statistics
 3. Message control modules: storage
 4. Schemas
 - conceptual
 - detail sub-schemas
 5. Catalogs: distributed nodes

Data storage and retrieval

The Data Storage can be logically divided into three distinct parts:

1. The Data Base Management System (DBMS) is a software tool for the management and control of a pool of data to be shared by all authorized members of the user community.
2. The Data Base is a collection of data, organized to satisfy the information requirements of a community of users.
3. The External Data is data that is physically stored and managed in conventional (non-DBMS) media.

The data storage functions are:

1. Data structures
 - schema
 - sub-schema(s)
2. Physical storage control
 - buffer management
 - data directory
3. Access methods
 - internal representation
4. Data base recovery
 - journalizing
 - error message ID
 - data recovery
 - catastrophic
 - on-line
5. Contention/deadly embrace
6. Validation
 - 2nd level—DB access

DATA STRUCTURE AND RETRIEVAL ARCHITECTURE⁴

Based on the functional framework (Figure 1), the third category, data storage and retrieval, could be broken down into a seven-layer architecture similar to the ISO reference model. As presented in Figure 3, the highest layer would be the application layer. This layer would provide for tenant tracking

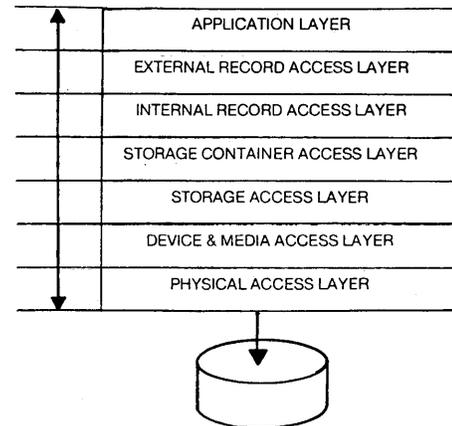


Figure 3—Data storage and retrieval architecture

and its functions would relate to the application. The lowest layer would be physical and it would relate to the actual physical recording, such as bits-per-inch on the magnetic tape or tracks on the disk platter. The layers between the application layer and the physical layer assemble the elements for the appropriate execution. An example would be a banking-application accessing disk. The banking application would reside in the application layer. The subordinate layers would formulate the information for the disk access. The physical layer (bottom layer) would then perform the disk access. Then the data would be passed back through the various layers to the banking application, which resides in the application layer. The functions of each layer of the architecture are listed as follows:

Application layer

- Support of end-user access to data
- Support of external access layer
- Tenant tracking

External access layer

- Multiple user views of data
- System components
 - ULI
 - DD/D/C
 - DBMS
 - external data
- Security and integrity
 - data access
 - function
- Data independence mapping
 - transformation between program needs and storage
- Offers the same services as the ISO presentation layer
- Supports application layer
- Supports internal layer

Internal access layer

- Access to and update of remotely stored data
- Logical integration of existing files
- Integration of textual and structured data

- Parallel processing
- Security and integrity
 - data access
 - data function
- Data dependence
 - files
 - records
 - items
 - item groups
- Access support
 - sequential
 - direct
 - primary/secondary key
 - data structure
 - area
- Storage structure
 - sequential
 - direct
 - indexed
 - hierarchical
 - network
- Storage media
 - magnetic disk
 - magnetic tape
- Data location transparency
 - local
 - remote
- Supports external access layer
- Supports storage container access layer
- Supports internal access level

Storage container access layer

- Maintains multiple copies of areas (files and records) for
 - rapid access at multiple locations
 - “time stamped” control
 - recovery purposes
- Space management for
 - records based upon database keys
 - recognition that record content has been changed
 - release of the area space whenever it is no longer needed
- Transfer of area (records and files) between homogeneous systems
- Supports storage access level
- Supports internal access level

Storage access layer

- Responsible for allocation and control of
 - main memory
 - cache memory
 - main mass storage
 - archival storage
 - file transfer
 - etc.
- Supports storage container access level
- Supports device access level

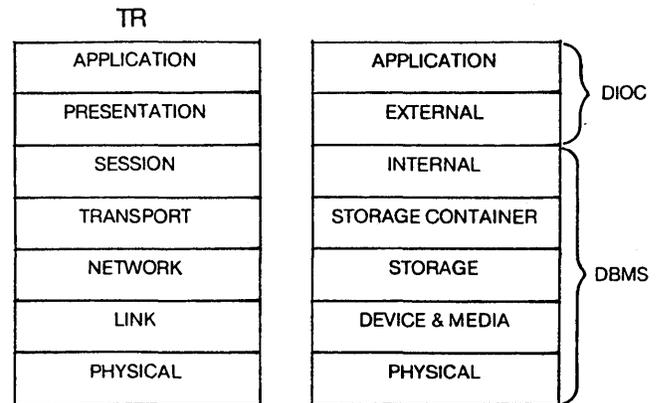


Figure 4—Residency of architecture of DBMS machine

Device and media access layer

- Supports
 - holding, mounting and demounting storage media
 - controlling movable read-write heads
 - awareness of physical transport of storage media
 - protocol for controller and device communications
- Supports storage access layer
- Supports physical access layer

Physical access layer

- Physical recording
 - magnetic tape (bits per inch)
 - disk (tracks per inch)
- Supports device and media access layer

RESIDENCY OF DATA STORAGE AND RETRIEVAL ARCHITECTURE (DSRA)

I will explain the residency of the DSRA in two examples to provide a better understanding of the implementation possibilities. Also, you should realize that part of the ISO architecture may reside in the DIOC and part in the TR or the ISO architecture may reside fully in the DIOC or fully in the TR. However, for clarity in Figures 4 and 5, I have chosen to present a distinct ISO architecture and a distinct DSR architecture. Since there are numerous papers on the ISO reference model, I will not discuss the TR and its ISO relationship any further.

In addition, it should be understood that all the layers of the architecture and their associated functions have to exist partially in the DIOC and partially in the other component. It should be noted, however, that the placement of the layers and their associated functions is dependent upon the DBMS machine and the software function and interface, which are discussed below.

DBMS Machine

If the component is a machine such as that represented in Figure 4, much of the intelligence required could be performed in this separate machine, creating the benefit of less work to be performed by the DIOC and allowing more time

for it to perform other functions. Therefore, the DIOC would need only to perform such functions as tenant tracking, determining whether the data is in a particular DBMS machine, and interfacing to other system components. In addition, the DIOC would perform the other functions within the Application Layer and External Access Layer.

However, the DBMS machine would have to perform all of the functions of the Internal Access Layer, Storage Container Access Layer, Storage Access Layer, Device and Media Access Layer and the Physical Access Layer.

Software Function and Interface

The second factor in decisions about the placement of the architectural layers is software. If the system component such as External Data is a conglomerate of software operations including sequential storage access, then you normally would not expect the system component External Data to perform with the amount of intelligence and efficiency that the DBMS machine could. Therefore, most of the intelligence would reside in the DIOC and very little in the software component as depicted by the brackets in Figure 5.

In addition to tenant tracking and having interfaces with other system components, the DIOC has to perform the functions of the Application Layer, External Access Layer, Internal Access Layer and the Storage Container Access Layer. This is the majority of the functions of the DSRA. The sequential storage access software would perform only the functions in the Storage Access Layer, Device and Media Access Layer and the Physical Access Layer.

In conclusion, this discussion has, I hope, provided enough background for you to expand your awareness of implementation concerns to include the other system components such as the ULI and the DD/D/C, whether these components be software interfaces or independent equipment.

RELATIONSHIP BETWEEN ISO AND DSRA⁴

Since both the ISO and DSR architectures work in close association, it could be said that they have an orthogonal relationship. If you start at a terminal and connect to the Physical Layer of the ISO architecture (Figure 6), you will pass through the various layers into the Application Layer. You then would pass through the various layers of the DSR architecture to the Physical Access Layer for media connection, such as disk.

In a similar manner, you could walk backwards from the media access (Physical Access Layer) of the DSRA up through all the layers to the Application Layer and back through to the terminal, which connects to the Physical Layer of the ISO architecture.

SUMMARY

In today's complex world, a large user often has various types of equipment from different vendors. This equipment performs functions to aid the work at manual and semi-automated work stations and also could perform all of the work at automated work stations. When requirements change, the user does not have the time or the capital to reprocur and convert to a single vendor.

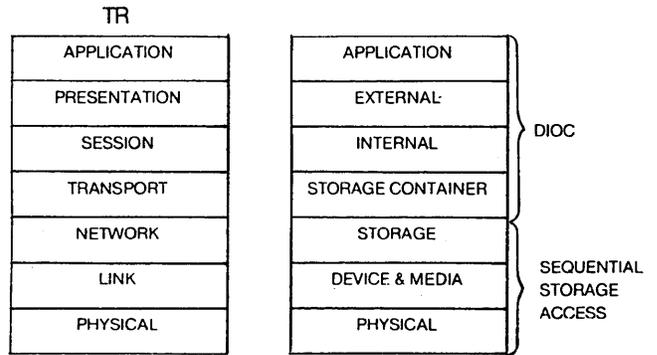


Figure 5—Residency of architecture for sequential storage access

Thus, the time has come for us in the information environment to be concerned that the user will be able to intercommunicate and perform processing in a cooperative mode.

This paper has suggested that the required system components integrated with the ISO architecture and the DSR architecture should be the foundation for such a unified system.

ACKNOWLEDGMENT

ISO developed the communications architecture. There are over fifty members in the ANSI/X3/SPARC Database Systems Study Group and in the Distributed Systems Group. Their contribution is acknowledged. In addition, thanks is expressed to Elizabeth Fong for editing.

REFERENCES

1. "Info-80 Conference Program," October 1980, p. 14
2. International Organization for Standardization (ISO/TC97/SC16), "Reference Model for Open Systems Interconnection" (Version 4 as of June 1979), (m.p.: International Organization for Standardization, August 1979)
3. ANSI/X3/SPARC Database Systems Study Group, Database Interface with Distributed Systems Task Group, "Working Notes on Distributed Database Management," August 1980
4. Bachman, Charles, "Second Annual Database Symposium of The Washington Area," April 1980

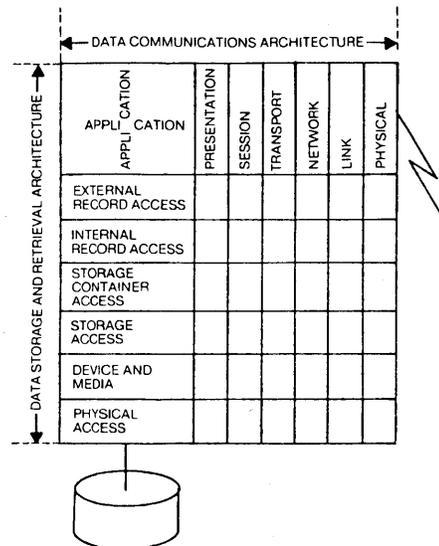


Figure 6—Orthogonal architectures in distributed computer systems

Optimization of the file access method in content-addressable database access machine (CADAM)

by SADAYUKI HIKITA, HARUAKI YAMAZAKI, KIYOSHI HASEGAWA,
and YUTAKA MATSUSHITA

*OKI Electric Industry Co., Ltd., Research Laboratory
Tokyo, Japan*

ABSTRACT

The design of the database machine CADAM with an efficient file access mechanism is presented. CADAM can be used widely for various applications cost-effectively. For this purpose, the functions of database systems are distributed over three basic components, and a conventional magnetic disk device instead of a disk attached with microprocessors and logic circuits is used. One of the most important points in designing CADAM with high throughput is to minimize the time for transferring data from the disk to the cache memory. We analyze the optimal size of physical access unit between them for various applications in order to show that the size should be adjustable.

INTRODUCTION

Recent semiconductor technology has accelerated distributed systems. Many studies on specialized processors based on that technology have been reported. The database machine is one of them for database management which is to improve the system performance of database processing. For designing such a database machine the following three points should be taken into consideration.

The first point is to reduce the number of interactions between a host computer and a database machine for enhancing the performance. In XDMS² a data request is given to the database machine by a procedural language such as COBOL, which needs a number of interactions between them. Such a data request using a procedural language can be said to be an obstacle to enhancing the performance. If a higher level language or a nonprocedural query language is used, the number of interactions between a host computer and a database machine will be much reduced^{3,4}. Thus if a high bandwidth communication channel and a nonprocedural query language are both adopted, the performance bottleneck in the interface will disappear.

The second point is to shorten the processing time in the directory management or set operations of union, intersection, complement, and division which are specified by a query. Such set operations especially may need exhaustive comparison. This may require much processing time and may

extremely degrade the performance of the database system. If such an operation is processed by a single processor, the performance bottleneck of the total system will result. The database machines proposed in RAP³, DBC⁴, and CASSM⁷ enhance their performance by using a number of microprocessors, each of which shares one of the functions such as interface control between a host computer and a database machine, security check, data retrieval, and attribute value comparison. Furthermore, since the function of the data retrieval and exhaustive comparison require a large amount of processing time, many microprocessors are required to realize higher performance. On the other hand, existing database management systems use the hierarchical data model or the network data model in which complicated data structure must be handled. This data structure has to be modified in every data modification request, which leads to deterioration in performance. The relational data model proposed by Codd¹ does not need such a complicated data structure but needs data stores called flat tables. But much processing time is required to scan the flat tables by a single processor. In most of the database machine architecture, a flat table is simultaneously scanned by many microprocessors; this process is equivalent to realizing the content-addressable memory^{3,4,7}. Thus the performance bottleneck for handling so large an amount of processing could be eliminated by using a number of microprocessors.

Although memory device technology is rapidly progressing, we cannot find a large-capacity secondary memory other than a disk memory from the viewpoint of cost/performance (CCD and Bubble memories are potential candidates). Since the access time difference between IC memory and disk memory is very large, data transfer between them would cause a performance bottleneck in a database machine. In other words, the performance of a database machine depends on how efficiently data are transferred between the main memory and the secondary memory. This relates to the third point of the performance bottleneck.

The three points described above are extremely important in designing an efficient database machine. Several studies have been reported on the former two points^{3,4,5,6,7,12}. Studies regarding the third point are not so plentiful^{8,9,10,11}. In this paper the outline of the content-addressable database access machine (CADAM) equipped with an efficient file access

mechanism is described. Furthermore, the optimal physical file access unit is quantitatively analyzed to show that it should be adjusted according to database applications.

THE OPTIMIZATION OF THE FILE ACCESS METHOD

If all tracks of the disk storage are always read in one disk revolution and at the same time compared in parallel for every data access request as presented by RAP³, it can be said to be the fastest data access scheme. However, this scheme will be extremely expensive, because as many microprocessors and logical circuits are required as there are disk heads of a fixed-head disk. In addition, in RAP, as many disk revolutions are required as there are qualifications of a query. In this sense, RAP's approach does not always give quick response when such a complicated query is processed.

A new method, in which only a specified relation instead of all tracks of the disk is read, has been proposed in RAP¹². The relation read from the disk is compared in parallel by using numerous microprocessors^{9,10,12}. However, the efficiency of disk access may not be optimal even in this method. Since the data really needed may be a part of the relation rather than the whole relation, most of the data read from the disk may be useless and may have to be discarded.

Another method, called DBC, in which all the tracks in a cylinder are accessed simultaneously and compared in parallel, has been proposed by Hsiao and colleagues^{4,6,11}. This method is much more economical than that of RAP³ because not so many microprocessors and logical circuits are required as in RAP. Compared with an existing disk, this gives faster access to the disk because all tracks in a cylinder are simultaneously accessed. In this method, data are identified by a cylinder number rather than by each record address.

When the relation needed is stored in more than one cylinder, it is sufficient to read only the necessary cylinders rather than the whole relation. That is why DBC is more economical than RAP³.

However, when the data really needed are only a part of the cylinder, all data in all tracks in the cylinder have to be read from the disk. In this case, the method of DBC cannot be said to be very economical. If the traffic of the data access is fairly high, the DBC method may be said to be cost-effective. However, if not, this cylinder-based access method is fairly costly. Thus Hsiao's cylinder-based DBC can be said to be cost-effective provided that the capacity of a database is extraordinarily large and the traffic of the access request is high. However, there exist various applications ranging from small (low) to large (high) storage (traffic). A database machine should be used in various applications. Therefore a database machine must be designed so that cost/performance is optimized according to given applications.

On the basis of these discussions, we propose the database machine CADAM (Content-Addressable Database Access Machine), which has the following characteristics for the data access method.

1. It uses commercial based floating head disks unmodified.
2. A cache memory is introduced between the main memory and the disk so as to shorten the disk access time. The capacity of the cache memory is adjustable according to each application.
3. Requested records may be evenly distributed over disk tracks in one application, and the records may be located in the continuously adjacent disk tracks in the other. Therefore, the physical data access unit (called PAU) is optimized by taking the characteristics of the application into consideration. Each data item can be accessed by the PAU number.

BASIC OPERATION OF CADAM

In this section, basic operation of CADAM is described by focusing on the file access efficiency. CADAM supports a relational data model. Its database consists of a number of relations stored in disk units, each of which is composed of a number of PAUs. A database user program in a host computer is interfaced with CADAM by a mapping-oriented query language such as SEQUEL^{13,14}. CADAM is located at the back end of a host computer. A query specifies one of four basic commands (retrieve, insert, delete, replace), and the following basic operations.

MAPPING OPERATIONS

RESTRICTION, which selects tuples by Boolean search conditions; and PROJECTION, which selects specified attributes of a relation.

JOIN

Combine two relations into a new relation based on equi-join.

SET OPERATIONS

UNION, DIFFERENCE, INTERSECTION and DIVISION.

AGGREGATE OPERATIONS

such as MAX, MIN, AVERAGE, and COUNT.

CADAM consists of three basic components: interface processor (INP), directory processor (DP), and file access processor (FAP). INP receives queries one after another from the host computer, analyzes them, and decomposes them into several transactions, which include only MAPPING OPERATIONS of a single relation. A transaction is sent to and executed in DP and FAP, and the output is returned to INP. INP is further divided into two processing submodules; one is for a query analysis, decomposition, and output transfer to the host computer; the other is for basic operations, except MAPPING OPERATIONS. When a number of queries are processed in CADAM, some may require only the MAPPING OPERATIONS, but others may further require JOIN or SET OPERATIONS, in which much processing time is necessary.

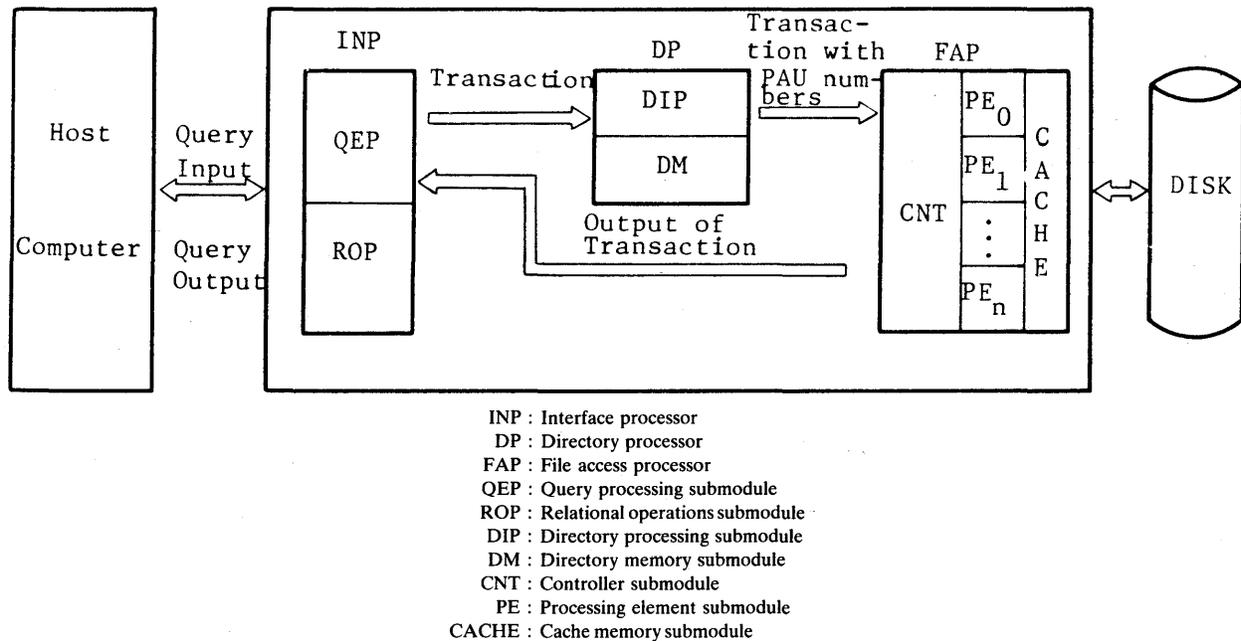


Figure 1—Basic operation of CADAM

In this situation, queries in CADAM requiring only MAPPING OPERATIONS of a single relation do not have to wait until a query further requiring JOIN or SET OPERATIONS is completed, since the two processing submodules in INP can operate concurrently and independently. This feature cannot be found in CASSM⁷, RAP^{3,9,12}, or DBC^{4,5,6,11}.

DP consists of the directory processing submodule and the directory memory submodule. The value of specified attributes of a relation is partitioned according to the predefined scale of the value. The directory memory submodule holds a mapping table converting from a partition number to a PAU number or numbers. The directory processing submodule decides the partition number from the attribute value of a query. Furthermore, the independently retrieved PAU numbers by each search condition are ANDed, Ored in order to get final PAU numbers satisfied with all search conditions. The attribute(s) used in a search condition most often is (are) called clustering attribute(s). All records of each partition of the attribute are clustered in a PAU or PAUs. Furthermore, all PAUs are clustered according to the value of the clustering attribute(s). On the other hand, the attributes used not so often are called nonclustering attributes. All records in each partition of the nonclustering attributes may be evenly scattered on many PAUs of a relation.

FAP receives transactions with PAU numbers attached by DP. FAP consists of three submodules: the controller submodule, the processing element submodule, and the cache memory submodule. The controller attaches transactions transferred from DP to a queue and selects one of the PAUs from the queue which can be read in the shortest time to the cache memory submodule from disk units. In other words, the processing order of the transactions in the queue is rearranged so as to shorten the overall disk access time as far as the database consistency is preserved.

The specified PAU read from the disk should be divided according to the number of the processing elements of FAP. Each processing element compares the value of records with the specified value of query simultaneously, which results in the realization of content addressability. The output of the transaction in the processing element submodule is returned to the controller submodule every time a single PAU is processed. The output of the transaction is returned back to INP from FAP every time a single relation is processed.

The basic operation described above is shown in Figure 1.

In this section the file access method of CADAM and the distribution of the function over the components of CADAM are shown. A database machine should be applied to various applications cost-effectively. In the next section the transfer time between the cache memory and the disk is analyzed, and the reason why the size of PAU should be adjustable to applications is shown.

THE ANALYSIS OF OPTIMAL PAU SIZE

The minimization of the time required to transfer PAUs from the disk to the cache memory is a critical problem to improve the performance in CADAM. In this section, the optimal PAU size to achieve this minimization is analyzed, provided that the two types of queries (random query and clustering query) are given.

The Random Query Model

If the set of records which satisfies a query with non-clustering attributes may be evenly distributed over the disk storage space, then that query is called a random query. For

example, the query "Retrieve EMP.AGE = 35" on the EMP relation might be a random query, if AGE is a nonclustering attribute. If the records are clustered on the basis of a clustering attribute, then that query is called a clustering query, which will be described in the following section.

The following notations are used:

- c : number of cylinders for a relation in the disk device
- m : total number of the tracks for a relation in the disk
- n : number of tracks per cylinder ($= m/c$)
- s : mean seek time
- a : mean latency delay
- t : number of records in a track
- p : probability that the random queries in the queue of FAP access a track
- γ : probability that a record is not accessed by the queries in the queue

The relation to be accessed is assumed to be larger than a cylinder capacity and smaller than a disk unit capacity.

Let P_c be the probability that the given cylinder is accessed by the queries. Then P_c is expressed as follows in terms of p :

$$P_c = 1 - (1 - p)^n$$

Let us denote $1 - p = q$; then

$$P_c = 1 - q^n \quad (4.1)$$

Therefore, the probability that k cylinders in the disk are accessed is given as follows:

$$Pr \{k \text{ cylinders accessed}\} = {}_c C_k P_c^k (1 - P_c)^{c-k} \quad (4.2)$$

Therefore, the mean number of cylinders to be accessed by the queries is given as $c(1 - q^n)$.

By similar discussion, the mean number of tracks to be accessed is given as mp .

Note that the probability where no cylinders are accessed is also defined as a nonnegative value in this model. This probability can be interpreted as the frequency of the occurrence of the illegal queries.

Let us assume a PAU consists of i tracks within a cylinder where $n \equiv 0 \pmod{i}$.

Then the probability that the given PAU is accessed is

$$1 - (1 - p)^i = 1 - q^i \quad (4.3)$$

Since the mean latency delay is a , $2a$ is the time for one disk revolution. Therefore the time required for a PAU transfer from the disk to the cache memory is $(a + 2ai)$.

Thus, the mean time for the transfer of all required PAUs from the disk to the cache memory T_R is given as follows:

$$T_R = (\text{mean seek time}) \times (\text{mean number of cylinders accessed}) + (\text{time for a PAU transfer}) \times (\text{mean number of PAUs accessed}).$$

Therefore the following equation is derived:

$$T_R = c \cdot s(1 - q^n) + a(1 + 2i) \frac{m}{i} (1 - q^i) \quad (4.4)$$

γ is the probability that a given record is not accessed by the queries, and t is the number of records in a track, so the probability P is given as follows:

$$P = 1 - \gamma^t$$

Therefore q and T_R are given as follows:

$$q = \gamma^t$$

$$T_R = c \cdot s(1 - \gamma^{tn}) + a(1 + 2i) \frac{m}{i} (1 - \gamma^n) \quad (4.5)$$

The Clustering Query Model

If all records of a relation are clustered according to the value of the clustering attributes, and a query uses the clustering attributes, then such a query is called a clustering query. Furthermore, all PAUs that store a single relation are also clustered according to the value of the clustering attributes. When a clustering query uses a comparator such as *greater than* or *less than* in its qualification, the retrieval efficiency is further improved by the clustering technique.

Let b be the number of tracks per cluster, assuming $b \leq n$ and $n \equiv 0 \pmod{b}$.

Now, let P_b be the probability that a cluster is accessed by the queries with the clustering attributes. Then the probability P_c that a cylinder is accessed is as follows:

$$P_c = 1 - (1 - P_b)^{\frac{n}{b}} = 1 - q_b^{\frac{n}{b}} \quad (4.6)$$

By the same reason shown in section above, the mean number of cylinders to be accessed is as follows:

$$cP_c = c(1 - q_b^{\frac{n}{b}}) \quad (4.7)$$

Let us assume that a cluster consists of k PAUs and a PAU consists of i tracks. Furthermore, assume that a cluster is contained in only one cylinder.

Let a cluster consist of b tracks; then $b = k \cdot i$. Therefore the mean number of clusters to be accessed is $\left(\frac{m}{b}\right) P_b$. Since the time of transferring a cluster from the disk to the cache memory is given as $(a + 2ai)k$, the time of transferring all required PAUs from the disk to the cache memory is given as follows:

$$\left(\frac{m}{b}\right) P_b \cdot k(a + 2ai) = \left(\frac{m}{i}\right) P_b(a + 2ai)$$

Here, if we assume $b = 1$, that is, the size of a cluster is one track, then this query model is equivalent to the previously stated random query model.

Thus, the mean value of time T_c of transferring all required PAUs from the disk to the cache memory in addition to the disk seek time is given as follows:

$$T_c = c \cdot s(1 - q_b^{\frac{n}{b}}) + P_b \cdot \left(\frac{m}{i}\right)(a + 2ai) \quad 1 - q_b = P_b \quad (4.8)$$

In Equation 4.8, the first term is the expected seek time of all clusters required by queries and the second term is the expected transfer time of all clusters required by them.

We assume that the number of records read from the disk to the cache memory is same both in the random query model and the clustering query model. Since m is the number of tracks of a relation, t is the number of records in a track, and $1 - \gamma$ is the probability that a record is accessed, the number of records read in the random query model is $mt \cdot (1 - \gamma)$.

On the other hand, in the clustering query model the number of records of all clusters accessed is $\left(\frac{m}{b}\right) bt$.

So the number of records read in the clustering query model is $m \cdot t \cdot P_b$.

Thus

$$\begin{aligned} mt \cdot (1 - \gamma) &= m \cdot t \cdot P_b, \\ P_b &= 1 - \gamma, \\ q_b &= \gamma \end{aligned}$$

Finally, equation 4.8 is expressed as follows in terms of γ instead of P and q .

$$T_c = c \cdot s \cdot \left(1 - \gamma^a\right) + (1 - \gamma) \frac{m}{i} (a + 2ai) \quad (4.9)$$

Generally the mean transmission time T from the disk to the cache memory in the combined model of the random query model and the clustering query model is as follows

$$T = \alpha T_c + (1 - \alpha) T_R \quad (0 \leq \alpha \leq 1),$$

where α is relative frequency when the clustering queries are used. Thus, by Equation 4.5 and Equation 4.9 the following equation is derived:

$$\begin{aligned} T &= c \cdot s \cdot \left[\alpha \left(1 - \gamma^a\right) + (1 - \alpha) \left(1 - \gamma^m\right) \right] \\ &+ \frac{anc}{i} (1 + 2i) \left[\alpha (1 - \gamma) + (1 - \alpha) \left(1 - \gamma^i\right) \right] \end{aligned} \quad (4.10)$$

NUMERICAL EXAMPLES

Figure 2 and Figure 3 describe the transfer time T versus PAU size i , assuming:

- $t = 10$ records/track
- $b = 32$ tracks/cluster
- $c = 200$ cylinders/relation
- $s = 30$ ms (mean seek time)
- $n = 32$ tracks/cylinder
- $a = 10$ ms (mean latency delay)

For simplicity, the curves are shown with a continuous value of i in both figures, but the condition $b = 0 \pmod{i}$ restricts the possible values of i to $i = 1, 2, 4, 8, 16, 32$.

Figure 2 describes the transfer time T versus i with α as a parameter. The minimizing value of i changes from $i = 1$ to $i = 32$ depending on α .

Note that the parameter γ describes the probability that a record is not accessed by the queries in the queue in FAP. In Figure 2 the fairly small value of $\gamma = 0.91$ is assumed; that is, $1 - \gamma = 0.09$ for the probability of a record accessed. This means the required response time for each query is not severe and the queue in FAP can have many queries or transactions. Therefore CADAM can process a comparatively large number of queries at a time to improve the total throughput of the system rather than to improve the response time. This figure shows that the optimal PAU size is a track if the relative frequency of clustering queries (α) is approximately less than 0.8. However, if the value of α is greater than or equal to 0.8, the optimal PAU size is the cluster size.

On the other hand, Figure 3 describes the T vs. i for each γ , assuming $\alpha = 0.99$. In this case T is minimized by the value of $i = 2$ for all γ approximately. In this figure, the comparatively large value of γ is assumed. For example, $\gamma = 0.9995$ means approximately 32 ($= 32 \times 200 \times 10 \times 0.0005$) records are accessed by the queries. This means that FAP has the

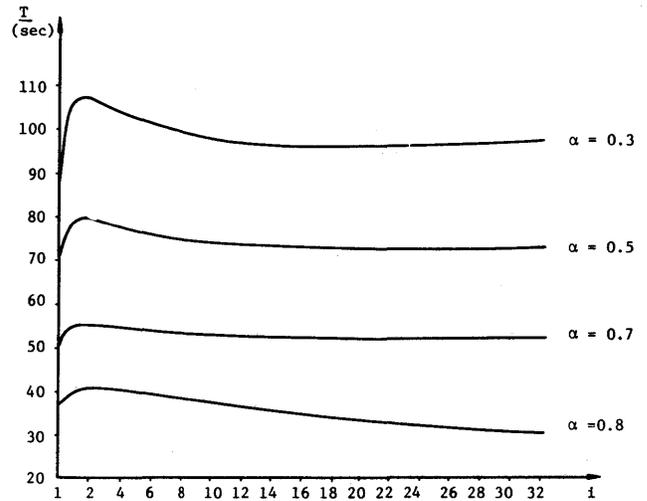


Figure 2— T vs. i for $\gamma = 0.91$

short waiting queue where a single query (transaction) or small number of queries (transactions) are processed at one time, since a quick response for each query is required.

In this example, the optimal PAU size seems to be the cluster size, since $\alpha = 0.99$, which means that almost all the queries are clustering ones. However, Figure 3 shows that the optimal PAU size is two tracks. This is an interesting result and is explained by the following reason. Even though the relative frequency (α) of the random queries is very small (0.01), the transfer time T is affected and increased by them. So the optimal PAU size is not the cluster size.

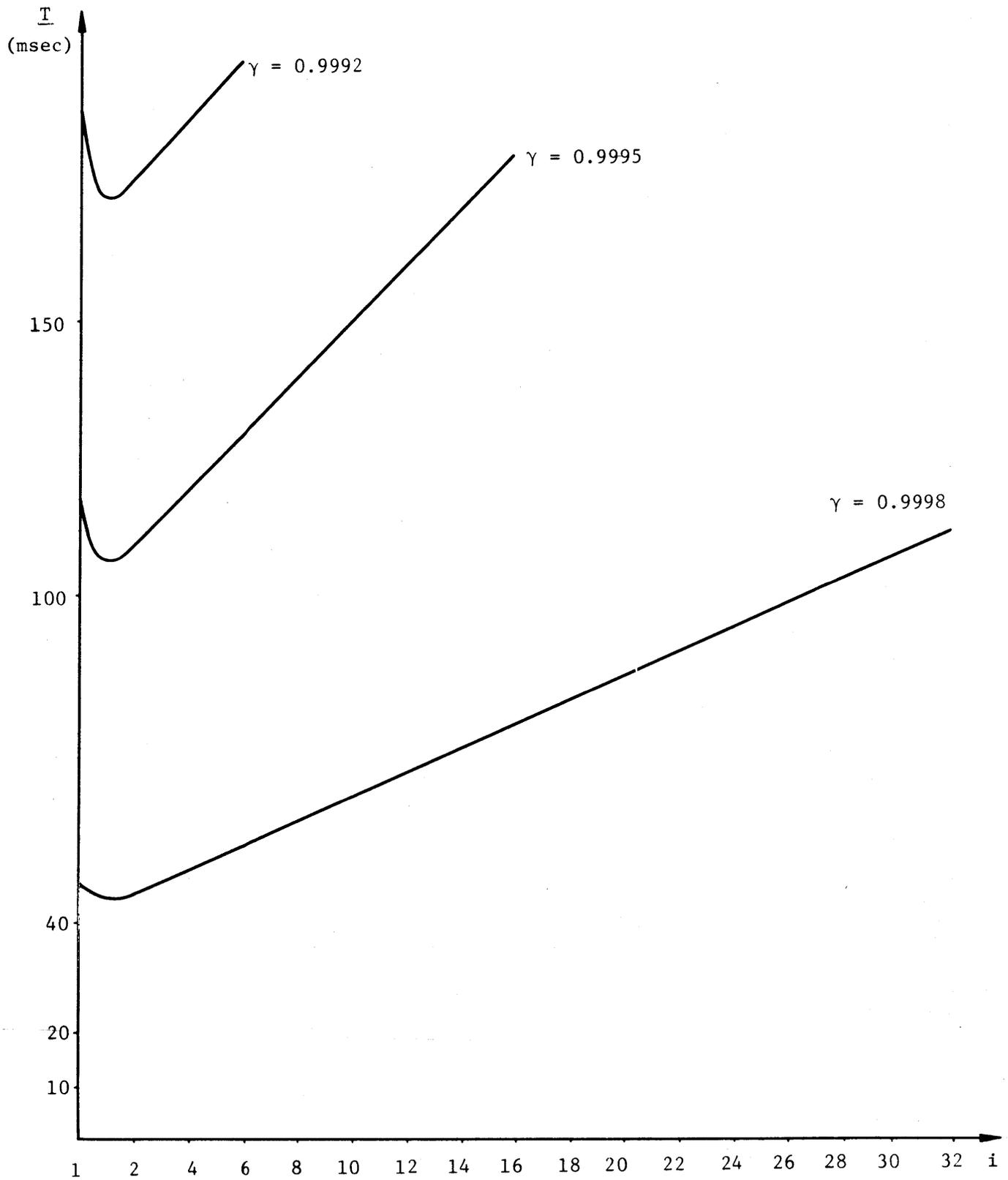
CONCLUSION

As analyzed in Section 5, the optimal PAU size changes from the smallest (one track) to the largest (cluster) depending on parameters α and γ .

Furthermore, this optimal PAU size is very sensitive to these parameters. A slight change of α or γ produces a great change of the optimal PAU size. In the real application of the database system, these parameters reflect the field where that database system is applied. Those are whether the system especially requires a quick response or high throughput, whether each query requires a large number of records, what percentage of the queries are clustering queries, etc. Therefore, in order to design a database machine system that can be widely used, the architecture must be adjustable to the different PAU sizes.

On the other hand, there is an approach that modifies a disk unit to access a cylinder simultaneously by adding some logic units to it^{3, 4, 7}. However, this approach cannot improve the cost effectiveness if the cluster size is sufficiently small or clustering attributes do not exist (α is small).

Taking into account this strong dependency of the optimal PAU size on α and γ , the flexible addressing mechanism and cache/disk mapping mechanism have been adopted for diverse use of CADAM. CADAM is a promising architecture

Figure 3— T vs. i for $\alpha = 0.99$

for supporting a cost-effective file access method through the adjustable PAU size distributing database functions among three components (INP, DP, FAP).

We are now developing this database machine. The problems of data consistency for concurrency control, optimization for a query execution, a security system, and recovery from failure are also under research and development.

REFERENCES

1. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." *CACM* 3 (1970), pp. 377-387.
2. Canaday, R.H., R.D. Harrison, E.L. Ivie, J.L. Ryder, and L.A. Wehr. "A Back-end Computer for Data Base Management." *CACM*, 17 (1974), pp. 575-582.
3. Ozkarahan, E.A., A.A. Schuster, and K.C. Smith. "RAP—An Associative Processor for Database Management." *AFIPS, NCC Proceedings*, 44 (1975), pp. 379-387.
4. Baum, R.I., D.K. Hsiao, and K. Kannan. "The Architecture of a Database Computer. Part I: Concepts and Capabilities." Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1.
5. Hsiao, D.K., and K. Kannan. "The Architecture of a Database Computer. Part II: The Design of Structure Memory and Its Related Processors." Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-2.
6. Hsiao, D.K., and K. Kannan. "The Architecture of a Database Computer. Part III: The Design of the Mass Memory and Its Related Components." Ohio State University, Tech. Rep. No. OSU-CISRC-76-3.
7. Hearly, L.D., G. J. Lipovski, and K.L. Doty. "The Architecture of a Context-Addressed Segment-Sequential Storage." *Fall Joint Computer Conference Proc.*, 1972, pp. 691-701.
8. Casey, R.G., and I.M. Osman. "Replacement Algorithms for Storage Management in Relational Databases." *The Computer Journal*, 19 (1976), pp. 306-314.
9. Schuster, E.A., E.A. Ozkarahan, and K.C. Smith. "A Virtual Memory System for a Relational Associative Processor." *AFIPS, NCC Proceedings*, 45 (1976), pp. 855-862.
10. Ozkarahan, E.A., and K.C. Sevcik. "Analysis of Architectural Features For Enhancing the Performance of a Database Machine." *ACM, TODS*, 2 (December 1977) pp. 297-316.
11. Hsiao, D.K. and K. Kannan. "Simulation Studies of the Database Computer (DBC)." Ohio State University Tech. Rep. No. OSU-CISRC-TR-78-1.
12. Schuster, S.A., H.B. Nguyen, E.A. Ozakarahan, and K.C. Smith. "RAP 2—An Associative Processor For Data Bases." *Proceeding, 5th Annual Symposium on Computer Architecture*, 1978, pp. 52-59.
13. Astrahan, M.M., et al. "System R: Relational Approach to Database Management." *ACM TODS*, (1976), pp. 97-137.
14. Chamberlin, D.D., et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control." *IBM J. Res. Develop.* (November 1976), pp. 560-575.

Parallel sort and join for high speed database machine operations

by MAMORU MAEKAWA

University of Tokyo
Tokyo, Japan

ABSTRACT

This paper proposes a parallel joining and sorting algorithm that completes in about $\log_2 N$ steps for N records. The algorithm is intended for large database systems. The algorithm and its required processor interconnection are simple and realistic. Thus, a performance improvement of three to four orders of magnitude can be realistically expected by applying this algorithm to database machines. Analysis is primarily made by simulations.

INTRODUCTION

Background and Objectives

In database processing, set operations of records such as sorting and joining are most fundamental. The importance of sorting operations has been well recognized by data processing people. The joining operation is one of the fundamental operations of relational data base systems and is a very costly operation. A simple method requires N^2 comparisons for a pair of two files of N records each. The number of comparisons can be reduced by improving algorithms but the minimum number of comparisons for a pair of two arbitrary files can never be less than $2\log_2(N!) \approx 2N\log_2 N$. Parallel processing is the only method that can substantially reduce the operation time. Unfortunately, parallel joining algorithms have seldom been proposed, so virtually no analysis of parallel join algorithms is available. However, many results from the analysis of sorting algorithms are applicable since joining and sorting are similar operations.

Batcher,¹ Stone,⁴ Thompson and Kung,⁵ and others have made analyses of parallel sorting algorithms. The Batcher's algorithm, which is the fastest algorithm known so far, requires $\log_2 N(\log_2 N + 1)/2$ comparisons for a file of N records. This algorithm is based on the sorting network proposed by Batcher himself. Stone⁴ shows that the Batcher's algorithm can also run on the perfect shuffle interconnection that he has proposed. If the interconnection is less complete, then a longer operation time is required, as shown by Thompson and Kung.⁵ Thus the interconnection is important. Since the theoretical lower bound is $\log_2(N!) \approx N\log_2 N$, we expect that

parallel processing by N processors would reduce the lower bound to $N\log_2 N/N = \log_2 N$. Thus the Batcher's algorithm requires $(\log_2 N + 1)/2$ times more than the theoretical minimum.

This paper is an attempt to obtain parallel joining and sorting algorithms whose operation times are close to their theoretical lower bounds. The algorithms are mainly analyzed by simulations. It also attempts to make a required processor interconnection that will be realistic even for large files. Although the algorithms are equally applicable to internal or external operations, they are primarily intended for large external files, and if they are applied a performance improvement of several orders of magnitude can be realistically expected.

Problems in Parallel Algorithms

There are two important points in developing parallel algorithms, namely the interconnection of processors and the algorithm itself. They are strongly related to each other. A simple interconnection imposes restrictions on algorithms, and sophisticated algorithms generally require more complete interconnections. For large database systems, shared memory schemes usually used in multiprocessor systems are not applicable because such schemes are severely limited in size. A network of independent processors with their own local memories is the only possible approach. It is important that the network is technically and economically feasible. Particularly crucial is the number of lines from/to a node, and every effort should be made to reduce it.

Evaluation of Parallel Algorithms

Ordering or matching of arbitrary KN records to perform a sorting or joining operation would theoretically require at least $\log_2[(KN)!]$ comparisons. Therefore, parallel processing of KN records by N processors would reduce this lower bound to $\log_2[(KN)!]/N \approx KN\log_2(KN)/N = K\log_2(KN)$. This $K\log_2(KN)$ can be further divided into two components, $K\log_2 N$ and $K\log_2 K$. The first component, $K\log_2 N$, can be interpreted as the minimum number of comparisons required to distribute KN records over N processors, thus making each

processor hold K records. The second component, $K \log_2 K$, is then the minimum number of comparisons for each processor to order its K records. According to this interpretation, any parallel algorithm which can distribute KN records evenly over N processors in $K \log_2 N$ comparisons is optimal. In general, however, the resulting record distribution will not be even over N processors and the distribution process itself takes more than $K \log_2 N$ comparisons. Therefore, the total number of comparisons, T , can be expressed as

$$T = \alpha K \log_2(\alpha K) + \beta K \log_2 N,$$

where αK ($\alpha \geq 1$) is the maximum concentration of records in a processor and β is the increase in ratio of the number of comparisons for distributing records. The optimal case is of course $\alpha = \beta = 1$. We can observe from the above equation that for a large K α is more important than β , so efforts should be concentrated more on distributing records more evenly than on reducing the number of comparisons for distributing records.

Another important point to be remembered about parallel algorithms for large database systems is that a record transfer between processors usually requires a much longer time than a record comparison, because records are usually large and a transfer of records between processors usually involves a relatively complex protocol handling. Therefore the number of record transfers is much more important than that of record comparisons. In fact, this paper evaluates algorithms mainly in terms of the number of record transfers (called "steps" in this paper) rather than in terms of record comparisons.

PROCESSOR INTERCONNECTION

We assume that the system is composed of interconnected autonomous, identical processors, where each processor is a pairing of memory and logic.

When this interconnection is represented as a graph, where processors are shown as vertices and lines interconnecting them as edges, it is clear that the diameter of the graph is the absolute minimum number of steps of record transfers required for a sort or join operation to be completed, regardless of the method used. The diameter of a graph is minimized when the graph is complete. However, when the size of the graph becomes maximal, the system becomes very costly and may be unfeasible even for a reasonably small system. The problem is then to find an interconnection or graph whose diameter and size are both within some practical limits. Such interconnections are found by Maekawa.^{3,6} The optimal interconnection found allows each processor to reach any other processor in no more than $\log_2 N$ steps and requires only $2(N-1)$ edges altogether. In this paper, however, for the sake of simplicity and better understanding we will base our discussions on cube interconnections. They have the same property as regards the transfer delay as the optimal interconnections do, and the same algorithms can be applied to both. The cube interconnections require $(N \log_2 N)/2$ edges altogether and are thus inferior to the optimal interconnections.

JOIN ALGORITHM

In this section, we propose and analyze parallel join algorithms. Since a join operation can be considered as repetitions of a search operation—e.g., each record of one file is searched against the other to find matching records—its operation time can be shortened by performing search operations in parallel. If all the search operations are performed completely in parallel, then the total execution time would be shortened to the execution time of a single search operation, namely $\log_2 N$ steps. Of course, this is an ideal case, and the success of any parallel algorithm is determined by how much parallelism can actually be obtained. In the join operation, the key question is how much parallelism can be obtained in tracing down the search tree. (The problem of how the search tree is prepared will be discussed in the next section.) The simplest way of obtaining the maximum degree of parallelism is to provide N search trees, one for each record. However, this method is clearly not realistic, for $N(N-1)$ nodes of the search trees would be required. For our interconnection the total number of necessary search tree nodes is $N \log_2 N$. This $N \log_2 N$ is the minimum number of search tree nodes for a complete parallel search. For any number less than $N \log_2 N$ it is extremely difficult to achieve a high degree of parallelism.

In our system, the major causes of delays are the limitations of the lines connecting processors and of the memories of the processors. The interconnection method, the number of search tree nodes, and the delays caused by the limitations in line capacity and memory size are all strongly related. The interconnection structure should be given the top priority; it determines the feasibility of system building, both technically and economically. The number of search tree nodes has the second priority and is determined as the minimum number necessary for the complete parallel operations. This number is $N \log_2 N$, and this $N \log_2 N$ nicely matches with the number of lines. Our problem is, based on these conditions to devise algorithms that minimize the delays caused by the line and size limitations and to analyze them. Following the arguments so far, we define the assumptions and algorithm next.

Assumptions

The assumptions are as follows.

1. The system consists of $2N = 2 \times 2^D$ processors, and their interconnection is the $(D+1)$ -cube. The file R is held in processors 0 through $(N-1)$ while the file S is held in processors N through $(2N-1)$. The case of $N=8$ is shown in Fig. 1.
2. The distance $d(C_i, C_j)$ is defined between two processors C_i and C_j when they are directly connected, as

$$d(C_i, C_j) = \begin{cases} (\log_2 |i - j|) + 1 & i \neq j \\ 0 & i = j \end{cases}$$

If they are not directly connected, the distance between them is undefined. Similarly, the line connecting C_i and C_j is called the line of distance $d(C_i, C_j)$ of processor C_i or C_j . The lines of distance d connect two $(d-1)$ -cubes to form a d -cube so that a cube is recursively defined.

3. The file R is already ordered and is placed in the lower

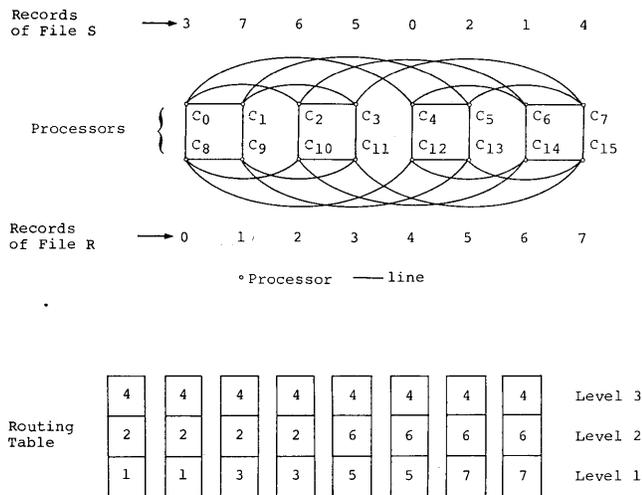


Figure 1—Interconnection of the 4-cube and routing table

half of the system, as shown in Figure 1. The file S is arbitrarily ordered. A joining operation is performed by moving the records of file S among its processors so that the records are placed at the processors located immediately above their corresponding processors of the file R (namely, the processors of distance $D + 1$); then the records are immediately moved down through the lines of distance $D + 1$, at which the join of the records is immediately performed because the records of the two files match. In the above operation the lower half of the system is not used, and it will be ignored in the discussion in this section.

4. Record transfers are synchronized. Records are transferred to directly connected processors in a single step. The lines are bidirectional, but only one record can be transferred at a time.
5. Each processor initially holds K records. The record values are in the range between 0 and $(N - 1)$.
6. Each record has a descriptor whose value can range between 0 and D .
7. To assist routing, each processor of the file S is provided with a routing table of size D , as shown in Figure 1. These tables effectively serve as the search tree as a whole. As seen in Figure 1, the top entries (level 3) hold the value that divides the file R into two halves, the 2nd entries (level 2) further divide the halves into quarters, and so on. These routing tables are set according to the ordered file R before the joining operation begins. This setting is discussed in the section "The Determination of Routing Tables and Sorting Algorithms."

The Basic Algorithm

The algorithm of the join operation for each processor is defined as follows:

1. Initialize the descriptor, p , of the record at the processor to $D + 1$.
2. Decrease p by 1.

3. When $p = 0$, the record is transferred to the lower half of the system through the line of length $D + 1$, at which the record is immediately joined with the corresponding record of the file R .
4. Compare the record with the level p entry of the routing table of its processor. When this processor belongs to the right half of the 2^p -processor segment containing the processor in question and the record is smaller than the entry, it is moved to the processor of distance p ; otherwise it remains there. Here, a segment of 2^p processors is a group of processors whose numbers are between $2^p i$ and $2^p(i + (2^p - 1))$ for some integer i , or a p -cube. Similarly, when the processor belongs to the left half and the record is larger than the entry, the record is transferred to the processor of distance p ; otherwise it remains there. Then the algorithm goes back to step 2. Since, as mentioned before, a record transfer takes a much longer time than a record comparison, only the number of record transfers are counted as our performance measure.
5. The joining operation is completed when p becomes 0 for every record and all the records have been transferred to the corresponding processors of the file R and have been joined.

The above operations are performed as much in parallel as possible. If $K > 1$, there may coexist at each processor the records initially held in the processor and the records transferred from other processors. This algorithm always gives priority to the transferred records over the initially held records. In the algorithm, record transfers are notably dominant over other operations, such as record comparisons. Therefore, the "step" of the algorithm is defined as the record transfer, usually consisting of the operations 2 through 4. However, if this cycle of the operations 2 through 4 does not include a record transfer, it will not be counted as a step. This step is the unit in the following simulation analysis.

Simulation Analysis

Since the mathematical analysis of the algorithm is extremely difficult, simulations are necessary to evaluate it. Simulations are first made in the cases of $K = 1$ and $K = 20$ where the record values are all unique for each set of N records or permutations of N different records. The results are shown in Tables I and II. In these tables, parameter q is the number of simultaneous inputs allowed into a processor. Since only a single record can be transferred in one direction on a line at a time, q can not exceed $\log_2 N$. In the cases that the record values are independent or randomly chosen, the results are shown in Tables III and IV for $K = 1$ and $K = 20$, respectively.

Observations

First, note that the simulation results are very encouraging. In particular, the results are enlightening when $K > 1$.

TABLE I—Simulation result I:
Unique record values and $K = 1$

N	q	Mean number of steps	Standard deviation	90% Confidence interval of the mean
64(= 2 ⁶)	1	8.1	0.71	8.1 ± 0.12
	2	6.7	0.71	6.7 ± 0.12
	6	6.7	0.77	6.7 ± 0.13
128(= 2 ⁷)	1	10.1	0.95	10.1 ± 0.16
	2	8.3	0.71	8.3 ± 0.12
	7	8.3	0.77	8.3 ± 0.13
256(= 2 ⁸)	1	11.8	0.89	11.8 ± 0.15
	2	10.0	0.77	10.0 ± 0.13
	8	9.9	0.77	9.9 ± 0.13
1024(= 2 ¹⁰)	1	15.8	1.00	15.8 ± 0.16
	2	13.5	0.89	13.5 ± 0.15
	10	13.2	0.84	13.2 ± 0.14
4096(= 2 ¹²)	1	19.6	0.77	19.6 ± 0.41
	2	17.2	0.89	17.2 ± 0.48
	12	16.5	0.77	16.5 ± 0.41

Observation 1

For $K > 1$, the mean number of steps per set of N records is very close to $D = \log_2 N$. When the record values are all unique for each set of N records the mean number of steps per a set of N records exceeds $\log_2 N$ by at most 1.5 for $N \leq 4096$. If two-record input is allowed ($q = 2$), the mean number of steps is even less than $\log_2 N$. When the record values are independent (record value duplications are allowed), the mean number of steps per set of N records exceeds $\log_2 N$ by 2 to 5 for $N \leq 1024$. This is less than a 50% increase over $\log_2 N$. If two-record input is allowed, the excess is only 1 to 3.

In large data base systems, which are the major application area of our algorithms, each processor would usually hold a much greater number of records than 20. Thus the above result is very encouraging. Considering the simplicity of the algorithm and the technical and economic feasibility of the interconnection, we can realistically expect a performance improvement of three to four orders of magnitude by employing this algorithm.

Observation 2

When $K = 1$, more steps are required. However, when the record values are all unique, the number of steps exceeds $\log_2 N$ by only 2 to 7, or a 50% increase for $N \leq 4096$. When the record values are independent, the number of steps may reach 2 to 3 times of $\log_2 N$. This is still a very good result. In fact, it is rather surprising that the number of steps is within 2 to 3 times $\log_2 N$ considering that "quick sort," one of the fastest sequential algorithms, requires twice as much as its theoretical lower bound, $N \log_2 N$.²

Observation 3

The standard deviations are sufficiently small that in actual operations a very uniform performance can be expected.

DETERMINATION OF ROUTING TABLES AND SORTING ALGORITHMS

The algorithm proposed in the previous section assumes that the routing tables are known. In actual systems, some information about the distribution of the file in question is usually available, and from this information the routing tables can be easily prepared. The distributions of human names and ages, for instance, are well known. If the distribution is not known, then it can be obtained by just once sorting the file by some means. Then the routing table can be easily prepared by keeping the obtained information with the file. This distribution information need not be as exact as search trees. It also need not be updated each time a file is updated. It can be an approximate distribution function. Thus the determination of the routing tables is not as difficult as it seems. Furthermore, the amount of distribution information will not be large.

A general procedure for a join operation for two files where the distribution of one file, say file R , is known is first to move the records of the file R to their proper processors based on the routing tables determined by the known distribution, and second to perform the joining operation described in the previous section for the other file, say file S . The first step is basically the same as the joining operation except that the last record-to-record joins are not necessary. The total steps of this joining operation are then twice the steps of a join oper-

TABLE II—Simulation result II:
Unique record values and $K = 20$

K	N	q	Mean number of steps	Standard deviation	90% Confidence interval of the mean	Mean number of steps per a set of N records
20	64(= 2 ⁶)	1	126.0	2.55	126.0 ± 0.43	6.3
		2	100.7	1.76	100.7 ± 0.55	5.0
		6	98.6	1.64	98.6 ± 0.27	4.9
	128(= 2 ⁷)	1	149.7	1.95	149.7 ± 0.60	7.5
		2	119.5	1.55	119.5 ± 0.48	6.0
		7	116.6	1.45	116.6 ± 0.24	5.8
20	256(= 2 ⁸)	1	174.1	2.00	174.1 ± 0.33	8.7
		2	138.5	1.38	138.5 ± 0.23	6.9
		8	134.8	1.38	134.8 ± 0.23	6.7
	1024(= 2 ¹⁰)	1	221.9	0.95	221.9 ± 0.51	11.1
		2	176.8	1.18	176.8 ± 0.63	8.8
		10	172.4	1.55	172.4 ± 0.83	8.6
4096(= 2 ¹²)	1	270.7	1.84	270.7 ± 0.99	13.5	
	2	215.5	1.34	215.5 ± 0.72	10.8	
	12	208.9	0.71	208.9 ± 0.38	10.4	

TABLE III—Simulation result III:
Independent record values and $K = 1$

N	q	Mean number of steps	Standard deviation	90% Confidence interval of the mean
64(= 2 ⁶)	1	12.9	2.07	12.9 ± 0.64
	2	12.4	2.03	12.4 ± 0.63
	6	12.7	2.12	12.7 ± 0.66
256(= 2 ⁸)	1	19.4	1.63	19.4 ± 0.51
	2	19.1	1.30	19.1 ± 0.40
	8	19.5	1.75	19.5 ± 0.54
1024(= 2 ¹⁰)	1	27.8	1.95	27.8 ± 0.60
	2	28.2	2.12	28.2 ± 0.66
	10	27.8	1.95	27.8 ± 0.60

ation for a file of N records, namely $2(\log_2 N + \alpha)$. If files R and S can be simultaneously handled, then the whole operation would be completed in $(\log_2 N + \alpha)$.

The first step of the above procedure can be considered as a sorting algorithm because the records will be ordered. Therefore there is essentially no difference between joining and sorting in our algorithm.

Our remaining problem is to find an algorithm to determine the routing tables when there is no distribution information available. This problem might not be practically important but it is necessary for the completeness of our discussion. In this section, such an algorithm is proposed and analyzed. All the assumptions remain the same except that the routing tables are initially blank and $K = 1$.

Algorithm

The first entries of all the routing tables are filled with the value T_1 , which is the average of the four records at processors

TABLE IV—Simulation result IV:
Independent record values and $K = 20$

N	q	Mean total number of steps	Standard deviation	90% Confidence interval of the mean	Mean number of steps per a set of N records
64(= 2 ⁶)	1	159.8	6.70	159.8 ± 2.08	8.0
	2	135.8	6.45	135.8 ± 2.00	6.8
	6	132.9	5.95	132.9 ± 1.84	6.6
256(= 2 ⁸)	1	231.3	10.40	231.3 ± 3.22	11.6
	2	196.7	6.92	196.7 ± 2.14	9.8
	8	190.8	6.24	190.8 ± 1.93	9.5
1024(= 2 ¹⁰)	1	299.1	8.28	299.1 ± 2.57	15.0
	2	255.2	7.55	255.2 ± 2.34	12.8
	10	254.7	7.72	254.7 ± 2.39	12.7

0, $N/2 - 1$, $N/2$, and $N - 1$. Then the records are moved through the lines of distance D so that the left half of the system holds records smaller than T_1 , while the right half holds records larger than or equal to T_1 . The second entries of the left half are next filled with the average of its four corresponding records, and the second entries of the right half are filled with its four records. Then the records are moved through the lines of distance $D - 1$. Similarly, the third entries are filled with the four records of the newly created quarters. This step continues until all the entry values are determined.

However, if the record distribution is unbalanced after one set of moves, corrections are made. If the ratio of the numbers of records of the two halves is not within the given threshold, we perform corrective record transfers. The threshold is defined by the following inequality

$$1 - \theta \leq \text{ratio} \leq 1 + \theta$$

where θ is some positive number. Since the number of records is discrete, the range is expanded to include the two boundary integers.

The corrective transfers are defined below. Let us identify the left and right halves by H_L and H_R and denote the number of resulted records of each H_L and H_R by p_L and p_R . Let us further denote by B_L and B_R the left and right boundary values of the segment to be divided into halves. If the ratio of p_L and p_R is not within the threshold, the present level entry value is recalculated by

$$E' = E + \frac{p_R - p_L}{2p_R} (B_R - E) \text{ if } p_L < p_R$$

$$E' = E - \frac{p_L - p_R}{2p_L} (E - B_L) \text{ if } p_L > p_R$$

where E is the current entry value that resulted in having p_L and p_R records in H_L and H_R . After the new values are set, the records are retransferred. This correction is continued until a satisfactory division is produced. This whole procedure is repeated until all the entries are determined.

Simulation Results

The simulation results for the above algorithm are shown in Table V. The distribution of the number of records at a processor is shown by its most important value, the maximum number of records concentrated in a node. A correction is counted as one step. Although divisions of records can be made in parallel in each level except the top level, the total number of steps is computed as the sum of the maximum number of steps of each level.

We can observe that the record concentration at a node is less than 4. This is satisfactory. The number of steps is 2 to 3.5 times $\log_2 N$ for $N \leq 1024$. This is relatively satisfactory.

Further improvements of the algorithm are possible. There is the potential for two major improvements; one is to use distribution information and the other is to improve the details of the algorithm. The first method is very effective and

TABLE V—Simulation result V:
Algorithms for determining routing tables

N	θ	Mean number of steps	90% confidence interval of the mean	Maximum number of records at a processor	90% confidence interval of the maximum number of records at a processor
64	0.1	11.23	11.23 ± 0.50	2.27	2.27 ± 0.06
256	0.1	22.00	22.00 ± 0.89	2.90	2.90 ± 0.12
1024	0.1	35.29	35.29 ± 2.84	3.00	3.00 ± 0.00
64	0.001	12.80	12.80 ± 0.71	2.10	2.10 ± 0.17
256	0.001	25.76	25.76 ± 0.78	2.71	2.71 ± 0.19

is also applicable in most cases. If detailed distribution information is available, then both the record concentrations and the total steps can be kept small. However, even if the detailed information is not available, some gross or partial information such as the middle value, the mean value, the maximum and minimum is a great help. In particular, the mean and the middle values are very useful because the higher level entries are more important.

The improvements of the algorithm may involve many things; the number of sample records, the way of picking sample records, the correction methods, and the way of transferring records are all important. In record transfers, the whole record need not be moved, only the key portion. Also all the lines of different distances may be used from the beginning for faster record transfers. As for the correction methods, more sophisticated functions can be used to interpolate and extrapolate the record value distributions. Finally, it may

be more efficient to vary the number of sample records according to the level.

CONCLUSIONS

The major application area of the algorithms is in large database systems. In such systems, each processor holds a large number of records and the algorithm can take advantage of this. Since the interconnection is technically and economically feasible, a performance improvement of three to four orders of magnitude can be realistically expected by applying the algorithm to large database systems such as database machines.

Although basic aspects of the algorithm are described in this paper, more detailed analyses and refinements are desirable. The remaining analysis includes more exact mathematical analysis to further the understanding of the algorithm, applications of the optimal interconnections, the effect of an unbalanced distribution of records over processors, the effect of different distributions, and improvements of the routing-table algorithms.

REFERENCES

1. Batcher, K. E., "Sorting Networks and Their Applications," *SJCC, AFIPS Proc.*, Vol. 32 (1968), Washington, D.C., pp. 307-314.
2. Knuth, D. E. *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*. Addison-Wesley, 1973.
3. Maekawa, M. "Extensibility and Adaptability of Distributed Processing Systems," *Proc. COMPSAC 80*, Chicago, Oct. 29-31, 1980.
4. Stone, H. S., "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Computers*, Vol. 1 (1971), C-20, No. 2, pp. 153-161.
5. Thompson, C. D., and H. T. Kung, "Sorting on a Mesh Connected Parallel Computer," *Comm. ACM*, Vol. 20, No. 4 (April 1977), pp. 263-271.
6. Uemura, S., and M. Maekawa, "Database Machines," Information Processing Society of Japan, July 1980.

Highly parallel associative search and its application to cellular database machine design

by SAKTI PRAMANIK

Indiana University-Purdue University
Indianapolis, Indiana

ABSTRACT

This paper describes a fast associative search algorithm based on parallel searching by pattern-splitting. The text is read as a sequence of substrings and searching is parallel within each substring. Substring length can be arbitrarily chosen and this division is independent of the logical partitioning of the data, such as tuple and domain. This provides a flexible storage structure across tracks of the secondary storage. The algorithm developed is used in the design of a hardware associative search. It works directly on the secondary storage and is a basis for database machine design. The design is cellular in structure and can be implemented by using LSI technology.

INTRODUCTION

A good part of nonnumeric computing is spent in searching and sorting. This searching is associative because a user wants to access data by its content or a logical ID. The existing computer architecture however requires data to be accessed by its address. This difference in the mode of data access requires a very complex software mapping of the user's query into the physical address of the data. In fact, in a generalized database management system this mapping is performed in three distinct levels, namely external, conceptual, and internal data models.⁵ A lot of auxiliary data structures are used to implement this mapping. This complexity is imposed by the conventional Von Neumann machine architecture of the existing computers. This architecture is quite appropriate for numeric processing but is very inefficient⁶ for nonnumeric applications.

The machine instructions of the conventional computers are designed to process data residing in random access main memory. Though significant progress has been made in the size and speed of these memory devices, they are still not cost effective for storing a huge amount of data that need to be kept on-line. These data are stored in less expensive secondary storage like rotating disks. The trend in memory technology indicates that newer devices like bubble,⁹ CCD,⁷ holographic,⁷ and electron-beam accessed memory⁸ will become available commercially, but they have characteristics similar

to those of the rotating disks, i.e., accessing a block of data at a time. The existing computer architecture is not very efficient in processing information on secondary storages. This is because a lot of data has to be transferred into the main memory, while only a small fraction of it is really needed. To avoid unnecessary transfer of a huge amount of data, the information can be searched directly on the secondary storage. When the search is successful the needed data is transferred into the main memory. A technique called "LOGIC PER TRACK DEVICES" first developed by Slotnick was used to search data directly on a head per track disk.^{22,16} He used search-hardware logic for each track. Here data is accessed sequentially within a track and randomly between tracks. Searching goes on in parallel between tracks, increasing the search speed considerably. A significant amount of research work has been done in developing computer architecture for database applications based on this logic-per-track devices approach.^{4,10,14,15,21} For large textual database similar search technique is also very applicable. Hardware algorithm of pattern matching in text has been developed by many authors.^{3,13}

This paper describes fast associative search algorithms based on cellular logic.¹³ Here the text is read, one substring at a time, from a circulating memory and a search-hardware logic finds all possible matches within the substring. There is one search-hardware logic for a group of tracks and a substring can be stored across these tracks. The complexity of each search-hardware logic is reduced considerably by splitting both the pattern as well as the text substrings. It is also seen that this splitting helps in achieving more parallelism.

BASIC PATTERN-MATCHING HARDWARE BASED ON CELLULAR LOGIC APPROACH

The hardware algorithms of pattern matching use highly parallel character-matching techniques. This is explained through an example below. In this example the pattern is assumed to be 'ABA' and it is to be matched against the text string 'ABABAB'. Multiple matches will be considered in this example. The first character 'A' of the text is first compared against all the characters of the pattern and the result is stored in a bit vector. The bit vector in this case is '101', '1' represent-

Text characters	A	B	A	B	A	B
Bit Vectors	1	0	1	0	1	0
	0	1	0	1	0	1
	1	0	1	0	1	0

Figure 1a—Bit vectors

Text characters	A	B	A	B	A	B
Resulting Bit Vectors	1	0	1	0	1	0
	0	1	0	1	0	1
	1	0	1	0	1	0

Figure 1b—Resulting bit vectors

ing a match and a '0' representing a mismatch. Similarly the bit vector for the second text character 'B' is '010', third character 'A' is '101', and so on. The bit vectors for all the text characters are shown in Figure 1a below. Now the bottom two bits (the count is the number of pattern characters minus one) of each vector is 'AND'ed bitwise with the top two bits of the next bit vector. This is done sequentially, starting with the leftmost bit vector. The resulting bit vectors are shown in Figure 1b. A match is represented by a '1' bit at the top of a resulting bit vector. Since the pattern is three characters long we start with the third resulting bit vector for interpreting a match. The complete match is represented by the circled bits of Figure 1b. The corresponding characters of the text-string represent the ends of the matching text.

This technique of finding matches can be used to develop a hardware algorithm for pattern matching. Here we are interested in cellular hardware design involving interconnection of identical cells. A cell in this design consists of a single character comparison hardware as shown in Figure 2 below. Here y_i represents the i th character of the pattern and x is a text character. The relationship between f_i , s_i and f_{i-1} is given in the algorithm below. A complete pattern matching hardware for a pattern $y_1, y_2 \dots y_m$ is obtained by cascading m cells as follows: Here text characters are read one at a time and are compared against all the characters of the pattern.

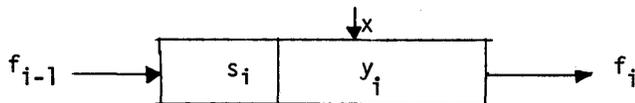


Figure 2—A basic cell

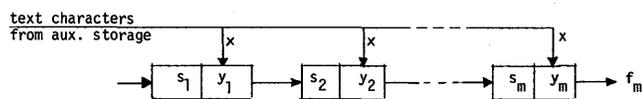


Figure 3—Pattern matching hardware

Algorithm 1.

Step 1. Initialize $s_2 \leftarrow s_3 \leftarrow \dots \leftarrow s_n \leftarrow 0$

Step 2. Set $s_1 \leftarrow '1'$

Read next character x of the string, and compare it against all the characters of the pattern simultaneously.

Step 3. $f_i \leftarrow '1'$ if $s_i = '1'$ and $y_i = x$ for $i = 1, 2, \dots, m$, $f_i \leftarrow '0'$ otherwise

Step 4. $s_{i+1} \leftarrow f_i$ for $i = 1, 2, \dots, m-1$

$f_m = 1$ implies current text character represent the end of a matching text.

Step 5. Go to Step 2.

A pattern of length j where $j \leq n$ can be taken care of in the above circuit by storing it right-justified and setting s_{m-j+1} to '1' in Step 2 of the algorithm.

The above algorithm reads one character at a time from the text. We may however like to read more than one character at a time and process it before reading the next text substring. For example, we may store character bit serially on two adjacent tracks as follows and read the text string 'ABCABCAB' as 'AB' then 'CA', and so on. (See Figure 4.) Now consider a pattern 'ABCA', and split it into two subpatterns 'AB' and 'CA', and store them in two adjacent cells as shown in Figure 5. Now a cell needs to be redefined as in Figure 5; only the equality match in each cell is not enough now. All the partial matches also have to be considered. For example, the text substring 'BC' and the subpattern 'CA' will have three possible partial matches as shown in Figure 5. Thus both the cell definition and the algorithm above have to be extended to take care of the general case. Some of the advantages of this general model are more parallelism in searching, greater flexibility in memory usage, and better utilization of the storage device. This is achieved however, at the expense of more complexity in the design of a cell and the algorithm. But we will see in later sections that this complexity can be minimized considerably by using cellular hardware design.

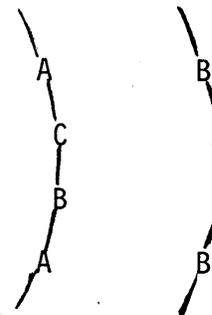


Figure 4—Storing text string 'ABCABCAB' on two adjacent tracks

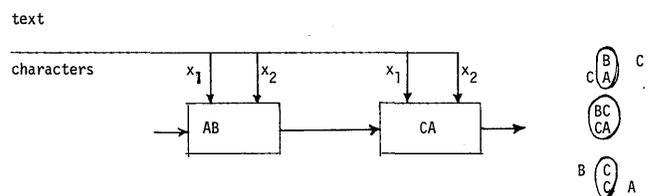


Figure 5—Two-character cells

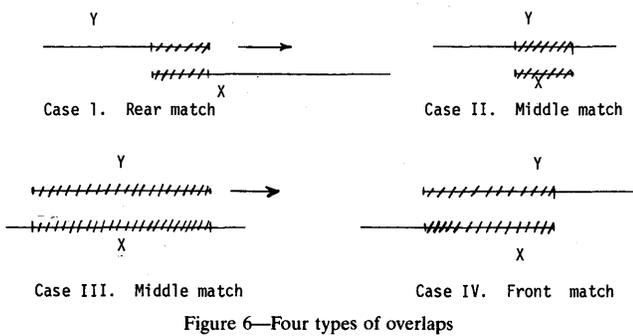


Figure 6—Four types of overlaps

COMPUTATION OF PARTIAL MATCHES AND GENERATING PARTIAL MATCH VECTOR

Let X and Y denote the text and pattern substrings respectively. X_H and X_T are used to denote the head and tail of the substring X . If the substring $X = 'abcd'$, then X_H will be the character 'a' and X_T will be the character 'd'. Similarly the Y_H and Y_T are the head and tail of the pattern Y . Now we slide the pattern Y over the text X from left to right, as shown in Figure 6 below. As Y slides over X the amount of overlap between Y and X changes. This is shown by the shaded portions. Each time Y is moved to the right by one character position, the characters that are overlapping are compared. If a match occurs between the overlapping characters, the result is a bit '1' else it is '0'. There are four possible types of overlaps between Y and X , as described in Table I below. Each of these four types of overlaps is also shown pictorially in Figure 6. We will differentiate only three types of overlaps namely rear match, middle match, and the front match. The rear matches as shown in Figure 6 correspond to all possible overlaps of the rear part of the pattern Y . We will define rear match bit vector simply denoted by the letter 'R', as the bits representing all the rear matches. Thus the pattern 'ABAB' and the text string 'ABA' for example, will have two rear matches as shown in Figure 7, below. The resulting bit vector **R** is also shown in the figure. Similarly, the results of front and the middle matches will be denoted by the bit vectors "F" and "M" respectively. The **M** and **F**, for the example above, are also given in the figure. We combine these three bit vectors into one and call it partial matches vector table, PMT in short. The PMT for the example above is shown in Figure 7. It should be noted that reading the bits of PMT downwards corresponds to sliding the pattern from left to right over the text substring. The top bit of the PMT corresponds to the rightmost character of the pattern being aligned over the leftmost character of the text substring.

In general, if the pattern Y consists of m characters, i.e., $Y = y_1 y_2 \dots y_m$ and text substring X has n characters, i.e., $X = x_1 x_2 \dots x_n$, then the sizes of **R**, **M**, **F** and PMT will be given by Figure 8. It should be noted that the interpretation of M_i s is different for different relative values of m and n . The length of a PMT is $m + n - 1$ bits. The following relations are also true

$$len_R + len_M = len_M + len_F = \text{Max}(m, n)$$

where len stands for the length of a section of PMT.

TABLE I—Definition of four types of overlaps

Y_T has passed X_H	Y_H has not passed X_H	I
	Y_H has passed X_H	II
Y_T has passed X_T	Y_H has not passed X_H	III
	Y_H has passed X_H	IV

PATTERN MATCHING IN LONG TEXT STRING

Pattern matching in long strings by merging PMT's

Pattern matching in a long text string is found by splitting it into smaller substrings and obtaining PMT's of the pattern for each of these substrings. Then the PMT's are merged as shown below to determine the complete matches in the whole text string. For example, the text string "ABABAB" can be split into the three substrings S_1, S_2 and S_3 , as shown in Figure 9 below. For a pattern "ABA" the PMT's for each of S_1, S_2 and S_3 are shown. Now the bottom two bits of each PMT is 'AND'ed with the top two bits of the next PMT and the result replaces the top two bits. The complete matches are interpreted from the top two bits of the 2nd and the 3rd PMT's. Here the first bit of the 2nd and 3rd PMT's equal to 1, implying matches at the first character of the 2nd and the 3rd substrings.

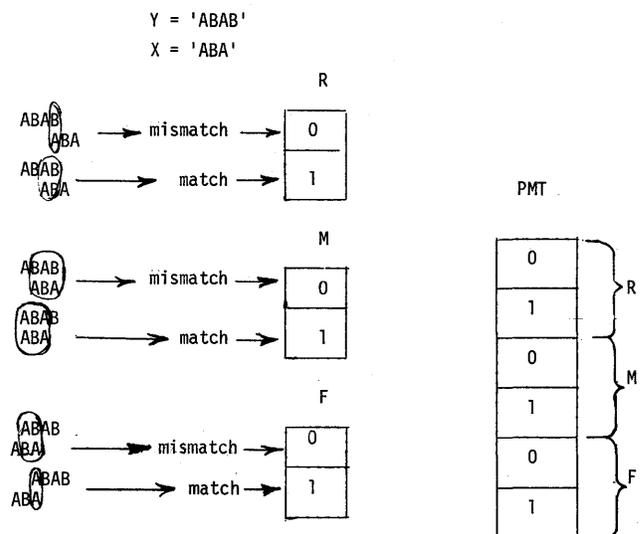


Figure 7—Partial match bit vectors

For this figure the values of l and i are as follows:

- $l = \min(m, n)$ and $i = 1, 2, \dots, l - 1$
- $R_i = 1$ when $y_{m-i+1}, y_{m-i+2}, \dots, y_m$ matches x_1, x_2, \dots, x_i
 $= 0$ otherwise
- $F_i = 1$ when y_1, y_2, \dots, y_i matches $x_{n-i+1}, x_{n-i+2}, \dots, x_n$
 $= 0$ otherwise
- $K = n - m + 1$ and $i = 1, 2, \dots, K$
 when $m > n$
- $M_i = 1$ when $y_{k-i+1}, y_{k-i+2}, \dots, y_{m-i+1}$
 matches x_1, x_2, \dots, x_n
 $= 0$ otherwise
 when $m \leq n$
- $M_i = 1$ when y_1, y_2, \dots, y_m matches $x_i, x_{i+1}, \dots, x_{m+i-1}$
 $= 0$ otherwise

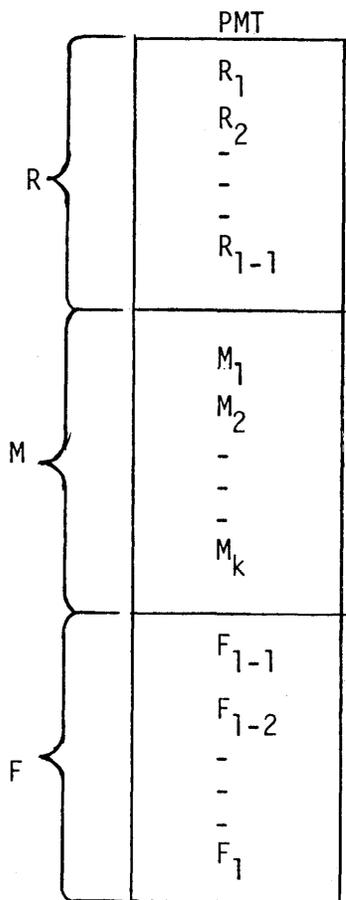


Figure 8—PMT for different relative sizes of X and Y

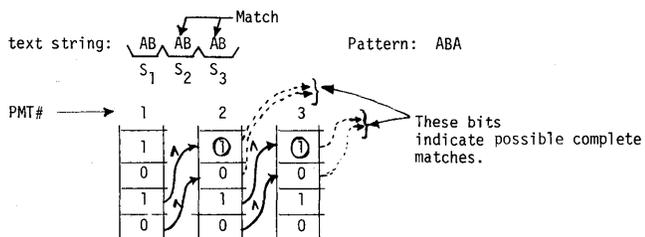


Figure 9—Merging PMT's of the example problem

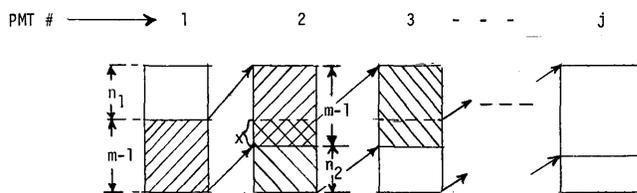
General merging algorithm

Let X_1, X_2, \dots, X_j be the j text substrings of lengths n_1, n_2, \dots, n_j respectively. These substrings will have j number of PMT's as shown below. If S_i^k represent the k th bit from the top of the i th PMT, the algorithm will be as follows:

Algorithm 2.

- Step 1. $i \leftarrow 1$
- Step 2. $S_{i+1}^k \leftarrow S_{i+1}^k \wedge S_i^{k+1}$ for $k = 1, 2, \dots, (m - 1)$
- Step 3. When $S_{i+1}^k = 1$, and $k \leq n_{i+1}$, and $n_1 + n_2 + \dots + n_i + k \geq m$ it is a complete match, else it is not. K th character of the $(i + 1)$ th substring corresponds to the match
- Step 4. $i \leftarrow i + 1$
- Step 5. $i = j \Rightarrow$ done else GO TO Step 2.

Step 2 above is a bitwise 'AND' operation of the bottom $(m - 1)$ bits of the i th stage with the top $(m - 1)$ bits of the $(i + 1)$ th stage. Step 3 gives the bits that will indicate, after 'AND'ing, the complete match conditions. 'AND'ing of bits of one stage with those of the next is necessary for finding complete match conditions. This is because every bit in a PMT represents only a partial match condition (i.e., only a part of the pattern is being matched). By 'AND'ing the corresponding bits of the successive PMT's we are in effect finding match conditions for the concatenated text string. If $n_i \geq m$ the M bits of the PMT will correspond to the complete match conditions. Thus the M bits of the PMT, when $n_i \geq m$ do not participate in the 'AND'ing because they already represent complete match conditions. The R and F bits of the PMT's will



The length of the i th PMT $l_i = m + n_i - 1$

FIGURE 10—Merging PMT's of varying lengths

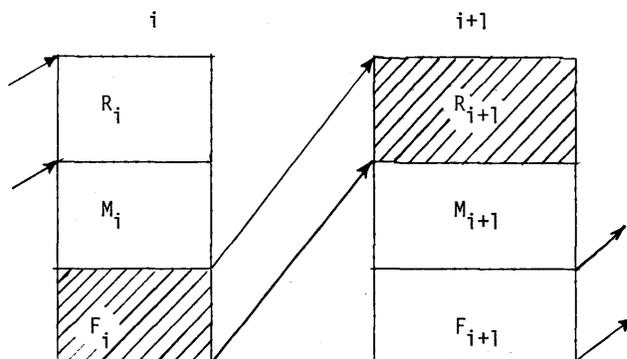
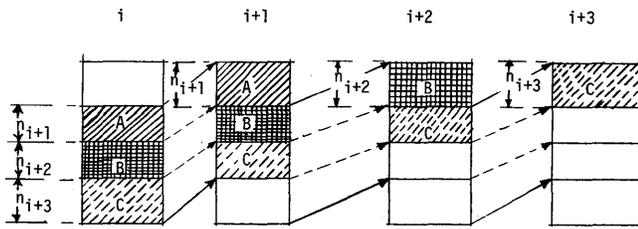


Figure 11—Merging PMT's when $n \geq m$

FIGURE 12—Sequential merging of stages i thru $i+3$

always correspond to partial matches and participate in the 'AND'ings process. Figure 11 shows the merging process when $n_i \geq m$. The algorithm given above however, is general and works for both $n_i \geq m$ and $n_i < m$.

Figure 11 shows that the middle M bits are unaffected by the 'AND' operations. Thus each adjacent pair of PMT's can be 'AND'ed in parallel because the 'AND'ed bits are disjoint between adjacent pairs of PMT's. In Figure 10 where $n < m$, the middle x bits are not disjoint and the 'AND'ing has to be carried over to several stages.

Suppose n is greater than or equal to m and we keep reducing n while keeping m same. As the length n shrinks so does the middle section M of Figure 11 and it becomes 1 when n equals m . At this point the length of the section F is still $m-1$. Reducing n further will increase the size of M again. But the length of F will start to decrease becoming less than $m-1$. Since the bottom $m-1$ bits always participate in the 'AND'ing some of the M bits will not become involved. At this point more than half of the bits from bottom are 'AND'ed with equal number of bits from the top of the next PMT. Thus some of the 'AND'ed bits are not disjoint any more.

Parallelism in merging

The algorithm 2 given in the previous section will work for varying length substrings. This algorithm is sequential and it computes PMT's of one substring at a time and then merges it with the result of the previous merge. Of course, the computation time of the PMT's can be overlapped with the time of merging these PMT's. Moreover, several PMT's can be merged together simultaneously, achieving parallelism in merging also. But simultaneous merging is done at the cost of a uniform design. Instead we can merge groups of several adjacent stages (i.e. PMT's) sequentially and between the groups in parallel. This is explained in Figure 12 below. The bits of stage i are 'AND'ed with the bits of stages up to $i+3$. For example, the C bits of stage i are 'AND'ed with the C bits of stage $i+1$. These resultant bits of stage $i+1$ are 'AND'ed with the C bits of stage $i+2$, and so on. The figure above shows that the bits of stage i have no effect on those of stages after $i+3$. Therefore, only the stages i thru $i+3$ have to be merged sequentially and we merge PMT's by 'AND'ing bottom $(m-1)$ bits of each stage with the top $(m-1)$ bits of the next. Assuming the substrings are of equal lengths in Figure 12, the groups to be merged sequentially are as follows: (1,2,3,4), (2,3,4,5), (3,4,5,6), and so on. If the length of each substring is n then the number of stages within a group is $\lceil \frac{n}{m} \rceil + 1$. For varying length substrings this will be given by the

following condition: stages i thru $i+j$ have to be merged in sequence if

$$n_{i+1} + n_{i+2} + \dots + n_{i+j-1} < m - 1$$

$$\text{and } n_{i+1} + n_{i+2} + \dots + n_{i+j} \geq m - 1$$

Since the 'AND' operations are associative we could use the following repetitive steps to perform the above merging:

1. Merge all adjacent pairs in parallel—
(1,2), (2,3), (3,4), ...
2. Repeat Step 1 $\lceil \frac{n}{m} \rceil$ times.

HARDWARE IMPLEMENTATION OF THE ABOVE ALGORITHMS AND THEIR APPLICATION TO NONNUMERIC PROCESSING

Introduction

The concepts and algorithms developed in the previous sections can be applied to database machines where the content searching is done in hardware. In database management systems and text processing systems bulk-data is stored on the auxiliary storage, which can be head per track disk, bubble memory, CCD devices, or others of this type. These memory devices are sequential-access within each cell called track and random-access between cells. Because a piece of data once read off the device cannot be read again until after one complete rotation, pattern-matching algorithms must use the character only once for comparison with the pattern. Since characters are read at high speed, the comparison has to be done at a comparable speed to keep it up with the speed at which characters are being read. The "Logic per track" technique has been applied by many authors to match patterns on the storage device directly when the data is on the fly. We will discuss several techniques here which are based on the concepts developed in the previous sections. This hardware-associative-search is also appropriate for pattern matching in large-text file editing systems. Many efficient pattern matching algorithms for text editing systems are in existence.^{9,1} But these are complex software algorithms and the complexity is due to the Von Neumann machine architecture. Data-handling mechanics of on-line text editors is rather inefficient, because the machines they are implemented on require a complex interaction between the auxiliary and the primary storages.^{17,18,19} The technique discussed here is the hardware solution of the content searching problem. The searching is done directly on the auxiliary storage.

A hardware algorithm for finding a match for the pattern y_1, y_2, \dots, y_m in a text string consisting of fixed length substrings x_1, x_2, \dots, x_n is given below. The hardware organization of the algorithm is shown in Figure 13. The output lines marked * in the figure do not depend on status bits s_i 's. When $n < m$, however, all output lines including some of the feedback lines may become dependent.

Algorithm 3.

Step 1. $S_1 \leftarrow S_2 \leftarrow \dots \leftarrow S_{m-1} \leftarrow '1'$

Step 2. Compare \bar{X} and Y and compute the PMT

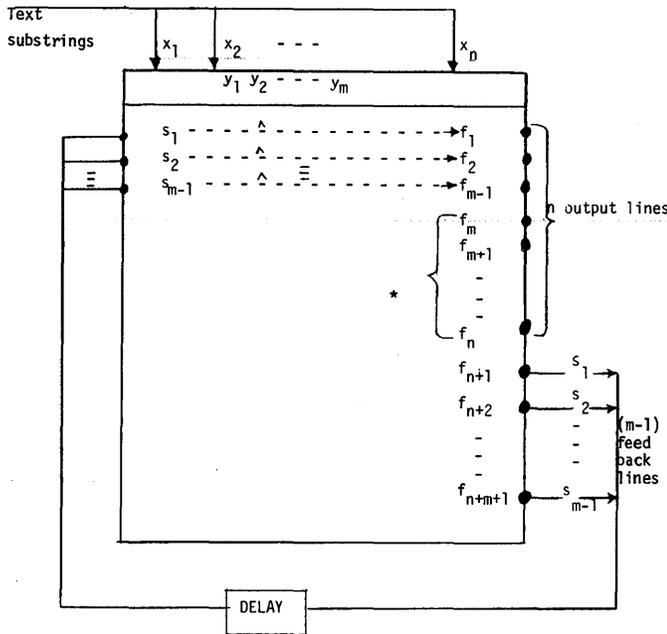


Figure 13—A single stage matching circuit

- Step 3. Perform $f_i \leftarrow f_i \wedge S_i$ for $i = 1, 2, \dots, m - 1$, Output lines f_i for $i = 1, 2, \dots, n$ represent match at the i th character of the text substring.
- Step 4. $S_i \leftarrow f_{n+i}$ for $i = 1, 2, \dots, m - 1$
- Step 5. Go to Step 2.

Implementation issues and modular design

For large patterns the hardware to find PMT's or to implement it in a single chip may be impractical. On the other hand, we can arbitrarily break a pattern into smaller subpatterns and cascade the results of each of these subpatterns. Thus, the implementation of each of these stages can be made very simple and repetitive. Cascading of stages also becomes simple because there will be fewer lines between stages, as shown in Figure 15 below. Here we will be using the stage of Figure 13 as the basic building block. The output lines of each stage will be cascaded forward to the next stage as follows. The top $(n - 1)$ output lines of i th stage are connected to the bottom $(n - 1)$ status lines of the $(i + 1)$ th stage. The n th output line of the i th stage is connected to the m th status line of the next stage through a delay. There are altogether $m + n - 1$ status lines out of which $m - 1$ are coming from the feedback lines of the same stage. The functioning of the circuit of Figure 14 is described in algorithm 4 below. It is assumed that there are j numbers of stages.

Algorithm 4.

- Step 1. Initialize: $s_1^1 \leftarrow s_2^1 \leftarrow \dots \leftarrow s_{m-1}^1 \leftarrow 1$
The superscript represents the stage number.
- Step 2. (a) $s_m^1 \leftarrow s_{m+1}^1 \leftarrow \dots \leftarrow s_{m+n-1}^1 \leftarrow 1$
(b) Read the next text substring and compute the PMT's for all the stages.

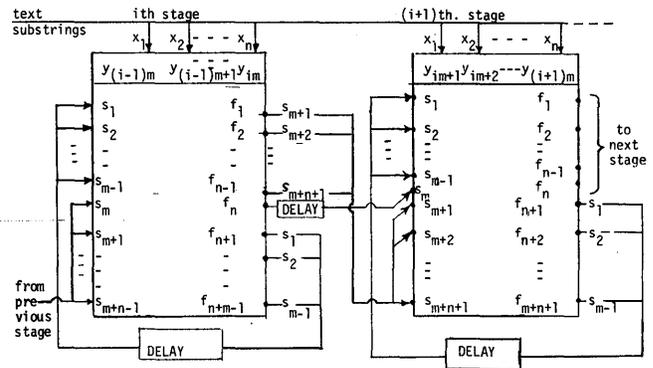


Figure 14—Multistage matching circuit

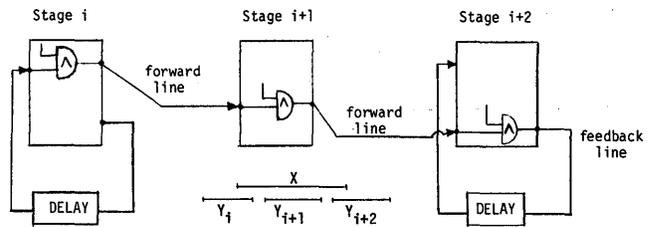


Figure 15—Gate delays for merging

- Step 3a. $f_i^k \leftarrow f_i^k \wedge s_i^k$ for $i = 1, 2, \dots, n - 1, k = 1, 2, \dots, j$
- Step 3b. $s_{m+1}^{k+1} \leftarrow f_i^k$ for $i = 1, \dots, n - 1, k = 1, 2, \dots, j - 1$
- Step 3c. $f_{m+i}^{k+1} \leftarrow f_{m+i}^{k+1} \wedge s_{m+i}^{k+1}$ for $i = 1, 2, \dots, n - 1, k = 1, 2, \dots, j$
- Step 3d. Iterate steps 3a thru 3c $\lceil \frac{n-1}{m} \rceil$ number of times. No iteration is necessary when $n \leq m$.
- Step 4. $S_i^k \leftarrow f_{n+i}^k$ for $i = 1, 2, \dots, n - 1, k = 1, 2, \dots, j$
- Step 5. $s_m^{k+1} \leftarrow f_n^k$
- Step 6. Go to Step 2.

Steps 2 through 5 are called the major cycle while 3a thru 3c are in the minor cycle. The iteration of the minor cycle is required when $n > m$. This is because a text substring may extend over several subpatterns requiring concatenation of the subpatterns within the same major cycle. When $n \geq m$ no such iteration is necessary because a substring does not extend over subpattern. In the latter case a subpattern will extend over text substrings requiring concatenation of the substrings per subpattern. This is taken care of by the major cycle. Thus the algorithm above is greatly simplified when $n \leq m$. Steps 3a, 3b, and 3c can be performed in parallel and then iterating them once more. Similarly Steps 4 and 5 are also done in parallel.

9765—AFIPS Pramanik-4, 1514-1518 2/9765

Computation of cycle time

Text substrings are read from the auxiliary storage sequentially one at a time and are compared against all the

subpatterns simultaneously. A cycle begins with the reading of a text substrings and ends before the next substring-read starts. Thus a search time per substring must be less than or equal to the cycle time. The search time consists of the time to compare the substring against each of the subpatterns and the time to merge the result of these comparisons. In fact, the time to compare against each of the subpatterns in the current cycle can be overlapped with the merging time of the previous cycle. The relative sizes of n and m determines the merging time. When $n \geq m$, $\lceil \frac{n}{m} \rceil + 1$ stages have to be merged in sequence, consequently the merging time is $\lceil \frac{n}{m} \rceil + 1$ gate delays. When $n < m$ the merging time is two gate delays. Figure 15 shows the merging time of 3 gate delays because $\lceil \frac{n}{m} \rceil = 2$. It is theoretically possible to minimize this time by merging several stages simultaneously. Here the reduction in merging time is achieved at the cost of a modular design as shown above.

Cellular design and generalized associative search model

We can make the design of each stage above simpler by choosing the subpattern arbitrarily small. The design can be further simplified by selecting arbitrarily smaller text substrings. Moreover, the design of each stage can be made independent of the size of the text substring read from the storage device. Thus the model developed above is now generalized to searching several text substrings at a time instead of only one.

$$Y_1 = y_1 y_2 \dots y_m, Y_2 = y_{m+1} y_{m+2} \dots y_{2m}, Y_i = y_{(i-1)(m+1)} y_{(i-1)(m+2)} \dots y_{im}$$

---> nth. forward line, ---> (m-1) feedback lines, —> (n-1) forward lines

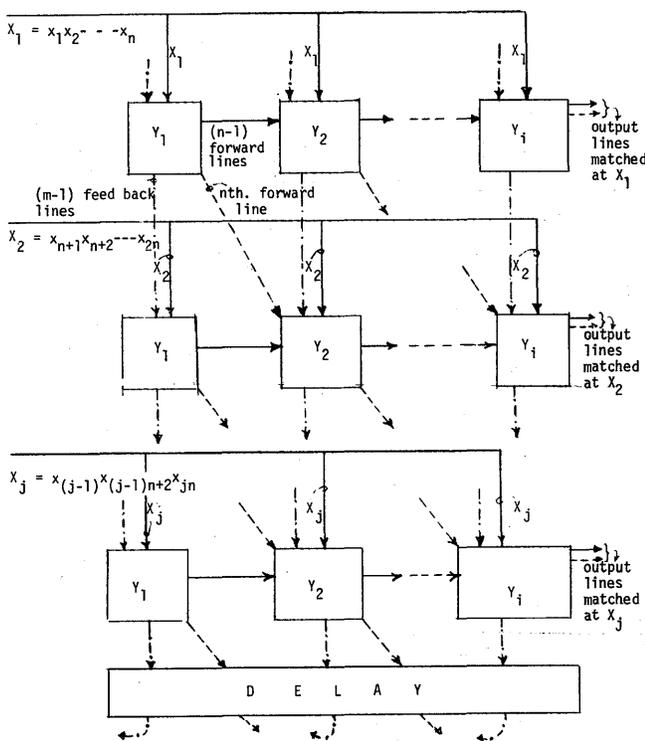


Figure 16—Generalized matching circuit

We will achieve this by cascading stages in two dimension as shown in Figure 16 below.

The output lines of the right most stages in the diagram give the result of complete matches. The result of partial matches can be obtained from the output of intermediate stages. The stages in the second column for example, gives the matches for the pattern Y_1, Y_2 . The DELAY at the bottom provides synchronization between the text substring x_j in the current cycle and x_{j+1} of the next cycle. X_{j+1} through x_{2j} are the text substrings of the next cycle.

Initialization of this circuit is done by setting the incoming feedback lines of the first stage of first row to 1's. There are no forward lines coming into the stages of the first column. These lines are permanently set to 1's.

The cycle time is determined in gate delays and each gate delay is due to the propagation of a forward or a feedback signal through a stage. Comparison of a substring with a subpattern for all the stages is assumed to be done in parallel. These operations can also be overlapped in time with the propagation of the forward and feedback signals of the previous cycle. Thus, the maximum delay per cycle for the above circuit is $i + j - 1$ gate delays.

Implementation of a single stage

A stage can be implemented by interconnecting simple cells as shown below. A cell consists of a single character text substring x_j and a single character subpattern y_i , as shown in Figure 17 below.

Thus a cell defined above is a special case of a stage where substrings are one character long. A stage can be constructed by interconnecting several cells in the same way stages were interconnected in Figure 16. Since the subpattern and the text substrings are one character long, there won't be any forward and feedback lines from each stage. The output lines f_{ij} of each cell correspond to the n th forward line of each stage of Figure 16. Thus a stage of n text characters and m pattern characters is as shown below.

The maximum delay for this circuit is m gate delays. This can be reduced by processing outputs of several cells simultaneously by look ahead circuits. The following design of a stage with 3 text characters and 3 patterns characters has only one gate delay.

CONCLUSION

This paper describes several hardware algorithms for finding pattern matches in a linear text string stored on a circulating

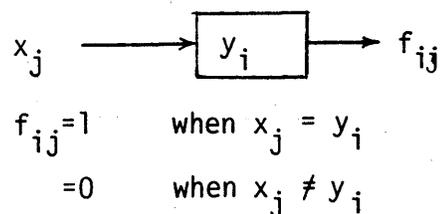


Figure 17—A basic cell

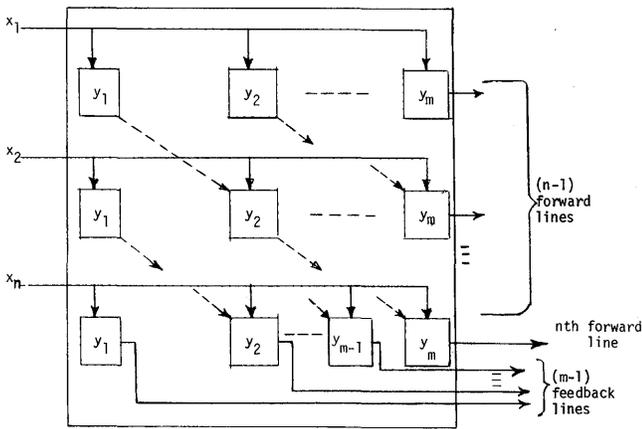


Figure 18—Design of a stage

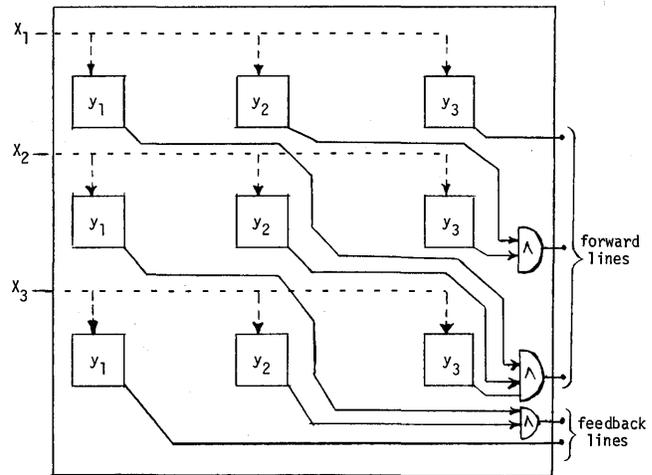


Figure 19—Design of a single stage with one gate delay

type device. The search process here is highly parallel because the text string is split into smaller substrings, and the search progresses within each cell in parallel. Reduced complexity of the hardware is achieved by properly cascading many simple cells, where each cell can be a single character comparator.

REFERENCES

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "The Design and Analysis of Computer Algorithms" Addison-Wesley, 1975: Reading, MA.
2. G.F. Amello, "Charged-coupled devices for memory applications", *AFIPS Conf. Proc.* Vol. 44 pp. 515-522, 1975.
3. R.M. Bird, J.C. Tu and R.M. Worthy, "Associative/parallel processors for searching very large textual data bases", *3rd Workshop on Computer Architecture and Nonnumeric Processing*, Syracuse Univ., Syracuse, NY, May 17-18, 1977.
4. G.P. Copeland, G.J. Lipovsky, and S.Y.W. Su, "The Architecture of CASSM: A cellular system for nonnumeric processing", *1st Annual Symp. on Computer Architecture*, 1973.
5. C.J. Date, "An Introduction to Database Systems" 2nd Edition, Addison-Wesley, 1977: Reading, MA.
6. J.F. Gimpel, "Algorithms in SNOBOL4", *Wiley Interscience*, 1976.
7. A.K. Gillis, et al., "Holographic memories-fantasy or reality?", *AFIPS Conf. Proc.* Vol. 44 pp. 535-539, 1975.
8. N.C. Hughs, et al, "BEAMOS—A new electronic digital memory", *AFIPS Conf. Proc.* Vol. 44, pp. 541-548, 1975.
9. D.E. Knuth, J.H. Morris, and V.R. Pratt, "Fast pattern matching in strings", *SIAM J. Comput.* Vol. 6, No. 2, pp. 323-350, June 1977.
10. C.S. Lin, D.C.P. Smith, J.M. Smith, "The design of a rotating associative memory for relational database applications," *ACM Trans. Data Base Systems*, Vol. 1, pp. 53-65, 1976.
11. Stephen W. Miller, Ed. "Memory and Storage Technology—@Vol. II" *AFIPS Press*, 1977.
12. N. Minsky, "Rotating storage devices as partially associative memories," *Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control*, 1972.
13. Amar Mukhopadhyay, "Hardware algorithms for nonnumeric computation", *IEEE Trans. Comput.*, Vol. C-28, June, 1979.
14. E.A. Ozkarahan, S.A. Schuster, and K.C. Smith, "RAP-An Associative processor for database management", *National Computer Conference*, 1975.
15. B. Parhami "A highly parallel computer system for information retrieval", *Proc. Fall Joint Computer Conf.*, 1972.
16. J.L. Parker, "A logic per track retrieval systems", *IFIP Congress*, 1971.
17. S.B. Pramanik, Edgar T. Irons "A data-handling mechanics of on-line text editing systems with efficient secondary storage access", *National Computer Conference*, 1979.
18. S.B. Pramanik, "A mathematical model of character string manipulation", *National Computer Conference*, 1980.
19. S.B. Pramanik, "MAP EDITING", Ph.D. Thesis, Yale University, 1974.
20. C.V. Ramamoorthy, J.L. Turner, and B.W. Wah, "A design of a fast cellular associative memory for ordered retrieval", *IEEE Trans. Comput.* Vol. 2-27, Sept. 1978.
21. S.Y.W. Su, G.P. Copeland, and G.J. Lipovsky, "Retrieval operations and data representation in a context addressed disk system", *Proc. ACM Programming Languages and Information Retrieval Interface Meeting*, 1973.
22. D.L. Slotnick, "Logic per track devices", *Advances in Computers*, Academic Press, 1970.

A generalized database access path model

by GEORGES S. NICOLAS

The MITRE Corporation
McLean, Virginia

ABSTRACT

Access paths are essential for the operation of database access methods, and their structures can greatly influence the search efficiency and storage requirements. In this paper we develop a general access path model that is shown to characterize a wide class of file organizations. An integrated file organization, which includes all the basic file organizations as special cases, is subsequently derived. If used as the underlying structure for a database, this integrated file organization can be subjected to optimization techniques to yield an optimal database performance.

INTRODUCTION

Access paths are essential for the operation of database access methods, and their structures can greatly influence the search efficiency and storage requirements. All of the important structures underlying a database system can be characterized by modelling the corresponding access paths. When optimized, the access paths structures will produce efficient organizations for the corresponding database files.

Generalized access path and file organization modelling have been studied by many researchers. The most important results are reported by Hsiao,¹ Yang,² Severance,³ and Yao.^{4,5} However, in every case the model presented suffers from lack of completeness and/or of significant practical value.

In this paper, we will develop a general access path model capable of characterizing a wide class of file organizations. This model will be used to identify a general file organization, which will be shown to include all the basic file organizations as special cases. More details on this general access path model can be found in an earlier article by Nicolas.⁶

REVIEW OF BASIC POINTS

Using the relational model terminology,^{7,8} we consider the database as consisting of a collection of named relations in normal form. These time-varying relations are of assorted degrees; and as time progresses, each n -ary relation may be subjected to transformation, extraction, deletion, and alteration of some or all of its existing tuples. Consequently, we

think of a file as a normalized relation or as a two-dimensional table in which rows correspond to records and columns correspond to attributes. The records in a file correspond to occurrences of a single record type that describes a specific class of real-world entities. A record is subdivided into a number of fields corresponding to the associated file attributes. The fields can, in general, be classified into two kinds: *key fields* and *data fields*. The contents of key fields, called *keys*, are used to distinguish one record from another and they can be used to specify the retrieval criteria in queries. A key is a *generic key* if it appears in more than one record in the file; otherwise it is called a *non-generic key*.⁹

Define a *keyword*, named K , to be a unique ordered triple (f, a, v) , where f represents a file name, a represents the name of one or more concatenated key attributes in the file, and v represents a key value from the set of values in the attributes represented by a .

The physical storage space on secondary storage devices, where the database is assumed to reside, is divided into fixed-size blocks that are considered the units of secondary storage allocation. A *storage cell*, hereafter referred to as simply a *cell*, is defined as a logically related set of blocks bearing numerically sequential addresses. It is completely specified by a cell identifier (CID) and a cell size. The cell is also considered to be a unit of data transfer between the processor main memory and the secondary storage devices.

Cells are divided into two categories: *index cells* and *data cells*. Index cells are used for storing indexing information, and data cells are used to store the records of the database files. A data cell may contain records from more than one file. Each record is assigned a unique identifier, called the record identifier (RID), which indicates the whereabouts of the record in the secondary storage space. The RID has two components: the CID and the record address relative to the cell (AID).

In the next section, the general access path model will be developed.

ACCESS PATH STRUCTURAL MODEL

The memory elements needed to represent an access path structure can be modelled by the following two basic storage objects: The *data object* (DO), which normally contains the

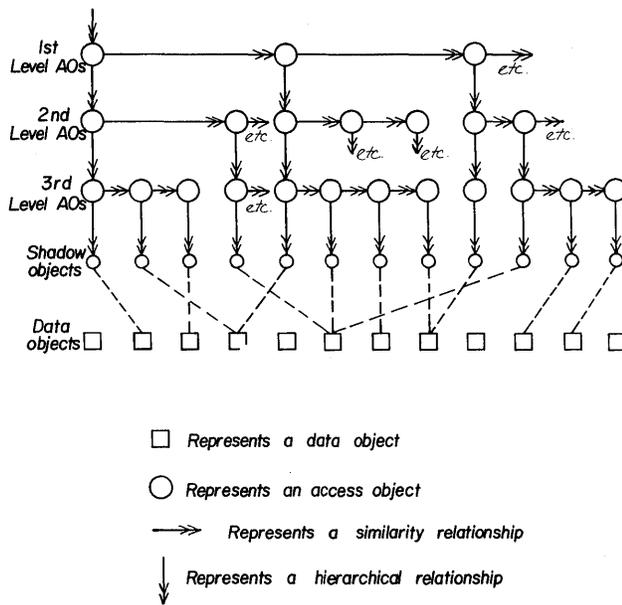


Figure 1—A graphic representation of the access path structural model

desired data such as a data record, and the *access object* (AO), which usually contains intermediate information leading to another object, such as the address of the next object (i.e., the address of either an AO or a DO).

The concept of “successor object” or “next object” is sufficient to describe the structural relationships that may exist among the various AOs and DOs and that are also necessary for the representation of the various access paths. The concept is used here to represent a two-dimensional relationship. The first dimension, the *similarity relationship*, pertains to the successor object on the same level, and the second dimension, the *hierarchical relationship*, pertains to the successor object on the next lower level of the hierarchy.

A binary-tree-like graph, as shown in Figure 1, is sufficient to represent a general access path model. The circles and squares represent the AOs and DOs respectively, and the double-headed arcs (\leftrightarrow) are used to represent the successor object relationships. The horizontal arcs represent the similarity relationships, while the vertical arcs represent the hierarchical relationships.

Since a DO can be reached from more than one AO, we introduce here an imaginary object, called a *shadow object** (SO), the purpose of which is purely limited to simplifying the graphical representation of the access path model. Those shadow objects are shown as small circles in Figure 1. For each DO there exist as many SOs as there are AOs associated with this given DO. This association, a function from the set of SOs onto the set of DOs, is represented in Figure 1 by the dashed lines between the SOs and the corresponding DOs. As an example, in multilist or inverted list file organizations, a data record with d key fields may be associated with d lists, and consequently can be reached via d different access paths. This record, a DO, is thus associated with d different SOs, one SO for each list.

Let a DO represent a data record, and let the first, second, and third level of AOs, as seen in Figure 1, represent the hierarchy of the keyword, CID, and RID levels respectively. The vertical arcs representing the transitions between the levels of the hierarchy are interpreted as follows: a vertical arc emanating from an AO on the keyword level serves to reach all the corresponding cells associated with the keyword. In turn, a vertical arc emanating from an AO on the CID level serves to reach all the corresponding records associated with the keyword and located in the cell. Finally, a vertical arc emanating from an AO on the RID level serves to reach a unique shadow object representing a data record associated with the keyword. Thus the vertical arcs serve to identify unique access paths from each keyword to all of its associated data records. We interpret the horizontal arcs as serving to represent a particular *sorting order* of the elements of each of the three levels, as dictated by the particular file organization. As an example (see Figure 1), the third AO on the first level reaches two AOs on the second level, the first and second of which reach one and three AOs respectively on the third level. In turn, each one of those AOs on the third level reaches a unique shadow object assigned to a data record (DO).

Model Interpretation

Let an AO be associated with an ordered pair (x, y) of object (AO or DO) addresses, whereby x and y represent the addresses of the two successor objects on the next and same levels respectively. A successor object can be located in any one of four possible locations with respect to the given AO:

1. In a cell not containing the given AO, but with a variable displacement from the cell.
2. In a cell not containing the given AO, but with a fixed displacement from the cell.
3. In a cell containing the given AO, but with a variable displacement from the AO.
4. In a cell containing the given AO, but with a fixed displacement from the AO.

The time consumed by the access method is modelled by the time spent traversing the arcs from the AOs to their successor objects. This time is then a function of the actual values of (x, y) associated with the given AOs. Hence, the arcs can be marked to indicate the amount of time consumed by the corresponding traversals.

Here an arc is assumed to be marked if it takes any one of the following four possible graphic representations: the solid shank with a heavy tip (\rightarrow), the dashed shank with a heavy tip (\dashrightarrow), the solid shank with a light tip (\rightharpoonup), and the dashed shank with a light tip (\dashrightarrow). These four arc representations correspond to the four previously listed cases describing the possible locations of the successor object with respect to its associated* AO. Note that the shank is associated with the displacement (fixed or variable) of the next object from the current object, and the tip is associated with the location

* They are also referred to as virtual records.

* A special value for y can be used to indicate that the corresponding AO has no successor on the horizontal level. In this case, no horizontal arc will be shown in the graph.

(same or different cell) of the next object with respect to the current object. Consequently, the shank models the storage requirement while the tip models the access time. Also note that it is not necessary to mark the AO nodes, since the associated (x, y) values are directly related to the corresponding arc representations (i.e., a given unique value for x or y dictates a unique arc representation, and vice versa).

Since $x(y)$ can be any one of four possible types, the set of vertical (horizontal) arcs for a given level of AOs can be one of fifteen ($2^4 - 1$) possible combinations (the power set with the exception of the null set ϕ). As an example, all arcs being \rightarrow is one combination, all arcs being \rightarrow is another combination, and some arcs being \rightarrow and the rest of the arcs being \rightarrow is yet another combination. Hence, an AO level can take one of $15 \times 15 = 225$ possible representations, and the three AO levels produce a total of $(225)^3$ possible representations. Consequently, it can be said that there exist a corresponding $(225)^3$ classes of file organizations.

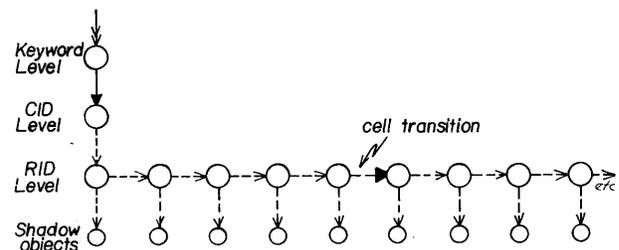
Fortunately, it can be easily shown that the majority of the $(225)^3$ possible representations are impractical, inefficient, or insignificantly different. Hence, they can be eliminated from further considerations. As an example, consider the horizontal arcs of the first-level AOs: the three combinations of all arcs being \rightarrow , all arcs being \rightarrow , and some arcs being \rightarrow and the rest of the arcs being \rightarrow are not practical. In addition, the three combinations of all arcs being \rightarrow , all arcs being \rightarrow , and some arcs being \rightarrow and the rest of the arcs being \rightarrow are less efficient than other possible combinations. Thus, only nine combinations should be considered out of the fifteen possible combinations for the horizontal arcs of the first-level AOs. As an additional example, consider the vertical arcs of the third-level AOs. The only feasible combinations are: all arcs being \rightarrow , all arcs being \rightarrow , and some arcs being \rightarrow , some \rightarrow . Thus, only three combinations should be considered out of the fifteen possible combinations for the vertical arcs of the third-level AOs. It is concluded that the number of potential file organizations is several orders of magnitude smaller than $(225)^3$. Moreover, these potential file organizations are realized by the basic file organizations (discussed in the next sub-section) and their many possible variations. The examples* in the next sub-section will serve to illustrate how the structural access path model can characterize the various potential file organizations.

File Organizations

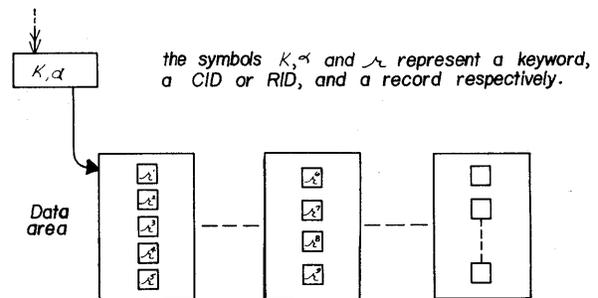
The basic file organizations can be classified into the following categories:⁹⁻¹¹

- A. Sequential Organization
- B. Primary Keyed Organization
 - B.1. Direct Organization (also called Hashed or Mapped Random Organization)
 - B.2. Indexed Organization
 - B.2.1. Indexed Sequential Organization
 - B.2.2. Indexed Random Organization

* For the examples, the DOs are not shown in the corresponding graphs.



a) Access path Model Representation



b) File Structure Representation

Figure 2—An example of sequential file organization

C. Secondary Keyed Organization

- C.1. Multilist or Multi-chained Organization
- C.2. Inverted List Organization
- C.3. Cellular List Organization

An elaborate file organization is usually made up of some combination of the previously listed basic organizations (e.g., rings, two-way chains, trees, plexes, and inverted bit-map organizations).

There are two general methods for gaining access, via keywords, to data records. The first is the index or table look-up, and the second is an algorithmic translation or hashing. The index, required for secondary keyed organizations, is by itself a file usually stored in the same medium as the data files. Hashing functions, applicable only to primary keyed organizations, do not normally require any extra storage. Being a simple and special kind of file, the index file is cast in a primary keyed organization allowing a relatively large scope for compaction and optimization techniques.

In the remainder of this subsection it will be shown how the basic file organizations can be characterized by the generalized structural access path model. In each of the Figures 2 to 8, an example of both an access path model and a file structure representation for the basic file organizations are shown. As we discuss the access path model in each of these figures, the reader should examine the corresponding file structure representation exhibited in the same figure.

Figure 2 represents a typical Sequential file organization. One entry point to the data records is needed, as represented by the single AO on the keyword level. The only AO on the CID level does not have a horizontal successor. However, its vertical successor occupies a fixed location in the same cell. The first five AOs on the RID level share the same cell, while

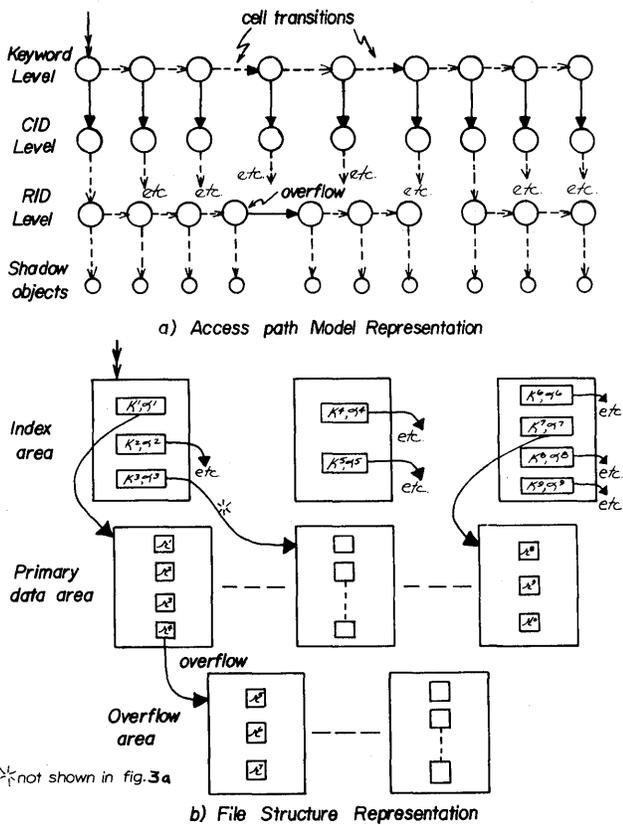


Figure 3—An example of Indexed Sequential file organization

the next four AOs share the next physically contiguous cell. Consequently, as also seen from the shadow objects, the first five records are contained in the same cell, while the next four records are contained in the next physically contiguous cell. Note that no storage space is required for the CID and RID level AOs, as evident from the nature of the arcs (i.e., \rightarrow and \dashrightarrow with dashed shanks).

Figure 3 represents a typical Indexed Sequential file organization. The keyword level represents the lowest level in a multilevel index structure. The figure shows that the first three index elements are in one cell, whereas the next two and the last four elements are located in the next two physically contiguous cells, respectively. For each AO on the keyword level there exists one and only one AO on the CID level. In turn, for each AO on the CID level there exists one or more AOs on the RID level. The figure indicates that the first four records are physically contiguous in the same cell, while the next three overflow records are physically contiguous in an arbitrary overflow cell. Note, for example, that none of the RID level AOs, except for those leading to overflow cells, require any storage space (this is due to the arcs with dashed shanks).

Figure 4 represents a typical Indexed Random file organization. The keyword level represents the lowest level in a multilevel index structure. The figure shows that the first three index elements are located in one cell, the next two elements are located in the next physically contiguous cell, and the last four elements are located in an arbitrary cell. The vertical arcs

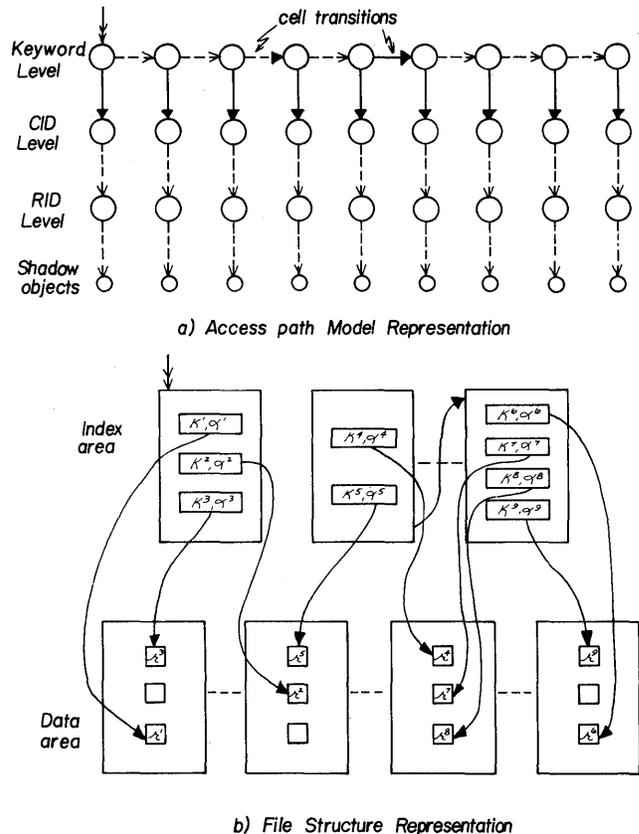


Figure 4—An example of Indexed Random file organization

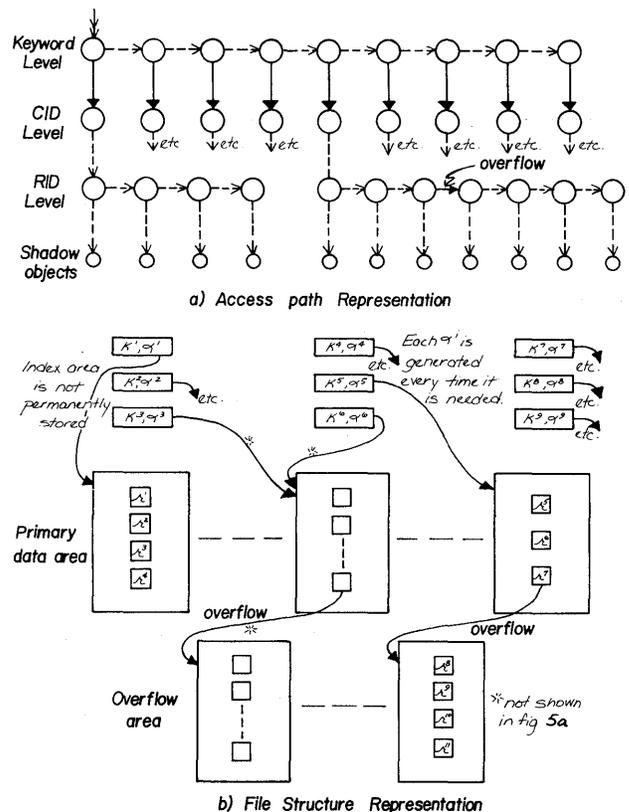
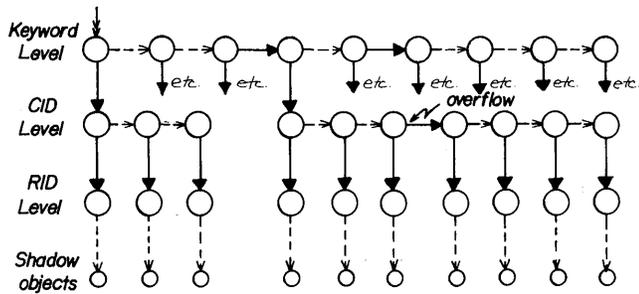
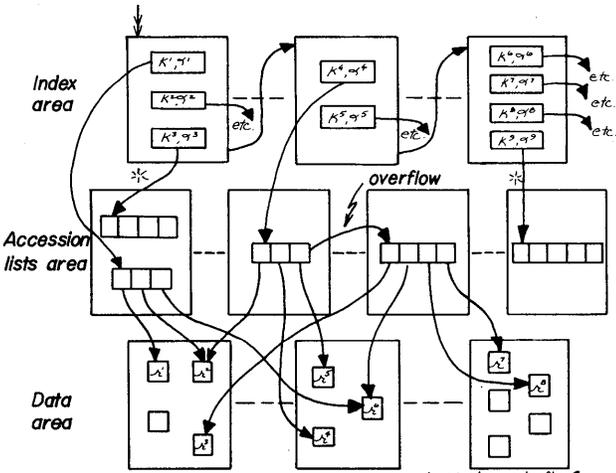


Figure 5—An example of Mapped Random file organization

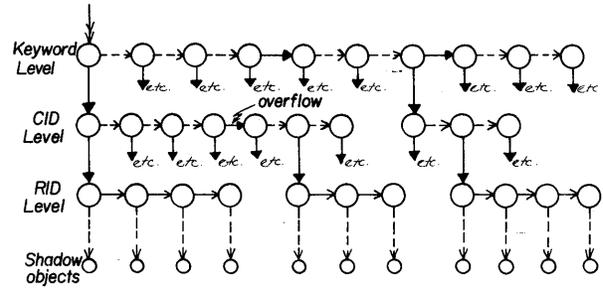


a) Access path Model Representation

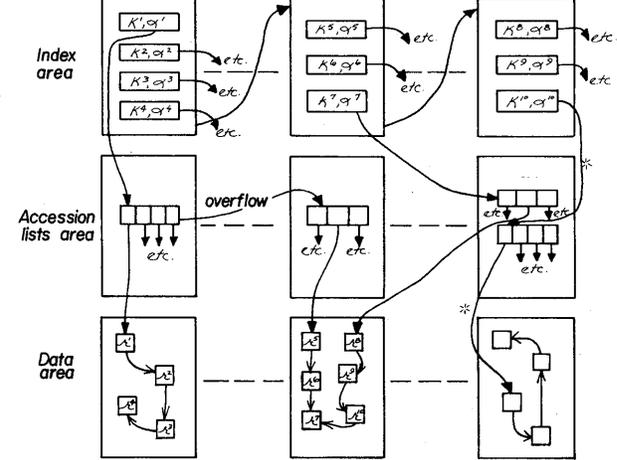


b) File Structure Representation

Figure 6—An example of inverted List file organization



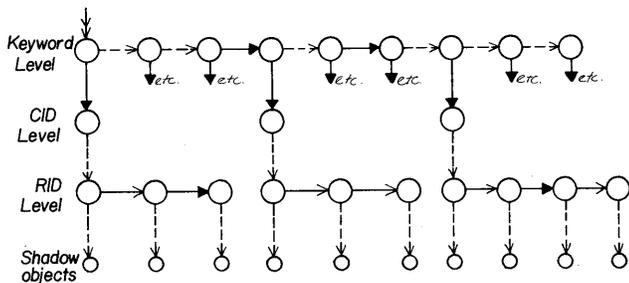
a) Access path Model Representation



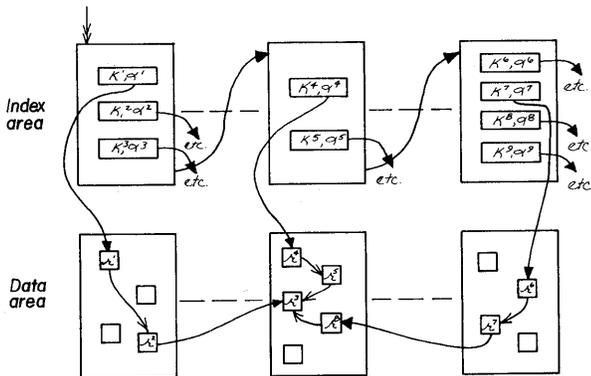
*not shown in Fig. 8a

b) File Structure Representation

Figure 8—An example of Cellular Multilist file organization



a) Access path Model Representation



b) File Structure Representation

Figure 7—An example of Multilist file organization

of the keyword level AOs represent the RIDs corresponding to the unique keywords in the primary key attribute. Note that there exists a one-to-one correspondence between the sets of AOs at the various levels and that none of the AOs at both the CID and RID levels require any storage space.

Figure 5 represents a typical Mapped Random file organization. The AOs at the keyword level represent the arguments used by the appropriate mapping function. A vertical arc at this level represents the RID of the first record in a group of commonly mapped records. The values of RIDs are generated by the appropriate mapping function only when needed and thus need not be permanently stored in the database.

Figures 6, 7, and 8 represent typical Inverted List, Multilist, and Cellular Multilist file organizations. The AOs at the keyword level represent the unique keywords from the various indexed attributes. The CID level AOs, for both Inverted List and Cellular Multilist organizations, represent the various sets of RIDs and CIDs, respectively. Each set is associated with an AO at the keyword level. The horizontal arcs of the RID level AOs, for both the Multilist and Cellular Multilist organizations, represent a chain of pointers linking the records associated with the same AO at the CID level. For each AO at the keyword level there exists one and only one AO at the CID level for Multilist organizations. However, in a cellular organization, for each AO at the CID level there exists one or more AOs at the RID level such that the corresponding records are located in the same cell. The rest of Figures 6 to 8 is self-explanatory.

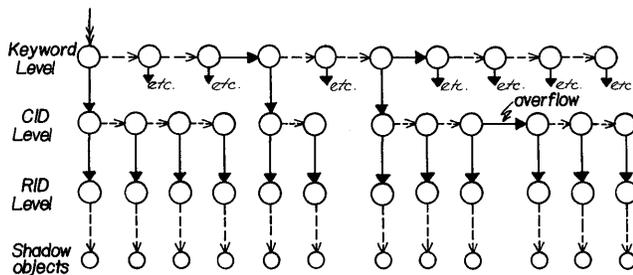


Figure 9—Access path model representation of type T_i file organization

INTEGRATED FILE ORGANIZATION

It is shown in this section that only three principal file organizations are necessary to replace the basic file organizations and their possible variations.

Figure 9 shows a one-to-many mapping from the set of keyword level AOs to the set of CID level AOs, and a one-to-one mapping from the set of CID level AOs to the set of RID level AOs. This kind of access path structure was also shown to characterize the Inverted List file organization (see Figure 6). Hereafter, this structure will be referred to as type T_i structure (remember *i*nverted).

Figure 10 shows a one-to-many mapping from the set of keyword level AOs to the set of CID level AOs, and a one-to-many mapping from the set of CID level AOs to the set of RID level AOs. This kind of access path structure was shown to characterize the Cellular Multilist file organization (see Figure 8). However, the RID level arcs in Figure 8 assume the swapped values of their counterparts in Figure 10. The structure of Figure 10 is adopted here, since, in contrast to the one shown in Figure 8, it does not require any modification to the data record structure. Hereafter, the structure shown in Figure 10 will be referred to as type T_c structure (remember *c*ellular).

Figure 11 shows a one-to-one mapping from the set of keyword level AOs to the set of CID level AOs, and a one-to-many mapping from the set of CID level AOs to the set of RID level AOs. The access path structures for the basic file organizations shown in Figures 2 to 5 and Figure 7 are special cases of the structure shown in Figure 11. The proofs are straightforward. As an example, if at the RID level, we let the horizontal arcs assume \rightarrow and \rightarrow and all the vertical arcs assume \dashrightarrow , then we ended up with the Multilist file organization (see Figure 7a). Hereafter, the structure shown in

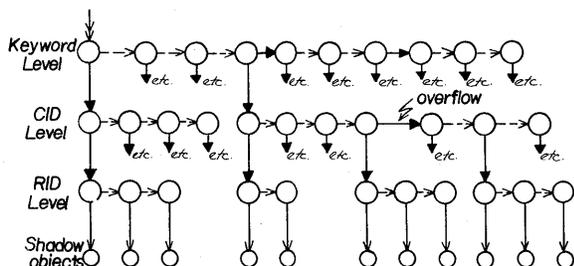


Figure 10—Access path model representation of type T_c file organization

Figure 11 will be referred to as type T_d structure (remember *d*istributed).

Finally, we conclude that the three principal access path structures (T_i , T_c , T_d) are sufficient to cover the seven basic file organizations and their possible variations. Hence, an access path structure in which a tree with a keyword level AO as a root takes on any of the three principal structures (T_i , T_c , or T_d) can form the architecture of a corresponding integrated file organization. This integrated organization can be used as the underlying file organization for a database, and consequently, it can be subjected to performance optimization.

CONCLUSION

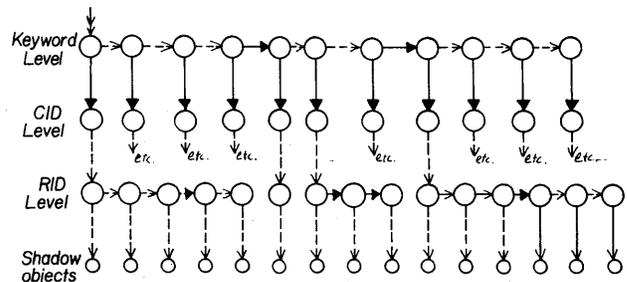
A general access path model has been developed in this paper. This model is capable of characterizing a large class of file organizations. Three principal access path structures, which can be used to model an integrated file organization, have also been identified. The integrated file organization, which includes all the basic file organizations as special cases, is capable of supporting the multidimensional key access feature. This organization is a general and flexible organization, and any of its components can change its structure dynamically. Hence, if it is used as the underlying structure for a database, the integrated file organization can be subjected to optimization techniques⁶ to yield an optimal database performance.

ACKNOWLEDGMENT

The author wishes to thank Dr. Jerome R. Cox of Washington University in St. Louis for his constructive comments during the development of this access path model.

REFERENCES

1. Hsiao, D. *Systems Programming, Concepts of Operating and Data Base Systems*. Addison-Wesley Inc., Reading, Massachusetts, (Chapter 6), 1975.
2. Yang, C.S. "A Class of Hybrid List File Organizations," *Information Systems*, 1975, Volume 3, pp. 49-58.
3. Severance, D.G. "A Parametric Model of Alternative File Structures," *Information Systems*, 1975, Volume 1, pp. 51-55.



Sequential, Indexed Sequential, Indexed Random, Mapped Random, and Multilist organizations are special cases of type T_d File organization

Figure 11—Access path model representation of type T_d file organization

4. Yao, S.B. "Evaluation and Optimization of File Organizations Through Analytic Modeling," *Ph.D. Dissertation Presented to the University of Michigan*, Ann Arbor, Michigan, 1974
5. Yao, S.B. "An Attribute Based Model for Data Base Access Cost Analysis," *ACM Transactions on Data Base Systems*, March 1977, Volume 2, Number 1.
6. Nicolas, G.S. "Access Path Optimization in Multidimensionally Accessed Database Files," *Sc.D. Dissertation Presented to the Sever Institute of Washington University*, St. Louis, Missouri, 1979.
7. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, June 1970, Volume 13, Number 6.
8. Codd, E. F. "Further Normalization of the Data Base Relational Model," *Data Base Systems, Courant Computer Science Symposia Series*, Volume 6. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1972.
9. Lefkovitz, D. *Data Management for On-Line Systems*. Hayden Book Company, Rochelle Park, New Jersey, 1974.
10. Martin, J. *Computer Data-Base Organization*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.
11. Weiderhold, G. *Database Design*. McGraw-Hill Book Company, New York, 1977.

Database programming with data abstractions

by BURT LEAVENWORTH

IBM Corporation, Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT

This paper describes how a general purpose programming language supporting the notion of data abstraction can be used as a data definition and manipulation language for database management systems. The examples used here are based on a functional data model and on a database system called NDB. However, the approach is not limited to or biased towards any particular data model or architecture. The strong typing properties of the data abstraction language are carried over to the realm of database manipulation operations and provide useful consistency checking. The advantage of the approach is that the user deals with but one database programming language, thereby avoiding a separate query and host language.

INTRODUCTION

There are three basic approaches¹¹ to providing database access from a general-purpose programming language:

1. Defining subroutines that execute database requests when called (for example, DL/I).⁵
2. Embedding database constructs in an existing language and using a preprocessor to translate these constructs into runtime calls on the database system (for example, SEQUEL2).²
3. Designing a new programming language in which database facilities are integrated into the language environment (for example, Date).³

This paper describes how a programming language supporting data abstractions (hereafter called XPLS) plus its supporting module interconnection language (hereafter called the external structure) can be used as a database definition and manipulation language. The approach is to use the notion of programming in the large⁴ for the purpose of data definition and that of programming in the small⁴ for the purpose of data manipulation.

XPLS is an experimental data abstraction language that was designed as an executable design language,⁷ based on CLU.⁸ It has a PL/I-like syntax and is supported by a preprocessor to

the PL/I compiler. It was not designed originally with database applications in mind.

The XPLS language has been combined with a database system called NDB,¹² which provides a set of PL/I data manipulation routines. A number of languages have been developed recently that exploit data abstractions and strong type checking but are based on a relational database (see, for example, Rowe and Shoens).⁹ However, the approach described here is not biased toward any particular database system or data model.

A DATA ABSTRACTION LANGUAGE

XPLS is strongly typed, with respect not only to its primitive types but also to new types defined by the programmer. This means that type consistency can be checked by the machine at translation (preprocessing) time.

The language has an external structure mechanism, which allows the designer to specify the interface properties of modules and view the design of an application at a high conceptual level. This component allows the separate compilation of individual modules whereby external references within the module can be checked for consistency.

Four types of modules are supported:

Procedure

XPLS procedures are like conventional procedures except that they accept abstract data objects as parameters and (optionally) return abstract data objects as values. This type of module is usually called an external procedure, in contrast to an encapsulated procedure (see the following section).

Capsule

Capsules allow the definition of abstract types in terms of the characteristic operations on these types. A capsule consists of the definition of the internal data representation for the abstract type and the definition of the operations (called encapsulated procedures) on this representation. Both of these components are hidden from the user.

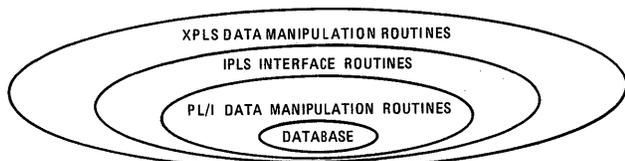


Figure 1—Database/data abstraction environment

Iterator

Iterators produce the elements of a collection, one element at a time, where the collection is an abstract data object the representation of whose data elements is hidden.

Interface Module

An interface module is a procedure written in a dialect or adjunct of XPLS called IPLS. The purpose of an IPLS procedure is to allow an “escape” into PL/I code while preserving the type integrity of the overall program. This approach guarantees that an IPLS procedure cannot violate the data space of XPLS programs.

For further details on the XPLS language, see Leavenworth.⁶

Figure 1 is a layers-of-abstraction view of how IPLS forms the interface between XPLS and a database.

FUNCTIONAL DATA MODEL

The data model we use is based on the functional data model and DAPLEX language described by Shipman.¹⁰ This model matches closely the architecture of NDB,¹² which is described later. The basic constructs of the functional model are the *entity* and the *function*. Entities represent real-world objects; functions, which map entities to other entities, are used to model an object’s properties. For example, a student may be considered an entity and a particular class in which the student is registered another entity. The function *courses* applied to a particular student would return a *set* of entities—that is, all the classes that the student is registered in. Finally, if a class has an *instructor of* property, then it is possible to derive an *instructor of* property of a student from the *instructor of* property of the classes in which he is enrolled. The function which models this type of property is called a *derived function*.¹⁰ The basic constructs of the functional model and the corresponding elements of XPLS are shown below.

<i>Functional Constructs</i>	<i>XPLS Elements</i>
Entity	Data abstraction
Function	Encapsulated procedure
Set	Parameterized set
Derived function	External procedure

Figure 2 shows a sample database taken from Shipman.¹⁰ The nodes represent entity types, and the arrows indicate functions mapping their argument types into their result types. The double-headed arrow indicates that the function returns a set of elements.

MODELING WITH DATA ABSTRACTIONS

The external structure allows the designer to specify for each data type the operations characterizing the type, and for each operation the operation name, types of its parameters, and return type (if any). For the purpose of describing data models, the operations are used as value-returning access functions, which are applied to entities (instances of a data type) and return scalar values or other entities. To model a database, our approach involves defining abstractions corresponding to the entity types of the application. The definition of these abstractions takes the place of the conceptual schema for a particular database. For example, the external structure definition for an employee type would have this form:

```

TYPE EMPLOYEE
DEFINES
(* := EMPLOYEE
NAME(*) → STRING
SALARY(*) → INT
MANAGER(*) → EMPLOYEE)
    
```

where the asterisk is used as shorthand for the employee type.

For a given data model, a parameterized set abstraction (generic type) would be defined by a systems programmer or database administrator. The particular set abstraction that we use for the functional model is given below (we show an abbreviated version). The operations shown are the retrieval and data manipulation functions for the functional model. The set abstraction is used by the applications programmer and can be instantiated with any of the entity types given by the example database.

```

TYPE SET << T:TYPE >>
DEFINES
(* := SET << T >>
SOME(*,PROC(T)→BOOL)→BOOL
IN(T,*)→BOOL
UNION(SET << * >>)→*
SEL(*,PROC(T)→BOOL)→*
THE(*,PROC(T)→BOOL)→T SIGNALS
(EMPTY)
    
```

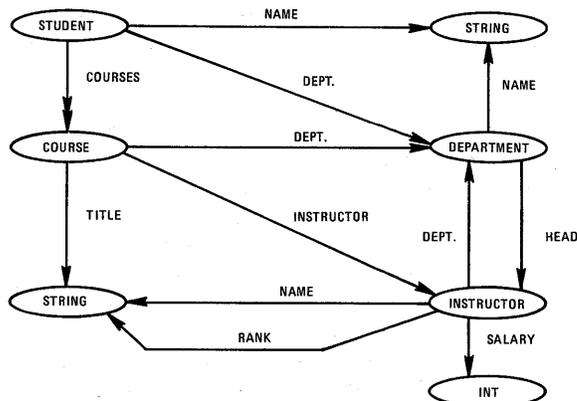


Figure 2—University database

```

SLICE << T1:TYPE >> (*,PROC(T)→T1)→
  SET << T1 >>
INSERT(*,T)
DELETE(*,T)
EACH ITER(*)⇒T
FILTER ITER(*,PROC(T)→BOOL)⇒T
    
```

SET is a generic type (has type-valued parameters) with the type parameter T. Examples of the use of this type are given later. The SLICE function is also generic. Its use will be shown in one of the examples to follow. The parameter PROC(T) → BOOL, which appears in several of the definitions, represents a predicate having the parameter T and returning a value having the type Boolean. The purpose of the THE function is to return the unique element of a set (first argument) satisfying a predicate (second argument). If this element is not found, the function signals an exceptional condition. The EACH iterator returns each element of a set, one at a time, and the FILTER iterator returns just those elements of the set satisfying a predicate (second argument).

Figure 3 gives the external structure definitions for the university database of Figure 2. These definitions represent a conceptual schema for this application (note that, for the sake of simplicity, string bounds have not been shown).

DATABASE INTERFACE

We will describe enough of the architecture of NDB¹² that the reader can understand how the interface routines are designed. This architecture presents a view of data in terms of entities and named binary relationships. In NDB, a single kind of data element, called a V-element, is used to model entities and values. The V-element, a composite structure, is shown in Figure 4. The three components of a V-element are

1. The identifier of an element representing the entity type of the V-element
2. An identifier that yields access to all connections of this element
3. A value that is a variable-length string

Given the identifier of a V-element, an application program may extract the value by using a function. V-elements may be used to represent real-world objects by using a unique character string as the value. All identifiers have the attributes FIXED BINARY(31), and the data values stored in V-elements have the attributes CHARACTER(1000) VARYING.

As stated previously, IPLS procedures are used to “escape” into PL/I code while preserving the type integrity between programs. IPLS procedures are called by XPLS procedures (only abstract scalar arguments may be passed), but not vice versa. Two transfer functions having single parameters are provided: AC (abstract to concrete) and CA (concrete to abstract). The idea is to transform abstract scalar arguments to PL/I scalar types, which can then be manipulated by standard PL/I code. Abstract scalar values are finally passed back to the calling procedure by using the appropriate transfer function. IPLS procedures are type-checked in the same manner as for XPLS procedures.

```

TYPE STUDENT
DEFINES
(* := STUDENT
  NAME(*)→STRING
  DEPT(*)→DEPARTMENT
  CHANGE_DEPT(*,DEPARTMENT)
  COURSES(*)→SET<<COURSE>>
  CREATE(String,DEPARTMENT)→*
  ENROLL(*,COURSE)
  DROP(*,COURSE) )

TYPE COURSE
DEFINES
(* := COURSE
  TITLE(*)→STRING
  DEPT(*)→DEPARTMENT
  INSTRUCTOR(*)→INSTRUCTOR
  CHANGE_INSTR(*,INSTRUCTOR) )

TYPE INSTRUCTOR
DEFINES
(* := INSTRUCTOR
  NAME(*)→STRING
  RANK(*)→STRING
  DEPT(*)→DEPARTMENT
  SALARY(*)→INT
  CHANGE_RANK(*,STRING)
  CHANGE_SAL(*,INT) )

TYPE DEPARTMENT
DEFINES
(* := DEPARTMENT
  NAME(*)→STRING
  HEAD(*)→INSTRUCTOR
  CHANGE_HEAD(*,INSTRUCTOR) )
    
```

Figure 3—Conceptual schema for university database

An example of an IPLS (function) procedure named ZVAL, which returns the value of an element, given the element-id as argument, is shown below.

```

ZVAL: PROC(X:INT) RETURNS(STRING<<1000>>);
DCL Y CHAR(1000) VAR;
Y=DBV(AC(X));
RETURN(CA(Y));
END ZVAL;
    
```

This function transforms the abstract INT (integer) scalar parameter denoted by X to a fixed binary concrete scalar

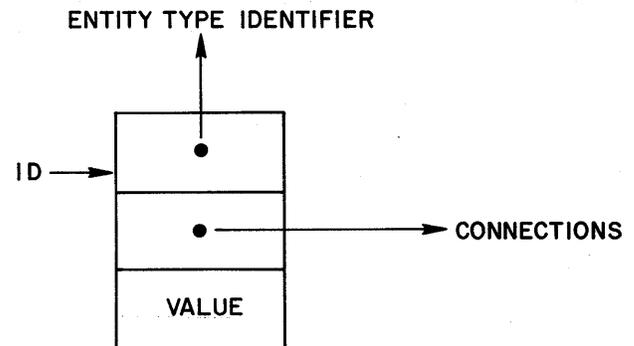


Figure 4—V-element

using the AC function. The NDB function DBV is applied to this scalar and returns a CHAR(1000) VAR value which is stored in Y. Finally, this value is converted to an XPLS abstract scalar string and returned.

We now show the definition of the EACH encapsulated iterator with its enclosing SET capsule. This gives an example of the use of IPLS routines to sequence through the database.

```
SET: CAPSULE << T:TYPE >> EXPORTS
  (... ,EACH,...) WHERE T HAS (COERCE)
  PROC(INT) RETURNS(T);
  EQUATE REP INT;
  ...
EACH: ITER(X:*) YIELDS(T);
  DCL Y INT;
  Y = ZFIRST(X);
  DO WHILE(¬ZEMPTY(Y));
    YIELD(T&COERCE(ZGET(Y)));
    Y = ZNEXT(Y);
  END;
END EACH;
  ...
END SET;
```

The EACH iterator uses the following IPLS functions (shown below in external structure format):

```
ZFIRST(INT)→INT /*returns first element of set */
ZNEXT(INT)→INT /*returns next element of set */
ZGET(INT)→INT /*returns value of element */
ZEMPTY(INT)→BOOL /*tests for empty set */
```

To write these routines, and others, one must understand the database architecture. However, although the applications programmer must understand the data *model* defined in the external structure, he or she does not need to know the database architecture.

The EQUATE statement indicates that the internal representation for SET is an INT. This integer represents the XPLS abstract equivalent of the V-element identifier for the NDB set. After Y is initialized to the first element of the set, the DO WHILE loop obtains the element identifiers for each element of the set, using the ZGET function. Although these are the correct objects that will be yielded by the EACH iterator, they are of type INT; and it is necessary to transform them (using COERCE) to type T to satisfy the type checker. Note that the WHERE clause requires that the parameterized type T have an encapsulated (function) procedure COERCE, which performs the necessary conversion.

DATABASE QUERIES IN XPLS

Consider the following declarations:

```
DCL STUDENTS SET << STUDENT >>;
DCL ST STUDENT;
DCL EE STUDENTS SET << STUDENT >>;
DCL COURSES SET << COURSE >>;
DCL INSTRUCTORS SET << INSTRUCTOR >>;
DCL INSTR INSTRUCTOR;
```

```
GENERIC NAME
  (WHEN (STUDENT) STUDENT&NAME,
   WHEN (INSTRUCTOR) INSTRUCTOR&NAME,
   WHEN (DEPARTMENT) DEPARTMENT&NAME)

GENERIC DEPT
  (WHEN (STUDENT) STUDENT&DEPT,
   WHEN (DEPT) COURSE&DEPT,
   WHEN (INSTRUCTOR) INSTRUCTOR&DEPT)

GENERIC COURSES
  (WHEN (STUDENT) STUDENT&COURSES)

GENERIC TITLE
  (WHEN (COURSE) COURSE&TITLE)

GENERIC INSTRUCTOR
  (WHEN (COURSE) COURSE&INSTRUCTOR)

GENERIC RANK
  (WHEN (INSTRUCTOR) INSTRUCTOR&RANK)

GENERIC ENROLL
  (WHEN (STUDENT, COURSE) STUDENT&ENROLL)

GENERIC DROP
  (WHEN (STUDENT,COURSE) STUDENT&DROP)
```

Figure 5—Macro definitions for conceptual schema

The types SET << STUDENT >> and SET << COURSE >> are *instantiations* of the parameterized type SET. Using the university database in Figure 2, we can write an XPLS program to answer the following query:

```
Q1. Find all courses taken by EE students.
EE_STUDENTS = SET << STUDENT >> &SEL
  (STUDENTS, (S:STUDENT→BOOL;
  DEPARTMENT&NAME(STUDENT&DEPT(S))
  = 'EE'));
COURSES = SET << COURSE >> &UNION
  (SET << STUDENT >> &SLICE
  << SET << COURSE >> >>
  (EE_STUDENTS,STUDENT&COURSES));
```

The SEL encapsulated function has the compound name SET << STUDENT >> &SEL, where the prefix SET << STUDENT >> is the (instantiated) type and the suffix SEL is the operation name. The second argument of SEL is a predicate having a single parameter S of type STUDENT, which returns a Boolean value. The expression following the semicolon is called the body of the function. The effect of the first statement of the program is that EE_STUDENTS refers to all the STUDENTS satisfying the predicate. The SLICE function is generic and has the effect of applying the (selector) function STUDENT&COURSES to each member of EE_STUDENTS. This will produce the type SET << SET << COURSE >> >>, and UNION applied to the aggregate will produce the type SET << COURSE >>.

As a result of the compound names and instantiations, the program becomes very cumbersome to write and read. This situation can be remedied by the use of generic declarations, as shown in Figure 5 and Figure 6. This facility has been inspired by the generic functions in PL/I and is roughly simi-

```

GENERIC COUNT <<T:TYPE>>
  (WHEN (SET <<T>>) SET <<T>> €COUNT)

GENERIC IN <<T:TYPE>>
  (WHEN (T,SET <<T>>) SET <<T>> €IN)

GENERIC UNION <<T:TYPE>>
  (WHEN (SET <<SET <<T>>>>) SET <<T>> €UNION)

GENERIC SLICE <<T:TYPE,T1:TYPE>>
  (WHEN (SET <<T>>,PROC(T)→T1)
  SET <<T>> €SLICE <<T1>>)

GENERIC SOME <<T:TYPE>>
  (WHEN (SET <<T>>,PROC(T)→BOOL)
  SET <<T>> €SOME)

GENERIC SEL <<T:TYPE>>
  (WHEN (SET <<T>>,PROC(T)→BOOL)
  SET <<T>> €SEL)

GENERIC THE <<T:TYPE>>
  (WHEN (SET <<T>>,PROC(T)→BOOL) SET <<T>> €THE)

GENERIC EACH <<T:TYPE>>
  (WHEN (SET <<T>>) SET <<T>> €EACH)

GENERIC FILTER <<T:TYPE>>
  (WHEN (SET <<T>>,PROC(T)→BOOL)
  SET <<T>> €FILTER)

```

Figure 6—Macro definitions for set abstraction

lar. When these declarations are in effect, the query can then be written:

```

EE_STUDENTS = SEL(STUDENTS,
  (S:STUDENT→BOOL; NAME(DEPT(S)) = 'EE'));
COURSES = UNION(SLICE(EE_STUDENTS,
  STUDENT€COURSES));

```

At translation time the function invocation NAME(DEPT(S)) will effectively be expanded to DEPARTMENT€NAME(STUDENT€DEPT(S)). A more complicated example is the function SEL. Its type parameter SET <<T>> is matched with the type (SET <<STUDENT>>) of the corresponding argument, thereby binding the type parameter T to STUDENT. The expansion is then SET <<STUDENT>> €SEL.

The use of this facility allows the programmer to write XPLS queries very similar to those in conventional query languages.

The following queries are all written assuming the use of these generic declarations.

Q2. What are the names of all EE students taking courses from assistant instructors?

```

FOR ST IN FILTER(STUDENTS,
  (S:STUDENT→BOOL;
  SOME (COURSES(S),(C:COURSE→BOOL;
  NAME(DEPT(C)) = 'EE' & RANK
  (INSTRUCTOR(C)) = 'ASST PROF'))));
CALL PRINT(NAME(ST));
END;

```

The FOR statement works essentially as follows. The iterator (FILTER, in this case) will select just those students who satisfy the predicate and will bind them one at a time to the variable ST. The predicate has a single parameter S of type STUDENT and returns a Boolean value. Each time the iterator yields a STUDENT, the body of the FOR statement, which is the print statement, is executed. This process is repeated until the iterator yields no further students.

Q3. Among the students who are also instructors, list those who are taking a course that they teach.

We first define an external function, COURSES_TAUGHT, which is applied to an individual instructor and the set of courses to yield the courses which the instructor teaches.

```

COURSES_TAUGHT: PROC
  (I:INSTRUCTOR,CS:SET <<COURSE>>)
  RETURNS(SET <<COURSE>>);
  RETURN(SEL(CS,(C:COURSE→BOOL;
  INSTRUCTOR(C) = I)));
END COURSES_TAUGHT;

```

Then the answer to the query is:

```

FOR ST IN EACH(STUDENTS);
  INSTR = THE(INSTRUCTORS,
  (I:INSTRUCTOR→BOOL;
  NAME(I) = NAME(ST)));
  EXCEPT WHEN(EMPTY) CONTINUE;
END;
  IF SOME(COURSES(ST),(C:COURSE→BOOL;
  IN(C,COURSES_TAUGHT
  (INSTR,COURSES)))) THEN CALL PRINT
  (NAME(ST));
END;

```

The preceding program illustrates the use of an exception handler. If the THE function has a normal return, then the IF statement is executed; otherwise, an exception is signalled, and the loop is continued (the IF statement is skipped).

Q4. Drop introductory physics from John's courses and add organic chemistry.

```

ST = THE(STUDENTS,(S:STUDENT→BOOL;
  NAME(S) = 'JOHN'));
CALL DROP(ST,THE(COURSES,
  (C:COURSE→BOOL; TITLE(C) =
  'INTRODUCTORY PHYSICS')));
CALL ENROLL(ST,THE(COURSES,
  (C:COURSE→BOOL; TITLE(C) =
  'ORGANIC CHEMISTRY')));

```

CONCLUSIONS

We have described how a general-purpose language supporting data abstractions can be integrated into a database man-

agement system. The systems programmer or database administrator defines abstractions (using the external structure as a data definition language) based on a given data model and database and writes interface routines and capsules corresponding to the application entities. The applications programmer then uses XPLS as a data manipulation language, using the data model to navigate through the data space. The functional model was used as the data model in this paper.

The advantages of using this approach are as follows:

- The strong typing properties of XPLS are carried over to the realm of database manipulation operations and provide useful consistency checking.
- A general-purpose data abstraction language becomes by extension an integrated database programming language. In this way, a separate host language and query language are avoided.

The approach is not biased toward any particular data model or architecture.

ACKNOWLEDGMENTS

I am indebted to Jerry Archibald, Les Belady, Hamed Ellozy, Leigh Power, and Bob Taylor for their many valuable comments and suggestions regarding the work described here.

REFERENCES

1. Archibald, Jerry L. *The Yorktown External Structure User's Guide* Technical Memo No. 14, Software Technology Project, IBM Research Center, Yorktown Heights, New York (Dec 1979).
2. Chamberlin, D.D., et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control." *IBM Journal of Research and Development*, 20, (1976), pp. 560-575.
3. Date, C.J. "An Architecture for High-Level Language Database Extensions." *Proceedings 1976 ACM SIGMOD International Conference on the Management of Data*, pp. 101-122.
4. DeRemer, F. and H. Kron. "Programming-in-the-Large versus Programming-in-the-Small." *SIGPLAN Notices* (June 1975), pp. 114-121.
5. IBM Corp. *IMS/VS Application Programming Reference Manual*. Form SH20-9026.
6. Leavenworth, B. *XPLS Reference Manual*. Technical Memo No. 19, Software Technology Project, IBM Research Center, Yorktown Heights, New York (Aug. 1979).
7. Leavenworth, B. "The Use of Data Abstraction in Program Design." *Proceedings Software Development Tools Workshop Conference*, Pingree Park, Colorado (1979), to be published by Springer-Verlag.
8. Liskov, B.H., et al. "Abstraction Mechanisms in CLU." *Comm. ACM*, 20, 8 (Aug. 1977), pp. 564-576.
9. Rowe, Lawrence A., and Kurt A. Shoens. "Data Abstraction, Views and Updates in RIGEL." *Proceedings ACM 1979 National Conference*, pp. 71-81.
10. Shipman, D. "The Functional Data Model and the Data Language DAPLEX." *ACM TODS*, 1980 (to appear).
11. Stonebraker, Michael, and Lawrence A. Rowe. "Observations on Data Manipulation Languages and Their Embedding in General Purpose Programming Languages." *Proceedings Third International Conference on Very Large Data Bases* Tokyo, Japan (Oct. 1977), pp. 128-143.
12. Winterbottom, N., and G.C.H. Sharman. *NDB: Non-programmer Data Base Facility*. Technical Report TR.12.179 (Sept. 1979), IBM United Kingdom Laboratories Ltd.

Feature analysis of selected database recovery techniques*

by BHARAT BHARGAVA and LESZEK LILIEN

University of Pittsburgh
Pittsburgh, Pennsylvania

ABSTRACT

Database recovery techniques in a real-time environment for so called single-division databases are investigated. A classification of database recovery goals and a classification of database system crashes is presented. It is shown that the (best) recovery goal is a function of a crash category against which the system is to be protected. In particular, for the broadest category of hidden hard crashes an actual past state is an attainable recovery goal. It is described how to reach this goal using a generic recovery technique, based on an idea of a database recovery block. The specific recovery techniques implementing the generic technique are described. Then the representation of each specific recovery technique in terms of atomic "primitives" is demonstrated. The claim is made that this "divide-and-conquer" approach can facilitate the analysis of the database recovery techniques.

INTRODUCTION

Database technology is one of the most rapidly growing areas of computer science.¹⁰ The technology makes it possible to reduce data redundancy, as compared to independent file systems, simultaneously improving data availability. But it also introduces the potential for disaster; the database is now more vulnerable to destruction through hardware and software malfunction. The loss of "quality" in a database, especially its total destruction, may be considered a threat to the organization owning the database, because data is one of its most vulnerable assets. The problems can be further aggravated if a database system is to function in a real-time environment. This case is investigated in this paper.

To avoid confusion let us indicate the meaning of the words fault, error and crash (failure) as used here.⁴ A fault is a malfunction in a hardware, software, or human component of the system that may introduce or allow to be introduced errors. These are items of data or pieces of program incorrectly stored or transmitted within the system or lost altogether. In due course, an error may cause a crash, which is cessation of normal, timely operation by all or part of the system, or

delivery to the outside world of incorrect data. We interpret a detection of an error at time t as a crash at time t ; the moment an error is detected, the system must take some special actions and its normal operation is disrupted. But it is also possible that some crashes will become manifest directly and not through detection of errors that cause them. For example, in the case of a major hardware breakdown the fault and the crash are simultaneous.

Clearly it is impossible to avoid hardware or software crashes in any computing system. Thus the only way to protect a database is through the use of recovery techniques that allow one to restore the correct database state in the case of partial or total database destruction.

The steps in the ideal recovery process could be as follows:⁴

- the fact that the system has crashed is recognized, either through error detection or directly,
- the type of crash is determined,
- the faults in the system which caused the crash are identified,
- the extent of the damage is determined, in the database, programs, system files and elsewhere,
- a method of recovery is selected,
- faulty programs and hardware units are repaired,
- the database is repaired or reloaded, as appropriate,
- restart programs are run which reset the state of the system, undo and reprocess any incorrectly applied transactions, and re-open contact with the users, and
- normal processing is resumed.

Many of the above operations are so complex that we do not know how to implement them. After a crash, hard detective work must be done to diagnose the original fault. If it is a hardware problem, some help might be had from diagnostic software or test equipment. Locating software faults is usually more difficult. Many of the faults which occur in real-time systems are transient, depending on particular combination of events and input data. They may be extremely difficult to reproduce, and if they are ever traced at all, it is usually as a result of ingenuity or guesswork by the maintenance programmer.⁴ It is not a surprise that such informal diagnostic methods, let alone correction methods, are far from on-line implementation. Yet there exist some approximate solutions to the problems of on-line diagnosis and correction of faults.

*This work was partially supported by the grant GTRS 5680-C-00026 of the U.S. Department of Transportation.

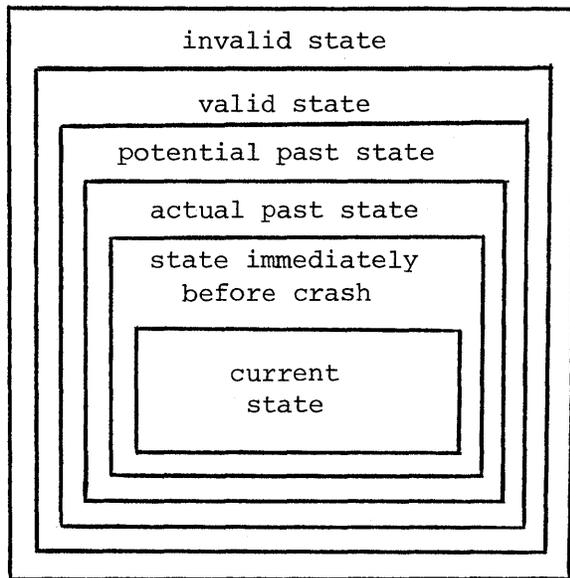


Figure 1—The hierarchy of database recovery goals

In the following considerations the notion of a division is used. We define a division as a logical subset of a database such that integrity assertions of different division are mutually disjoint, and the sum of all divisions constitutes the database. In general, database systems can have more than one division. If an error is detected in one division, it is possible to carry out the recovery process for that division alone. While a given division is under recovery, incoming transactions for other divisions may continue to be processed. This approach increases system availability but makes the recovery procedures more complicated.

We are going to refer to a method for increasing reliability of software system components, which is proposed by Randall.⁸ Its basic idea is that all procedures are encapsulated in so-called recovery blocks. Each recovery block comprises a predicate called "acceptance test" (AT) and a collection of alternative procedures for accomplishing the same task. On entry to a recovery block, the primary alternative is tried. If it succeeds, i.e. passes the AT, a normal block exit follows. If it fails, all variables are restored to their values on entry to the recovery block, then the second alternative is tried, etc. (In general, an acceptance test has a limited ability to detect errors, so it is possible that erroneous results pass the test.)

After classifying database recovery goals and classifying database crash categories in the next two sections, we show that the best attainable database recovery goal is a function of a crash category. Finally, we present some selected database recovery techniques and analyze their features.

DATABASE RECOVERY GOALS

Each database recovery technique can be viewed as containing three phases:

1. backing up to a past state;¹¹ this phase names goals of database recovery, as presented below,

2. restoration of the immediate before-crash state (even if this state is not known explicitly),
3. reexecution of after-crash database operations.

The hierarchy of the database recovery goals, somewhat different from the one presented by J.S.M. Verhofstad,¹¹ includes (see Figure 1):

1. The current (after-crash) correct database state (this can be a recovery goal only if the database is completely protected from crash effects),
2. the correct database state as it was immediately before crash (what "immediately" means is defined by a single update operation),
3. the actual past database state, i.e. a snapshot of the correct database as it was some time ago ("some time ago" will be defined more clearly by the notion of Database Recovery Block),
4. the potential past database state, i.e., a correct state that is a combination of actual past states of database divisions (these states of database divisions could never exist at the same time, but each of them did exist at some time in the past),
5. the valid database state, in which only a proper subset of database divisions is in a correct state, and
6. the invalid database state, in which all database divisions are incorrect (for a single-division database this goal is equivalent to 5).

The goals higher in this hierarchy, i.e., those with smaller indices, are more difficult to attain than those below. This interpretation underlies Figure 1.

At first glance it seems that goals 1 through 4 are defined in the dimension of time while goals 5 and 6—in the dimension of the database correctness. Our claim that all goals 1 through 6 are really related in one dimension—that of the database correctness—is based on the following approach: more recent correct database state is "more correct" than any previous correct database state.

DATABASE SYSTEM CRASH CATEGORIES

For our purposes we distinguish the following crash categories:

1. soft crashes, i.e. crashes which do not damage the database contents,^{3,5}
2. hard crashes, i.e. crashes damaging database contents,^{3,5} which may be divided into
 - a. overt hard crashes, i.e. crashes that are caused by instantaneously detectable errors or faults, and instantaneously detected after they occur, and
 - b. hidden hard crashes, i.e. crashes that are caused by errors detected only some time after these errors occurred. For example, if an erroneous data item is written into a database, this crash can remain hidden for a long time before it is recognized through the detection of the original error or its consequences.

In real-life situations, very few crashes are hard.⁵ However hard crash recovery is very time consuming once it happens. Clearly, hidden crashes are the most dangerous; as long as underlying errors are not detected, their effects continue to contaminate a database.

DATABASE RECOVERY GOAL AS A FUNCTION OF CRASH CATEGORY

Protecting against all possible types of crashes is in most cases impractical.³ This implies the importance of a function

$$\text{recovery goal} = f(\text{crash category})$$

We understand this shorthand notation in the following way. Given a crash category against which we want to protect the database, we aim to achieve the best attainable goal for this crash category. For example, for soft crashes goal 1, which is the best, can obviously be reached. For hidden hard crashes only goal 3 can be reached; more precisely, we will show in this report how to attain the goal 3 and we do not know how to attain better goals, namely 1 or 2, for this crash category. In this sense a crash category implies a recovery goal.

Recovery goal 1, the current (after crash) database state, is attainable for soft crashes. The generic "technique" is just null.

Recovery goal 2, the immediate before-crash database state, is attainable only for overt hard crashes. For crashes of this category we record the database state (e.g., just the old item value) before each update. If the crash happens during the update, we simply restore the old item value. This achieves the required database recovery goal, as a crash is discovered instantaneously. Specialized techniques for recovery goal 2 are not investigated here.

Recovery goal 3, the actual past database state, is attainable for the much broader category of hidden hard crashes. The generic recovery technique and its most prominent implementation approaches are discussed in the next section.

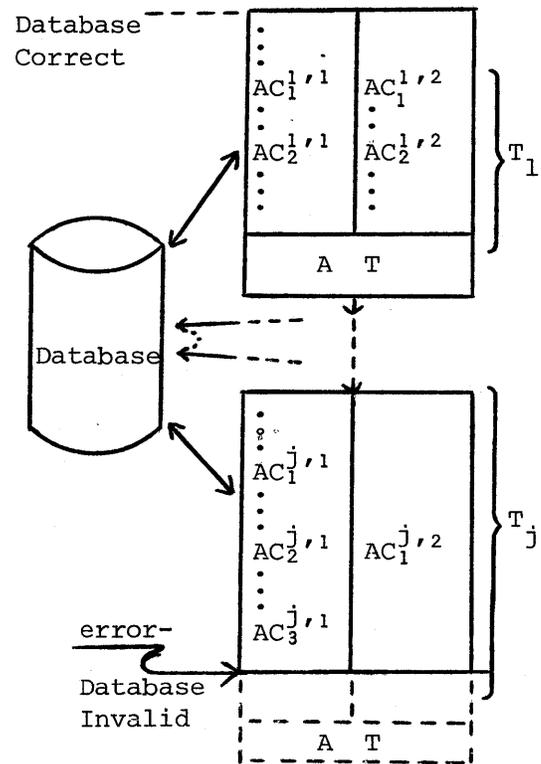
Recovery goal 4, the potential past database state, is not considered here. It seems to be of value in a multi-division database, whereas we assume below only a single-division database.

Recovery goal 5, valid database state, and especially goal 6, invalid database state, are of no practical value—they leave the database seriously damaged and completely destroyed, respectively. Because the proposed mechanism allows us to maintain at least potential past database state, the valid and invalid database states should be seen merely as a closure for the theoretical classification of the database recovery goal hierarchy.

TECHNIQUES FOR RECOVERY FROM HIDDEN HARD CRASHES

Generic Technique for Recovery from Hidden Hard Crashes

For the hidden hard crashes, the database is being contaminated, from the moment an underlying error occurs to the



- T_i - i-th transaction,
- AT - acceptance test of a transaction,
- $AC_k^{i,j}$ - k-th acceptance check within the j-th alternative of T_i

Figure 2—Database access by a sequence of transactions—"unsafe" system approach

moment it is detected, through an uncontrolled propagation of incorrect database entries.

One means of error detection is the use of acceptance checks (AC). Each transaction can include a number of acceptance checks. Acceptance checks are predicates on values of database items and values of variables of the transaction. It is important to discriminate between acceptance checks (AC) and the acceptance tests (AT) of a recovery block; the former can be placed anywhere in the body of the transaction, the latter are placed only at the exit from the recovery block implementing this transaction. Acceptance checks are specialized error-detection mechanisms (looking for only some types of errors) that can be used to decrease the time interval between the error occurrence and the error detection. Acceptance tests should be able to detect all kinds of errors, but they allow errors to remain undetected until the very end of the currently executed transaction alternative. In general, acceptance checks guarding against more specific types of errors have more diagnostic power. This fact is really not exploited in our preliminary mode.

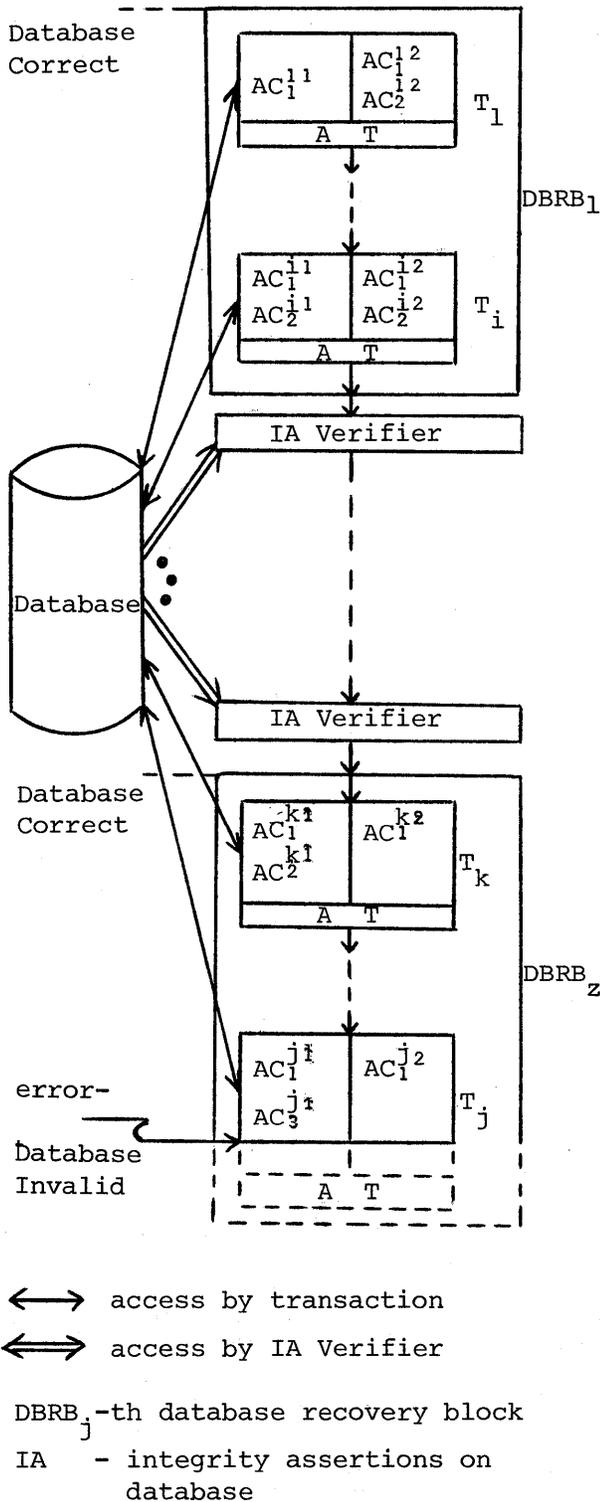


Figure 3—Database access by a sequence of transactions—"safe" system approach

Let us present two scenarios of database system operation. We start with a pessimistic scenario for the "unsafe" system operation (see Figure 2). By definition, the database is initially in a correct state. The processing starts with the transaction

(process) T_1 . Let the first alternate of T_1 include two acceptance checks ($AC_1^{1,1}$ and $AC_2^{1,1}$) on the database, which give positive validation of its contents. At some moment the transaction T_j starts. The first two acceptance checks positively validate the database. But the third one ($AC_3^{j,1}$) finds out that the database is severely damaged.

The diagnosis is generally impossible. Each acceptance check verifies database integrity only partially. So the fact that $AC_2^{j,1}$ answered positively gives us no help—maybe $AC_2^{j,1}$ did not at all check the integrity constraints that $AC_3^{j,1}$ did. We know only that a hidden crash has happened, caused by an error that occurred after the moment T_1 has started and before the moment $AC_3^{j,1}$ of T_j has detected the error. It is important to point out that even if there are backup copies of the database in the system, in the case of a hidden hard-crash we have no guarantee that any of these copies is correct. Therefore we cannot use them for recovery.

The only recovery possible in this case is the total system abortion, followed by the restart from the point where T_1 has started initially. The chances that the crash will not happen again are based on the chance that the underlying error is transient or that the transactions are implemented as recovery blocks.

Let us now present a pessimistic scenario of the "safe" system operation. We propose a mechanism confining the database contamination so that a faster recovery is possible than for the "unsafe" system operation (see Figure 3). The assumption of crash transience or implementation of transactions according to a recovery block scheme is still essential here. (Note that not only acceptance checks but also acceptance tests are generally not completely effective; the transaction results could pass their acceptance test and still be a source of errors.)

The system components T and AC are as described above, but new system components have been added. A Database Recovery Block (DBRB), defined dynamically by time interval, encompasses a number of transactions. A DBRB is created in such a way that we are assured of a correct database state at the entry to this block. Before entering a DBRB other than the first one, where integrity assertions are true anyway, the integrity of the database is verified by means of Integrity Assertion (IA) Verifier. (The efficient organization of the IA verification is a problem in itself. What one needs is minimization of the number of database accesses for purposes of the verification. We plan to investigate this subject later.)

After the positive IA verification a database logical snapshot is made and the new, z th DBRB is initiated. (In the case of a negative IA verification, the previous snapshot is restored and processing restarts from that point.) Suppose that $AC_3^{j,1}$ in T_j , which in turn belongs to DBRB_z, discovers that the database is invalid. Diagnosis is immediate; a hidden crash has happened, caused by errors that occurred during the execution of the current DBRB, i.e. DBRB_z. The following steps are taken to resume processing:

1. Transactions are not allowed to query the database. All incoming transactions are queued.
2. The transactions, which are implemented as recovery blocks, are reconfigured; some permutation of their alternatives is scheduled for execution. This permutation

is different from that of the ones that are marked so far as trouble-makers and are placed on a suspicion list. (The suspicion list can be used for an off-line diagnosis and repair of transactions. The repaired transactions are removed from the list.)

3. The most recent database snapshot is restored.
4. All transactions that
 - a. were active at the moment of crash, or
 - b. were completed during the current DBRB before the moment of crash,
 are processed again. All these transactions are notified of the recovery if necessary.
5. Incoming transactions which were stored during recovery are processed.

In the case that errors that happened during a DBRB are not detected by Integrity Assertion Verifier at the end of this DBRB, a mechanism to restore earlier snapshots must be given. The mechanism is not a simple one by any means. If an error is detected for the i th time on end in the same DBRB during an attempted recovery, we can

1. try to run the DBRB one more time, assuming error transience or using one more permutation of transaction alternatives, or
2. back up to the previous snapshot and thus to the earlier DBRB.

With i growing, obviously the probability of the latter decision grows. But optimization of the decision is not easy.

We assume that the extent, the precision, and thus the cost of IA verification are much higher than those of any acceptance check. This relatively high cost is the reason that one cannot afford IA verification too often. Thus acceptance checks are still useful as means of earlier, specialized error detection. The costs of IA verification are nothing extravagant. T. Gibbons advises "It is wise to run a series of check programs on the database, to find all the errors before attempting a restart."⁴

Comparison of the performances of the "unsafe" and "safe" approaches under pessimistic circumstances shows the advantages of the latter. From now on we discuss the "safe" approach exclusively.

Assumptions for the Analysis of the Generic Technique for Database Recovery

We analyze the generic database recovery technique under the following assumptions:

- A1. The database functions in a real-time environment.
- A2. The database has a single division.
- A3. Transactions are implemented accordingly to the recovery block scheme.
- A4. Database recovery from the hidden hard crashes is considered.
- A5. Integrity Assertion Verifier is completely effective (i.e. detects all errors). (At first sight this assumption

seems to collide with our view of recovery-block acceptance tests as not completely effective. But there are important differences between the two:

1. Integrity assertion verification is performed less often than acceptance test execution of any transaction. Thus integrity assertions can be more detailed and comprehensive with the comparable overhead.
2. Integrity assertions are for general use, while acceptance tests are transaction-specific. Thus integrity assertions can be more thoroughly tested. Note that the assumption A5 could be discarded by a modification of our model as proposed above, namely by including a mechanism for the restoration of earlier snapshots when needed.)

A6. "Recovery" software is completely reliable. (Unlike the software of transactions, the "recovery" software, as a standard package, can be thoroughly tested and made quite reliable.)

Description of Database Recovery Techniques

The generic database recovery technique can be implemented in many ways. Our candidates are:^{9,11}

1. Complete Database Dump—Before entry to each DBRB, the whole database is dumped (copied).
2. Incremental Dump—An initial or periodic database dump creates a basis. Before entering the next DBRB, all blocks/files updated in the previous DBRB are copied, i.e. incremental dump is created. This permits the restoration of the last snapshot, using the complete database dump, if necessary, and using the results recorded on incremental dumps. (Note that incremental dumps alone would not ensure recovery; blocks/files of the database not changed at all are not recorded on any incremental dump.)
3. Audit Trail—An audit trail (a log) records sequences of actions performed by transactions on files/blocks inside a given DBRB. It can be used to restore the latest snapshot. It can also be used to back up particular transactions, which is important when one needs to allow for abortion of a single transaction.
4. Differential Files—The main file (the frozen database) stores the latest snapshot, and the differential file is a log recording all later updates, executed inside of a DBRB. The merge of the differential file with the main file is done only after positive verification of the logical database made up by the main and differential files (i.e. before entering the next DBRB).
5. Backup/Current Versions—Copies blocks/files just before they are updated for the first time inside a DBRB. From then on only this copy of block/file is accessed. The "original" is a backup version used, if necessary, for database recovery. Using the latest backup copies for each block/file, the latest snapshot can be reconstructed.
6. Multiple Copies—More than one copy of each block/file is stored. The different copies are identical except during

an update. There are two variants of this technique. The first uses an odd number of copies and applies "majority voting" to select the correct data value. Fewer than half of the copies are ever updated at a time. The other variant uses only two copies, but each has an "update-in-progress" flag. A flag set indicates that the associated copy is under update and thus possibly in an inconsistent state. Only one copy at a time can be updated. Copies not under update at a moment of crash are consistent, if there are no hidden crashes.

7. Careful Replacement—The principle of this technique is the avoidance of updates "in place." Altered data are put in a copy of the original. The original is deleted only after the alteration is complete and has been certified. Note that two copies exist only during update.

Database Recovery Techniques— A Qualitative Analysis of Usefulness

Analyzing the potential usefulness of the presented database recovery techniques, we have found out that two of the techniques, multiple copies and careful replacement, can not be used for recovery from the hidden hard crashes. The multiple copies technique can be successfully used to recover from overt hard crashes or even, using majority voting, for error detection. But when hidden hard crashes occur, all copies could be equally contaminated and useless. The careful replacement technique deletes the original as soon as the new copy is certified. By definition, errors causing overt hard crashes are detected instantaneously and the technique can protect against them. But if hidden hard crashes occur and the IA verification is not completely effective (does not detect all integrity violations), the errors may be detected only some time after this verification. By then there is no way to restore the original, which has been deleted immediately after the IA verification.

Thus for the further analysis we are left with the following five database recovery techniques: complete database dump, incremental dump, audit trail, differential files, and backup/current versions.

Let us now try to answer the question: Which database recovery techniques could be used in the cases that (1) DBRB's are relatively short, (2) DBRB's are relatively long?

In the first case, clearly, we can afford undoing the results of database updates to back up to the most recent snapshot, so we do not need to prepare extensive physical database snapshots at the entry to a DBRB. Just logging the updated item values would suffice. Thus the audit trail technique seems suitable here.

In the second case, undoing the results of database updates would take too long. We must record database state (remember that we assume single-division database) at each DBRB entrance. The techniques that can be used here include

- complete database dump,
- incremental dump with an initial or periodic complete database dumps,
- differential files,
- backup/current versions.

Primitives for Database Recovery Techniques

We claim that it is both feasible and useful to present the database recovery techniques in terms of certain primitive actions, which we want to consider as atomic elements of the selected database recovery techniques. The feasibility is proved by the presentation of the set of these primitives, which follows.

Our long-term goal is the time-cost comparison of the database recovery techniques. Instead of analyzing each technique separately, we will analyze each primitive. As each recovery technique is a sequence of these primitives, the resulting recovery technique cost can be easily obtained. This is one of the aspects of the usefulness of the primitives. Others, we hope, will include the increased clarity of the description of these techniques.

Below we define the primitives and later we show how to construct the selected database recovery techniques out of these primitives.

In the definitions the notion of a set of "corresponding" pages, or of a "generic" page, is used: whenever page B of file Y was initialized as the copy of page A of file X, we say that these pages are corresponding or that both pages map into the same generic page, even if the content of page B, due to its updates, no longer is identical to the content of page A. In a sense, a generic page is the generalization, beyond a single file, of a page version. The function "pg" (as "page"), used in the figures for the next section, maps any file into the set of its generic pages. The function "gp[F]" maps a set of generic pages into the corresponding pages of a file F. (Note that the inverse of pg is not a function.)

The primitives are as follows:

C/DUMP(X)—Make complete database dump, and call it X.

COPY(Y, DB, X)—Copy all distinguished, i.e. with their IDs in X, pages of the database into the file Y (if a page has two versions in Y—delete the old one).

ERASE(X)—Erase block/file X.

HALT—Halt normal database processing after transactions currently writing into database write their results completely. This primitive ends a DBRB.

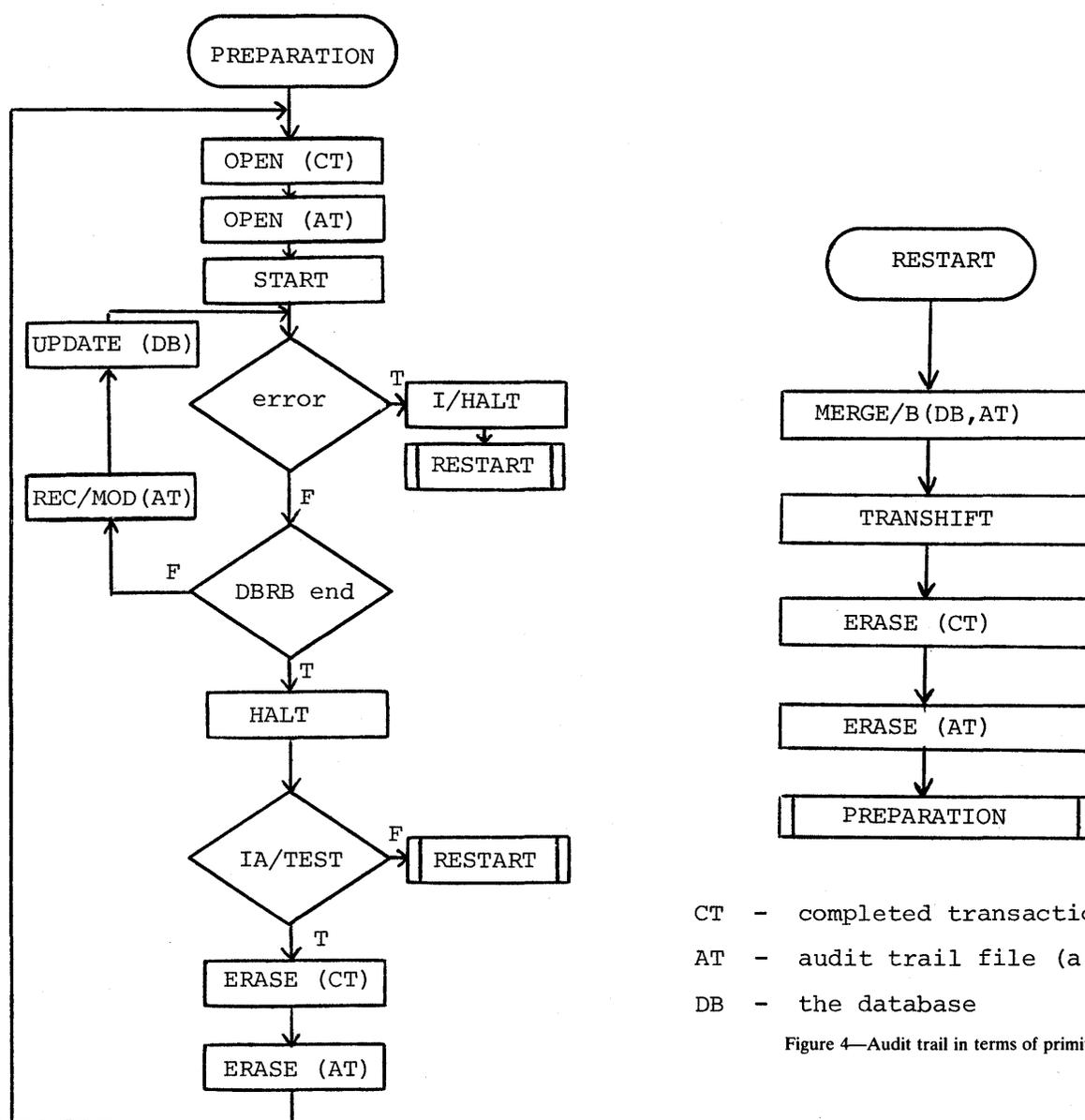
IA/TEST—Test original database consistency, using IA Verifier.

//HALT—Halt normal database processing immediately upon detection of an error by an acceptance check. This primitive initiates restart of the current DBRB unconditionally, so we need not wait for writing transactions as in HALT.

LOG/IA/TEST(X)—Test, using the IA Verifier, the consistency of the current logical database. The current logical database consists of the most current values of database items that are stored in the database or in the block/file X. This corresponds to a logical merge of X with the database followed by IA/TEST.

MERGE/parameter(DB, X)—There are two variants:

1. *MERGE/B(DB, X)*—Merge the database with the log X backwards (i.e. use the oldest recorded values of data of X to restore the correct database).
2. *MERGE/F(DB, X)*—Merge the database with the log X



CT - completed transaction log
 AT - audit trail file (a log)
 DB - the database

Figure 4—Audit trail in terms of primitives

forward (i.e. use the newest recorded values of data of X to build the correct database).

OPEN(X)—Open file X.

OVERWRITE(X, Y, Z)—Replace (e.g., by pointer switching) pages of X specified by page identifiers stored in Z with the corresponding pages of Y. If Z is omitted—each page of Y replaces the corresponding page of X.

REC/ID(X)—Record in X identifiers of database pages to be modified.

REC/MOD(X)—Record data (e.g., a 4-tuple: transaction ID, item ID, old item value, new item value) about modifications on a log X.

START—Start normal database processing. This primitive initiates a new DBRB.

TRANSIFT—Shift into system input queue:

- a) all transactions recorded on “completed transaction log”, i.e. finished but not saved transactions,

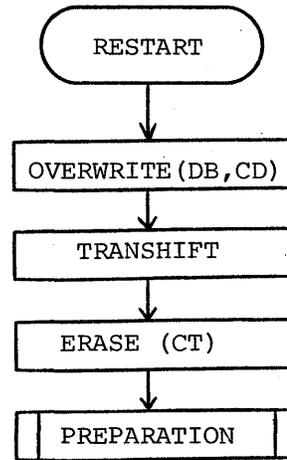
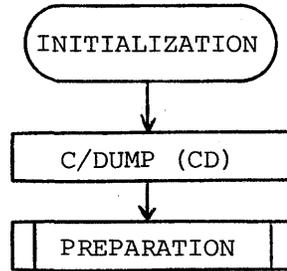
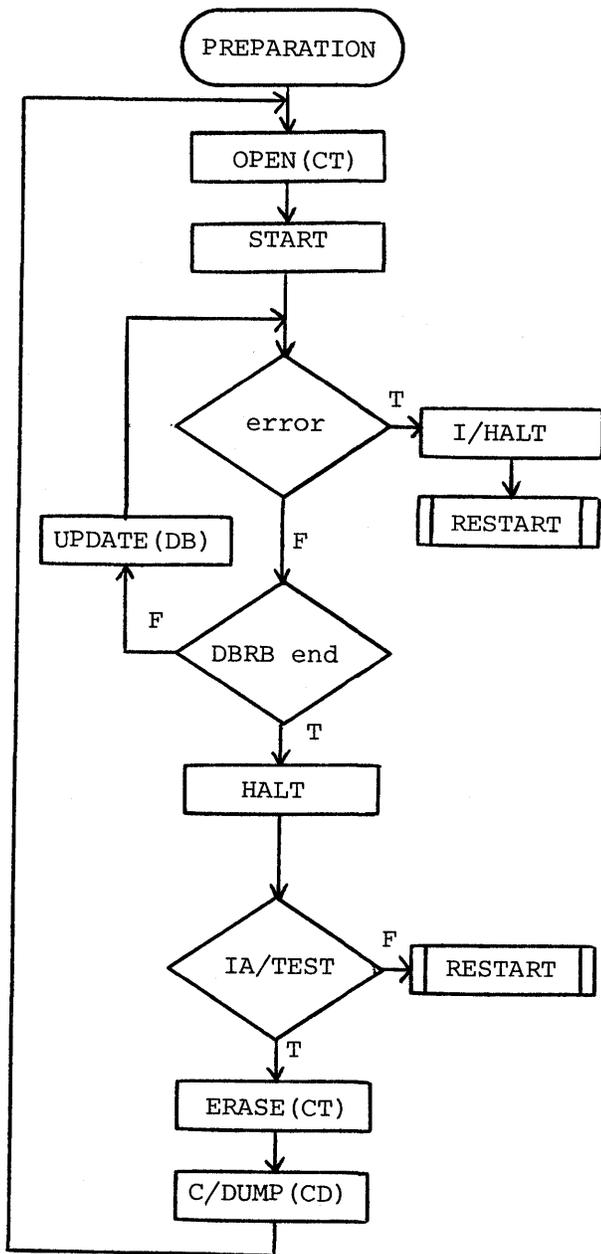
- b) all other transactions present in the system, i.e. unfinished transactions, and sort transactions of system input queue in the arrival time order.

UPDATE(X)—Write an update in the file X. This primitive specifies which file should be updated when more than one file includes the same generic page that is to be updated.

The Selected Database Recovery Techniques in Terms of Primitives

Using the primitives defined above, we have built the following selected recovery techniques:

1. audit trail (see Figure 4),
2. complete database dump (see Figure 5),



CD - complete database dump
 CT - completed transaction log
 DB - the database

Figure 5—Complete database dump in terms of primitives

3. incremental dump (see Figure 6),
4. differential files (see Figure 7),
5. backup/current version (see Figure 8).

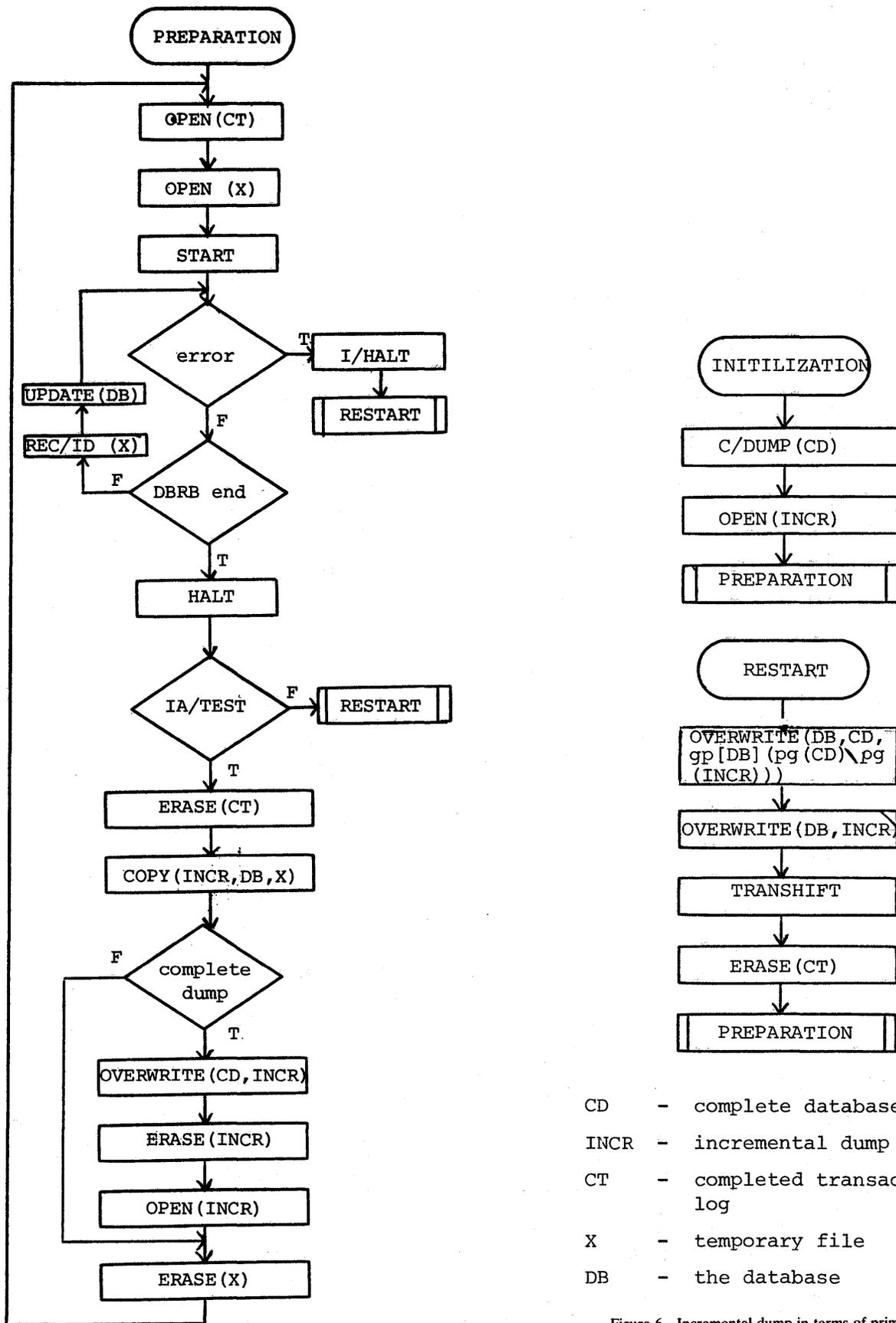
The flowcharts of these recovery techniques combined with the definitions of the primitives should be self-explanatory (you may wish to consult short description of the techniques in the section "Description of Database Recovery Techniques.") The completed transaction log, referred to in the above-mentioned figures, records all transactions that are completed (their results are already written into the database), but with updates not saved yet, that is, the end of the DBRB in which transaction finished its execution has not been reached. This allows it to reexecute completed transactions, if necessary.

For comparison we present in Figure 9 the list of the primitives used by the selected database recovery techniques. This demonstrates how much in common the techniques have.

FUTURE RESEARCH AND EXTENSIONS

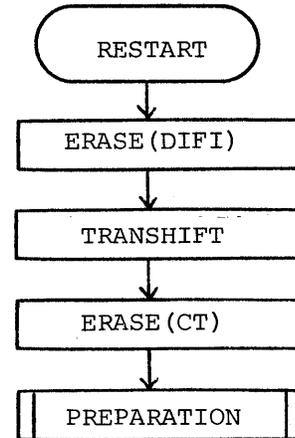
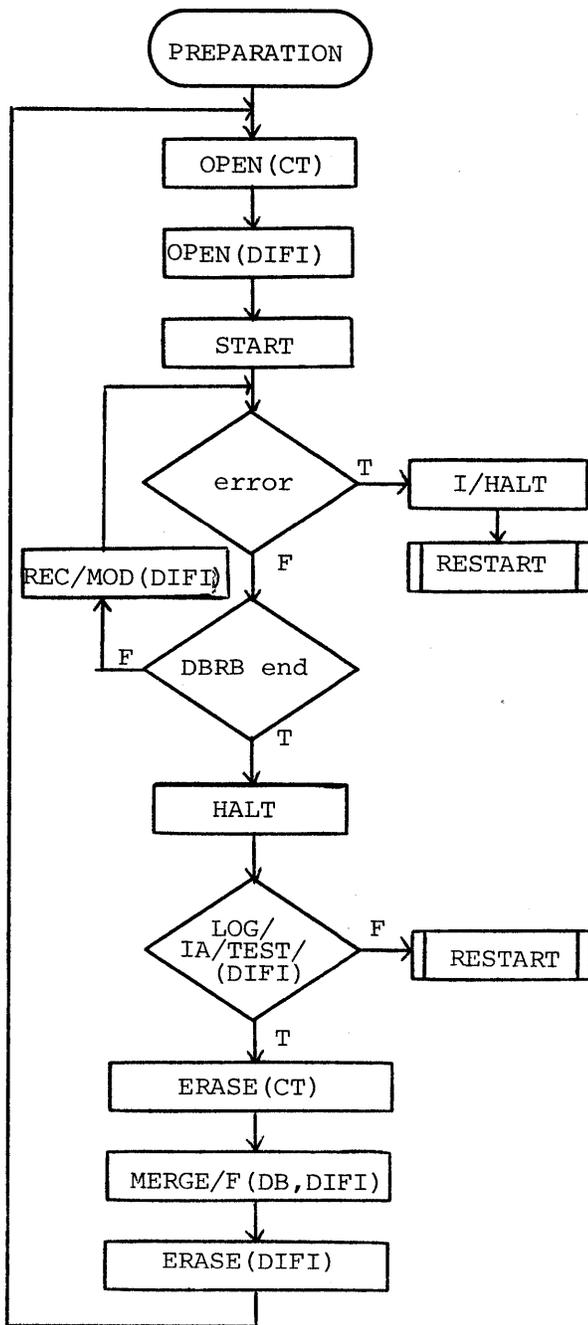
It is our intention to compare the performance of the above database recovery techniques for hidden hard crashes. We plan to base the analysis of the recovery techniques on the analysis of the primitives constituting them, which is to be made first.

The database recovery cost considerations will be limited to the time-cost analysis, as the storage cost does not seem to be essential in the real-time environment. Time costs can be



- CD - complete database dump
- INCR - incremental dump file
- CT - completed transactions log
- X - temporary file
- DB - the database

Figure 6—Incremental dump in terms of primitives



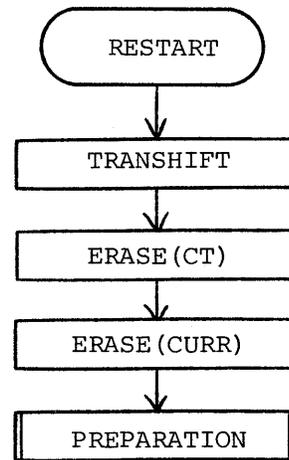
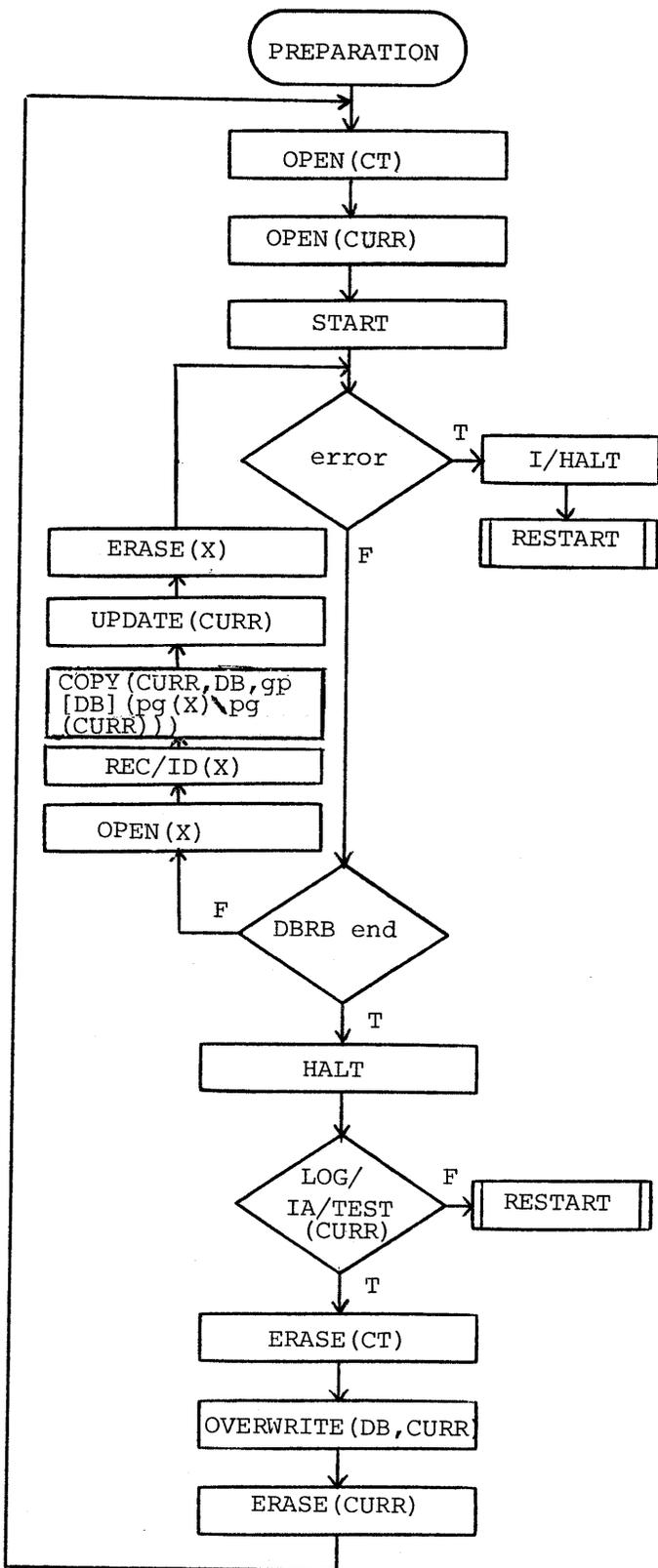
- CT - completed transaction log
- DIFI - differential file (a log)
- DB - the database

Figure 7—Differential files in terms of primitives

classified as fixed and variable costs.² Fixed time costs, independent of the number of errors detected, cover all preparatory actions necessary for restart when an error is detected. Variable time costs, incurred only if an error is detected, cover all restart actions. The fixed time costs, as completely predictable, can be more easily incorporated within real-time constraints of the system operation during system design. But the variable time costs are the threat to real-time constraints of the system operation (these constraints could be defined as the maximum time the system can be left nonoperational without grave consequences). Thus in our opinion only the time-cost analysis is essential and the variable time cost is the

main criterion of the cost analysis for a recovery technique in our environment.

A designer or a database administrator defines Database Recovery Blocks by specifying the intervals of regular database processing between consecutive recovery preparation phases. Long DBRB will increase chances that the restart will be time consuming, involving the reexecution of many transactions and keeping the system nonoperational too long. Short DBRB will increase the costs of the preparatory actions (snapshots, etc.), increasing the chances of breaking the real-time requirements. Thus a compromise is clearly needed. This compromise will affect operational costs of a given database



- CT - completed transaction log
- CURR - current version
- DB - the database
- X - temporary file

Figure 8—Backup/current version in terms of primitives

recovery technique. We want to find the minimum cost schedule for all of the above techniques.

Only the database recovery techniques for hidden hard

crashes have been discussed. These techniques can obviously cope with the overt hard crashes too, but they are much more expensive than specialized recovery techniques. The tech-

RECOVERY TECHNI- QUE PRIMITIVE	AT	CD	ID	DF	BC
C/DUMP (X)	-	IP	I	-	-
COPY (Y, DB, X)	-	-	P	-	P
ERASE (X)	PR	PR	PR	PR	PR
HALT	P	P	P	P	P
IA/TEST	P	P	P	-	-
I/HALT	P	P	P	P	P
LOG/IA/TEST (X)	-	-	-	P	P
MERGE/par (DB, X)	R	-	-	P	-
OPEN (X)	P	P	IP	P	P
OVERWRITE (X, Y, Z)	-	R	PR	-	P
REC/ID (X)	-	-	P	-	P
REC/MOD (X)	P	-	-	P	-
START	P	P	P	P	P
TRANSHIFT	R	R	R	R	R
UPDATE (X)	P	P	P	-	P

AT - Audit Trail
 CD - Complete Database Dump
 ID - Incremental Dump
 DF - Differential Files
 BC - Backup/Current Versions
 I - Primitive used in initializa-
 tion phase
 P - Primitive used in preparation
 phase
 R - Primitive used in restart
 phase
 X, Y - File names
 DB - The database

Figure 9—Use of the primitives by the selected database recovery techniques for database recovery from overt hard crashes will be investigated later, using the analogous approach.

There are a number of possible extensions to our work:

1. increasing the concurrency of normal database processing by exploitation of elements of a recovery mechanism¹;
2. concurrent execution of recovery actions and normal database processing, for example, dumping concurrent with regular processing^{4,6};
3. concurrent execution of a few recovery actions, such as checking database files concurrent with dumping of these files⁴ or processing several logs (or log sections) in parallel (e.g., the Audit Trail Tag File method⁴);
4. creating single transaction backup facilities by use of deferred commit^{4,7} or use of transaction save points;⁶
5. independent dumping of sections of a database, especially when these sections have varying level of activity or the database is large (compare the noncontemporary file dumps method⁴);
6. investigation of special database recovery implementation methods, for example the duplexing of logs and files^{4,7} or the use of multiprocessor systems; and
7. investigation of after-implementation tunability of recovery methods.

In the refinement of our approach we will include some of these ideas.

ACKNOWLEDGMENT

We would like to thank Ms. Beverly Hill for the preparation of the figures.

REFERENCES

1. Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Systems*, June 1980.
2. Chandy, K.M., J.C. Browne, C.W. Dissly, and W.R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," *IEEE Trans. on Software Engineering*, March 1975.
3. Garcia-Molina, H. *Reliability Issues for Completely Replicated Distributed Databases*. Princeton University, Dept. of EECS, Technical Report #266, 1980.
4. Gibbons, T. *Integrity and Recovery*. Hayden Book Company, 1976.
5. Giordano, N.J., and M.S. Schwartz, "Data Base Recovery at CMIC," 1976 SIGMOD International Conference on Management of Data.
6. Gray, J., P. McJones, M. Blasgen, et. al. *The Recovery Manager of a Data Management System*. IBM Technical Report RJ 2623.
7. Gray, J. *A Transaction Model*. IBM Technical Report, February 1980.
8. Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, June 1975.
9. Sayani, H.H., "Restart and Recovery in a Transaction-Oriented Information Processing System," *ACM SIGMOD Workshop on Data Description, Access and Control*, 1974.
10. Sibley, E.H., "The Development of Data-Base Technology," *Comp. Surv.*, March 1976.
11. Verhofstad, J.S.M., "Recovery Techniques for Database Systems," *Comp. Surv.*, June 1978.

Data compression procedures utilizing the similarity of data

by YAHIKO KAMBAYASHI

Kyoto University
Kyoto, Japan

and

NARAO NAKATSU

Aichi University of Education
Aichi, Japan

and

SHUZO YAJIMA

Kyoto University
Kyoto, Japan

ABSTRACT

In large database systems, we usually encounter the situation when a set of similar data is to be stored. This paper discusses efficient data compression procedures utilizing similarity of data. These procedures are suitable for compressing versions of programs, a series of data produced in an office etc.

The procedure to compress one string utilizing regularity of data is as follows:

1. Calculate all maximum repeated substrings in the given string.
2. Since each repeated substring is required to be stored only once, replace the second and later occurrence of the same substring by the code which shows the position of the first occurrence of the substring.

The procedure to compress two strings w_1 and w_2 utilizing data similarity is as follows:

1. Calculate all maximum common substrings of w_1 and w_2 .
2. Find a minimum cover for w_2 using the maximum common substrings contained in w_1 .
3. Encode w_2 by codes, each of which shows a substring of w_1 .

These procedures are shown to require time only proportional to the total length of data and thus they are efficient. Combinations and variations of these two procedures are also discussed in the paper.

1. INTRODUCTION

In large database systems, we usually encounter the situation when a set of similar data is to be stored. This paper discusses

efficient data compression procedures utilizing the similarity of data.

Popular data compression methods such as Huffman coding¹ and arithmetic coding² will not produce the best possible results in such cases. Examples of similar data are as follows:

- versions (or variations) of programs or papers;
- a set of monthly data of some organizations such as databases; and
- periodically dumped files of a large file;

There are two known methods to handle the problem.

1. Prepare a dictionary of common phrases which appear in the set of data. Then assign a proper code according to the frequency of each phrase.^{3,4}
2. Store one or more sets of original data called the reference data. For other data, store the difference from the reference data.

The first approach is suitable when there are a lot of data. In above situations (a), (b) and (c), the most recently generated program or the most recent datum are rather frequently retrieved, while other old data are not used often. If we store such data by the second approach, the most frequently referred datum is used as a reference datum. So the time for reconstructing the most frequently used datum is not necessary. The first approach always requires computation to recover an original datum. Furthermore the second approach is less redundant when the number of data is not large and these data have strong similarity.

The differential file approach⁵ is regarded as an example of the second approach. Compression of the data difference is not considered in this approach. Kang et al.⁶ discussed the

difference compression problem on a relation, which is equivalent to the string data compression problem when all data are of equal length. Consider three data, $w_1=abbcbbe$, $w_2=abbebe$, and $w_3=acbebe$. If w_1 is stored as a reference string in the system, w_2 and w_3 can be stored as $\frac{1}{4e}$ and $\frac{1}{2c4e}$, respectively. Here $i|j_1a_1j_2a_2$ shows the sequence is equal to w_i except that the j_k th symbol is replaced by $a_k(k=1,2)$. w_3 has another expression $\frac{2}{2c}$ and it is shorter. Kang et al. used the minimal spanning tree algorithm to obtain the least redundant codes to express the given set of data. Major problems of this approach are as follows:

1. The approach is not suitable for variable length data.
2. Permutation of a datum is not handled efficiently.

If we consider versions of programs, sometimes the same procedure blocks are just permuted. So the above two problems must be solved.

We have developed new data compression procedures from this motivation. Our procedures utilize the consecutive retrieval property of files^{7,8} to represent similarity. That is, the same part is represented by the interval of the reference datum. Consider the two data $w_1=abeca$ and $w_2=abecacabe$. Let $w(i)$ be the i th symbol of w and $w(i:j)$ be the substring $w(i)w(i+1)\dots w(j)$. w_2 can be expressed by a set of substrings of w_1 . In this case, w_2 can be expressed by $w_1(1:4)w_1(3:5)w_1(2:3)$. By a proper encoding of $w_1(i:j)$, a compact expression for w_2 can be obtained. Generalization of this idea is used in this paper.

Basic concepts to be used in this paper are given in the next section. In Section 3, data compression procedures for single text string and two text strings are described, which are realized by encoding the second and latter occurrence of the same substring. Using the procedure developed by Weiner,¹⁰ all maximum repeated sequence can be obtained in time proportional to the length of the given string, and thus these procedures can be implemented efficiently. These procedures can be extended to the cases when there are more than two strings. This approach, however, requires long decoding time for some text strings. An improved procedure for multiple text strings is discussed in Section 4, which is a combination of the above procedures, a procedure to find all maximum common substrings and a covering procedure. We can also utilize the procedure developed by Weiner to find all maximum common substrings. Usually a covering problem is time-consuming, but we will show that there is a straightforward procedure to solve this particular covering problem. By these reasons, the procedure presented in Section 4 requires time only proportional to the total length of text strings, when the number of the given strings is two.

The procedures discussed in this paper are summarized as follows:

1. A linear time data compression procedure for a single string by detecting all maximal repeated substrings (Section 3).

Although more than one string can be compressed by concatenating all the given strings, the method requires long decoding

time for a string which is located at the last part of the composite string.

2. A linear time data compression procedure for two strings by detecting all maximal common substrings (Section 4).

One of the two strings is selected as the reference string. Extensions of the procedure for a multiple data compression are rather easy. There is an efficient procedure for such cases although the procedure may not produce the optimum solution.

3. A linear time data compression procedure for two strings by combining the above two procedures (Section 5).

The assumptions used in this paper are as follows.

1. The given strings have strong similarity, that is, there are substrings appearing frequently.
2. We are interested in only the coding procedures where the decoding procedure can be applied sequentially from the top of encoded strings. $w = ababa$ can be encoded as $abw(1:3)$ or $w(3:5)ba$. The former can be decoded from the top of the string while the latter cannot. In general, if we permit nonsequential decoding, both coding and decoding procedures will have more computation time.
3. We do not introduce an imaginary string as a reference string. If we introduce such a string, there are cases when a better compression can be obtained. A procedure to find the imaginary string which will give the optimum solution is time-consuming. (The Steiner tree problem can be regarded as a subclass of this problem and even a special case of the Steiner tree problem is known as NP-complete.)
4. We assume that all the characters appear in all data to be compressed. A little modification is required, if some character is not used in some datum (see Section 4).

2. BASIC CONCEPTS

In this section, basic definitions used in this paper are presented.

Definition 1

A string w is defined over a finite set of symbols Σ . Let $w(i)$ be the i th symbol of string w and $w(i:j)$ ($i \leq j$) be the substring $w(i)w(i+1)\dots w(j)$ of w . $|w|$ denotes the length of w . For a string w , u is called a repeated substring (RS for short) of w if u appears in w at least twice. The occurrences of u in w are numbered sequentially from the left to right. Thus the leftmost occurrence of u is called the first occurrence of u . The string $w(i:j)$ is called a maximal RS (MRS for short) when $w(i:j)$ is an RS of w and neither $w(i-1:j)$ nor $w(i:j+1)$ appears in w elsewhere.

In Definition 1, an RS, u may overlap with another u in w .

Definition 2

For strings w_1 and w_2 , u is called a common substring (CS for short) of w_1 and w_2 if u is a substring of both w_1 and w_2 . Let u be a CS of w_1 and w_2 , i.e. $u = w_1(i:j) = w_2(k:h)$. String u is called a maximal CS (MCS for short) of w_1 with respect to w_2 if neither $w_1(i-1:j)$ nor $w_1(i:j+1)$ is a CS of w_1 and w_2 .

In Definition 2, even if $w_1(i:j) (= w_2(k:h))$ is an MCS of w_1 with respect to w_2 , $w_2(k:h)$ is not always an MCS of w_2 with respect to w_1 , since w_1 can contain $w_2(k-1:h)$ or $w_2(k:h+1)$ without violating the condition of the maximality of $w_1(i:j)$. There may be more than one MCS and two MCSs may be overlapped.

Example 1

Let us consider the following two strings.

1 2 3 4 5 6 7 8 9 10 11 12 13 1 2 3 4 5 6 7 8 9 10 11 12
 $w_1 = b a a a a a a a b a a a$ $w_2 = a b b b a a a a b a a$

“aaaa” is one of the RSs of w_1 but is not an MRS of w_1 . $w_1(10:13) = “baaa”$ is one of the MCSs of w_1 with respect to w_2 since $w_1(9:13)$ is not a CS of w_1 and w_2 . Note that $w_2(4:7) = w_1(10:13)$ is not an MCS of w_2 with respect to w_1 , because $w_2(4:9)$ is a CS of w_1 and w_2 .

Basic terminologies of graph theory are used in this paper without definitions (see Harary⁹).

Definition 3

A weighted directed graph $G(V, E, c)$ is defined as follows. V is a set of nodes. E is a set of directed edges e_{ij} , where e_{ij} corresponds to an ordered node pair $(v_i, v_j)(v_i, v_j \in V)$. c is called a cost function which is a mapping from E to N (the set of nonnegative integers).

A directed spanning forest of G is a set of directed trees $F = \{T_1, T_2, \dots, T_k\}$ which satisfies following conditions.

1. Each directed tree $T_i = (V_i, E_i, c_i)$ has one root node from which one directed path exists to each node in V_i .
2. $\cup V_i = V$ and $E_i \subset E$ hold.
3. V_i and E_i are disjoint from each other, respectively.
4. $c_i(e) = c(e)$ for any e in E_i .

When $k = 1$ the tree T_1 is called a directed spanning tree of G . Cost of F is defined by $\sum_{i \in E_i} c_i(e)$. The minimum cost directed spanning forest is a directed spanning forest of G with minimum cost.

Example 2

Consider the graph shown in Figure 4. The minimum cost directed spanning tree of the graphs is shown by bold lines.

We define a data structure, a prefix tree, for a string w .¹⁰ This structure is used in our procedures for data compression.

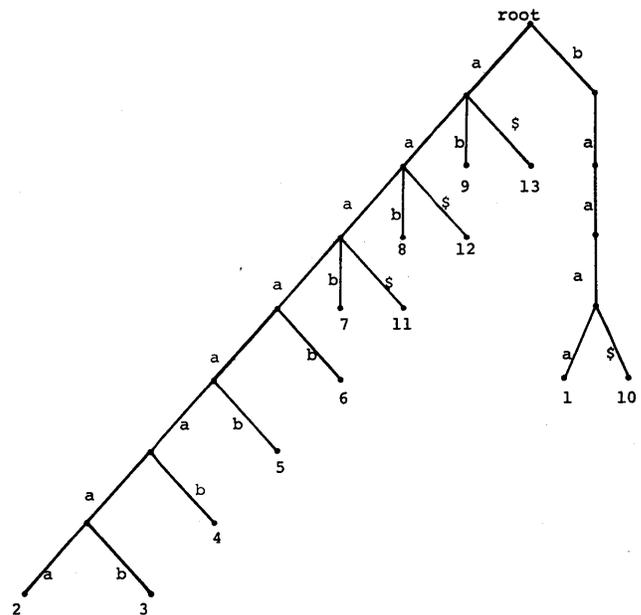


Figure 1—The prefix tree for the string $w_1 = b a a a a a a a a b a a a$

Definition 4

Let $w = w(1)w(2) \dots w(n)$ be a string over an alphabet Σ . Let $w' = w\$$ ($\$$ is an endmarker of w and $\$$ is not in Σ). A position identifier $W_i = w'(i:j)$ for each position i is defined as the shortest string which is not an RS of w' . If W_i is given we can uniquely identify the starting position i in w' .

Definition 5

A prefix tree P for a string w of length n is a tree which satisfies the following conditions.¹⁰ Let $w' = w\$$.

1. P has $n + 1$ leaf nodes each of which has unique label i and each edge of P is labeled by a symbol in $\Sigma \cup \{\$\}$. The leaf i has one-to-one correspondence with the position i of w' .
2. The string obtained by concatenation of edge labels on the path from the root to the leaf i is equal to the position identifier W_i .
3. For any N_i , all edges (N_i, N_j) have different labels, where N_i is an interior node of P and N_j is an immediate successor of N_i .

Example 3

The prefix tree for the string $w_1 = b a a a a a a a a b a a a$ is shown in Figure 1. For example, the position identifier W_1 for the position 1 of w_1 is “baaaa”. Note that all RSs of w_1 are represented in the prefix tree of w_1 . Let us consider two leaf nodes i and j which have the same predecessor in the prefix tree. If the level, the path length from the root, of the com-

mon predecessor is equal to k then W_i and W_j have the same prefix of length k . This means $w_1(i:i+k-1) = w_1(j:j+k-1)$. In this example, nodes 1 and 10 have a common predecessor of level 4. Nodes 2 and 5 have a common predecessor of level 5. Thus we have the following equations:

$$w_1(1:4) = w_1(10:13), w_1(2:6) = w_1(5:9).$$

For a given string w , Weiner showed an algorithm for constructing the compact prefix tree in time of $O(|w|)$. The construction algorithm is shown in Weiner.¹⁰ In the following section, we show data compression procedures using RSs, and in Section 4 we extend our procedures to the compression of more than two strings.

3. DATA COMPRESSION PROCEDURE UTILIZING MAXIMAL REPEATED SUBSTRINGS

In this section, efficient data compression procedures for one or two strings are presented. The following example shows our basic idea.

Example 4

Consider the string w_1 in Example 1. By Figure 1, we know that "baaa" and "aaaaaaa" are RSs of w_1 . That is, $w_1(1:4) = w_1(10:13)$ and $w_1(2:8) = w_1(3:9)$. If we know the above property, we need not store the whole string. w_1 is expressed by $w_1(1:2)w_1(3:9)w_1(10:13)$ which can be replaced by $w_1(1:2)w_1(2:8)w_1(1:4)$. Corresponding to this sequence we will define a coded string,

$$\bar{w}_1 = ba\#2,6\#1,3$$

Here, except the first one, $w_1(i:i+k)$ is represented by $\#i,k$. The decoding process is as follows. Since $\bar{w}_1(1) = "b"$ and $\bar{w}_1(2) = "a"$, we have $w_1(1) = "b"$ and $w_1(2) = "a"$. $\#2,6$ represents that the succeeding string of length $7(=6+1)$ is equal to the string starting from the position 2. In other words, $w_1(3) \dots w_1(9)$ corresponds to $w_1(2) \dots w_1(8)$. Thus

$$w_1(i+1) = w_1(i) \quad \text{for } i = 2, \dots, 8.$$

We have $w_1(3) = w_1(4) = \dots = w_1(9) = "a"$. The last part of w_1 is $w_1(1)w_1(2)w_1(3)w_1(4)$ which is known to be "baaa". Thus we can decode the expression \bar{w}_1 . When we use ASCII 8-bit Code for each symbol, 8×13 bits are required to store w_1 . Each integer in \bar{w}_1 can be coded by a bit string of length 3 in this case. So by our compression method, $8 \times 4 + 3 \times 4 = 44$ bits are required to store \bar{w}_1 .

In the above example, w_1 can be reconstructed by decoding from the first symbol of \bar{w}_1 to the last symbol. More storage reduction may be possible unless the sequential decodability of w_1 is assumed. For example, $w_1 = cdababcdabcd$ can be expressed as $\bar{w}_1 = \#7,3abcd\#5,3$. In this case, w_1 can be recovered by decoding $\#5,3$ and $\#7,3$ in this order. This coding schema is not sequentially decodable and is time-consuming for decoding. We only consider the compression procedure by which w_1 is sequentially decodable from \bar{w}_1 .

In order to get a best possible data compression, we have to decide whether the code $(\#i,j)$ or the original substring is to be stored. A threshold value T is determined for this purpose. When the length of an RS is greater or equal to T , the RS is stored as a pair of integers and if the length of an RS is less than T , we should store the original string. The formal procedure follows.

Procedure 1

Data compression procedure for one string using MRSs.

1. For a given string w of length n , construct the compact prefix tree of w . Note that each leaf is labeled by a number corresponding to a position in w .
2. Prepare arrays S and T of n elements and prepare a variable V_i for each interior node N_i of the prefix tree. Each element of S and each variable N_i is set to be ∞ .
3. Traversing the prefix tree in endorder, the following step (4) is applied to each interior node of the tree except the root node.
4. Suppose that the interior node N_i has leaves $\{j_k\}$ and interior nodes $\{N_l\}$ as immediate successors ($1 \leq k \leq k'$, $1 \leq l \leq l'$). Let V_i be the minimum element of the set $\{j_k\} \cup \{V_l\}$ ($1 \leq k \leq k'$, $1 \leq l \leq l'$). If N_i has no leaves as an immediate successor, consider next interior node. For each j in $\{j_k\}$ and $j > V_i$, $S(j+L(N_i)-1) \leftarrow \text{Min}(S(j+L(N_i)-1), j)$ and if $S(j+L(N_i)-1) = j$ then $T(j+L(N_i)-1) \leftarrow V_i$, where $L(N_i)$ denotes the level of N_i .
5. S and T are calculated. $R \leftarrow n$.
6. Find such i that $S(i) \leq R \leq i$ and $S(i)$ is minimum.

If above condition is not satisfied by any i then $R \leftarrow R - 1$.

If $i - S(i)$ is greater than a threshold value then replace $w(S(i):i)$ by $\#T(i), i - S(i) + 1$ and $R \leftarrow S(i) - 1$ else $R \leftarrow R - 1$.

If $R < 1$ then stop else repeat (6).

Procedure 1 requires $O(n)$ computation time, which can be proved easily by the fact that the compact prefix tree has only $O(n)$ nodes.¹⁰

For the purpose of sequential decodability of w , we have replaced the latter occurrence of an RS by the indicator to the former occurrence. In this paper we will discuss procedures to obtain a minimum representation of a sequence consisting of Σ , $\#$, and integers. Further reduction may be possible by a proper encoding of the resulting strings. For the string which has a long RS can be compressed efficiently by Procedure 1. For storing two strings, Procedure 1 can be also applied.

Procedure 2

Data compression procedure for two strings using MRSs.

1. For given two strings w_1 and w_2 , let w be $w = w_1w_2$.
2. Apply the Procedure 1 to w and obtain \bar{w} .

3. Store w_1 and w_2 as $n_1\bar{w}$, where n_1 is the length of w_1 . (We can separate w_1 and w_2 by n_1 when decoding is done.)

When we store two versions of programs, we may be able to find a long RS in $w = w_1w_2$, which will contribute to a good reduction of storage space. When \bar{w} ($w = w_1w_2$) is given, w_1 can be obtained without decoding the whole sequence, since \bar{w} is decoded sequentially from the beginning. In order to obtain w_2 , however, we have to decode the whole sequence. By this reason if there are two files w_1 and w_2 where w_1 is retrieved more frequently than w_2 , then w_1w_2 must be selected instead of w_2w_1 . When w_1 and w_2 are equally retrieved, then we can select shorter \bar{w} for $w_a = w_1w_2$ and $w_b = w_2w_1$. In Procedure 2, instead of encoding the length of w_1 we can use a separation mark \$, i.e. $w = w_1\$w_2$, which may be better in some cases.

Procedures 1 and 2 are efficient since they require time only proportional to the length of input strings.

4. DATA COMPRESSION PROCEDURES UTILIZING MAXIMAL COMMON SUBSTRINGS

By extending Procedure 2, we can efficiently compress n strings w_1, w_2, \dots, w_n . We construct the string $w = w_1w_2\dots w_n$ and apply Procedure 1 to w . This procedure is useful in some cases but it has the following problem. The string w_k can be recovered after w_1, w_2, \dots, w_{k-1} are recovered. So it is time consuming to reconstruct w_k when k is large. We should consider the decoding time as well as the storage reduction.

In this section, we introduce a new similarity of two strings. By this similarity, an efficient data compression procedure for two strings is presented and a procedure for more than two strings is also mentioned.

Definition 6

The set of substrings $C = \{w_2(i_1:j_1), w_2(i_2:j_2), \dots, w_2(i_k:j_k)\}$ is called an S -cover of w_2 with respect to w_1 if the following conditions are satisfied.

1. For any $w_2(i)$, there is at least one substring $w_2(i_h:j_h)$ in C such as $i_h \leq i \leq j_h$.
2. Each $w_2(i_h:j_h)$ is an MCS of w_2 with respect to w_1 .

$w_2(i_h:j_h)$ is redundant if $C - \{w_2(i_h:j_h)\}$ is also an S -cover of w_2 . An S -cover without any redundant element is called a minimal S -cover and the minimal S -cover with minimum number of elements is called a minimum S -cover.

In general, minimum cover problem is very difficult but in our case, the minimum S -cover can be proved to be calculated in linear time.

Definition 7

The similarity $s(w_1, w_2)$ from w_1 to w_2 is defined by the number of elements of the minimum S -cover of w_2 with re-

spect to w_1 . $s(w_1, w_2)$ is regarded as infinite (denoted by ∞) if there exists no S -cover of w_2 with respect to w_1 .

The basic idea of the data compression is shown in the following example.

Example 5

Consider the next two strings.

$$w_1 : \overset{1}{a}\overset{2}{a}\overset{3}{d}\overset{4}{c}\overset{5}{a}\overset{6}{d}\overset{7}{c}, \quad w_2 : \overset{1}{c}\overset{2}{a}\overset{3}{a}\overset{4}{d}\overset{5}{c}\overset{6}{b}\overset{7}{a}$$

$C_1 = \{w_1(1:4), w_1(5:7)\}$ is a minimum S -cover of w_1 w.r.t. (with respect to) w_2 . So $s(w_2, w_1) = 2$ by Definition 7. On the other hand, the symbol "b" does not appear in w_1 and there is no S -cover of w_2 w.r.t. w_1 and $s(w_1, w_2) = \infty$. $w_1(1:4)$ and $w_1(5:7)$ equal $w_2(2:5)$ and $w_2(3:5)$, respectively. When w_2 is used as a reference string of w_1 , w_1 can be expressed by a concatenation of substrings of w_2 . When $w_2(i:i+k)$ is coded as i,k , w_1 is expressed as 2,2,3,3,2, where the first value 2 means that the reference string is w_2 .

An outline of our data compression procedure for two strings is shown as follows. Two parameters α and β are used, where α and β are memory space for a symbol and a numeral, respectively.

Procedure 3

An outline of data compression procedure for two strings using MCSs.

1. Calculate all MCSs of $w_1(w_2)$ w.r.t. $w_2(w_1)$ (Procedure 4).
2. Find a minimum S -cover of $w_1(w_2)$ w.r.t. $w_2(w_1)$ and calculate $s(w_1, w_2)$ and $s(w_2, w_1)$ (Procedure 5).
3. Suppose that $\alpha|w_1| + \beta(2s(w_1, w_2) + 1) \leq \alpha|w_2| + \beta(2s(w_2, w_1) + 1)$. w_1 is stored as its original form.
4. If $\beta(2s(w_1, w_2) + 1) < \alpha|w_2|$ then w_2 is stored as a concatenation of substrings of w_1 , else w_2 is stored as its original form.

In (3) and (4), the selection is determined by the storage space requirement. When w_1 is referred more frequently than w_2 , w_1 should be a reference string. Even if $s(w_1, w_2) = \infty$, we had better have a finite $s(w_1, w_2)$ by adding a dummy string to w_1 or by storing uncovered substring of w_2 as its original form. Such processing is not considered in this paper. We will show procedures used in Procedure 3.

Procedure 4

Calculation of all MCSs of w_1 w.r.t. w_2

1. For given strings w_1 and w_2 , construct the compact prefix tree T for the string $w = w_1\lambda w_2$ ($\lambda \notin \Sigma$). For convenience, assume that the leaf node corresponding to the position i of $w_1(w_2)$ is labeled by $i(i')$, respectively).

2. Prepare an array E of length $|w_1|$. Each element of E is set to be ∞ .
3. For each interior node N_i , a set U_i and a variable V_i are prepared. Let each U_i be empty and let each V_i be zero. Traversing T in the endorder, following step (4) is applied to each interior node except the root node.
4. If N_i has a leaf j as an immediate successor, then $V_i \leftarrow U_i \cup \{j\}$.
If N_i has a leaf j' as an immediate successor, then $V_i \leftarrow V_i + 1$.
If N_i has an interior node N_k as an immediate successor, then

$$U_i \leftarrow U_i \cup U_k \quad \text{and} \quad V_i \leftarrow V_i + V_k.$$

After this processing, if $V_i \neq 0$ then $E(k + L(N_i) - 1) \leftarrow \text{Min}(k, E(k + L(N_i) - 1))$ for each k in U_i and then $U_i \leftarrow \phi$ (an empty set).

Consider next interior node and repeat (4) until all interior nodes are exhausted.

5. E presents a set of MCSs of w_1 w.r.t. w_2 , i.e. $w_1(E(i), i)$ is an MCS when $E(i) \neq \infty$.

Procedure 4 requires time of $O(n)$, where $n = |w_1| + |w_2|$. This can be proved in a similar way to Procedure 1. By using the set of MCSs of w_1 w.r.t. w_2 , $s(w_2, w_1)$ can be computed. The minimum cover problem is a famous NP-complete problem in general but the minimum S -cover can be obtained easily as described below. Following lemmas are useful.

Definition 8

For two MCSs of w_1 w.r.t. w_2 , $w_1(i_1:j_1)$ and $w_1(i_2:j_2)$, $w_1(i_1:j_1) < w_1(i_2:j_2)$ if and only if $i_1 < i_2$ and $j_1 < j_2$.

Lemma 1

For the set of MCSs of w_1 w.r.t. w_2 , the relation $<$ is a total ordering.

Proof: It is a direct consequence of the definition of an MCS(Definition 2). By Lemma 1, we suppose that $w_1(i_1:j_1) < w_1(i_2:j_2) < \dots$ for the set of MCSs, $M = \{w_1(i_1:j_1), w_1(i_2:j_2), \dots\}$

Lemma 2

An element $w_1(i_1:j_1)$ in M must be included in a minimum S -cover of w_1 .

Proof: Because the symbol $w_1(i_1)$ is not covered by any other element in M except $w_1(i_1:j_1)$.

Lemma 3

If $w_1(i_h:j_h) \in M$ is in a minimum S -cover of w_1 then there exists a minimum S -cover of w_1 that has $w_1(i_k:j_k)$ as its element, where $w_1(i_k:j_k)$ is the string in M such as $i_k - 1 \leq j_h$ and j_k is maximal for i_k .

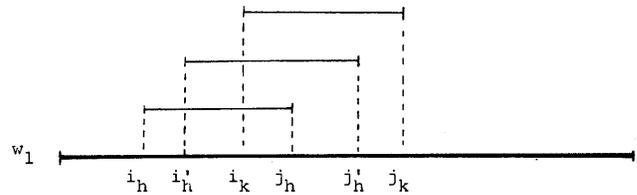


Figure 2—The situation considered in the proof of Lemma 3

Proof: Suppose that $w_1(i_h:j_h)$ is in a minimum S -cover. To cover $w_1(j_h+1)$, there exists a substring $w_1(i'_h:j'_h)$ in the minimum S -cover such as $i_h < i'_h < i_k \leq j_h + 1 \leq j'_h < j_k$ (see Figure 2). Then $w_1(i_h:j_h)$ can be covered by $w_1(i_h:j_h)$ and $w_1(i_k:j_k)$. So the S -cover obtained by replacing $w_1(i_h:j_h)$ by $w_1(i_k:j_k)$ is also the minimum S -cover of w_1 . Q.E.D.

A minimum S -cover can be obtained by Lemmas 2 and 3 in time proportional to the number of MCSs, that is $|w_1|$ at most. The set of MCSs of w_1 w.r.t. w_2 can be obtained in time proportional to $n = |w_1| + |w_2|$ (Procedure 4). Furthermore, the array E obtained in Procedure 4 arranges the MCSs by the ordering $<$. Procedure 5 shows a formal description of calculating a minimum S -cover using the resulting E of Procedure 4.

Procedure 5

Calculation of $s(w_2, w_1)$.

1. Calculate E by Procedure 4.
2. $\text{DISS} \leftarrow 0$, $\text{CAND} \leftarrow 0$, $R \leftarrow -1$, $i \leftarrow -1$ and set $M \leftarrow \phi$.
3. If $E(i) = \infty$ then go to (6).
4. If $E(i) \leq R$ then $\text{CAND} \leftarrow i$ and go to (6).
5. If $\text{CAND} = 0$ then $s(w_2, w_1) \leftarrow \infty$ and stop.
 $\text{DISS} \leftarrow \text{DISS} + 1$, $R \leftarrow \text{CAND} + 1$, $M \leftarrow M \cup \{w_1(E(\text{CAND}); \text{CAND})\}$ and $\text{CAND} \leftarrow 0$ go to (4).
6. $i \leftarrow i + 1$. If $i > |w_1|$ then go to (7) else go to (3).
7. If $\text{CAND} \neq |w_1|$ then $s(w_2, w_1) \leftarrow \infty$ and stop.
 $\text{DISS} \leftarrow \text{DISS} + 1$, $M \leftarrow M \cup \{w_1(E(\text{CAND}); \text{CAND})\}$.
 $s(w_2, w_1)$ is equal to DISS and M is the minimum S -cover of w_1 w.r.t. w_2 .

Each element of E is referred only once in Procedure 5 and Procedure 5 requires computation time of $O(n)$, where $n = |w_1| + |w_2|$. The next example shows how our procedure works.

Example 6

Consider the following two strings.

$w_1: \overset{1}{a} \overset{2}{a} \overset{3}{b} \overset{4}{d} \overset{5}{a} \overset{6}{b} \overset{7}{c} \overset{8}{a} \overset{9}{a} \overset{10}{b} \overset{11}{d} \overset{13}{c} \overset{15}{a} \overset{17}{b} \overset{19}{c} \overset{21}{b} \overset{23}{d} \overset{23}{c}$

$w_2: b d a b c b c a a b d c a b d a b$

By Procedure 4, an array E representing all MCSs in the ordering $<$ is calculated as follows:

$E \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \ 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23$
 $\infty \ \infty \ \infty \ 12 \ \infty \ 3 \ \infty \ \infty \ \infty \ \infty \ \infty \ \infty \ 6 \ \infty \ \infty \ \infty \ 13 \ \infty \ \infty \ \infty \ 17 \ 19$

By this fact, the set of MCSs of w_1 w.r.t. w_2 equals

$\{w_1(1:4), w_1(2:5), w_1(3:7), w_1(6:14), w_1(13,18), w_1(19:23)\}$.

By Lemma 2, the first substring $w_1(1:4)$ should be selected. Then we select $w_1(i_k:j_k)$ such as $i_k-1 \leq 4$ and j_k is maximal by Lemma 3. Thus $w_1(3:7)$ is selected. In this way, $\{w_1(1:4), w_1(3:7), w_1(6:14), w_1(13:18), w_1(19:23)\}$ is obtained and it is the minimum S -cover of w_1 w.r.t. w_2 . We have $s(w_2, w_1) = 5$ and $17\alpha + 11\beta$ storage space is required when w_2 is used for a reference string of w_1 . Similarly, $s(w_1, w_2) = 3$ is obtained and $23\alpha + 7\beta$ storage space is required w_1 is used as a reference string of w_2 .

Procedure 3 can be easily extended for more than two strings. An outline of the data compression procedure for more than two strings is as follows.

Procedure 6

Data compression procedure for more than two strings using MCSs.

1. Consider a set of strings $W = \{w_1, w_2, \dots, w_N\}$. Calculate $s(w_i, w_j)$ for any i and j . Consider a weighted directed graph $G(V, E, c)$, where $V = \{V_0, V_1, \dots, V_N\}$ and each V_i corresponds to w_i ($1 \leq i \leq N$), $E = \{(V_0, V_i)\} \cup \{(V_i, V_j)\}$ ($1 \leq i, j \leq N, i \neq j$) and $c((V_0, V_i)) = \alpha|w_i|$, $c((V_i, V_j)) = 2\beta(1 + s(w_i, w_j))$. This graph is called a similarity graph of W .
2. Find the minimum directed spanning forest of the above graph.
3. Encode each string according to the minimum directed spanning forest.

Procedure 6 is least redundant under our assumption stated in Section 1 but time consuming because all $N(N-1)$ combinations $s(w_i, w_j)$ are required to be calculated and finding the minimum directed spanning forest requires $O(N^4)$ computation time.

Instead of the minimum directed spanning forest, a star graph may be useful, that is, a most frequently referred-to string (say w_1) is used as a reference string of all other strings. In this case, only $N-1$ similarities $s(w_1, w_2), s(w_1, w_3), \dots, s(w_1, w_N)$ are required and the computation time is propor-

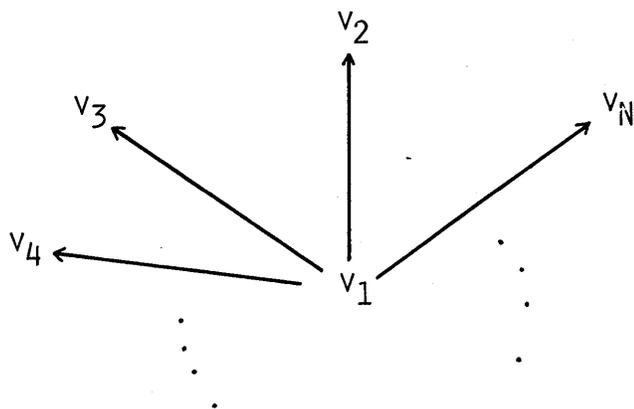


Figure 3—A star graph

tional to $O(N|w_1| + |w_2| + \dots + |w_N|)$ although it may not give the optimal result. In such a case, the reference string is the root of a star graph shown in Figure 3.

A dummy string w_0 may be used for a reference string in a star graph. Advantages of using a dummy string are (1) much storage reduction may be possible, (2) update is localized. However, finding a proper dummy string is very difficult.¹²

5. UTILIZATION OF BOTH MRSs AND MCSs

Combination of MRSs and MCSs may reduce storage space for more than two strings (The procedure for two strings presented in Section 3 cannot be improved even if we utilize MCSs). Primary processes are the same as presented in the prior section. The S -cover of w_1 w.r.t. w_2 is defined by using both MRSs of w_1 and MCSs of w_1 w.r.t. w_2 . The time complexity of this combined procedure is also proportional to the sum of lengths of two strings. Time complexity of this combined procedure for more than two strings is same as the procedure using MCSs only discussed in Section 4. Following example shows the basic idea of the combined procedure.

Example 7

Consider the following four strings.

w_1 : ^{1 2 3 4 5 6 7 8 9 10 11 12 13 14 15}
 $baaaaaaaaaab a a a b a$

w_2 : $abbbaaaaabad$

w_3 : $abbbbbbbbabba$

w_4 : $bbbabbababa$

w_1 is considered as the concatenation of $w_1(1:2)$, $w_1(3:9)$ and $w_1(10:15)$, each of which corresponds to $w_2(4:5)$, $w_1(2:8)$ and $w_2(4:9)$, respectively. $w_1(1:2)$ and $w_1(10:15)$ are MCSs of w_1 w.r.t. w_2 and $w_1(3:9)$ is an MRS of w_1 . So $s(w_2, w_1)$ is 3. w_4 is considered as the concatenation of $w_4(1:4)$, $w_4(5:7)$ and $w_4(8:11)$, each of which corresponds to $w_2(2:5)$, $w_4(2:4)$ and $w_4(6:9)$, respectively. When w_2 is used as a reference string of all other strings, other strings can be stored as follows:

$\bar{w}_1: 2, 3, 1, -2, 6, 4, 5$ $\bar{w}_3: 2, 1, 2, -2, 7, 1, 0, 3, 2$
 $\bar{w}_4: 2, 2, 3, -2, 2, -6, 3$

The first numeral 2 of \bar{w}_1 shows that w_2 is a reference string of w_1 . An MCS $w_k(i:i+j)$ is coded as i, j and an MRS $w_k(i:i+k)$ is coded $-i, j$. Thus (4,1), (-2,6) and (4,5) in w_1 represent $w_2(4,5)$, $w_1(2:8)$ and $w_2(4:9)$ respectively.

\bar{w}_3 can be also expressed as 4,4,1, -2,5,4,3 and this is shorter. When the set of strings is not updated and the decoding time is not so critical, Procedure 6 is useful. The similarity graph for these strings is presented in Figure 4. We should select an optimal coding schema by this similarity graph. For example, consider the directed spanning tree shown by bold lines in Figure 4. w_1 and w_2 are stored in their original forms and w_1 and w_3 are used as reference strings of

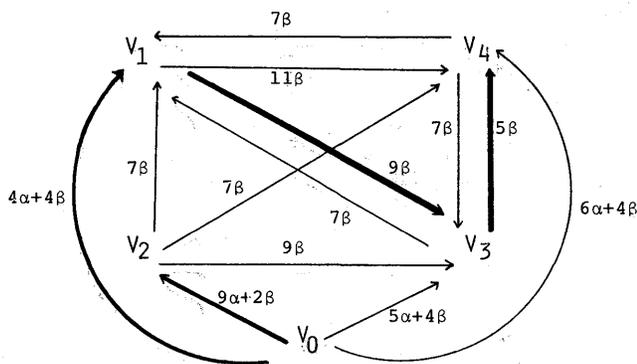


Figure 4—The similarity graph of the four strings in Example 7

w_3 and w_4 , respectively. As stated in the previous section, usually the utilization of star graphs (Figure 3) is more practical than the utilization of the minimum cost spanning trees although the latter gives the optimum results.

CONCLUDING REMARKS

There are increasing demands for generalized database systems which can handle text strings, programs, and picture data, as well as ordinary business-oriented data currently handled by commercially available database systems. These data are usually large and have some kinds of redundancy. We expect that the procedures developed in this paper are suitable to compress these kinds of data.

Further research problems are as follows:

1. Development of more efficient algorithms for large amounts of data: Although the procedures discussed in this paper are linear time algorithms, the coefficients are not small enough to be applicable to large data files. High-speed heuristic procedures which will not always produce the optimum code are also required. All the procedures presented in this paper can be easily modified to adopt such heuristic procedures.

2. Discussions on the optimum compression: Further compression is possible by adding redundant data that is used as

a reference datum. Steiner tree is known as a minimum spanning tree which utilizes some dummy nodes.¹² In many cases it is shown that this redundancy will contribute to decreasing the cost of the spanning tree.

Furthermore, our similarity does not satisfy the condition $s(w_1, w_2) + s(w_2, w_3) \geq s(w_1, w_3)$. So the generation of redundant data will reduce the total storage space.

ACKNOWLEDGMENT

The authors wish to express their thanks to Professor G. T. Toussaint of McGill University for informing them of the Steiner tree problem.

REFERENCES

1. Huffman, D.A. "A Method for the Construction of Minimum Redundancy Code", Proc. IRE, 40, 1098, 1952.
2. Rissanen, J. and Langdon, Jr., G.G. "Arithmetic Coding", IBM J. Res. Develop., vol. 23, no. 2, pp. 149-162, March 1979.
3. Wagner, R.A. "Common Phrases and Minimum Space Text Storage", CACM, vol. 16, no. 3, March 1973.
4. Ting, T.C. "Compacting Homogeneous Text for Minimizing Storage Space", International Journal of Computer and Information Sciences, vol. 6, no. 3, pp. 211-221, 1977.
5. Severence, D.G. and Lohman, G.M. "Difference Files: Their Application to the Maintenance of Large Databases", ACM Trans. on Database Systems, vol. 1, no. 3, pp. 256-263, Sept. 1976.
6. Kang, A.N.C., Lee, R.C.T., Chang, C.L. and Chang, S.K. "Storage Reduction Through Minimal Spanning Trees and Spanning Forests", IEEE Trans. on Computers, vol. C-26, no. 5, pp. 425-434, May 1977.
7. Ghosh, S.P. "File Organization: The Consecutive Retrieval Property", CACM, vol. 15, no. 8, pp. 802-808, 1972.
8. Tanaka, K., Kambayashi, Y. and Yajima, S. "Organization of Quasi-Consecutive Retrieval Files", Information Systems, vol. 4, no. 1, pp. 23-33, 1979.
9. Harary, F. *Graph Theory*, Reading, Mass.: Addison Wesley, 1969.
10. Weiner, P. "Linear Pattern Matching Algorithms", IEEE 14th Annual Symposium on Switching and Automata Theory, pp. 1-11, 1973.
11. Even, S. and Rodeh, M. "Economical Encoding of Commas Between Strings", CACM, vol. 21, no. 4, pp. 315-317, April 1978.
12. Pollak, H.O. "Some Remarks on the Steiner Problem", J. of Combinatorial Theory, Series A, pp. 278-295, 1978.

INFORMATION PROCESSING MANAGEMENT

Choosing application development tools and techniques

by V. KEVIN WHITNEY and JANE G. MORSE

*Arthur D. Little, Inc., Information Systems Section
Cambridge, Massachusetts*

ABSTRACT

Many different tools and techniques for assisting application system development are available today, but there is no good method available for choosing the proper ones to use in a particular application. This paper presents a fivefold classification of application systems and the tools and techniques most suitable for each class. Characteristics of each class of applications are explained, and methods of using this analysis to select application development tools and techniques are discussed.

INTRODUCTION

In most data processing shops today, as the maintenance workload continues to grow more burdensome, the backlog of new development projects is also increasing. Converting older production systems to newer and more maintainable technology without stopping the development of new systems means an ever increasing load on the applications development staff.

A bewildering array of tools and techniques to assist in this development and redevelopment is available today. All the following kinds of software and methodologies are designed partly or wholly to provide such assistance:

- Database managers
- Teleprocessing transaction monitors
- Structured analysis and design methods
- Librarians and precompilers
- Documentation support packages

In our work with many clients in the past few years, we have had the dubious privilege of reviewing unsuccessful development projects. A frequent problem, which could easily have been avoided in many of these cases, is the inappropriate choice of systems development tools and techniques. Complex projects are often attempted with only the simplest of tools, and sometimes simple projects are developed using overcomplex techniques. In retrospect it is easy to understand the underlying causes of a problem, but sometimes it is very difficult to recognize those same causes in a new development project.

One approach we have found useful in helping to decide which software tools and techniques are appropriate for a new development project is based on the observation that various products are designed to emphasize support for difficulties in different aspects of application development. Some tools help with display screen formatting, others with data storage and retrieval, yet others with transaction error recovery. You may not discern this from the software salesman, but a talk with the implementer of the package is certain to identify the aspects of applications development for which the product is most suitable.

Furthermore, different application systems seem to emphasize different computing functions. We have found that most application systems seem to be dominated by one of the basic computing functions: processing logic, output reporting, data management, transaction input, and task management. By classifying an organization's applications according to their dominant computing function and using that classification to choose the tools which assist most in managing that function, we can identify the application development tools that are most appropriate for an organization. This approach is simplistic, but it has proved useful in a variety of real cases where half an answer is better than none. It is interesting to note that our experience with this technique has been that it is particularly effective in giving early warning signals for situations where inappropriate tools and techniques are used and which as a result are likely to fail. In today's processing world, it seems that identifying the paths to failure is easier than identifying the paths to success.

In the following sections we describe the classification scheme we use for analyzing applications and then indicate which types of tools and techniques are most suitable for each application class. Finally, we identify some ways DP managers can apply the ideas for choosing the tools and techniques best suited to their installations.

APPLICATION SYSTEMS CLASSIFICATION

A data processing application system can usually be grouped into one of the five classes shown in Table I, based on its dominant computing function. Usually this assignment is easy. For example, small scientific application systems are

TABLE I—Characteristics of five classes of application systems*

Class	Dominant Function	Scope of Application	Number of Data Elements	Size of Typical Development Project	Main Barriers to Development Productivity
A	Processing and algorithm logic	Individual programs	10–30	1 person 3 months	Problem specification & solution verification
B	Output reporting	Collection of reports	50–300	2 people 9 months	Organizing many details of report format specification
C	Data management and structure	Applications on an integrated data base	100–400	3 people 15 months	Database design and modifications
D	Transaction input and processing	Group of related transactions	200–400	6 people 2 years	Keeping transactions self-contained and simple
E	Task management	An integrated business system	200–800	10 people 3 years	Specifying and implementing time-dependent processes

*An application system is a collection of processing modules and related data that exhibits a high degree of integration but is loosely coupled to programs outside the application system. Thus, an application system cannot easily be divided into subsystems that are designed, implemented, or operated independently of one another.

typically dominated by the processing or computing functions, and large accounting application systems are often dominated by the output reporting function. When an application seems to have two equally dominant computing functions, it is often seen to be two separate, but linked, application systems. For example, many modern systems involve a transaction input-dominated data collection portion and an output-dominated reporting portion.

No system, of course, consists entirely of a single function such as input or output. Although all five functions are found to a greater or lesser degree in all application systems, most applications can be classified according to their dominant function. When an application seems to fit two categories equally well, it is often best to assume that the application belongs to the more complex of the two categories.

Let us explore the characteristics of systems belonging to each of these five basic types and the development tools and techniques that are most particularly suited to them. Table 1 shows the characteristics of applications of each class.

- *Processing logic* is the dominant function in applications involving complex logic and computations, a small or moderate amount of data, and simple control structures. Examples of this type of application include most scientific programs and many of the analysis applications typically run on time sharing service bureaus. These applications involve only a few data elements and are normally developed in a few months.
- *Output reporting* is the dominant function for the class of applications that has extensive reporting requirements. The processing, control, and data relationships are simple. Usually these systems produce a great many different reports from essentially the same data. Applications

written using RPG and MARK IV are often of this class.

- *Data management* is the dominant computing function for the third group of application systems. These applications usually have a very stable database whose structure models the business. Data values are loaded into the database, the database is processed for updating, and reports are derived from the stored data. Applications with complex database structures are usually data-dominated.
- *Transaction input* dominates applications of the fourth type. Here data collection is the main data processing activity, and the system functions are driven by the arrival of the appropriate data. There are many types and sources of data. Data are gathered once, stored, and then used for many different types of processing. Transaction processing applications, like distribution company order entry and online teller banking, are good examples of input-dominated systems.
- *Task management* is the dominant computing function in applications that have very complex or very dynamic program structures. These applications are likely to have a very complex job control network or online control structure linking together a complex array of time-dependent processing modules. Small applications are rarely complex enough to be control-dominated. Integrated manufacturing control systems and corporate financial consolidation systems are often examples of this class.

This classification of applications by their dominant computing function is related to other application characteristics, such as their size or complexity. Although the chart is organized in a way that implies a strong correlation between size and application class, this need not always be the case. There

TABLE II—Development tools and techniques for each application class

Class	Dominant Function	System Design Aids	Programming Languages	Data Management Aids
A	Processing Logic	Decision tables Flow charts Function libraries	FORTRAN APL, PL/1 Problem-oriented languages	---
B	Output Reporting	Output-input matrix Data Dictionary Jackson methodology	COBOL (report writer) RPG, MARK IV	EASYTRIEVE Inverted DBMS
C	Data Management	Data Designer Data Dictionary	COBOL (CODASYL DML)	CODASYL DBMS
D	Transaction Input	HIPO Data Dictionary Jackson methodology	CICS/DMS COBOL	IMS/DB-DC TOTAL VSAM
E	Task Management	SADT Structured Design	PASCAL, ADA PL/1, ALGOL	Relational DBMS

are minicomputer-based applications that are task-management-dominated and super-computer applications that are processing-logic-dominated. There does seem to be a natural hierarchy of complexity among these application classes, and we will see that today's software development tools are better suited to implementing the simpler classes of applications.

As applications evolve and grow, they may change focus. Thus, when a simple payroll system dominated by its output reporting is upgraded to become part of a more general accounting system, it may need to be replaced with a data-based system. Later, as part of a more comprehensive financial management system, those data-based modules may need to be replaced with programs that are transaction input-oriented. Thus, as applications grow in complexity and sophistication within an organization, the dominant computing function of those applications may change, requiring a change in the application development methodologies used to develop subsequent generations of the system.

CHOOSING DEVELOPMENT PRODUCTIVITY AIDS

We have found that each of the classes of applications confronts the developers with different kinds of development problems to be overcome. As a result, we have found that application development tools and techniques suited to applications with one dominant computer function may not be appropriate for applications of another dominant computing function. For example, there is no need for a database management system (which is appropriate for data-dominated applications) when development is simple Fortran applications for engineering or scientific calculations. Table I shows the problems we believe characterize each class of application.

Each class of applications has a collection of development productivity aids that are most effective for solving problems associated with that application class. Some tools and techniques are useful for more than one class of application, of course; and sometimes it is difficult to identify a single class

for a particular application. But by identifying which classes of applications each tool or technique is best suited to we can simplify the decision of which to use for a new application.

Some of the tools and techniques suitable for each class of application are shown in Table II. The entries in the chart reflect our observations of some techniques that have worked well for our clients on various types of applications. It is not our intent to include all the development tools and procedures available, but only to illustrate the analysis.

For Class A *processing* applications, the main application development problem is specifying the problem and its solution in an appropriate computer language. FORTRAN, BASIC, and APL are commonly used for implementing scientific, business, and analysis applications. Higher-level, problem-oriented languages are also used in some specialized subject areas. Flow charts, decision tables, and structure diagrams are often useful in the design phase. Because Class A applications seldom have complex input, output, or data management requirements, the overhead of the more elaborate application development tools often outweighs any programmer productivity gains resulting from their use.

For Class B *output-dominated* applications, the main task is specifying the report formats and verifying that the coding to create them has been done correctly. Report writing languages like RPG, MARK IV, and the COBOL reporter writer feature are particularly helpful. The systems development methodologies based on the output-input matrix, such as the Arthur Anderson methodology,² are suitable only for output-dominated applications. In this approach, the development procedure begins by defining all the output reports and their data elements. Then the matrix showing the data inputs needed to generate these output data elements is generated. Finally, the programs based on the matrix are written. This approach works best, of course, in situations where the output reports remain relatively unchanged throughout the entire application development process. Inverted file data management systems, such as INQUIRE and ADABAS, are

often used in implementing output-dominated applications because they support good report generator packages.

For Class C *data-management-dominated* applications, the main productivity problem is designing the database correctly and modifying it effectively when changes are found to be required. DBMS software packages are, of course, the natural choice of implementation technology. Note, however, that some database management software systems are more appropriate than others for managing a complex database. The CODASYL-compliant systems with extensive backup and recovery features are particularly helpful in a stable database environment. A data dictionary, usually integrated with the DBMS, is normally essential. Unfortunately, none of the conventional systems development methodologies currently provide assistance for the important task of database design. At Arthur D. Little, Inc., we are developing DBAid³ to assist in the database design; and Data Base Design, Inc., has developed the Data Designer⁴ software for relational database design. Few other tools for improving development productivity for this class of system are generally available.

For Class D *input-dominated* applications, the chief barrier to high development productivity is the difficulty of partitioning the application into independent self-contained transactions, each with its unique inputs. Not only is the design task difficult, but most programming languages do not have special features for processing conversations of transactions or recovering from errors in partially completed transactions. Thus, teleprocessing monitors, such as CICS, are particularly helpful for implementing this application class. The application development methodologies that focus on identifying the flow of data through the elementary processes of the business work well for input-dominated applications. The Jackson methodology⁵ and IBM's HIPO⁶ are two examples of this type of application design. A data dictionary can be quite helpful, particularly if it has features that permit capturing data flow and usage parameters. DBMS packages used for implementations of this application class must provide fast access to the database, but they are not required to be able to manage complex data structures.

For Class E *task management* applications, there are few general-purpose aids for improving development productivity. Because control is the dominant computing function in this class of applications, there is usually complex time-dependent task control structure. PASCAL, PL/1, and the other block-structured programming languages and the well-developed network job control languages are best for these applications. The systems development methodologies most helpful for control-dominated applications are the top-down, iterative refinement techniques, such as SADT⁷ from Softech and Structured Design⁸ by Yourdon, which incorporate the specifications of the control structure as well as the data structure in the system design. For DBMS support, the relational systems are most likely to be helpful in providing the flexible database support for dynamic applications.

CASE STUDY

An example will help show the importance of choosing development tools appropriate to the particular class of applica-

tions. A large manufacturing company had decided to automate its order-processing services. As a rapidly growing business, it needed the system to speed paperwork processing. From the initial concept stage, the design was based on the use of online transactions to automate as much of the order-processing task as possible. It was a big project and required hundreds of data elements and many years of development effort.

The data processing department tackled the project, using tools and techniques that seemed appropriate. A CODASYL data management system was installed. An output-input matrix system development methodology was used to specify the application characteristics, and CICS was used to bring the system on line quickly. The system turned out to be much more elaborate than anticipated, with cost and time overruns as well as poor operational performance.

What went wrong? In this case, the application was clearly a transaction input-dominated (Class D) system. But the DBMS software used for implementation was the more structured type, suitable for stable, complex data-dominated problems. And the application development methodology started the design process by cataloging the output reports and their data elements, then building the system design on them. That resulted in a more complete and complex system than initially expected. It also forced the project into a monolithic design, which prevented an incremental, successively refined implementation. Worse than that, because the analyst's attention was focused on the output reports, the underlying business processes that were to be automated were never clearly understood and incorporated in the design. As a result, the implemented system did not fit well into the operational business practices and was eventually abandoned.

Could these problems have been avoided? In retrospect, we see that they could certainly have been anticipated. The application cried out for a transaction-oriented system, but the tools used were more appropriate for output reporting and data management projects. The rigid, output-dominated, design methodology used was certain to produce a system that would be inappropriate and difficult to modify. The database management system chosen for the implementation stressed an integrated design, rather than an incrementally modifiable and expandable system design. Using our method of choosing software tools, the data processing manager would have realized that this was a Class D system and used a more appropriate design methodology, such as the Jackson method, and a more suitable database management system than the CODASYL DBMS that was chosen.

WHAT'S A DP MANAGER TO DO?

Now that we have defined five basic application classes and described the development tools best suited to each class, let's outline a simple program to check an installation's tools and techniques for suitability to its application needs. It is really very straightforward.

First, the existing applications and those proposed or under development should be categorized in the five classes described in Table I. Normally, older applications will fall into classes dominated by processing logic and output reporting.

TABLE III—Techniques used for applications at D.P. Installation X

Application	Development Date	Application Class	Techniques Used Best Suited to This Class	Techniques Used Best Suited to Other Classes	Score
General ledger	1965	B	COBOL	—	+1
Plant materials control system	1970	B	RPG	—	+1
Order entry	1975	D	CICS	O-I Matrix CODASYL DBMS	-2
Engineering analysis	1980	A	FORTRAN CMS	—	+2
Distribution inventory	In development	D	CICS/DMS	O-I Matrix CODASYL DBMS	-2
Budgeting	In development	A	FOCUS	Data Dictionary CODASYL DBMS	-2

NOTE. Score is count of matching techniques less not-matching techniques.

Newer applications are more likely to fall into the more complex classes. In classifying the proposed applications, try to understand the class of the underlying business problem, rather than the proposed implementation technology.

Second, prepare a version of Table II showing the application development tools in use or potentially available at your organization. Don't be alarmed if the matrix is rather sparsely filled in; that is typical of many DP shops. It is often helpful to include tools and techniques the staff is interested in and to exclude those the staff is not interested in learning to use. Highlight tools that are current installation standards.

Then review the use of application development tools and techniques at your organization on an application-by-application basis. This can be organized as shown in the example of Table III. This analysis is interesting for all applications, but most important for new or planned applications. If the tools available and in use do not match the needs of the applications planned, then a potentially serious problem exists. Immediate action to install more appropriate application development tools and techniques is needed; this may require staff training, outside consultants, and new software or documentation.

Mismatches identified for applications currently under development are the most critical. Both of the two situations—simple needs/sophisticated tools and complex needs/simple tools—can lead to project failure. In the first case, the technological overkill can lead to overelaborate systems that are harder to develop and more complex to use. Very strong management control is needed to keep the systems being developed suitable for the actual need. In the second case, inadequate tools make project success technologically much more difficult. Either case requires prompt attention.

Another method of showing the overall match of techniques and tools to the characteristics of an organization's

systems is the matrix shown in Figure 1. This figure is derived from Table III as follows: For each application system in Table III, we place numbers or application names in the array in the cells corresponding to the class of the application and the class of the techniques used in developing that application system. A concentration of applications along the diagonal indicates that the tools in use are appropriate to the applications. If the applications are clustered above the diagonal, the

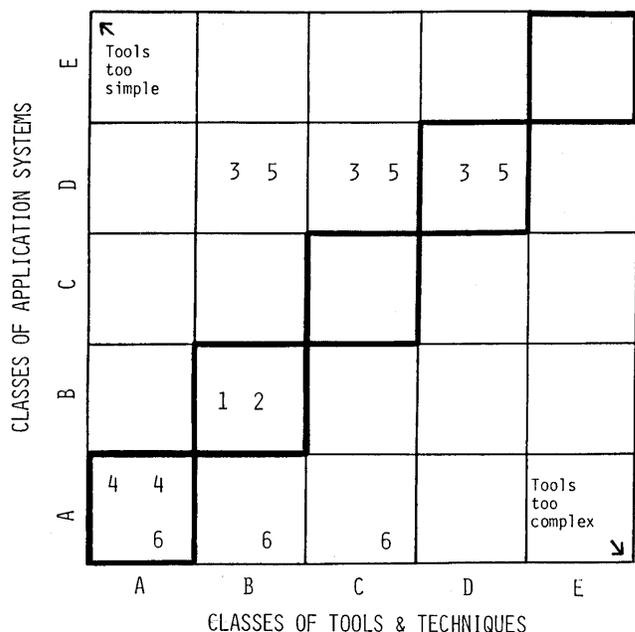


Figure 1—Array showing match of development tools and techniques to application systems

tools in place are likely to be inadequate; and if the focus is below the diagonal, the tools may be getting in the way of development. This array is useful in portraying overall trends in an organization's use of software development tools and techniques.

CONCLUSIONS

We hope this article has illustrated a simple but useful framework for choosing the proper systems development tools for the five classes of applications. Naturally, it will not fit all situations; but even a simple method is often better than none. No experienced manager expects to use the same project management techniques for both a three-person-month project and for a 300-person-month project. Similarly, we hope we have shown that the techniques that work best for an output-dominated application project are likely to differ from those that are best for an input-dominated application and that neither of these is likely to be most suitable for a large task control structure-dominated application.

This framework is also useful in selecting installation standards. There is value in limiting the number of tools in any one installation so that the staff can become familiar with them and so that starting a project does not require learning a new tool. If the standards groups review the kinds of applications in place and planned for the installation, they can favor tools and techniques that are suited to these kinds of applications. As the nature of the applications changes over time, it may be necessary to review the standard tools and techniques in use at the installation.

There is also a lesson here for career development paths of programmers and analysts. Professional growth must include learning when to use the various systems development tools and techniques as well as how to use them. What distinguishes the top analysts from the novices is not necessarily the proficiency they have with a few development techniques, but the

variety of techniques they can use effectively when required. The tools a novice will need for assignments differ from those of the expert. A good career development plan should provide exposure to a variety of tools, adding new ones when they are needed.

COBOL failed to be the universal programming language suitable for all kinds of applications that was sought 20 years ago. Similarly, there is no universal DBMS, teleprocessing task monitor, screen formatter, or system development methodology suitable for all classes of applications. But identifying the dominant computing function of an application can help insure a good choice of application development tools and techniques and thereby improve the probability of project successes. Naturally, the standards in place in an installation should be suited to the applications running there; and its career development program should provide staff experience with the several tools and techniques needed for its own classes of applications.

REFERENCES

1. *EDP and Systems Development: Some Management Issues*, Arthur D. Little, Inc., Cambridge, MA, 02138, 1980.
2. *Automation Concept for Business Information Systems*, Arthur Anderson & Co., Chicago, Illinois, 1974.
3. Curtice, R.M. and Jones, P.E., Jr., "Key Steps in the Logical Design of Data Bases," *Proceedings of the NYU Symposium on Database Design*, New York University, May 1978.
4. Holland, R.H., "Developing User Views," *Datamation*, Vol 26, No. 2, February 1980.
5. Jackson, M.A., *Principles of Program Design*, Academic Press, New York, New York, 1975.
6. *HIPO-A Design Aid and Documentation Technique*, GC20-1851, IBM Corporation, 1976.
7. *An Introduction to SADT™ Structured Analysis and Design Technique*, 9022-78R, SofTech., Waltham, MA, 02154, 1976.
8. Yourdan, E. and Constantine, L.L., *Structured Design*, Yourdan, Inc., New York, 1975.

A software requirements analysis and definition methodology for business data processing

by ISAO MIYAMOTO and RAYMOND T. YEH

University of Maryland
College Park, Maryland

ABSTRACT

A well-defined set of work breakdown structures for requirements analysis and definition activities, three levels of system modeling techniques, and the application methodologies are proposed. The experimental result in using the proposed requirements-engineering methodology is introduced and shows remarkable improvement in the quality of requirements documents. The methodology was originally developed for the business data processings.

THE REQUIREMENTS PROBLEM

The high cost of software has now reached an alarming level. Unless major breakthroughs are made, the data processing industry is headed towards becoming the most labor intensive one as more and more of its labor force will be tied up to maintaining old, ill-structured, and difficult-to-modify software.

Although there are many reasons for the high cost of software maintenance, lack of thorough attention to the early phase of software development is a major reason. For example, in two large command and control systems, the software had to be rewritten 67% and 95% respectively after delivery because of mismatches to user requirements.³ There are also many examples of total cancellation of projects due to lack of appropriate requirements and feasibility analysis. In general, it was found that design errors range from 36% to 74% of the total error count.⁹ Also the cost of fixing a design error is 1.5 to 3 times more expensive than to fix an implementation error.

The above discussion illustrates the importance of developing good requirement methodology as a means to control the maintenance cost. But there are other equally important reasons for paying more attention to the requirements phase, such as design validation, communication, operations control, etc. Therefore, a bad requirements document can severely compromise the design because of difficulty in validation, and that top-down design may not be possible. Furthermore, both the customer and management may lose control.

Although there are a number of approaches to requirements analysis,^{6,10} most techniques are inadequate. One of the major problems in current approaches is the lack of a set of

well-defined work breakdown structures that can be used as a systematic guide in using the methodology.

In this paper, we propose a requirements analysis and definition methodology for business data processing systems with a well-defined work breakdown structure. We have heavily borrowed notations from existing approaches.

REQUIREMENTS ENGINEERING METHODOLOGY

The requirements engineering methodology we propose here has three major components: a well-defined work breakdown structure; a validation methodology; and a set of modeling formalisms for each phase of requirements activities.

A framework of the work breakdown structures is illustrated in Figure 1 as a phase plan with objectives, inputs and outputs of each phase. The validation methodology for the requirements documents consists of a reviewing technique for customer, developer, and analyser, a well-defined set of checklists for consistency, completeness, and feasibility, a cause-effect graph checking technique for the requirements described in the form of natural language texts; and a flow-path predicates checking technique for the flow-oriented representations of requirements. The modeling formalisms consist of an abstract task modeling for the existing system in the enterprise, a conceptual system modeling for alternative ideas of future systems, and a requirements flow modeling for detailed level representation of the chosen alternative.

In this paper, the emphasis is placed on the work breakdown structures and the modeling but not on the validation methodology.

WORK BREAKDOWN STRUCTURES

In this section, a brief description of the requirements analysis and definition (RA&D) procedures is given. The items to be contained in each output document of each phase are given in the checklists shown in the figures. For the validation or requirements documents, another set of precise checklists is prepared.

The work breakdown structures developed here are motivated by ideas proposed by Boehm,⁴ and consist of three phased processes. These are the Investigation Phase, the Con-

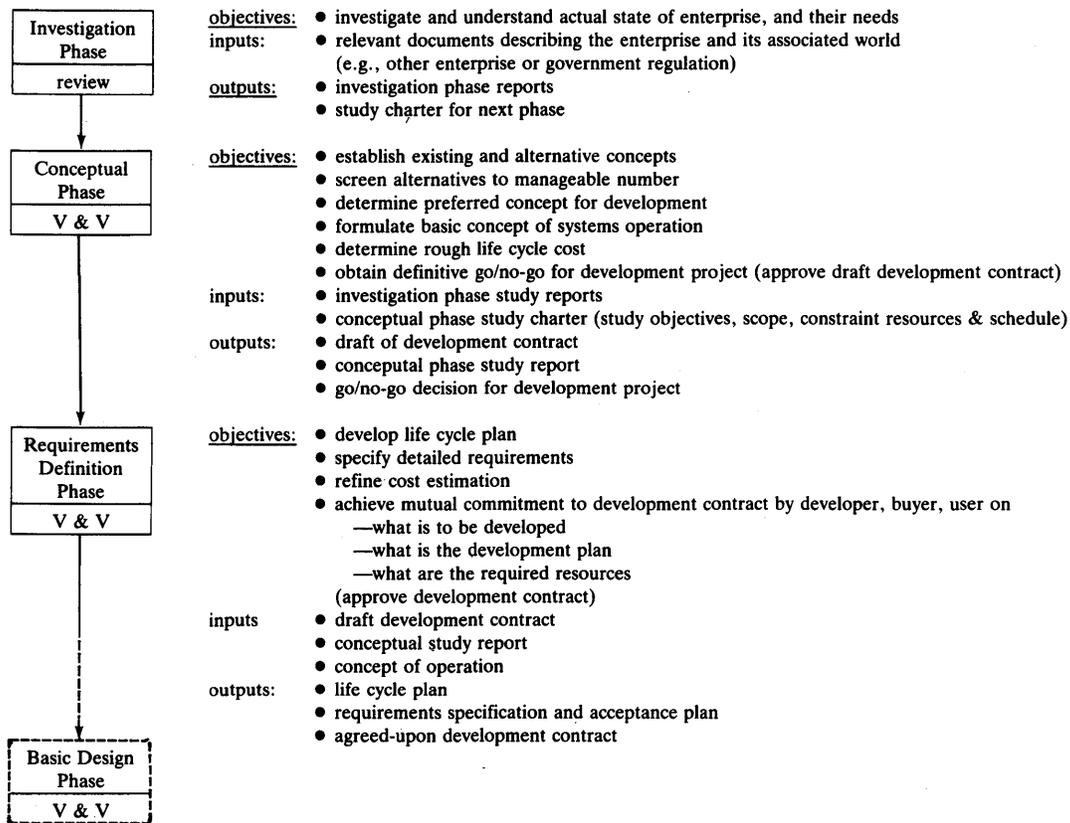


Figure 1—Proposed RA&D phase plan

ceptual Phase, and the Requirements Definition Phase. Essentially, the input information for these comes from real world environment of an enterprise that needs an improved information processing system. And outputs of these are the life-cycle plan of the system and the requirements specification, from which we can derive software designs (e.g., code design, output design, input design, file/database design, process design, reliability design, security design, design of operation, recovery design, etc.), and some aspects of hardware system design or selection (e.g., users' terminal devices, communication network, computing devices, peripheral devices, etc.). From the outputs of RA&D, all of these design or selection processes must be smoothly derived.

Investigation Phase

This is the front-end stage of the RA&D activities, and contains two sets of procedural steps. This phase is primarily information collection. Its goal is to bound the problem space and develop an overall logical model of the systems environment. The study results are to be written as two separate reports.

The inputs of this phase are the documents describing the object enterprise and the associated industrial world. If nothing of this sort is available, an investigation should be done mainly by interviewing. The outputs are the two investigation reports and the study charter for next phase.

Step I: Environmental investigation

This step is mainly achieved by documentation reviews. The proposed substeps are the following:

1. To investigate the characteristics of the industrial world in which the enterprise belongs, and the general and historical background of the enterprise.
2. To investigate the organizational macrostructure of the enterprise, such as staff divisions, production-line divisions, their geographical distributions, types of their business, associated suborganizations, etc.

By investigating the organizational macrostructure of the enterprise, we gain understanding of the business information flows and of the flows of materials and products associated with the information processing system.

3. To investigate the laws, regulations, or policies applied to the enterprise or the associated industrial world by the government or other organization.

These regulations might become external constraints for an information processing system design.

4. To investigate the current state of the associated market, and a share or competitive power of the enterprise.

The information obtained will help to estimate the future growth and direction of the enterprise and its implication on the information system design.

These results may contribute to deciding the objectives and scope of the new information processing system, and to deter-

1. Summary of report
2. Characteristics of associated industrial world
3. Historical and general backgrounds, and macro-structure of enterprise
4. Associated laws, regulations, and policies
5. Current state of associated market, share or competitive power of enterprise
6. Comments and references

Figure 2—The checklist for the environmental investigation report

mining the interviewing activities to assess users' hopes or requirements for new system or ideas to improve or solve current issues on an existing information processing system. The checklist for the items to be contained in the report produced by this step is shown in Figure 2. The report is described using natural language.

The documentation review is quite efficient. However, the investigation results should be validated by feeding back the results to the information sources (users and customers) and by detecting and correcting misunderstandings, incorrect items, ambiguities, and missing items.

Step II: Study of enterprise

The techniques used in this step mainly consist of interviews, questionnaires, and documentation reviews.

Based on the external investigation report, the interview of all levels of users or customers (i.e. management, clerk, operator, etc.) should be done to make clear and identify enterprise's objectives (e.g., to improve company profit and morale), system-internal objectives (e.g., to improve payroll operational efficiency and reliability or, to accommodate 100% growth in personnel), external constraints (e.g., to comply with the government's new tax reporting rule), external alternatives (e.g., to develop a personnel information system, to reorganize company financial organization, or to do nothing), internal alternatives (e.g., to develop new payroll program, or to buy a payroll program), and internal constraints (e.g., to perform payroll calculation in 128KB computer, or to represent files in particular form).

The major output is the report describing the macro-structure of the enterprise and a set of information about currently existing information processing systems in the enterprise.

The substeps consist of the following:

1. Defining the scope and objectives of the investigation, and planning the investigation activities (e.g. interviews, etc.);
2. Investigating and analyzing the enterprise's objectives and goals, i.e. external objectives of the information processing system (these may contribute to deciding system capabilities and their priorities during the development, and to estimating the benefits to be expected by the new system);
3. Investigating the organizational microstructures (sec-

- tions, departments, divisions, etc.) in the enterprise in detail and the types of business of each suborganization;
4. Investigating the management policies for operations, accounting, and information-processing systems of the enterprise as external constraints (these may provide some constraints on the installation, operation, and maintenance of the new system);
5. Investigating the inputs, outputs, and the resources used the enterprise's activities (the types, quantities and some characteristics of them should be analyzed);
6. Analyzing the business operations (physical operation for production or business, and the information process) and their objectives (internal objectives for information processing system) (the events, entities, and relationships among events and entities, and flows of physical operations should be made clear);
7. Investigating and analyzing the information-processing system from the viewpoint of current states of input information, processings (abstract task modeling), files, output information, and codes;
8. Analyzing the problems found in the currently existing information processing system;
9. Surveying ideas to solve or improve the problems and the enterprise's desires for new capabilities to be delivered by the information processing system;
10. Reviewing the above results and exploring incorrect items, misunderstandings, and ambiguities (usually an iterative process).

The items to be contained in the report produced in this step, given in Figure 3, are usually described using special forms with natural language texts, and partially described in the forms of abstract task models to be introduced later. Another output of this step is a study charter for the next phase. Through the investigation of the enterprise, the various types of issues regarding information processing might be recognized, and in principle the problems identified should be solved by the new system.

By the interview technique, we can easily pick up "whats" and "whys" from interviewees (users and customer), but the

1. Summary of report
2. Scope and objectives of investigation
3. Objectives and goals of enterprise
4. Micro-structure of enterprise, and types of business done in each sub-organization
5. Management policies in enterprise
6. Inputs, output products and resources used in the activities of enterprise
7. Physical operations for production or business, and their objectives
8. Input information, tasks, files, output information, input/output timing, and code information of currently existing information processing system
9. Issues of current information processing system
10. Ideas of improvement or solution of problems, and desires for new capabilities
11. References and glossary

Figure 3—The checklist for the internal investigation report

interview results may contain some bias of both of interviewers and interviewees. Thus the validation of the interview results is very important. To validate and correct the misunderstandings, incorrect information, ambiguous information, and inconsistent information. This can be done, for example, by repeating the interviews and feeding back the preliminary results to all levels of interviewees. Needless to say, this is a time-consuming process.

Conceptual Phase

In this second phase of RA&D, a logical model of the enterprise system concepts is established.

From the results of the investigation phase, it should be clear that the structure of the enterprise and the types of business done in each of the individual suborganizations are often distorted by the internal or external environments or by the power struggle among different suborganizations. Also, additional (nonessential) information is manipulated by the specific method or equipment used, because of specialities of the method or equipment.

Therefore, the basis of the system's structure should only consider the capabilities and information critically required for the enterprise's information processing.

In this phase, a conceptual model of the new system is to be constructed, and relationship among the input, file or database, task, and output information to be used in the new system, as well as the components in the enterprise that create or use these pieces of information, should be made clear. The basic approach is as follows. First, based on the individual task and system-level task diagrams, the tasks to be computerized should be determined. During this investigation, by paying attention to the information flow in the organization the relations between input information and output information, and their transformation processes, should be defined for each task. Next, the components that create the system's input information and the components that use the system's output information are examined. Then, interfaces between the environment and computerized tasks should be made clear. Then the input information required to generate the output information is investigated, and sometimes a lack of information might be found. In this case, a computer file is called for. By referring to the documents files and master files used in currently existing information-processing system, the necessary computer files might be determined. A daily accounting file and a journal file for keeping records of input information and data processed should be also considered, for example. These are to be done at the first step of the conceptual phase.

The best choice in this phase is to select a good solution among a wide range of alternatives, and to establish the feasible logical concepts of the system and its operation.

The inputs are the investigation phase study reports, conceptual phase study charter, and various ideas for new systems that come from users, customers, developers, operators, managers, or similar systems. The outputs are the draft of development contract, the conceptual phase study report, and the go/no-go decision for development project.

The substeps detailed in this phase are the following:

1. to construct a system diagram of the desired capabilities that are to be computerized by using a conceptual modeling technique (Conceptual System Modeling);
2. to list major objectives, alternatives, environmental constraints, and to refine them;
3. to characterize the likely system workload;
4. to define systems evaluation criteria;
5. to specify desired and acceptable levels (values) for each criterion;
6. to prepare models of cost-effectiveness, performance, and life-cycle cost for each alternative;
7. to screen alternatives using a screening matrix;
8. to pick the best alternative and investigate its feasibility (if infeasible, try others till all are exhausted);
9. to expand the systems architecture, concept of operation, and life cycle plan of the best alternative;
10. to validate the feasibility;
11. to develop a detailed plan for the next phase; and
12. to review and baseline results, to write a draft of the development contract, and to get a solid commitment to proceed.

The items to be contained in the report produced during this step may be checked by referring to Figure 4. The system concept of the best alternative idea should be described by using conceptual system modeling techniques to be described later.

The procedure itself contains some aspects of validation, feasibility analysis, and screening analysis.

The selection of the best alternative idea of the system concepts is done by screening analysis. Take, for example, the following evaluation criteria:⁴

- a. cost criteria (dollars of acquisition, dollars of operation, schedule, and key personnel)
- b. effectiveness of functions (functional capabilities, throughput, response time, accuracy, ease of use, ease of maintenance, staff morale and growth, sales potential, reputation, and side effects)
- c. risks (technology risk, availability/reliability, controllability, and others).

Additional evaluation criteria can be added according to the system objectives. And according to the importance level of criteria (for example, unimportant, optional, important, and critical), all of the alternative ideas might be rated as unacceptable, marginal, acceptable, and strong.

In order to establish the feasibility of alternative ideas, workload and environment characteristics, acquisition and operational costs, and all critical-importance criteria must be investigated in more detail. Then we should validate the feasibility in terms of responsiveness to all objectives, preference to external alternatives, sensitivity to assumptions, and compatibility with other initiatives.

Requirements Definition Phase

Based on the system concepts resulting from the former phase, the detailed requirements for the system's functions,

1. Summary of content
2. Background and a charter
3. System objective
4. Systems alternatives
5. Screening analysis and screening matrix⁹
6. Analysis of cost/effectiveness and validation
7. System concept of the best alternative in terms of basic architecture, concept of operation, and life-cycle plan
8. Feasibility analysis and validation
9. Decision of go or no-go
10. Draft of development contract
11. Detailed plan for next phase
12. Comments, recommendations, references, and glossary

Figure 4—The checklist for the conceptual phase report

desired properties of the system, and desired constraints are specified. Basic questions (such as: what is to be developed, what is the development plan, and what are the required resources?) must be answered by the end of this phase.

The inputs to the requirements definition phase are the draft development contract, conceptual study report, and the concept of operation; the outputs are the life cycle plan, requirements specification, an acceptance plan, and the agreement upon the development contract.

The major purpose of this phase is to produce and specify a well-organized set of requirements and constraints for development and operation. In addition to the purely user-originated requirements, the detailed requirements are to be created by analysts, developers, or managements, and their reasonableness should be discussed. The reason is to produce a complete requirements specification, which covers not only functional aspects of system but also the various properties of system and some constraints on development and operation, and from which we can derive smoothly the various types of system designs.¹⁰ The procedures are the following:

1. Based on the conceptual study report, to expand the basic architecture and concept of operation, and then to construct the refined abstract system model (Requirements Flow Representations).
2. To describe the properties of the system model. In detail, this is done through the following steps:
 - a. identifying all of the entities in the system and its environment;
 - b. specifying inputs to the system and functional response expected;
 - c. specifying system performance parameters and allowable value ranges;
 - d. specifying properties of the system (e.g. reliability, availability, maintainability, human engineering, etc.).
3. To specify requirements for inputs and outputs, terminals, code system; communication line, network, processing devices; system operations; recovery processing; and files and database. For each, functional, performance, and reliability aspects must be specified.
4. To verify the system model with respect to completeness, consistency, and correctness. One must review the system requirements by using checklists, verify the consistency by using traceability, verify the consis-

5. If the model is invalid, to remodel it by collecting additional information from users, customers, developers, or similar systems.
6. To validate the feasibility of the abstract system model (a) by non-real time simulation and (b) by taking into account technical risks, cost/schedule risks, and environmental risks.
7. To repeat above steps until a complete model is obtained.

The items to be contained in the specification are shown in Figure 5, and the document may contain the natural language texts and graphical representations. To the flow representation of requirements should be applied the modeling technique to be described next.

The requirements V & V techniques used in this methodology are:

1. review by customer, developer, analyst and manager,
2. checklists (e.g. consistency checking, completeness checking, etc.),
3. (if there is any control flow-oriented representation) flow path predicates checking,
4. cause-and-effect graph checking for natural language texts, and
5. decision table checking.

Some of the examples of completeness checklists are

- Are there any elements of requirements not stated?
- Are there any TBDs?
- Are decision criteria missing?
- Is interface missing?

1. Summary of contents
2. Overview of system in terms of objectives and environment of system
3. Functional requirements to system in terms of organization of functions, description of each function, flow representation (data-oriented or control-oriented), and test provisions
4. Performance requirements to system in terms of timings, resource utilization, accuracy, acceptable limits of degraded system performance, and validation points
5. Requirements to operations of system starting, normal operation, termination, and recovery processing
6. Interface requirements in terms of interfaces with external systems, and human interfaces
7. Requirements to database or file in terms of organization, capacity, and medium
8. Requirements to inputs, output, terminals, communication line, network, and computing devices
9. Requirement to code system
10. Developmental constraints on operational environment (i.e. hardware, software, data, man-machine interface), implementation (i.e. virtual machine, languages, operating system, DBMS), required documentations, development or maintenance cost, and end-product quality (e.g., reliability, availability, maintainability, portability, etc.)
11. Definitions and references

Figure 5—The checklist for the requirements specification

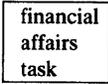
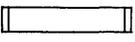
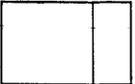
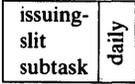
system/task	symbol	meaning	example
for SYSTEM (CF-AF) TASK DIAGRAM		information flow among tasks and subtasks within organization	order information 
		information flow between internal and external of organization	invoice 
		task or subtask	financial affairs task 
		associated section (external of organization)	customer 
for TASK (CF-AF) DIAGRAMS (in addition to above)		interfacing task	management & planning task 
		task or subtask with processing cycle	issuing-slit subtask daily 

Figure 6—Graphical notations for task/system task diagrams

- Is processing cycle for task missing?
- Are the recovery requirements missing?

By using the above techniques, we should validate the overall set of requirements in terms of completeness, consistency, testability, feasibility, compatibility and planned resources, off-nominal performance, and responsiveness to quality goals. The individual requirements we should check in terms of traceability to system objectives, testability, feasibility, completeness, and clear distinction between goals and requirements.

MODELING TECHNIQUES

There are three levels of modeling techniques in the methodology proposed here. The abstract task modeling, conceptual system modeling, and requirements flow representation are to be used in investigation phase, conceptual phase and requirements definition phase, respectively. An important point is that these models consist of only a part of the activities.

Abstract Task Modeling

Every information-processing activity in the enterprise should be modeled. A task is an abstract entity that performs (computerized or manual) information processings in the enterprise, and in many cases, it may correspond to the role of each component in the information processing in a rough sense. During the modeling, the following questions should be explored.

- What is processed in the task?
- What are inputs and outputs of the task?

- Where are they located?
- What is the processing method used?
- What is the mechanism or equipment used?
- What is the validity checking method for output produced?
- What is the degree of necessity?
- What are the exception handling method and frequencies?
- What are the processing timings?

Task-Name: "Order-Processing"

This task has following three subtasks.

Subtask-Name: "Receiving-Order&Price-Estimation"

description: "This subtask receives orders, inquiry for price information, and request for price estimation from customers on demand base. For every order, this subtask should check credit limit of each customer and price of product (by getting information from Financial Affairs Task), then forward the order information to Arrangement Request/Order subtask. For the inquiries, because it requires rapid processing, this subtask should contact with and request to Production Planning & Management Task. For the request for price estimation, according to the request, this subtask must get the information of product specification, production cost estimation, and product development schedule from Production Planning & Management Task (and indirectly from Design/Technical Management Task). Then based on this information, this subtask should estimate a sale price and a date of delivery, and must return this information to the customer."

Subtask-Name: "Arrangement Request/Order"

description: "This subtask should

Figure 7—Example of task/subtask description

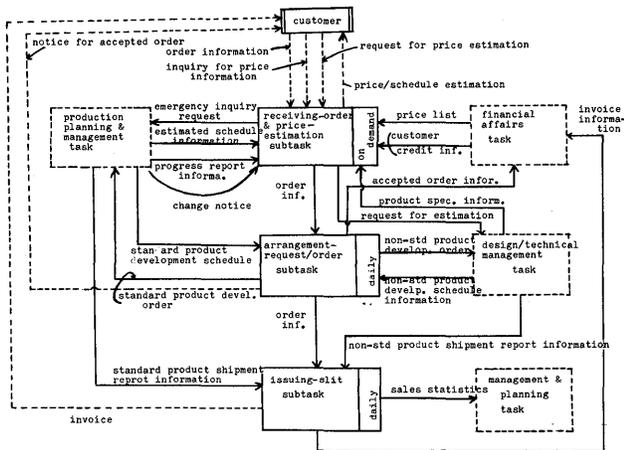


Figure 8—Task diagram "order-processing"

The results should be represented as a list of tasks, task diagrams, and system task diagram. For the diagrams, graphical notations are prepared and shown in Figure 6.

As a modeling strategy, any of the strategies such as top-down, bottom-up, outside-in, inside-out or their combination can be used.

For example, in a case of topdown strategy, the following modeling procedures can be applied:

1. find and list every information processing task in the enterprise;
2. list every input and output information of each task; and
3. represent the relationships and the information flows among tasks by using a modeling notation.

Note that above set of procedures should be applied to macro-level organizations (e.g., divisions) in the enterprise, then repeated to the next lower level (e.g., departments), and the next, until the lowest level is reached (e.g., sections or groups).

The examples of task modelling for the order processing and inventory control system are described in Figures 7 and 8.

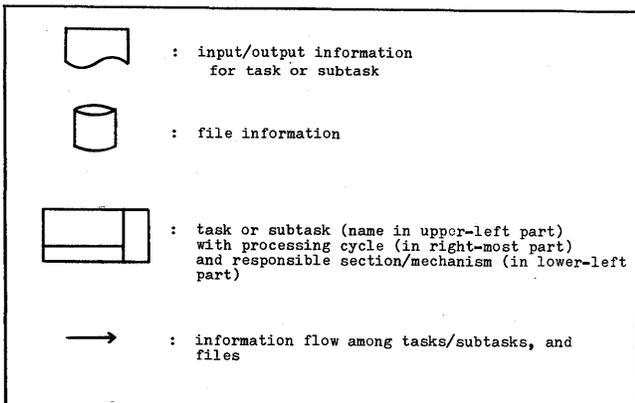


Figure 9—Conceptual modeling notations

Conceptual System Modeling

In the conceptual phase, there are two sets of procedures to be done. One is to construct submodels for each task, and another is to build a system model. The subsystem modeling procedures are to

1. determine tasks to be computerized,
2. define output information from each task or subtask to be computerized,
3. define input information for each task or subtask to be computerized,
4. determine computer files, and
5. represent the above results as a set of system submodels by using a modeling notation.

The system modeling procedures are to

1. represent each submodel of each task as the blackbox, and abstract the input information, the output information, and the file information,
2. by using abstracted information as the junction points (or interfaces), combine the abstracted submodel with another abstracted submodel,
3. repeat the combining job until all submodels are combined as a system model, and
4. represent system model by using the modeling notation.

The modeling notation is given in Figure 9 and the descriptions for the previous example system are given in Figures 10 and 11. The conceptual system modeling is based on a bottom-up approach.

Requirements Flow Representation

In order to smoothly understand and design a system based on the requirements specification, clearly some-type of flow representation for the requirements will be helpful. Therefore the requirements engineering methodology has applied three

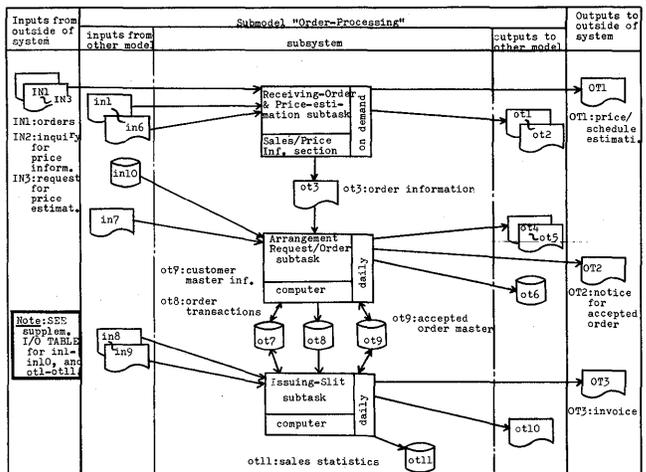


Figure 10—An example of a subsystem model

inputs			outputs		
inputs from outside of system	input from other submodels	input from files	output to file subtask within this submodel	output to other submodels	output to outside of system
IN1=orders	in1=progress report inf.	in10=standard product development schedule information	ot3=order information	ot1=request for estimation	OT1=price/schedule estimation
IN2=inquiry for price information	in2=product specification		ot7=customer master information	ot2=emergency inquiry request	OT2=notice for accepted order
IN3=request for estimation	in3=estimated schedule information		ot8=order transactions	ot6=standard product development order	OT3=invoice
	in4=change notice		ot9=accepted order master	ot4=non-standard product development order	
	in5=credit limit		ot11=sales statistics	ot5=accepted order information	
	in6=price list			ot10=invoice information	
	in7=non-standard product development schedule inf.				
	in9=non-standard product shipment report information				

Figure 11—Supplemental I/O table

levels of modeling, and the requirements flow representation technique we propose contains the integrated representation technique that combines data-flow oriented modeling and control flow oriented modeling. Since any kind of computer system and software has both data-oriented and control-oriented characteristics, it is unnatural to represent a system by using only one of two aspects. Therefore, it would be desirable to use both types of primitives. If a systems function can be represented as a series of transformations of sequential data (e.g. sequential file, printer outputs, communication message data), a data flow representation can be used. If a control sequence of processing has a more important role in the systems function, the control flow representation will be the better notation to use.

The modelling primitives concepts used in our methodology are mostly borrowed from existing approaches (e.g., Bell's⁷), as is shown in Figure 12.

As an interprocess communication mechanism, we used the concept of message buffer. This concept has the following restrictions on its use:

1. for each message buffer, only one producer process that sends data to message buffer, and only one consumer process that receives data from message buffer are allowed;
2. each process has no means of knowing the states of

message buffer (i.e. "empty" or "full"). They may have only send and receive operations. (If a process is accessed to an "empty" message buffer to receive data from, or a "full" message buffer to send data to, the process is always made to wait.)

By these, the system functions can be defined deterministically, independently from a scheduling of processes.

The following rules should be observed when one is using a message buffer:⁸

1. If a process C of data flow representation is decomposed into data flow representation, the message buffer used in C may be used by only one of the internal subprocesses of C. The usage in subprocess should follow the usage of process C. (E.g., if C sends data to the message buffer, the subprocess can only do same thing.)
2. If a process C of data flow representation is decomposed into control flow representations, the message buffer used in C may be used by any of the internal subprocesses of C. The usage in subprocesses must follow the usage in process C.
3. If a process C of control flow representation is decomposed into data flow representation, the process called by process C may be accessed by only one of the internal subprocesses of C.
4. If a process C of control flow representation is decom-

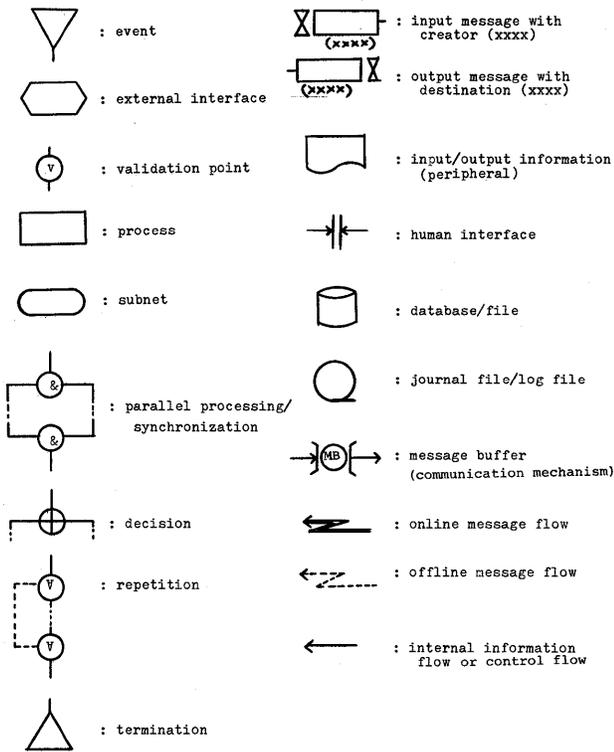


Figure 12—Primitive concepts for requirements flow representations^{1,2}

posed into control flow representation, the process called by process C may be accessed by any of the internal subprocesses of C.

Note that rules 1 and 3 are the restrictions to define system functions deterministically. If there are no parallel operations and no processes that keep internal states, then rule 3 can be removed.

The examples shown in Figure 13 and Figure 14 are the case in which the component of data flow representation is decomposed into control flow representation.

CONCLUDING REMARKS

The proposed requirements-engineering methodology has been used experimentally in the real world environment. The object system is not a business-oriented system but a control-oriented military system.⁵ Though the methodology was developed originally for business systems, the results of this use in real-time control system environment shows that the methodology has a wide range of application areas, and is not limited to business systems.

To summarize the experimental results, we achieved following things:

1. Through the experimental use, we achieved a great improvement of the quality of requirements documents. Because of the detailed work breakdown structures and validation techniques, the understandability, feasibility, and some other characteristics of quality of specifica-

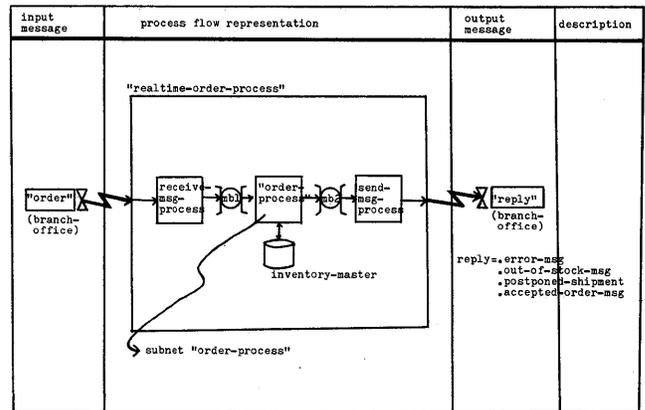


Figure 13—Example of data flow-oriented requirements representation (top level)

tions are improved. Actually, the rate of requirements problems per page has decreased definitely from 0.801 (= 185 errors/231 pages) of traditional methodology to 0.301 (= 205 errors/682 pages) of new requirements-engineering methodology. Figure 15 shows the requirements error distributions of traditional requirements methodology, and of the new requirements-engineering methodology proposed.

The most important point is the improvement in the understandability of documents because of the remarkable reductions of unclear descriptions and of missing descriptions. On the other hand, the relative percentages of incorrect and inconsistent/incompatible descriptions have increased. The reason for this phenomenon may be that due to the reduction of ambiguous descriptions and improved traceability of the specifications, the checkings for correctness and inconsistency became easier than before. Hence, validation can be done properly. As a result, more errors of these types were detected.

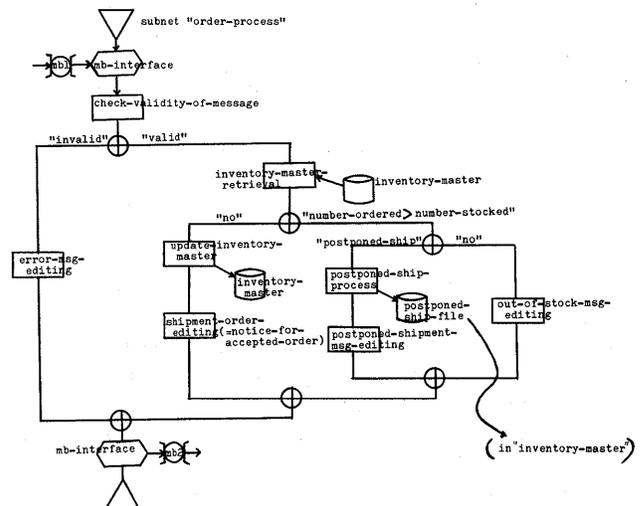


Figure 14—Example of control flow representation (2nd level of decomposition)

	Traditional		Improved	
Requirements out of scope	1.2%		2.9%	
Out of scope		1.2%		2.9%
Missing requirements	23.1%		17.6%	
Elements of requirements not stated		3.2%		2.5%
To be defined		3.7%		0.9%
Decision criteria missing		1.0%		1.4%
Interface missing		5.9%		4.0%
Processing cycle missing		1.5%		2.0%
Recovery requirements missing		1.0%		1.4%
Performance/reliability requirements missing		4.3%		2.5%
Developmental constraints		1.5%		2.0%
Others		1.0%		0.9%
Incorrect requirements	18.6%		22.7%	
Requirement not testable		7.3%		8.0%
Requirement satisfaction probabilistic		0.5%		0.9%
Timing requirement infeasible		2.8%		1.9%
Accuracy requirements not realizable		0.5%		0.9%
Parameter units incorrect		0.5%		1.4%
Equation incorrect		1.6%		1.9%
Unnecessary requirement		2.1%		2.4%
Required processing wrong		2.8%		3.5%
Requirement reference incorrect		0.5%		1.4%
Others		0.0%		0.4%
Inconsistent requirements	13.0%		25.1%	
Conventions not consistent		3.8%		5.0%
Requirements information not same in plural locations		9.2%		20.1%
Others		0.9%		0.0%
Unclear requirements	40.7%		23.2%	
Terms need definition		17.9%		8.2%
Requirements need restatement in other words		22.8%		15.0%
Others		0.0%		0.0%
Typographical errors	3.4%		8.5%	
Typos		3.4%		8.5%

Figure 15—Requirements error distributions of traditional and improved requirements methodologies⁵

2. Systems analysts can get clear requirements analysis and definition guidelines, and this implies the potential possibility of standardization, and makes data collection and analysis meaningful on cost and quality.
3. Projects may have less dependency on the experience of the project manager in requirements analysis and definition.
4. Planning and scheduling of requirements analysis and definition phases are made easier, and this allows the analysts to spend adequate time and resources.
5. User involvement in the RA&D phases can be expected more than before.
6. Because of use of data-oriented and control-oriented requirements representation techniques, the understandability of expected system behavior has been improved.

The new methodology does point to some problems. A typical one is concerned with the cost of requirements analysis and definition activities. In the traditional approach, the cost and resources used for RA&D were almost nothing (not zero, of course, but very small because almost nothing was done during the requirements analysis and definition phase). The cost of requirements engineering in the improved approach has become very large, and is considered to be budgeted. This may force the necessity of a two-stage contract system for RA&D, and development, has been appeared. Also the high

cost problem seems to be very severe for small and medium scale software projects, although for large scale projects it may pay off.

Finally, it should be pointed out that no automated tools exist for the proposed methodology.

ACKNOWLEDGMENTS

We are grateful to Drs. Barry Boehm, Roland Mittermeir, and Victor Basili for many helpful comments and stimulating discussion. This research was supported in part by the U.S. Air Force under Contract No. AFSORF 49620-80-C001, and by the U.S. Army under Contract DASG 60-80-C-0024.

REFERENCES

1. Alford, M. "A requirements engineering methodology for real time processing requirements," *IEEE Tr. Soft. Eng.* Vol. SE-3, No. 1, pp. 61-69, Jan. 1977.
2. Bell, T.E. et al, "An extendable approach to computer-aided software requirements engineering," *ibid*, pp. 49-60.
3. Boehm, B.W., "Software and Its Impact; A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.
4. Boehm, B.W., "Seminar on Software Requirements Analysis and Design," Berlin, 1979.

-
5. Miyamoto, I., "On the way to a practical requirements analysis and definition methodology," 14th Hawaii International Conf. on Systems Science, Jan. 1981, pp. 140-152, Vol. 1.
 6. Ramamoorthy, C.V. and Yeh, R.T., "Tutorial: Software methodologies," IEEE catalog No. EHO 142-0, 1978.
 7. Ross, D., "Structured Analysis (SA): A language for communicating ideas", IEEE Tr. Soft. Eng., Vol. SE-3, No. 1, 1977.
 8. Shigo, O., et al., "A software design system based on a unified design methodology," Journal of IPSJ, Vol. 21, No. 5, 1980 pp. 528-538.
 9. Thayer, T.A., et al, "Software Reliability Study," TRW System Report SS-76-03, 1976.
 10. Yeh, R.T., et al, "Software Requirements: A report on the State of the Art," in *Software Engineering*, Ramamoorthy & Vick, eds, Van Nostrand, 1981 (to appear).

A methodology for information system design

by COLETTE ROLLAND

Université de Paris
Paris, France

ABSTRACT

This paper concerns the design of large and integrated information systems (IS) in organizations. We propose to organize the IS development process in two interdependent steps, the first one centered on the semantic representation of the real world system, the second step including technical aspects of the solution ignored in the first one.

We present an original model to design the solution at the first step we named IS conceptual schema. This model allows a complete, consistent, nonredundant and economic representation including static and dynamic aspects of the real world system.

INTRODUCTION

We are concerned with the design of information systems (IS). We mean by IS a collection of data structured in a database, a collection of programs and transactions in a programs base and a collection of synchronization commands that control the triggering of programs and transactions upon data. Our experience is derived from the application of research results to the design of a complete information system for enterprises of the Electronics industry.

It is known that the development of a complete and consistent IS is a very hard task. This task involves, for a long period, a great variety of people doing particular activities, especially when the domain to inform is large and complex. We are now developing a complete information system for a company employing one thousand people. The development is planned for three years and involves ten information systems specialists on one hand and forty user-managers on the other hand.

Furthermore, when the system is complex it is not easy to undertake the development of the whole system at the same time without any risk of great confusion in the current activities.

A frequently used solution for this kind of complex development is to organize the process in several interdependent steps. In our approach, we have retained two steps: a conceptual step and a physical step.

For each step the solution is

1. obtained by a modelization using theoretical concepts and tools, and

2. expressed with a formal language.

The first step is centered on the semantic representation of the real world system. The first step solution, named information conceptual schema,¹ is a formal representation of the natural structure of facts perceived in their static and dynamic dimensions.² The second step includes the technical aspects of the solution ignored in the first one. It complements the initial solution by introducing parameters that were not necessary before.

The IS conceptual schema allows a complete, consistent, nonredundant, and economical representation of the real world system. It has the same role and advantages as the database conceptual schema in the design of a database but it is more complete. We make the hypothesis that the complete aspects of real phenomena must be represented in IS design. Not only must static aspects of the organization (as in a database) be represented but also the evolution of the items in time. The transformations of the organization components must complete the static structure expressed by the data schema.

From our point of view, the information system conceptual schema is a unique schema where the static structure, the transformations, and the time interrelations must be represented. Our IS conceptual schema is the integration of these three aspects.

To define the IS conceptual schema we need a conceptual model. This model provides elements for the construction of the set of data, the set of programs and the control of the time relations between data and programs.

There are many data models^{3, 4, 5, 6, 7} but very few proposals that attempt to integrate data, processes and dynamic command.^{8, 9} We have developed for this IS conceptual schema an original information system conceptual model,^{2, 10} that we now present.

A PROPOSED MODEL

What do we need to represent?

The model has been defined by an analysis of the real world phenomena, which lead to the two following conclusions:

1. In a dynamic perspective we have to represent three

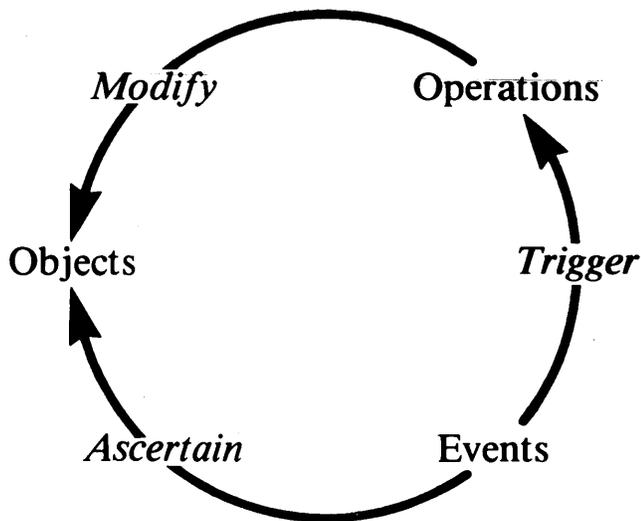


Figure 1—Definition of organizational dynamics

categories of phenomena, described according to their properties as *objects*, *events*, and *operations*;

- The dynamic dimension is completely represented by three categories of associations between the three categories of phenomena, *modify* (operation, object), *ascertain* (object, event), and *trigger* (event, operation).

An object is a durable, concrete or abstract component of the organization that can be particularised.^{8, 3, 7, 11} Examples might be the customer DURAND or the product number 33.

An operation is an action that can be executed at a given time in the organization and that modifies the state of one or more objects. For example, the operation "order analysis" number 312 creates the object "accepted order" number 202.

A modify association is an association connecting an operation and one or more objects. In it the operation modifies the objects.

An event is anything that can happen at a given time. It is the ascertainment of the state change of one or more objects by means of operations execution. For example, the event "order arrival" is the acknowledgement of the creation of the object order number 44 which triggers the operation "order analysis."

An ascertain association is an association connecting an event and one or more objects. It expresses that the state changes of objects are events.

A trigger association is an association connecting an event and one or more operations. It expresses that an event triggers one or more operations.

We propose a causal definition for the organization dynamics: the events cause the execution of operations issuing state changes of objects that could have events (see Figure 1).

Our representation is based upon a clear difference between state, state change, and event. A *state change* is different from a state. The state of an object can be durable, but a state change is instantaneous. A state change expresses the passage from one state to another one. An *event* is different from a state change; not all state changes of an object are events. For example, any modification affects the stock and

generates an multitude of state changes of the stock, but only some of them are events issuing restock orders. A *state change* describes a change, an *event* is a state change that triggers determined operations.

It should be noted that the phenomena of one category belong to classes, for example, the customer class or the order arrival class. In a class all the phenomena are described by the same collection of properties.

How to represent it?

The conceptual model must satisfy two main requirements: to be formal and to represent easily and homogeneously the classes of facts previously defined.

- We choose a typed relational model.⁴
- We introduce types for relations in order to represent different categories.
- We introduce time.

The correspondence between reality and conceptual representation is that

- each class of phenomena and each class of associations is represented by one or several relations;
- each property of a class of phenomena is represented by an attribute of relation; and
- the category of a class of phenomena is represented by its relation type (denoted C-object, C-operation, or C-event).

The three types of relations, c-object, c-operation, and c-event can be expressed in a normal form we name temporal normal form. We give the definition of concepts below.

The c-object concept

A c-object relation type is a *permanent relation*, i.e. a third normal (3NF) relation¹² where each attribute is in a permanent dependency¹³ with the identifier's relation. A *permanent dependency* between two attributes A and B (denoted $A \rightarrow B$) is an elementary direct and canonical dependency where $\forall a$ (occurrence of A) and the dependent b (occurrence of B), a and b have the same life duration. For example the 3NF relation CUSTOMER (NCLI, NAME, FIRSTNAME, ADDRESS, IDENTNUM, CA) is decomposed into the three c-objects

- CUSTOMER-PER (NCLI, NAME, FIRSTNAME, IDENTNUM),
- CUSTOMER-AD (NCLI, DATEM, ADDRESS), and
- CUSTOMER-ACTIVITY (NCLI, DATE, CA)

because the name, the first name and the identification number of the customer are permanent; the address can be modified and the turnover (CA) increases for each order of the customer.

We can interpret a c-object as being the biggest set of properties of an object having identical dynamic behaviour, that is

to say, properties created, modified or suppressed at the same time. A c-object represents a time-consistent aspect of the real-world objects class. Several c-objects represent a real world objects class. We named c-class the gathering of all these c-objects.

A c-object represents an atomic state of the information system.

The c-operation concept

A c-operation relation type is a permanent relation. The normalization of a c-operation must satisfy the following constraints:

- C-OP → C-OB
- C-OP → TYPE-CHANGE
- C-OP → TEXT-OP

We express the constraints using functional dependency notation.¹⁴ We make C-OP, C-OB the identifiers of c-operations and c-objects; we make TYPE-CHANGE the designation of the three state change types of objects (creation-destruction-modification) and TEXT-OP the name of texts of c-operations.

We define the c-operation by reference to the c-object concept. The c-operation is the expression of the smallest transformation that can happen to a c-object. More precisely, the first two constraints express the fact that the occurrences of one c-operation represent the operations that modify, in the same way, the states of objects corresponding to the same c-object. In other words, *a c-operation modifies in a unique way the state of one and only one c-object*. The third constraint expresses the fact that a c-operation represents an organization's management rule.

As a c-object, a c-operation represents a temporal aspect of a real world operations class and several c-operations represent the complete real operations class. For example, the real operations class "order analysis" is represented by three c-operations,

1. EXECUTION-ORDER ANALYSIS (NOR-AN, DATE-EXEC, NOR),
2. PERMANENT-ORDER ANALYSIS (NOR-AN, TYPECRE), and
3. MANAG-RULE-ORDER ANALYSIS (NOR-AN, DATETEXT, TEXTOP),

because there are many executions of "order analysis" operations (1), several management rules used at different periods of the c-operations' life (3), and only one type modification of objects corresponding to the c-object order (NOR) for all the life of the c-operation (2).

A c-operation represents an elementary transformation of the information system.

The c-event concept

A c-event relation type is a permanent relation. The normalization of a c-event must satisfy four constraints:

- C-EV → C-OB
- C-EV → TYPE-CHANGE
- C-EV → P-INIT, P-FIN
- C-EV → C-OP

We express the constraints using functional dependency notation¹⁴ and multivalued functional dependency.¹⁵ We denote by C-OB, C-EV, and C-OP identifiers of c-object, c-event, and c-operation relations; we make TYPE-CHANGE the designation of the three change types of objects and P-INIT, P-FIN the designations of predicates that express the states of a c-object.

We define the c-event by reference to the c-object concept. The c-event is the expression of the smallest noteworthy state change of a c-object. More precisely, the two first constraints express that a c-event represents the class of events that ascertains only one type of state changes of objects corresponding to the same c-object. The third constraint expresses that the state change that is the event is defined by the initial state and the final state expressed with predicates. For example, the events belonging to the c-event "restock" are connected with the modification of the c-object "stock" defined by the two predicates

- P-INIT : = any stock
- P-FIN : = stock < limit

The fourth constraint expresses that the events of a c-event trigger the operations corresponding to one or several determined c-operations. In other words, *a c-event is the state change type ascertainment of only one c-object, which triggers one or more c-operations*.

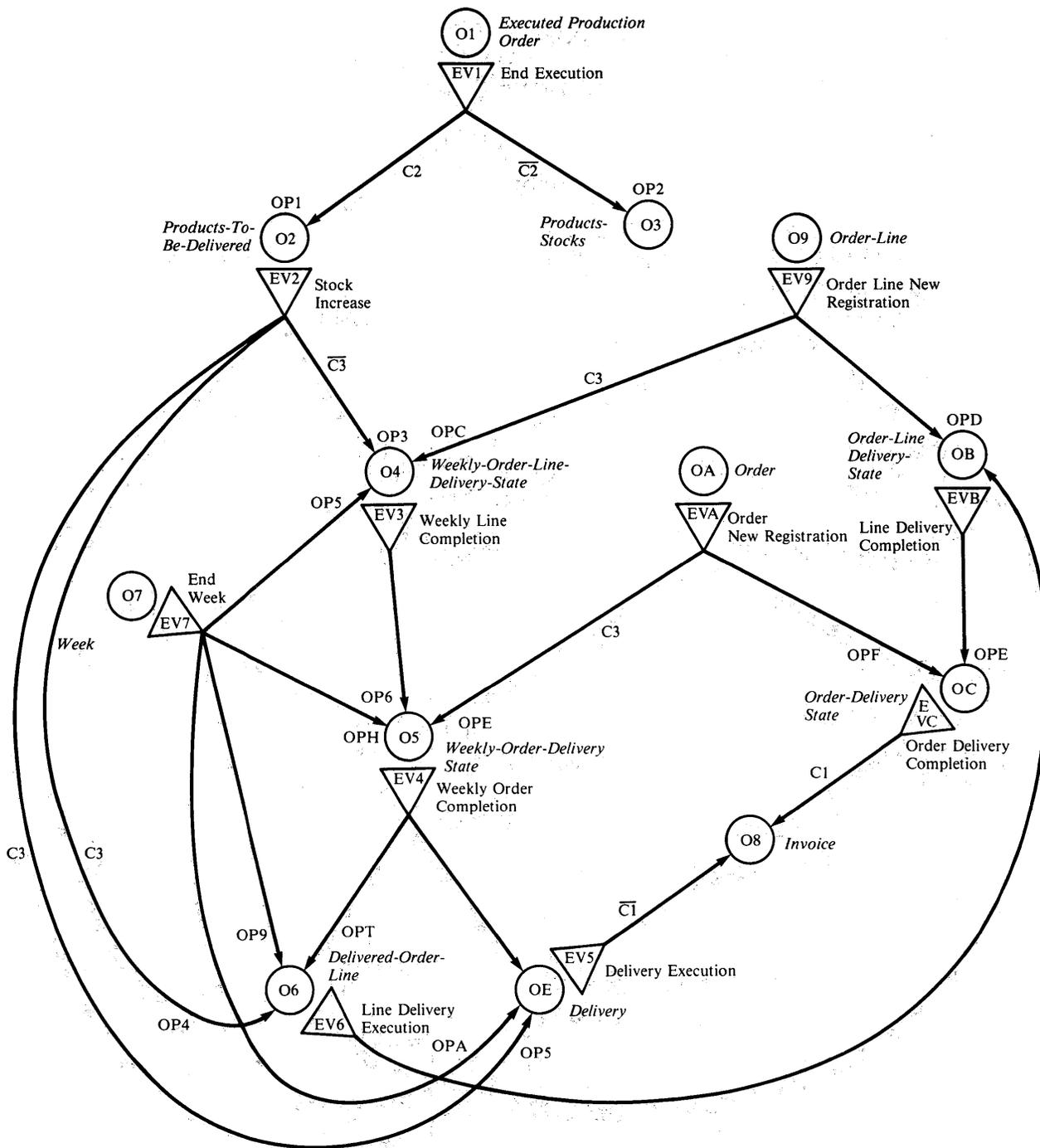
A c-event represents a temporal aspect of a real events class, and several c-events represent the complete real events class. For example, the real events class "order arrival" is represented by the following three c-events:

1. ORDER-ARRIVALS (NOR-ARR, DATE-ARR, NODER)
2. PERMANENT-ORDER EVENT (NOR-ARR, PI, PJ, TYPECRE)
3. TRIGGER-ORDER EVENT (NOR-ARR, NOR-AN, DATE-TRIG)

Relation (1) describes the arrivals of the event type "order arrival." Relation (2) describes the state change that defines the event. Relation (3) describes the triggering of the operations belonging to the type "order analysis" associated with the events "order arrival."

EVALUATION OF THE MODEL

Not many approaches define the IS in a dynamic way and attempt to integrate the data, the transformations, and the command actions. Our originality is in the capability to analyze structurally the interrelations between the three aspects. We give the opportunity to define in a complete description the functional behavior of the information system to be built. The following example expressed by the IS conceptual schema



Conditions

- C1 : Particular Customer
- C2 : An Order Corresponding to the Manufactured Product
- C3 : The Current Week Corresponds to the Week Certified in the Order Line

Notations

- C — Object
- C—Event
- C—Operation

Figure 2—Graphic representation of the dynamic conceptual schema

(described with graphic notation in Figure 2) illustrates the mutual dependencies that connect the c-objects, c-operations and the c-events of the problem.

The example concerns the information problem of the elec-

tronic sector. The firm sells approximately 40,000 products. Every month, the firm receives 3,000 orders. For one product the corresponding part of the order could be divided in quantities according to delivery date. We named the order part

corresponding to this quantity "order line." Several products of the same order could have the same "order line" delivery date. The delivery date retained after negotiation between the customer and the firm is expressed in week number in the year. When the week arrives and if all the "order lines" for this week and the order are available the delivery is done. At the end of the week all the available order-lines for an order are delivered. The delayed remaining "order lines" are delivered as soon they are available. Systematically each delivery is immediately billed. For certain customers the billing is postponed until the total delivery of the complete order is done.

In Figure 2, the creation of an O1 object is an event (EV1) that triggers

1. the update (OP2) of the products-stocks (O3) if condition C2 is false
2. The update (OP1) of the stock of products-to-be-delivered (O2) if condition C2 is true.

The arrival of a product in the stock of products-to-be-delivered is an event (EV2) that triggers

1. the update (OP3) of the weekly-order-line-delivery-state (O4) if the new product has been manufactured in time.
2. the immediate delivery (OP4, OP5) of the order part that has been manufactured with delay.

When an order line is completed (EV3) we update (OP6) the weekly-order-delivery-state (O5) and when all the current weekly order part is completed (EV4) we deliver it (OP7, OP8).

The delivery is an event (EV5) which triggers the invoicing (OPB) to the customer if he is not a special customer (condition C1).

The week-change is an event (EV7) that triggers the update (OPJ, OPH) of the objects weekly-order-line-delivery-state (O4) and weekly-order-delivery-state (O5) and the immediate delivery (OPA, OP9) of the uncompleted part of customer order that has been produced in the last week (O6, OE).

A new order line (O9) arrival is an event (EV9) which triggers the two operations (OPC, OPB) of creation of the two objects weekly-order-line-delivery state (O4) and order-line-delivery-state (OB) if the delivery delay corresponds to the current week.

The creation of a new order (OA) is an event (EVA) that triggers the operations (OPE, OPF) of creation of the two objects weekly-order-delivery-state (O5) and order-delivery-state (OC).

Each creation or a delivered-order-line (OE) is an event (EV6) that triggers the update (OPK) of the order-line-delivery-state (OB).

When an order line is completely delivered (EVB) we update (OPH) the order-delivery-state of the customer and when the order is delivered (EVC) we invoice the customer if he is a particular customer (condition C1).

THE USE OF THE MODEL

Based on the information system conceptual model, we have developed a complete method for the design of information

systems. This method is today used for the design of complex information systems implying automatic data processing, network communications and real time response. Our goal now is not to present the method but only to insist on few specific aspects of the model and on the possibilities of its use.

Description of the Information Problem

It is well known today that a central question in the IS design consists in a general difficulty of expressing exactly and competely the information problem to solve.

Users know their information problem. They are permanently confronted with aspects of it. But they have a partial and personal view and it is not their task to obtain a complete and consistent description of it. In fact, they expect the designer to do it and more directly it is his task. But to do it, he needs a model allowing the description of all the particular aspects of the problem to represent. The proposed model allows the definition of the information system conceptual schema that gives a global and complete view of the real world represented. The concepts of the model are precisely defined and formally connected and there is no possible ambiguity in their use nor real difficulty to build the conceptual schema.

We have noticed that the designer does not confuse events, objects, and operations when he uses the model. Better, he could, in certain situations, correct any misunderstanding due to a superficial analysis. In general, users and designers reach a more precise understanding of reality, especially in the definition of the events triggering the operations. A dynamic description of the reality expressed with simple and accessible concepts is a fruitful output appreciated by the users, especially because they get a better control of their information problem. As a result of the model's use we obtained more direct and efficient interactions between the users and the designers. The users were more confident in the capability of the designers to comprehend their information problem and solve it. The model makes the designer less dependent on the users. With a starting collection of facts he could develop his description, controlling the consistency, the correctness, and the lacks. He could determine independently of any suggestion when its analysis is completed. Conversely, he knows when he needs more facts in order to complete or correct his description.

Finally, with the model used for the description of the conceptual schema it becomes more natural to separate systematically the step of description of the information problem and the step of research of possible solutions. We are surprised at the confusion existing between the two aspects before using the model. In general, this confusion induced suspicion and doubts about the competence of the designer.

The Design and the Evaluation of the Solutions

As was illustrated in the example, the conceptual schema gives a complete definition of the information problem that must be solved by the future IS that the designer will define technically. Said in other words, the IS conceptual schema presents all the collections of objects, actions and events that

the technical information system will operate. Starting from the IS conceptual schema, the work of the designers is to define the technical solution to implement.

If the information problem is complex it is beneficial to undertake progressively the technical design and the construction of the future system. In this case, the information problem must be split in parts as independent of one another as possible. But the splitting is often arbitrary. It is clear that it neither burdens the design task nor complicates the interface and the implementation aspects. By way of the IS dynamic conceptual schema it is possible to clearly and simply evaluate where the frontier between the parts could be established. The dynamic associations between objects, operations, and events represented in the IS dynamic conceptual schema illustrate the connections between any part of the future system and arrange elements in order to reduce the arbitrariness of the partition. The experience demonstrates that the c-object class concept is a central but not unique parameter of the partitioning.

For any information system, complex or not, the solution is simultaneously technical and administrative. Technical because actions and events will be automatically processed, administrative because people will be totally responsible for the execution. It is known that the success of any system results in the balance between the two. Our scope is not to give the parameters for the successful solution but to insist on the fact that in this case also the designer must establish a frontier between the technical part and the administrative part. This frontier introduced in the diagram representation of the IS dynamic conceptual schema allows the designer to appreciate the consistency of any part and to evaluate the complexity and adequacy of the interfaces needed. Of course, the more complex the IS, the more extended and various the technical and the administrative parts and the more difficult is to establish and evaluate the interfaces. In the conceptual schema the range and the variety of the interfaces appear clearly and could alert the designer on the difficulties he would certainly meet.

The Evaluation of the Flexibility of the Technical Solutions

For large and complex ISs the flexibility of the solution is today an advantage that could with profit increase the lifetime of the system or improve the service given to the users. Many kinds of change are possible:

- a change in the organization's behavior that transforms the information problem
- a change in the nature of the service expected from the IS by the user involving the extension of the technical domain to the detriment of the administrative domain
- a change in the nature of the technical solution retained due to technological transformations.

For any kind of change the consequences are not similar and it is worthwhile to have the opportunity to evaluate the major consequences of the change and the technical possibility of taking it into account.

If we consider a change in the organization behavior, many

consequences could result for the information problem. Objects, operations, or events could be created or suppressed, implying the transformation of the technical solution. In the delivery example, the adoption of the rule "every available order line must be delivered without waiting for the other order lines of the week" results in the suppression of the c-objects O4, O5 and O7, the events EV3, EV4, and EV7 and the operations OP3, OPC, OP6, OPE, OHP, OP7, OP8. The operations OP4, OP5 become unconditional.²¹

A change in the nature of the service expected by the user could be appreciated in the same way. In the same example, the necessity to manage automatically the delivery of the products in stocks (O3) would force the designer to introduce new objects, operations, and events.

Using the IS conceptual schema it is easy to appreciate the probable transformations needed for the present system to be adapted. The estimation of the efforts, the time, and the costs resulting gives an idea of the flexibility of the present solution.

The Organization and Planning of the Technical Design

Starting from the concepts included in the model, we have developed a method for the organization and the planning of the technical design. As we have shown, the splitting of the problem in order to develop limited technical parts is essentially based on the connections between the c-class of objects (others economic aspects that are not illustrated here also interfere). For the construction of the planning for the technical design and the estimation of the resources we have established a simulation tool based for any technical part on the range, the variety, and the interferences between any c-class of objects, c-operations, and c-events. As a result, we obtain an estimation of the realization testing and implementation time and costs. These estimations based on the IS conceptual schema make possible to plan people's activities and to negotiate with the customer the costs and the schedule for the future IS.

CONCLUSION

From our experience of the IS design we can assume the following:

1. The interest in defining a conceptual solution,
2. The necessity of disposing a set of precise concepts allowing a rigorous analysis and design, and
3. The advantage for project management of using a complete method based on a formal model.

Our effort is now devoted to the development of tools in order to reinforce the method.¹⁶

REFERENCES

1. Ansi/X3/Sparc: report on data base management systems—Interim Report (1975).
2. Rolland, C., Foucaut, O. Concepts for design of an information system conceptual schema and its utilization in the Remora project—Fourth international conference on Very Large Data Bases (1978) Berlin.

3. Chen, P.P.S. The entity relationship model—toward a unified view of data. *ACM transactions on data base systems* (1976) Vol. 1, no. 1.
4. Codd, E.F. A relation model of data for large shared data banks. *Communications ACM*, Vol. 13, no. 6 (1970).
5. Delobel, C. Contribution théorique à la conception et à l'évolution d'un système d'information appliqué à la gestion—thèse d'état—Grenoble (1973).
6. Institut d'Informatique de Namur. Proc. of the international workshop on data structure models for information systems (1975).
7. Kent, W. Describing information (not data reality): technical report TRO 3012.
8. Benci, G., Bodart, H., Cabanes, A. Concepts for the design of a conceptual schema. Modelling in data base management systems. Proceedings of the IFIP Working Conference TC2 (1976).
9. Bodart, F., Pigneur, Y. A model and a language for functional specifications and evaluation of information system dynamics. IFIP TC8 WG8, (Working Conference on "Formal Models and Practical Tools for Information System Design. Oxford, England, 1979)
10. Rolland, C., Foucaut, O., Richard, C., Thiery, O. Information system design and computer-aided design—Euro-IFIP Conference (1979) London.
11. Lindgreen, P. Basic operations on information as a basis for data base design—Proc. IFIP TC2, North-Holland (1974).
12. Berstein, P.A. Synthesizing third normal form relations from functional dependencies—*ACM transactions on data base systems*—Vol 1, no 4 (1976)
13. Rolland, C., Leifert, S., Richard, C. A proposal for information systems design and management. *ACM/SIGDA Newsletter* (1980).
14. Codd, E.F. Normalized data structures: a brief tutorial ACM SIGFIDET workshop in data description access and control—San Diego, California (1971).
15. Fagin, R. Multivalued dependencies and a new normal form for relational data bases. *ACM transactions on data base systems*. Vol 2, no 3 (1977)
16. Rolland, C., Leifert, S., Richard, C. Tools for information system dynamics management—Fifth International Conference on Very Large Data Bases (1979) Rio de Janeiro.

**EDUCATION AND
SOCIETAL ISSUES**

CSDP: A model for continuing education in data processing

by DENNIS M. OLIVER, ROBERT A. ROUSE, and ROBERT J. BENSON

Washington University
St. Louis, Missouri

ABSTRACT

The Center for the Study of Data Processing (CSDP) was formed in cooperation between Washington University and a group of St. Louis corporations to create a local source of programmers and a business-oriented data processing (DP) curriculum and to provide inexpensive professional training for corporate personnel.

The Center oversees three academic programs. The B.S. provides continued academic training and a degree in systems and data processing. The 30-hour intensive degree specializes in retraining selected people for entry-level DP positions. The master's degree offers training in managerial issues.

The Center also offers 25 seminars yearly in data processing. Their subjects range from basic training to advanced managerial issues.

Much of the staff for seminars and evening classes comes from the cooperating corporations. This pooling of practical experience both in the academic and the professional programs has created an up-to-date curriculum and provided inexpensive, high-quality seminars for the region.

THE CENTER FOR THE STUDY OF DATA PROCESSING

In 1968 the School of Continuing Education at Washington University first offered a B.S. degree with a major in systems and data processing. Seven years later a master's degree in data processing (MDP) was added to the curriculum. In that same year, a group of St. Louis corporations joined with Washington University in an attempt to construct a professional development program that would offer continuing education to their programming and systems staff. To focus these activities, the Center for the Study of Data Processing was formed in July 1978 as a cooperative venture between local industry and the university.

The specific needs of the St. Louis data processing community have determined most of the Center's activities. One of these needs is for trained applications programmers and analysts. According to recent Department of Labor estimates, between now and 1985 the demand for programmers will increase by 50% and the demand for analysts by 65%. Able people are always and everywhere hard to come by. National

recruiting efforts are expensive, and the older urban areas of the Midwest have the additional disadvantage of trying to attract the young college graduate who can choose between California's beaches and Boston's museums.

A second need is for applications-oriented training. All too often the typical computer science graduate responds to the prospect of writing and designing business applications with a mixture of disdain and confusion. He or she has not been educationally prepared to function in such an environment, has little knowledge of business needs and procedures, and has even less interest in acquiring any.

A third industrial need is for continued training in DP. The cost of sending someone to a vendor for training is large, but often not as large as the cost of transportation and lodging for the attendee. A local training program offered on a regular basis, with content determined by general agreement among the participating organizations, can significantly reduce the training costs.

It is worth noting that these needs are not peculiar to the St. Louis area. They represent the needs of any urban area above the critical mass that allows cooperative support to flourish. A local source of training pays multiple dividends. The cost savings of recruiting and training are obvious, but an important additional benefit is the ability of local industry to directly influence program and curriculum content. The purpose of this paper is to present some of the ways in which such a cooperative venture can prosper and to point out how the experiences, contacts, and skills gained in one program have been used to build or improve other programs.

Currently the Center's areas of responsibility are the B.S. and M.D.P. programs, the professional development program (P.D.P.), an intensive B.S. program for students who already hold a bachelor's degree, and the development and maintenance of an audiovisual instructional facility as well as a research library.

The bachelor's degree program is by far the oldest and the busiest Center activity. About 25 courses with nearly 40 sections are offered every semester for about 900 students. Of these students, 120 are degree candidates, and figures for 1980 indicate that about 30 B.S. degrees will be granted in the next year. But the nature of the student body and faculty tell more about the program than these numbers.

Over 90% of the undergraduate degree candidates are currently employed, and about 75% of that group are engaged in

data processing on a full-time basis. Most of the rest are employed in closely allied fields, like accounting or data retrieval. In addition, about 90% of the faculty practices what it teaches: Systems analysis is taught by analysts, programming by experienced programmers, and DP management by administrators. The remaining faculty is drawn from Center staff, who also have DP experience, either from previous positions or through their duties in Washington University's computing facility. None, it might be noted, has a straight computer science background.

The particular nature of the faculty and students has significantly enhanced the value of the B.S. degree, since both groups continually encounter real-world DP problems on a daily basis, then bring that experience to bear upon the academic subject matters they deal with. The M.D.P. was established as a natural extension of this educational environment. The heart of the master's degree program is a curriculum of four courses emphasizing the control of DP activities through wider knowledge of technology, database management problems and principles, management information systems, and general managerial techniques. The remaining courses are selected from computer science or business offerings. The typical master's candidate is not a recent college graduate; he or she has five to ten years' experience in data processing and is expected to contribute that experience to the program.

The ability to pool experience is not only a benefit to program participants; it feeds several other Center activities. First, the knowledge gathered from M.D.P. students often finds its way back into the undergraduate curriculum, which has just undergone an extensive revision. Second, the issues encountered in the master's program (e.g., management problems, employee productivity, long-range planning) have created directions for research that are currently being pursued by Center staff. Finally, both the M.D.P. and the B.S. programs have served as contacts for and models of the professional development program, through which the Center has become an important adjunct to the training efforts of more than a score of large St. Louis data processing facilities.

For an annual fee these corporations become subscribing members of the Center. As such, they have guaranteed access to all Center offerings. The P.D.P. program presents about 25 seminars and classes every year (not counting in-house presentations), which total over 180 days of instruction. The offerings concentrate on three levels of experience: the programmer trainee and the new programmer; the programmer/analyst and systems analyst; and the data processing administrator (senior analyst, project manager, data base administrator, DP manager). More than half of the teaching and more than half of the material for these seminars is supplied by the Corporate Affiliates,* who are able to pool their experience through the Center.

* Affiliates as of October 1980 are as follows:

Angelica Corporation	May Department Stores Co.
Anheuser Busch Companies, Inc.	Missouri, State of
Banquet Foods Corporation	Missouri Pacific Railroad Co.
Blue Cross Hospital Services	Monsanto Company
Brown Group, Inc.	Peabody Coal Company
Citicorp Person-to-Person, Inc.	Pet Incorporated
Emerson Electric	Price Waterhouse and Company

Of the seminars offered for entry-level personnel and programmers (Intensive COBOL, Structured Programming, IBM Operating Systems, Testing Methods, and Programming Efficiencies) perhaps the most unique and successful is Intensive COBOL. It is offered four times a year and generally has 16 to 20 enrollees. It is assumed that the students have no knowledge whatsoever of COBOL or of programming, and in four weeks (20 class days with about 40 hours of lecture and 120 hours of programming) they are expected to complete 15 programming projects.† At month's end the successful student has a working knowledge of COBOL and experience in analyzing programming tasks and implementing the principles and techniques of structured programming. Although the course is mainly peopled by Corporate Affiliate employees, many individuals have enrolled on their own and have used the skills acquired there to embark on a new career.

The majority of the remaining seminars appeal to a group delimited on one side by the programmer/analyst and on the other by the project manager. These offerings divide themselves between technical and managerial issues. A few seminars are directly aimed at upper level DP personnel (Long-Range Planning, Financial Planning and Modeling), in an effort to offer them the broader skills needed to manage DP operations. This latter group also has access to a series of colloquia, which brings figures with a national reputation to the Center every few months. The general goal of the colloquia is to supply DP administrators with the widest possible perspective on current and future trends in data processing.‡

A new P.D.P. activity has proved quite successful: this past year a series of round-table discussions on topics of special interest to middle and upper DP management has been undertaken. The discussions have been conducted by employees of the affiliates who have common concerns or areas of responsibility. The three held so far have dealt with DP training, continuing support management, and word processing. The round-table format offers affiliate staff the ability to examine special problems in depth by creating (as in the M.D.P. classes) an occasion for pooling their knowledge and experience.

P.D.P. activities have had an immeasurable impact on the traditional academic programs the Center oversees. For instance, the projects and skills which were posited as goals for Intensive COBOL have been incorporated into accelerated evening COBOL classes, and the problems discussed in the Systems Analysis and Structured Programming seminars have been integrated into a half-dozen courses. Long-range planning and database design seminars have found their way into the core M.D.P. curriculum, as well as into some day school

First National Bank in St. Louis	Ralston Purina Company
General American Life Ins. Co.	Regional Justice Information System
Kellwood Company	U.S. Army DARCOM ALMSA
Laclede Steel Company	Wagner Electric Company
Mallinckrodt, Inc.	

† Besides a basic knowledge of COBOL, at month's end the student is able to use nested and compound IF statements; edit all incoming data; do multiple-file inputs, merges, updates, and printing; create sequential and ISAM files on disk; and do table-handling, sorting, subscripting, indexing, and 2- or 3-level breaks.

‡ The last six speakers were Ted Withington, Gerald Weinberg, Dick Brandon, Richard Nolan, John Toellner, and Harvey Poppel.

computer science courses. In the last two years the whole undergraduate curriculum has been closely restructured to fit more closely the practices and needs of DP shops in the business world. The content and structure of all programs, as the round-table format indicates, must constantly be reviewed in the light of changes in the DP environment.

The maturity and vitality of the undergraduate curriculum and the experience of Center staff in implementing and running this curriculum permitted the creation of an additional program: the 30-hour intensive degree in systems and data processing. This program is directed toward a specific audience: those who already have a college degree in a field where employment chances are slim, or those who are attempting to make a career change. Through two years of evening classes, the intensive degree candidate who, we assume (as we do in Intensive COBOL), has no prior programming experience, is given a thorough grounding in programming, computer technology, and business systems.*

Because of the demanding nature of the program (the 30 hours of credit is, in the actual effort demanded, more like 45 hours), admission is competitive. About one applicant in three is granted admission to the program. The quality of the student is high: the cumulative GPA of the most recent class is 3.4, about 30% of the students hold a master's degree or higher, and all have demonstrated (through college records or employment history) the ability to successfully handle such a curriculum. There are three profiles typical of the intensive degree candidate: the housewife in her thirties whose children are off to school and who wants a skill that will give her access to the job market; the high school teacher (usually of languages, English, or mathematics) who has decided to change careers; and the employee who wishes to change his or her lot and move from a support position (documentation writer, data gathering) to a data processing position.

The intensive degree is a timely activity for several reasons. First, we anticipate no immediate slackening in the demand for qualified DP personnel: Corporate Affiliates and the St. Louis business community at large can certainly supply entry-

level positions for these graduates. Second, both the high school teacher who feels, after years of training and years of teaching, that a financial or personal dead end has been reached, and the housewife with a solid education but no marketable skill will certainly be valuable employees because they bring to the job stability of attitude as well as an above-average ability to communicate. Third, the population in need of retraining increased dramatically throughout the 1970s, and a corollary shrinkage in the technically competent younger population has produced (and will continue to produce) skill shortages in technical and semitechnical occupations.

The goal of the intensive program has been achieved for the first class: in the 15 months since they began study, 24 of the 26 students have successfully made the career change, and more than half of them are currently employed by Center affiliates.

In addition to these various programs, the Center manages two facilities in support of all of its activities: an audiovisual room and a library. The A/V facility is used as a support tool for P.D.P. and evening teaching activities. A large-screen television and a videotape recorder are used to present special topics to classes and seminars. ASI (Advanced Systems, Inc.), a company involved in producing training tapes for data processing, has contracted to supply the Center with certain tapes made available on a selected basis to affiliate and university staff. The Center library is designed to support Center programs and to create a research capability for Center staff and M.D.P. students. Its budget, holdings, and administration are entirely independent of the main library facility at Washington University. Currently the library has about 2000 bound volumes and subscribes to about 100 periodicals.

The decade we have just entered is an arena of opportunity in DP education. Because of the shortfall in younger people, continuing education in general will be expected to undertake massive retraining efforts in many areas. The close affiliation between local industry and education allows schools of continuing education to respond much more quickly than could a traditional academic department to changes in the environment. The kind of relationship the Center has established with its affiliates allows us to perform stimulating and gratifying services, both for the affiliates and for the students. At Washington University the relationships have not come easily, nor have they been problem-free; but we believe that the Center's success can serve as a model for similar efforts in similar environments.

*The curriculum in the 30-hour program consists of eight hours of COBOL (from the beginning through data-base programming), four hours of computer technology, three hours of effective communications, four hours on financial, managerial, and business information systems, three hours of DP management, four hours of systems analysis and design, and a four-hour systems design and implementation project.

People teaching people: A cooperative education venture

by EDWIN F. KERR

Q.E.D. Information Sciences, Inc.
Wellesley, Massachusetts

ABSTRACT

Personnel problems faced by DP executives will continue to be a major obstacle in developing systems for the 1980's. Both corporate and DP management are now viewing personnel development expenditures as an investment rather than an expense. This paper stresses the need for a career development program and a well planned education and training activity. It discusses the difficulty of providing the training, particularly classroom instructor-led courses. It next discusses a cooperative EDP Education Program (EDPEP) covering the concept, the establishing of a network of regional learning centers and the impact it has made. The program provides regularly scheduled classes to member companies at under \$100/day. Since its inception in 1974, it has grown to 75 member companies. In 1980 over 3,600 students attended the courses.

INTRODUCTION

"A company is known by the people it keeps." This transposition of an old adage is central to professional development. It is especially important in data processing because business is overwhelmed by the need to find, develop and retain competent personnel. The rapid growth of the computer field has created such a great demand for technicians and managers in so short a time that development of these people has fallen behind and, with few exceptions, has been treated as low priority by corporate management.

I wrote this paragraph in an article "Let's Look at Peopleware Maintenance," published in *Infosystems* in 1975. Very little has changed in the five years since except that at last corporate management has accepted the fact that the personnel problems faced by DP executives are a major obstacle in developing systems for the 1980's. This was very much on our minds in 1974 when we were experimenting with ideas for a cost-effective cooperative education program. We knew that at some point, management would be forced to take a more active role and view professional development expenditures as an investment rather than an expense. We also were convinced that even though instructor-led education is the most difficult to provide, it is the most effective. Why?

Turn-On To Learning

Ask yourself when you first "turned on" to learning. Probably in a classroom situation when you realized the excitement of discovery and problem solving. Probably an image of a particular teacher comes to mind. Maybe the image of a particular classroom and the memory of sharing ideas and solving problems. Again a teacher provided the stimulus and encouraged student involvement.

However, in my 24 years in the computer industry, especially during the last 10 devoted to education, I have seen that this ideal learning environment has not been the norm. Material is generally transmitted through media with little student-teacher interplay. This reliance on media has generated problems. Students may be passive, there is little questioning or probing and students cannot learn from each other. Furthermore, there is no teacher to shift the emphasis to suit a particular need or environment. In short, this is not teaching, this is information transmission!

The essence of learning is exposing people to ideas, concepts, experiences, and then providing them with practical situations in which they can apply their knowledge and skills. I am not saying that media based education does not have a role. I am questioning the over-reliance on it by many companies. You were not "turned on" to learning by a television set.

COOPERATIVE EDUCATION

In 1974, when we began setting up career development and education programs, there were few classroom instructor-led courses. The public courses were questionable because of cost, travel and lack of dependability. They were and continue to be subject to cancellation on a moment's notice.

We wanted an option that provided instructor-led workshops at a reasonable cost on a continuing basis. These workshops had to provide high quality education, reduce or eliminate travel costs and provide a means whereby companies could help shape and fine tune the program.

EDP Education Program

To meet this need, Q.E.D., in association with Babson College, pioneered a unique EDP Education Program called

EDPEP. In 1974 eight Boston area companies joined with us in this experiment.

Q.E.D. provided the curriculum, course materials, instructors and program guidance. Babson provided the facilities, lunches and administration. The companies provided the students, the dollars, and Advisory Council members to help us fine tune the program. The intent was to gain support from enough companies so that we could plan, schedule and provide an on-going program without being concerned about spending large amounts of money on marketing in order to fill classes. The savings would be passed to the member companies in the form of low fees.

The success of EDPEP was immediate. From the original eight companies it has expanded to over 75 members. In 1980 over 3,600 students will attend approximately 180 courses at five regional centers. The five regions are located at Babson College serving New England; Fairleigh Dickinson University serving New York and Northern New Jersey; Drexel University serving Eastern Pennsylvania and Southern New Jersey; The University of Texas at Arlington serving the Southwest and Raleigh, North Carolina serving the Southeast. A typical curriculum at a regional learning center would include the following courses:

Systems development topics

- Systems Analysis
- Systems Design
- On-Line Systems Design
- Structured Programming Workshop: Techniques for Productivity
- Structured Design Workshop
- Computer Control and Audit
- Effective Methods of Quality Assurance
- Strategic Planning for Information Systems
- User's Role in Systems Development

Date base topics

- Data Base Fundamentals
- Data Analysis and Data Base Design

Data communication topics

- Data Communication Fundamentals
- Network Analysis and Design Workshop
- Distributed Computing Systems

Management topics

- Project Management and Control
- Leadership: Managing and Motivating People

Human relations topics

- Human Relations
- Effective Presentations
- Writing for Results: A Workshop for EDP Professionals

These courses are two to five days in length. Other environment-sensitive courses are run periodically at the request of member companies. The curriculum varies from region to region depending upon the recommendations of the advisory council.

The courses are constantly reviewed and improved so that they may continue to meet the ever changing needs of our industry. The advisory council plays a major role in the shaping of the program. Our instructor teams spend a great deal of time modifying courses to meet this ever changing need.

The concept of EDPEP is simple . . . a curriculum of courses is scheduled at each regional learning center. At each region we have a spring session (February-June) and a fall session (September-January).

Each course is scheduled at least once per session. Additional sessions are often scheduled to handle over enrollment.

A company may join EDPEP by purchasing a membership in any region it chooses. In effect, the organization is purchasing the right to use a block of seats. The company uses the inventory of seats by sending participants to the courses scheduled at the region. In addition, a company may send participants to courses scheduled at other regions. We keep track of the usage. A company may purchase as many memberships as it requires. Many of our members do exactly that. Further, a member company, rather than purchase an additional membership, may purchase seats in a particular course. The current cost for this is \$100 per student per course day.

I mentioned low-cost education as a requirement of EDPEP. The first membership costs approximately \$100 per student day. The additional memberships cost approximately \$85 per student day. This makes the cost of high-quality classroom education affordable, and at the same time it is education for which you can budget and schedule in confidence.

Although Q.E.D.'s EDPEP was never intended to meet the total education needs of an organization, many companies do use it as the center piece of their educational option mix. Others use it as one of several options, selected when instructor-led education is needed. Also many members supplement EDPEP by having Q.E.D. conduct courses for them in-house. In this way they can provide education to a larger number of people at a reasonable cost. The break-even point is from 12 to 15 people.

In 1979, we expanded the EDPEP concept to allow non-member companies to participate in selected courses. The cost for non-members is higher than for members, but it is more than competitive with other public courses. This non-member participation has grown steadily, often leading to new memberships. The companies who use the program on a selected open enrollment basis are pleased because they can plan that the courses will be held: this is often not the case with the normal public course market.

The Benefits of Cooperative Education

The benefits of cooperative education are many. (1) All of our members plan and schedule their usage of the program in advance. Many member companies have begun to do some extensive educational planning. This planning is done with confidence that the scheduled courses will be held. (2) The cost of membership is very low, enabling member companies to use classroom education as an option. (3) The flexibility in participation options has made the courses available to small and large companies. (4) A wide range of courses is taught in a variety of disciplines. We provide a comprehensive program.

(5) Another advantage is the flexibility of schedules. Companies are now beginning to look at EDPEP as a network that provides a broad range of courses at regional learning centers with many dates that can be selected to fit the need.

This movement among regions is gaining momentum which has created a need for course quality control so that the same course is taught at all regions. To do this, we develop instructor guides. Each new instructor must take the course that he/she will teach and then team teach it at least twice with an experienced instructor. Each major topic area is assigned a Resource Manager. Part of the Resource Manager's responsibility is to insure that quality is maintained in each course and that the same course is taught at each regional learning center.

There are two other important aspects of EDPEP that I will mention.

1. Student profiles/installation profiles
2. Regional advisory councils

Student Profiles/Installation Profiles

Because of the nature of the program, companies are able to select students well in advance. Each student completes a Student Profile form which is returned to Q.E.D. These forms provide us with the necessary background, experience, and job related information about each student. These are reviewed to insure the student is qualified to attend. If there is a question, we contact the appropriate advisory council member. Conversely, they contact us if they have a concern. Our objective is to insure that the mix of students is controlled; not so diverse that it is impossible to meet the objectives of the student and the course.

The instructor also reviews the profiles, as well as a profile of the companies represented. This includes the necessary information about the hardware, software, and environment from which the students come. These tools, coupled with a unique student introduction exercise we use, prepares the instructor and the student to participate in an effective course.

Regional Advisory Councils

I have mentioned the advisory council previously. It is time to discuss the role in detail. Each advisory council is made up of two members from each member company. One is the person in charge of professional development and the other a senior DP line manager. The advisory council is the link between members and Q.E.D. It meets as often as necessary to carry out its responsibilities. The council reviews student critiques and discusses course content and the overall curricu-

lum. This helps us to fine tune the program and to insure that it meets the members' needs. This has become increasingly important. We have designed a three part course evaluation form to provide much of the information for the council.

At the beginning of the course, the student answers the question, "What do you expect to learn from the course?" At the end of the course, the remainder of the form is completed. We go beyond the normal course critiques, which in some cases are a popularity vote for the instructor. We address such issues as how useful the course will be to the student on the job and will it make a contribution to productivity.

One copy is retained by the instructor, one copy returned to the Q.E.D. administration function, the third copy is kept by the student. The student reviews the course evaluation with his/her advisory council member. The instructor and Q.E.D. review the form to determine what improvements have to be made to the course. At the next advisory council meeting, each course is discussed in detail. The program we run today is far different than the program of 1974.

This mechanism has proven to be very important to the success of the program in a number of ways.

1. The student is forced to give some thought to expectations from the course. This focuses on the issue of student objectives versus course objectives.
2. The instructor gains immediate insight into student expectations and objectives which are related to the background information contained in the student profile.
3. Learning is reinforced because the students and advisory council members are forced to discuss what was learned and how it will be applied. This interest in career growth has made the students more receptive to the program because they see that the companies are committed to professional development.
4. The courses are continually enhanced to meet the changing needs of the member companies.

The council meetings are very productive since they give all members an opportunity to share ideas. The meetings have become a clearing house of ideas; as a result, they are well-attended. The topics range from immediate needs to long-term directions. Occasionally a guest instructor is invited to discuss a course in detail. Non-members are also welcomed in order for them to learn more about the program.

The benefits to the member companies are many, and the impact it has made on professional growth indicates to us that it is the model for things to come.

REFERENCES

1. Sullivan, R. *Handbook for Data Processing Educators*. Wellesley, Massachusetts: Q.E.D. Information Sciences, Inc., 1979.

Computers and the future of literacy

by FREDERICK L. GOODMAN

University of Michigan
Ann Arbor, Michigan

INTRODUCTION

My goal is to get people who make, buy, and use computers to consider more carefully the impact the computer is likely to have on our culture by recognizing the way it is both changing what it means to be "literate," and perhaps more importantly, changing how people become "literate."

In the simplest sense, to be literate is to be able to read and write. For an individual person this means that s/he must both live in a literate society as contrasted to a *pre*-literate society and have moved from a condition of *ill*iteracy within a literate society to a condition of literacy. Thus literacy may be a function of both the role that literacy plays in a society and the way an individual comes to be more or less literate. Within the last 50–100 years, the electronics industry has significantly changed the nature of what it means to be literate in many societies. Within the last 5–10 years the same industry has begun to change the way an increasing number of children learn to become literate by making sophisticated electronic devices available to them, especially in the form of micro-computers.

People in pre-literate societies use language to communicate; illiterate people in a literate society use language to communicate. The difference between pre-literates and illiterates on the one hand, and people who are literate on the other, is (or was) that pre-literates and illiterates must use oral (or gesturing) messages that are *momentary* with respect to time and *proximate* with respect to space, i.e., we can think of them as short-duration and short-distance messages. Writing, and reading, clearly involve messages that can have a long duration and can be communicated over long distances. Prior to the advent of electronic means of recording short-duration/short-distance (SD) messages, writing and reading were the only ways of converting such messages to long-duration/long-distance (LD) messages. Indeed, it is the availability of electronic transmission, recording and play-back equipment that requires the insertion of the phrase "or was" in parentheses in the second sentence of this paragraph. The transformation of SD messages to LD messages can now be accomplished electronically so preliterates' or illiterates' SD messages may be transmitted by telephone, recorded, stored, shipped, played back, and so forth. Indeed with the advent of television there has been a virtual explosion of LD messages

for a whole range of "nonlanguage" messages are now transmitted, recorded, etc.¹

Put simply, if literacy, taken to mean writing and reading, involves the creation and use of LD messages, then an analysis of literacy may be very much a matter of an analysis of the difference between SD and LD messages and *anything that changes the relationship between SD and LD messages may alter the nature of literacy.*

My intent here is to focus on the role of the computer in altering this relationship but it is appropriate to comment first upon the impact of television upon literacy. A conventional concern among educators, parents, and the public at large involves television's role in making children passive, lazy, and perhaps spoiled, in the sense that they are conditioned to rapid-fire, attention grabbing communications. These may well be legitimate matters of concern but I do not wish to attend to them here. I simply wish to point out that a lengthy chain of technological innovations, from the printing press to radio, television, and the computer, has multiplied the availability of LD messages, vis-à-vis any one person to such an extent that an individual is literally surrounded with "there and then" (LD) messages, many of which come crashing in as "here and now" (SD) messages. Television is a dramatic source of such messages, bringing yesterday's horror-filled war action from the other side of the globe into today's living room. But I submit that it is not simply that one form of technology lies at the heart of the problem if the problem is seen as a dramatic mismatch between the availability of LD messages and the individual's ability to deal with all this at the SD message level. *A basic premise of this argument is that LD messages can only be decoded at a rate that is determined by a person's ability to process SD messages, i.e., the rate at which a person may hear or see what is placed in front of him/her at any one moment in time.* It is not simply television that "overwhelms" children; I submit that schooling can also "overwhelm" children insofar as it represents an enormous source of LD messages.

Education in pre-literate societies is mostly a matter of teaching by example and by passing along one's traditions orally. Reading and writing is, by definition, impossible in such a society. If a society possesses a written language, but many people are illiterate, education divides immediately into an "example and oral" mode of operation or a "schooling"

mode which means, typically, "reading and writing." The irony is that "reading and writing" not only are the keys to "power" as educators from Horace Mann to Paulo Freire² point out; "reading and writing" may also be the keys to an awareness of one's relative "importance." Once you begin to see the role LD messages might play in your education and sense that the experience of countless people over thousands of years is available to you via the LD message route, it shouldn't be too surprising that a sense of despair follows more commonly than does a productive sense of wonder. The potential "there and then" messages are so numerous they can easily drown anyone intent on living in the "here and now."

This point deserves more attention than I can give it here for I feel constrained to return as quickly as possible to consideration of the role of computers in all this. The significant step in so doing is to focus on the role of *reading as contrasted to writing* in the process of education. The answers to the question, "Why should you teach someone to read?" are obvious. If you can read, all the world's LD messages (in your language) are available to you. But why do we ask people to *write*? If you mean by that question, "Why do you ask a person to learn to write?" the answer may be, "So s/he can function in society, fill in forms, write checks, apply for jobs, etc." But if you ask the question a little differently, "What is gained when a person translates SD messages into LD messages? . . . a person's answer might look something like this:

1. So I can return at a later date and see what I was thinking/saying.
2. So I can make my thoughts clearer (for there is something about writing that forces greater clarity than just "thinking out loud").
3. So I can communicate rather specifically with other people—at a later time, in other places—perhaps so they can understand me, do what I want them to do, or have them see if I know something.
4. So I can at least suggest something to others that is interesting or important to me, perhaps so they can see the matter differently, more richly, more productively than I.

This list could be extended in many ways; it is not intended to be exhaustive, just illustrative. I wish to illustrate that when a person writes something it is either entirely personal (as in items 1 and 2) or *it commits someone else to do something*. With the exception of "making lists for oneself" (or perhaps keeping records or writing a diary) or writing "to clarify one's thoughts," if one's writing isn't read by someone else "it is wasted." Ideally, someone should give the writer some "feedback," ranging from a parental pat-on-the-head to an informed, detailed critique.

A cursory look at mass education illustrates the difficulties associated with going to the expense and trouble of making this commitment to paying attention to what students write. Most LD messages may be potentially "long in duration" but they are extremely brief in nature. Indeed, multiple-choice markings and "short-answers" are the rule, not the exception in institutions devoted to mass education. Society simply does not supply the resources for converting frequent, voluminous, lengthy LD messages into SD form by teachers. Thus learning

is characterized by either *talk* (with students mostly receiving SD messages and occasionally sending some) or *reading* (conversion of LD messages to SD by students). The net effect of this is for the student to spend most of his/her time in a "decoding" or passive mode.

It is into this "passive at home due to television" and "passive at school due to the nature of mass education in an information glutted world" environment that the electronics industry has introduced micro-computers. Given the approach I have been taking here, it should come as no surprise that I think of micro-computers as devices, like all computers, that allow their operators not only to write them LD messages, but devices that will really "pay attention" and "do something with" those messages. Indeed, computers are, essentially, very active LD message environments.

For the first time in history it is becoming economical for children to grow up in an environment that allows them to experience to a much fuller extent than before what it means to be actively literate—to actually deal extensively with the encoding of LD messages and the consequences thereof rather than just the decoding of LD messages. The sense of "power" that the phenomenon of literacy gives to a society, the micro-computer gives to the individual. You send the computer a series of messages that have implications for one another for a long time to come, that interact with messages that have been sent by other people at other times in other places (e.g., the programmers of the language being used), and something "happens." You are held very precisely accountable for what you have said in the past; things combine in both predictable and not-so-predictable ways; you can study your messages and alter them; you recognize that you are issuing "commands." In short, you begin to actually experience the consequences of "writing," what it really means to "be literate"—as contrasted to what it means to "communicate" on a short-term basis or to be a mere spectator in a literate society.

The "power" that I claim this implies is dramatically illustrated by Joseph Weizenbaum's description of the "compulsive programmer" in *Computer Power and Human Reason*. The "compulsive" programmer, as contrasted to the "professional" programmer, derives his satisfaction from "having bent a computer to his will," not from "having solved a substantive problem."³ "The computer challenges his power, not his knowledge."⁴ "He seeks reassurance from the computer, not pleasure."⁵

I am seeking to contrast the sense of *impotence* that is likely to accompany an awareness of all the LD messages that are "out there" to be found, organized, and made relevant (the position one is likely to be in if one assumes a passive, decoding posture vis-a-vis all the LD messages of the world) with the sense of *power* that an active, encoding posture encourages. Clearly Weizenbaum's "compulsive programmer" stands at one end of a continuum and serves to exaggerate the person's plight. I mean to compare him/her to the intensely "literate" person in the conventional sense of the word, a person who also can mistake the real world for the symbolic world that s/he lives in, complete with the "delusions of grandeur" that accompany his/her sojourns in quest of the "grand system," behavior which Weizenbaum also attributes to the "compulsive programmer."⁶

If my only goal were to highlight the relationship between literacy and the computer, when the latter is seen as an environment that accepts, manipulates, uses, abuses, and comments upon one's LD messages, allowing the student to become either a little more active and powerful—or at the other extreme “compulsively” active and powerful—I think the goal would be an easy one to meet. The larger problem is to gain some perspective on *the limits of this particular kind of LD messaging activity*. I value the feelings of efficacy, power, associated with what has been said so far, and I do not particularly fear the excesses suggested by the extreme illustration of the “compulsive programmer.” Anything, if taken to excess, needs attention and so forth. . . .

What I would like to understand better is how children who become active in the LD environments of computers will differ from those who do manage to participate “actively” in more conventional LD environments. What is different about writing a computer program that interacts cleverly with the programs of others and writing a clever essay on *The Odyssey*? If there is a significant difference, and I suspect there is, then the matter is very important because it is going to be so much easier and economical to get children, in school and out (with all the attendant benefits of learning in an environment of one's choice), to become “actively literate” vis-à-vis computers, than to become “actively literate” in a conventional manner. I am asserting that more people are likely to become more actively “literate” because of the opportunities presented by computers, but that there may be a cost associated with this in terms of the deflection of people away from other styles of “literacy.” Not only are electronics infiltrating all aspects of society, thereby changing what it means for a society to be literate; there may also be a way in which an individual is likely to be initiated into the intricacies of literacy that is “different” because of the availability of microcomputers at certain points in their childhood and youth.

I submit that we might not only be changing the nature of the dart board at which we are teaching people to aim (literacy), we are changing the nature of the darts (the means of

achieving literacy)—and good “new” darts are getting very, very inexpensive compared to good “old” darts. How will the use of the “new darts” change the nature of the game?

Earlier I claimed that, “For the first time in history it is becoming economical for children to grow up in an environment which allows them to experience, *to a much fuller extent than before*, what it means to be literate. . . .” (underlining added at this point). By this I meant that they would find it much more attractive, interesting, and rewarding to “write” rather than just to “read,” to encode LD messages rather than just decode them. The challenge now is to try to figure out if there are some dimensions of “fullness” that are restricted rather than enhanced by “writing” in the modality of modern computers, and if so, what could and should be done about it.

NOTES

1. My initial exposure to the concept of “SD” and “LD” messages came as a result of a collaboration with Laurence B. Heilprin in 1965. A paper entitled “An Analogy Between Information Retrieval and Education” appeared as a challenge paper in the September, 1965, American Documentation Institute Symposium on Education for Information Science. It was published both in the Proceedings of that Symposium (Washington, D.C.: Spartan Books) and in *American Documentation*, Vol. 16, No. 3, July, 1965. Heilprin's use of the SD and LD notation involved only short-duration and long-duration messages. The idea that distance is involved as well occurred to me in reading the *Encyclopaedia Britannica*'s account of “Forms of Writing” (15th edition, v. 19, pp. 1033-45). I am also indebted to that *Encyclopaedia Britannica* article for clarifying the relationship between “pre-literate” and “illiterate.”
2. The 19th century architect of the American “common school,” Horace Mann, is probably familiar to most readers. Freire, a contemporary radical Brazilian educator, has argued brilliantly for the need to connect the ability to read and write with a person's political awareness. See either his *Pedagogy of the Oppressed* or *Education for Critical Consciousness*.
3. Joseph Weizenbaum, *Computer Power and Human Reason* (San Francisco: W. H. Freeman and Co., 1976), p. 117.
4. *Ibid.*, p. 119.
5. *Ibid.*, p. 121.
6. *Ibid.*, p. 118.

COMPUTERS AT WORK

Keeping CAI humane in the humanities

by HELEN J. SCHWARTZ

Oakland University
Rochester, Michigan

ABSTRACT

Existing courseware in the humanities includes text feedback (such as readability formulas), drill and practice (such as grammar and punctuation drill), and tutorials (including tutoring in composition and poetry interpretation). Sample programs show limitations in hardware and the state of the art in programming and rhetoric can cause problems unless the instructor announces such limitations and uses them as a basis for instruction beyond the capabilities of the courseware. Tutorials should be open-ended and thoughtful in anticipating students' responses. In summary, CAI for the humanities should be designed to respect and build on the students' responses in order to serve their needs.

INTRODUCTION

The potential of computer-aided instruction (CAI) for the humanities has become clearer as computer programs have actually been written and used. Over 120 programs are listed in the most complete bibliography available, Anastasia Wang's *Index to Computer-Based Learning* (1978), and even this listing is incomplete and scheduled for a new edition.¹ Now, in the early stages of development, we need to think about what the computer can do *well* in light of subject matter in the humanities and in accord with humanistic teaching methods. Otherwise, the computer will not be a kindly genie or mentor, but a monster which, as poet Howard Nemerov warns, can brutalize the mind to the level of the machine.²

This article gives a progress report, reviewing some typical programs to show what kinds of computer applications exist in the humanities and to see the capabilities, limitations and dangers. Based on this review, I'll suggest some questions for evaluating CAI programs and their integration in an educational setting.

Let's look first at some examples of the three kinds of CAI available at present: (1) text feedback, (2) drill and practice, and (3) tutorials.³

TEXT FEEDBACK

Computerized text analysis can sometimes see surprising and problematic configurations in writing that a writer can miss

(even when s/he knows what s/he's supposed to do). But ultimately it is not the machine but the writers who must proceed after text analysis to the human hammering out of meaning.

One simple example of such a computer application is a program of a "readability formula." The user types in his/her text, and the computer calculates the reading difficulty of the passage according to the scale of the formula. Several such formulas have been computerized, and the programs are easy to use and easy to get.⁴

Such formulas are roughly reliable in flagging problems at the extremes—"Dick and Jane" prose and gobbledygook. You'd think that just *knowing* the formula, based usually on average sentence length and word familiarity, would help the writer. But sometimes writers are influenced by the pride of creation. For example, when I checked part of a paper I was writing, I realized from computer feedback that the style (18th grade level!) was inappropriate for a speech—and I revised accordingly.

But text feedback must be joined to human judgment. Experts agree that a readability score doesn't measure comprehensibility, because the formulas cannot test for sense (or nonsense!), for grammar or organization.⁵ If the student has mastered sentence structure and punctuation, and s/he is willing to type and run the program, then a computerized readability formula can suggest whether revision is badly needed. But human judgment and response are crucial to help the student improve *comprehensibility* along with *readability*.⁶

DRILL AND PRACTICE

Computer programs that drill students can provide patient, individualized, though rather impersonal, instruction. But the teacher must supervise drill and practice and carry instruction beyond CAI's capacities, or CAI drill's inherent limitations will be harmful.

One such program involving grammar and punctuation drill is available through the Engineering School's Department of Humanities at the University of Michigan. This program illustrates several good features possible in CAI drill:

1. The student controls the process by selecting modules, calling for help or explanations, and choosing when to stop.

2. The number and difficulty of problems are geared to the individual student's pace in mastering principles.
3. The program's form is effective for learning, using fill-in-the-blanks questions drawn randomly from a large data base.

CAI drill can work well (even after the novelty fades) because it can give the student a sense of mastery. But it's important to be aware of its limitations in providing that sense of mastery. Ideally, the program would evaluate an answer to see if it is acceptable usage. But currently two problems prevent a program from doing this: first, the computer system usually has space limitations which restrict what the program *may* do; second, the state of the art limits what the program *can* do.

According to Professor Leslie Olsen, the administrator of the program, the need for keeping the program to a size reasonable for small or space-cramped systems led to an important programming decision: the program accepts only one answer as "right" and responds to all other answers as "wrong." Yet for many sentences, there are several acceptable usages. To get around this problem, the instructions tell students to change a sentence presented to them only if it is *wrong*, but to mark acceptable uses (even if not preferred) as "ok." For example, here's a sample sentence on using commas in a series: "The Iroquois Indians, the Dutch and the English fought the French." If you put a comma before the "and," the program would say "Too bad" and explain, ending with the announcement that "a comma before the 'and' is optional." So your acceptable answer would be acknowledged, but in the computer's calculation of your progress you're marked wrong and get additional problems.

Now, technically, you *would* be wrong, since the instructions say to make a change *only* if the form of the sentence given is unacceptable. But notice how this programming decision subtly affects the educational goal of the program. The program cannot really promote a student's personal writing style by answering the question, "Is the *student's* preferred choice acceptable? Instead, it tests for editorial skills: "Is the *given* usage acceptable?"

Here the teacher must take over to help the student develop a consistent personal style and choose the best of acceptable usages in the larger context of a paragraph and essay. Such contextual and individualized editing is beyond the courseware's capability.

Another problem arises from limitations on space. At times, the introductory explanations include material that is missing from responses to student answers. For example in the section on agreement of pronoun and antecedent, the introductory explanation raises the issue of sexism in language, but the explanatory responses do not include this information:

EXAMPLE: "Every child who passes their test will be given a prize."

"RIGHT" ANSWER: his

EXPLANATION GIVEN WITH ALL OTHER

ANSWERS: Too bad.

"Every child" is singular and any pronoun referring to "every child" must be singular. Therefore, *you must use the singular pronoun "his" [my italics]* in

"his test." "Every child who passes his test will be given a prize."

Notice that the explanation *mandates* ("must") the use of "his" and doesn't even *mention* the alternate forms—"his or her," "his/her," "her"—explained in the introduction to this module. Now, remember, the program can accept only one "right" answer. "Their" is clearly wrong and should be changed. But "his/her" is judged wrong, despite pronouncements of publishing houses and professional groups prescribing non-sexist language. Thus, space limitations lead to truncated explanations and right/wrong decisions about input that can frustrate or mislead students if the teacher does not use the program well.

Here's where the integration of CAI into the class as a whole can turn the limitations of the program into assets for education. Professor Olsen suggests three ways to increase the effectiveness of the program:

1. Alert the students to the limitations of the program and use these limits as a starting point for discussing questions of style and contextual editing.
2. Make the use of CAI pay off for the student. Make clear that a student's improvement in grammar on assignments will improve his/her grade.
3. Give CAI a social dimension by encouraging group work at terminal sessions. This encourages peer-tutoring, is more fun, and cuts waiting time. It can also dispel frustrations when the program judges an acceptable answer as "wrong."

In this way, students are motivated to learn and also realize that they must ultimately pass beyond the lessons of the computer program. The student in consultation with the instructor or with peers makes the final judgment—thus having people take responsibility for what they do better than the computer does.

TUTORIALS

While drill and practice can lead a student to master *skills*, a good tutorial program can help him/her to internalize a *process* of learning. Such a tutorial program can guide a student to discover his/her own original ideas and then refine them using a generalizable disciplinary approach. The programs do *not* depend on the programmer anticipating every "right" answer or "right" approach by the user. For example, James Garson and Paul Mellama have developed EMIL, a tutorial for helping a student construct formal proofs in logic.⁷ And Ellen Nold has led the way in devising creative, open-ended programs in composition.⁸ Such mentor programs can provide an "Open Sesame" to the adventuring student.

For example, three programs developed by Hugh Burns stimulate rhetorical invention in composition according to three strategies involving (1) Aristotelian topics, (2) Burke's dramatic pentad, or (3) the tagmemic matrix. The tutor provides the *method* through questions; the student provides *content* with his/her responses.

After a student typed in his name and a 2-3 word summary

Is the program "polite" in the tone of its responses?

2. Does the program respect and promote the originality and individuality of its user?

Does it anticipate the questions, problems and input of the user in a useful way?

Does it know and *announce* its limitations (in the program or in supplementary material)?

Does it allow the student a safe "playground" for developing skills or ideas—without grades or punishment?

3. Is the program easy to use?

Is it *fun*?—not boring, not cutesy?

Is the amount of training the user needs to operate the machine and the program within an acceptable limit for your purpose?

Are there enough terminals to make use possible at times and places convenient for the student?

Is the computer system reliable (with minimal "down" time)?

Is there technical assistance available to the student for using the program and coping with any malfunction?¹²

4. Is the program integrated in an educational setting to "support the essential social character of human learning" and motivation?¹³

Does use of the program involve or lead to student-teacher or student-peer interaction?

Does the student perceive that the program develops skills that will gain him/her recognition?

5. Is the student involved in evaluating and modifying the program and its integration into the educational setting?

Is there a channel of communication the student can use to register complaints, problems, suggestions, reactions?

Does the student perceive that s/he can suggest modifications or develop new applications?¹⁴

This last question suggests that educators who use CAI, as well as their students, will want to be able to develop and modify CAI programs. And this is as it should be if we are to develop the best potential of CAI. If computer programs are to be mechanical mentors, and not monsters, they must be developed and tested by human mentors experienced in

teaching and in the unique qualities and procedures of their discipline.

NOTES

1. The 1981 edition of Wang's *Index* can be ordered from Instructional Media Laboratory of the University of Wisconsin in Milwaukee.
2. Nemerov, H., "Speculative equations: Poems, poets, computers," *American Scholar*, 36 (1966-67), 414.
3. For a fuller discussion, see K.M. Jaycox, *Computer Applications in the Teaching of English*, Illinois Series on Educational Applications of Computers, 19e (Urbana, Ill.: University of Illinois, 1979).
4. The sources for several formulas are listed in G.R. Klare, "Assessing Readability," *Reading Research Quarterly*, 10 (1974-75), 87-91. And two programs written in BASIC for microcomputers are reproduced in the April 1980 issue of *Creative Computing*: D. Goodman and S. Schwab, "Computerized Testing for Readability;" and R. Carlson, "Reading Level Difficulty."
5. Gunning, R. *The Technique of Clear Writing*. N.Y.: McGraw-Hill, 1952, p. 13; Klare, G.R. *The Measurement of Readability*. Ames, Iowa: Iowa State University Press, 1963, pp. 24-25; Redish, J. *Readability*. Washington, D.C.: Document Design Center, 1979.
6. See H.J. Schwartz, "Fighting Gobbledegook in Technical Writing with Computer Magic: A Preliminary Study," Annual College English Association Conference, April 1980; and H.J. Schwartz, "Teaching Stylistic Simplicity with a Computerized Readability Formula," International Convention of the American Business Communications Association, December 1980.
7. Garson, J.W. "Giving Advice with a Computer," *Proceedings of the National Educational Computing Conference*, June 1980, pp. 42-45.
8. Nold, E.W. "Fear and Trembling: The Humanist Approaches the Computer." *College Composition and Communication*, 26 (October 1975), 269-273.
9. Burns, H.L. and G.H. Culp. "Stimulating Invention in English Composition through Computer-Assisted Instruction." *Educational Technology*, 20 (August 1980), p. 8.
10. Burns and Culp, 7.
11. In *Collected Poems* (New York: Harcourt, Brace, 1938), no. 31.
12. For a more technical rating mechanism, see Appendix A in J.R. Dennis, *Evaluating Materials for Teaching with a Computer*, Illinois Series on Educational Applications of Computers, 5e (Urbana: University of Illinois, 1979).
13. Dwyer, T.A. "Some Principles for the Human Use of Computers in Education." *International Journal of Man-Machine Studies*, 3 (July 1971), 221-222.
14. Several experts note that the problem analysis involved in writing a program may provide a greater learning experience than the running of the program: M. Masterman, "Computerized haiku," in J. Reichardt, ed., *Cybernetics, Art and Ideas* (Greenwich, Conn.: N.Y. Graphic Society, [1971]), p. 183; M. Critchfield, "Beyond CAI: Computers as Personal Intellectual Tools," *Educational Technology*, 19 (October 1979), 10, p. 24.

The effects of computers on library staff and users: How can the administrator cope?

by RICHARD W. BOSS

Information Systems Consultants Inc.
Bethesda, Maryland

INTRODUCTION

A library normally prepares for the automation of one or more functions by undertaking systematic technological and economic planning. A set of specifications is normally drawn up, setting forth the functional requirements for the system, whether that system is to be a "turnkey system" (one supplied by a vendor who provides hardware, software, installation, training and maintenance—some 85% of libraries choose this course of action), a custom development by a systems house, or an in-house effort. The budget will be drawn up at least several months in advance, setting forth the anticipated costs. While most library cost projections tend to be too conservative, the effort is nevertheless made. The area of automation planning that is most often neglected is that of the psychological impact of the new system on the library's staff and users.

Automation brings with it many negative side effects. Among them are

1. *Strain on resources.* A library may realize too late that it lacks the human and financial resources to see the project through to its conclusion. There can then be painful reallocations. If these are not made and the project is abandoned, there may be recriminations among the staff, library administration, and higher administrative authority.
2. *Fear of job loss.* Automation is associated with staff reduction in the minds of many individuals. Insecurity among employees often runs high, particularly among those who do not regard themselves as mobile or who have limited marketable skills.
3. *Fear of changes in duties.* Even in organizations where there is no fear of job loss there may be fear about duties being changed to ones that will be less liked. Underlying that fear is often an even more serious, unspoken one that the individual may not be able to perform the new duties as well as the old.
4. *Organizational changes.* One of the effects of automation least frequently mentioned is the impact on organizational structure. The departments in a library have historically developed around particular functions and files. When all departments are tied to on-line files, the

interrelatedness among departments increases and the ability of a single department to control access to particular files comes to an end.

5. *Patron criticism.* Libraries are service-oriented and sensitive to the criticism of their users. New technologies often have their vocal critics who decry the new approach as "dehumanizing."
6. *System breakdown.* Computer systems can normally be expected to be "down" or non-operational from 1 to 4 percent of the time. When this occurs in the case of a public service function such as circulation during a busy period, criticism can be swift and sharp. The staff, not the library administration, bears the brunt of it. Their response to the criticism may worsen, rather than improve the situation.

These are but some of the risks associated with the introduction of a complex technology. Yet library administrators continue to acquire automated systems. And it is well that they do so, for as Tom Galvin of the University of Pittsburgh has said,

"The manager who makes fewer mistakes may actually be declining in managerial effectiveness and sidestepping or avoiding the very administrative and supervisory responsibilities which are, or ought to be, the essential content of his or her work."¹

There are risks and they should be accepted, if the purposes for automation are sound. There can be significant benefits: work can become more satisfying for library staff and service can improve for library users.

A great deal of librarianship is clerical and repetitive, even for professional librarians. Checking books in and out, preparing overdue notices, and handling reserves or holds are all mundane tasks that require a friendly, alert staff member, but ideally one with a powerful tool such as an automated circulation control system. Cataloging is costly, time-consuming and, until the on-line shared cataloging concept was developed, duplicated all over the country. Literature searching through dozens of bound and paperback indexes and abstracts can be agonizing. Librarians who have become accomplished searchers of on-line bibliographic data bases hate to do man-

ual searching. With automation, work can become more substantive.

Library users should be the principal beneficiaries of library automation. They should be able to check books out faster, inquire about the status of a book and get a reliable answer, and have the library duplicate high-demand books promptly. They should be able to find books on the shelves sooner as the result of shared on-line cataloging. Literature searches should provide them with more suitable references and the assurance of evenness throughout the period searched.

The problem that library administrators face is that it may be a year or more from the time of the decision to automate before the benefits of automation become apparent. What does one do in the interim? How does one plan automation so that those who may be affected, positively or negatively as they might see it, will support the automation effort—or at least, withhold their opposition—during the planning and implementation period?

There are several things which can be done to improve staff reaction to a planned introduction of automation:

- *Participation:* The involvement of those most affected by the new system may gain understanding and commitment.
- *Orientation:* Carefully prepared presentations comparing the planned with the existing system and explaining the pros and cons of each.
- *Demonstrations:* In-house demonstrations, with hands-on experimentation allowed may be successful if the terminals to the proposed system are “user-cordial” (that term will be discussed later).
- *Site visits:* Visits to other libraries that have successfully installed the same or a similar system may be more effective than in-house demonstrations, especially if time is provided for individuals to talk with their counterparts at the other institutions.
- *Handouts:* Clear, concise written materials might be distributed to the staff. A letter from the director is a good format.
- *Reassurances:* If the library is able to tell staff that no one will be terminated, written or oral statements of reassurance to that effect should be made.
- *Neutralizing critics:* Special efforts could be made to involve, orient, and reassure those who work around a particularly negative person.
- *Reassignment:* A person who appears to be unalterably opposed and who might adversely affect the program might be transferred to another position that is equally attractive, but less sensitive.
- *Gradual implementation:* Pilot installations or function-by-function implementation may be undertaken to establish the performance and benefits of the system.

The critical test will come when the system is installed. Its ease or lack of ease is usually the biggest single factor in acceptance or rejection. As Mooers' Law says, “An information retrieval system will tend not to be used whenever it is more painful and troublesome for a customer to have information than for him not to have it. A terminal is the computer to most users. Resistance to it means resistance to the system.”

What too many purchasers overlook is the terminal. Virtually all terminals in use in libraries are standard off-the-shelf terminals with few if any features to make them easily acceptable to a user who is less than fully-trained and experienced. On occasion a vendor is sensitive to the issue of user cordiality and has developed software to prompt the user through a series of multiple-choice steps. That has solved the problem of the first-time user, but it created an equally serious one for the skilled operator. Just how many times a day can a person stand to have a computer flash the message, “Hello, how are you? Which of the following would you like to do?”

User cordiality has to be defined as responsiveness to the user at his/her level of ability and confidence. It should be possible for the skilled operator to override the prompting mode and “command” the system.

The Lister Hill Center for Biomedical Communications, the research and development arm of the National Library of Medicine, has been the only library agency that has systematically studied user cordiality. They have sought to eliminate the need for a terminal operator to know the correct method for entering a query—where the blanks go, where the commas are to be placed, and whether periods are needed after initials. They have sought to complement the simplicity of design with a series of instructions that are displayed as needed. At the same time, they have tried to provide the means for the skilled operator to bypass the instructions. These efforts are part of the development of a conceptual library information system that will include acquisitions, cataloging, circulation, and several other functions necessary to a fully integrated system. The software documentation is available from NTIS for approximately \$2,000.

Lister Hill has not been alone in its concern, however. Greater user cordiality is being incorporated into many of the commercially vended systems. The on-line catalog terminal of CLSI is one of the simplest terminals of all. It is a touch terminal that merely requires the user to respond to options by pressing the appropriate part of the screen. Ambiguities in the instructions are still being worked out, but the terminal already is a delight for the person who has never used a computer terminal before. The company recommends this terminal for use by library patrons, with a more flexible terminal for staff.

Even a system with user cordiality built in is not truly simple. Considerable training is required for the performance of any other than the most basic functions. The specifications of many libraries stipulate that the vendor is to train the library staff. We normally recommend that the number of staff trained by the vendor be limited both to reduce training costs and to assure the library of an ongoing capability for training new staff. The specifications should stipulate that those trained by the vendor shall be able to train others.

It is highly desirable to have an orientation program for the entire staff shortly after the system is installed. A representative of the vendor should demonstrate the system, ideally with opportunity for some hands-on use by staff. It is even more important that a member of the library's administration explain again why the library is automating circulation. Our experience has been that staff fears are most often based on unclear objectives. Our interviews suggest that administrative staff share a common perception that improvement of service,

not staff reduction, is the reason for automating. That should be stressed.

We recommend a carefully planned program to tell patrons of the automation plans. Again, the objectives should be stressed. We recommend that several of the following approaches be selected.³ All have been successfully used by libraries throughout the country. It is our experience that the libraries which have not undertaken such systematic orientation have experienced patron resistance.

- *Newspaper stories:* Articles in a newspaper describing a library automation program are the most common form of promotion. Success has been mixed, however. A press release carefully setting forth what the library wants to say is a must. Unplanned interviews often lead to emphasis of the high cost, the difficulties of making the system work, etc.
- *Leaflet:* A handout describing the why, how, and when of a new system is inexpensive and usually effective. The leaflet should emphasize the prospective benefits to the library user, but should also set forth the possible disadvantages, especially possible short-term reliability problems.
- *Radio and television:* A few libraries have arranged interviews or demonstrations on radio or television. This is particularly effective when the library representative has a good media personality.
- *Displays:* Displays in the library or in another high-traffic area can be very positive promotions. Simple explanations of how the computer works are of particular interest. Photographs of the various components of the system in use are inexpensive and often equally useful for newspaper stories.
- *Support groups:* The friends of the library, whether a formally organized group or not, can be an important source of support from the earliest planning stage through implementation. They often are heterogeneous groups of people who will pose the questions that are on the minds of library users generally.
- *Contacts with opinion leaders:* The library administration have usually identified the people who are most influential in their community, company, or academic institu-

tion. Personal letters and personal contacts are a good way of informing these individuals and soliciting their opinions. The timing of these contacts is critical. It should be before a decision is reached, before a newspaper story appears, and before the system is implemented. One library had a very successful open house by invitation only shortly before launching its system. Those invited were all those "who had provided advice during the course of the planning." The open house and the demonstrations were reported in the local press.

- *Remedial activity:* Even the best of systems will frustrate some library users at times, by any action from losing a record temporarily to tying two different ones together. A phone call to someone who has had a bad experience with the library's new automated system can keep the library's administration and staff alert to progress and also possibly avoid a broadcast of the bad experience by the affected library user. Staff should be instructed to record the names of those who were inconvenienced so that follow-up can be made.

These techniques are already being used by many libraries as part of their ongoing efforts to promote the library as a whole. The higher risks associated with the introduction of automated library systems warrant these efforts by all libraries.

All of these comments assume that the choice of an automated system has been made wisely. No amount of special effort can long conceal a poor choice—poor because automation was not really the solution to the problem posed or because an inadequate system was selected in an effort to hold costs down. Good choices do not sell themselves, however; there has to be some special effort.

REFERENCES

1. Galvin, Thomas J. "Management in Interesting Times," keynote address at the Army Library Institute, El Paso, Texas, May 22, 1978.
2. Mooers, C. N. "Mooers' Law on Why Some Retrieval Systems are Used and Others are Not," *American Documentation* vol. 11, 1960, p. 204.
3. Boss, Richard W. *Library Manager's Guide to Automation*. Knowledge Industries Publications, Inc., 1979, pp. 80-81.

Libraries as local database producers

by ROBIN CRICKMAN

University of Minnesota
Minneapolis, Minnesota

Sale of citations retrieved from bibliographic database services began more than a decade ago. Data as well as literature citations were later made available from the same companies. Recently, other information vendors have entered the market and offered databases on such things as automobiles for sale or barter exchange of skills.

Information offerings aimed at the consumer market are beginning to be common. Perhaps the best-known service aimed at the general public is available from The Source, which sells access to databases on airline schedules and social amenities such as cultural events and restaurants in major American cities. The Europeans and the Canadians are actively developing systems designed to provide access to a variety of rapidly changing information, such as weather, stock prices, news, and cultural events. The individual consumer using the foreign system needs only a simple terminal to access whatever information is of interest. A small fee is charged for each use.

At a different level, the coming of the microcomputer and the development in home hobby computing has meant that many people now have at least limited capability not only to consume information but also to provide the organization for its storage and retrieval. The technology is present. These hobbyists are learning database collection and management as rapidly as possible. Bulletin board systems that run on modest microcomputers are a significant service activity for many a hobby computer club.

One seeks in vain, however, for an online people's information system. Nowhere can the general public find easy-to-operate systems at a very low cost that provide information about the community rather than a mechanism by which individuals can exchange information of interest to them. Until recently, of course, such an endeavor would have been too expensive. In addition, few individuals in the community had any familiarity with computer equipment. Until there are users for a database, there is very little reason to create one. Now some people have equipment in their homes that is able to access a community information system. More people will probably own such equipment as time goes by. And there are literally thousands of high school students who encounter small computer terminals in their classrooms. The next generation may well use terminals as easily as most of the current one uses television sets.

There is sufficient skill and equipment in the population that databases, if offered, would be used by at least part of the

community. What information might these databases provide? One possibility is the availability and quality of locally provided goods and services. Organizations such as Consumers' Union, the publisher of *Consumers' Reports*, provide one source of unbiased information on consumer purchasing decisions. Their evaluations are available for much nationally marketed merchandise. When a service or product is produced and sold locally, however, unbiased information on availability and quality of the offered good or service can be difficult to obtain—for example, many find trying to select a high-quality physician difficult. General guidelines can be found in consumer publications, but exactly which plumber or auto repair shop delivers high-quality service at reasonable prices is information available mostly from personal advising through one's network of acquaintances. A database that combines information on availability and quality of goods and services, then, should be of considerable community interest.

Another database that might be useful would be one offering locally significant, ephemeral information. One example of such information is prices in local supermarkets. Many people would find it useful to list their weekly grocery items and have a computer report which store or stores offer the best prices on those items. Such a service would be especially helpful to persons of limited income with a large number of household members to feed.

A different type of ephemeral information is related to social and cultural activities taking place within the community. The occurrence and availability of some of these activities can be announced through the usual channels of information, such as newspapers, radio, and television. Other organizations sponsoring activities have more limited interests and treasuries and must depend on free or low-cost channels of information, such as public service announcements. Still other groups publicize their meetings only to current members—as often because the publicity effort is excessive for the number of outsiders the group would attract as because they desire to maintain an exclusive membership. A database on local events would provide access to both heavily attended activities and those of very limited interest. It would allow a new bicycle enthusiast in town to find when and where the bicycle club(s) meet. It would also give anyone in the community a resource to consult when that person wishes to select an activity. A community database on local events is particularly likely to succeed if some organization can offer the hardware and soft-

ware to allow access. The one thing that most local groups have in abundance is volunteer labor to assist in input and maintenance of database entries. The thing they have in short supply is skill in creating the software and funds to support the purchase of the hardware. A communal effort might provide sufficient funding for several community organizations together to purchase a system, but some mechanism for coordinating the database entries would still be necessary.

Who should be providing databases of interest to a local community? Commercial services that want entry into more limited markets than the nationally vended databases are a logical possibility. Some local information has been sold by traditional media such as newspapers for many years now. Much of the information that might logically be part of a community information system is not readily amenable to for-profit providers. There are several reasons for this. The first is the collection of the information: hiring individuals to do this would be fairly expensive, and so would entering the information into the database. However, if volunteers were willing to collect the information as part of other activities (such as their weekly shopping trip) and enter it into the system, the effort could be modest. Further, personal evaluations of locally provided goods and services could be sought from anyone in the community who has used a service or patronized a shop and used for another database. Such a process would require that the appearance of unbiased evaluation be maintained if the database were to be credible to the community. A for-profit organization selling evaluations might have a difficult time gathering consumer opinion and a more difficult time convincing the public that the information gathered was not changed in any inappropriate way.

A second problem with community information services is that those who are most in need of the information are not those with the resources to pay for the information. For example, information about grocery purchases is likely to be most valuable to the poorest elements of the population; and those with the most time for social and cultural events are those who are not in the labor force, because they are too young, retired, or temporarily unemployed. These individuals often have limited resources, and the organizations that would benefit from their participation can afford only very limited expenditures to attract them.

A community organization would be a sensible place to coordinate and house the community information system. It should probably be a nonprofit organization, with the ability and resources to support such a system. I would like to suggest that the public library is a reasonable choice. It has a number of advantages over other public agencies.

Let us start with technical aspects. Although it is true that the public library is no more likely than the public school or the welfare office to know a great deal about computer hardware and software, the librarian does know a great deal about the storage and retrieval of information; they are one of the most important aspects of a librarian's work. Thus, the library is one place to find expertise in local demand for information and knowledge of how best to organize it for ready retrieval in the context of local interests and approaches. Further, librarians have considerable experience in helping people locate information. This combination of skills in the creation of database and the use of the information placed in it would

make librarians and the library a strong technical base for the location of this system.

Equally important is an appropriate social climate for the system. Here also the library has distinct advantages over other organizations. The library is an agency that is not usually seen as delivering service to any particular segment of the population. It does not suffer from the age bias of the public school. It does not have to overcome any stigma to its use, as the welfare office might. It is not type-cast, as the Red Cross or the Salvation Army might be. Using a service provided by the public library would neither make the poor feel stigmatized nor make the wealthier feel that they are using a resource that should be kept for those who cannot afford to pay for it. The traditions of the library support use by both poor and wealthy.

The library also has a tradition of volunteer assistance to the provision of its services. Every library has its "Friends" organization, and there is considerable skill in some libraries in the coordination of volunteer labor. Thus the library can readily use volunteer assistance to build a community database and to keep it accurate and up-to-date.

The library has a third important strength in its tradition of full and free access to information. To have a truly useful database, someone is going to have to exercise considerable skill in deciding what will be in the database and what will not. This form of selection must not be allowed to turn into censorship, or the value of the database to the community will be defeated. Librarians begin learning early in their professional training the distinction between selection and censorship. Public libraries in America have a long tradition of being an unbiased source of information. That tradition, coupled with the skills in which librarians are trained, will be of considerable value in the maintenance of a community information system. Without some selection, evaluation of services could degenerate into a vituperative attack on a service or product provider by a few disgruntled individuals or be so bland as to provide no valuable information whatsoever.

There are still challenges to be addressed before such a service can become a reality in a library. While libraries often have a tradition of active volunteers who support their program, few libraries have the experience to organize and coordinate a volunteer effort to gather and tabulate information on the scale that a community database would require, even if the information generators assist by supplying some of the information.

Then there is the need for algorithms to consolidate opinion data on services. How can an unbiased evaluation be prepared? Can the computer's technical abilities contribute? Is there some means to aggregate many people's opinion in a routine fashion, so that it is not necessary to read a large number of comments on a product or service to determine whether it has been satisfactory? What about the differences in values among the members of the community? What may be high quality to one person may be unacceptable to another; what is a friendly attitude to one might be overbearing to another. Just how are the disparate value structures behind the evaluations to be reconciled? How can the individuals in a community be persuaded to enter their evaluations of services or goods?

What is to be done about people who want to interrogate

the database but do not have their own terminals? Should they call and have another person do the search for them? Would it be acceptable to expect them to present themselves at the database site?

There will be many questions to resolve before community

information systems become part of the expected resources of our nation's towns and villages. The resources are there, human as well as computer. Those of you who also see this as an idea whose time has come can readily contribute willingness to solve the problems and organizational effort.

Data files as library materials: policies, procedures, and politics*

by RICHARD C. ROISTACHER

*Bureau of Social Science Research
Washington, D.C.*

ABSTRACT

Increasingly, organizational and public policies call for machine-readable data files to be treated as publicly available archived material. Data producers will be required to produce public-use files and documentation as well as the printed reports and on-line access they presently provide. An archivable machine-readable data file (MRDF) consists of three objects: The data file itself (either with or without a machine-readable dictionary), a set of documentation materials describing the file to the secondary user, and a set of bibliographic materials which lead the secondary user to the documentation and the file. Computing and data centers know how to produce data files. They can develop procedures for documenting files and producing bibliographic identification materials. By developing working relationships with technical libraries, data centers can relieve themselves of bibliographic, clerical, and financial burdens associated with the maintenance of a data archive. This paper outlines some procedures for the documentation and bibliographic identification of machine-readable data files. Some strategies for developing working relationships with technical libraries are suggested.

POLICIES

Until fairly recently, most machine-readable data files (MRDFs) were treated much like working notes. While published tables and reports are often criticized and reworked, the data files underlying the reports are not usually subject to public inspection. In the social sciences, it was discovered that copying a tape costs several orders of magnitude less than redoing a study. As a result, many large social data collections are now sent to data archives where they are made available to secondary analysts. Some studies, such as the biennial National Election Study, have no primary analyst, but are designed for dissemination through data archives to a clientele of secondary analysts.

Federal agencies are beginning a policy of making statistical data files available for public use. In addition to such statistical series as the Census Public Use Samples and the Current

Population Survey, many one-time surveys and formerly internal files are beginning to appear in public-use format. The National Technical Information Service (NTIS), which originally disseminated only hard copy and microform editions of printed materials, now disseminates an increasing number of machine-readable data files. The U. S. Department of Justice requires all of its research grantees and contractors to provide the National Criminal Justice Data Archive with public-use versions of any machine-readable data files generated in the course of research.

An increasing number of public and corporate entities are likely to adopt data access and archiving policies in order to comply with canons of public accountability, public regulation, and scholarship. As a result, data and computing centers will have to develop their own data archiving policies and procedures. The development of a data archiving policy is outside the scope of this paper. The basis of any such policy is that any data file be considered a potential resource for public use, subject to the constraints of privacy and legal confidentiality. The remainder of this paper will be an outline of some procedures for documenting public-use data and for developing a working relation with technical libraries in order to share the burdens of data archiving.

PROCEDURES

An archived machine-readable data file consists of three objects: The data file itself, either with or without a machine-readable dictionary; a user's guide describing the file to the secondary analyst; and a set of bibliographic materials that lead the secondary analyst to the file and its documentation.

The Data File

The public-use data file usually differs little if at all from the operational file from which it was produced. Where the operational file bears information on the level of the individual person, it may be necessary to delete personal identifiers from the public-use file. A small but important set of files, most of them from the Bureau of the Census, require elaborate confidentiality procedures to protect the identity of small aggregates of people or small geographic areas. Most public-use

*This work was supported by Grants No. 78-SS-AX-0028 and 79-SS-AX-0026 from the Bureau of Justice Statistics, U.S. Department of Justice.

files, however, require no more than the deletion of personal identifiers.

At this point in the discussion it is traditional to mention that public-use files should not be padded with extraneous data fields, that numeric data items should be purged of alphabetic characters, and that funny representations of numbers should be transformed to more usual forms. It is clear that the enlightened reader of this paper would never allow him/herself to labor under such a burden as an ill-formatted file, only to provide some stranger with a file more pure than the original.

The User's Guide

MRDF documentation consists of information that describes the file's identity, organization, contents, physical characteristics, and relation to computer hardware and software. Items of documentation have been called codebooks, tape layouts, data dictionaries, program manuals, etc. Roistacher¹ has suggested that the comprehensive manual be called a "user's guide." This term has been chosen for its generality, simplicity, aptness, and conformity with Federal Information Processing Standards (FIPS) for documentation.

All machine-readable data in the form of numeric or coded records to be preserved for later analysis should be documented in the same fashion. It matters little whether a file consists of responses to a questionnaire, income figures for counties, or temperatures and barometric pressures. A file's format and documentation style is affected by whether the file represents a data matrix, a tree, a hierarchy, a graph, or a multidimensional table. However, its format and documentation are not affected by the nature of the data that are stored in the matrix, tree, hierarchy, or table.

The preparation of a documentation is a task for the data producer. While an archive may polish the original documentation into a user's guide, the archive should not have to do any substantive documentation itself. In particular, there should be no need for the archive to apply to the producer for materials not provided with a file's documentation.

A user's guide has five major sections:

1. Preliminaries—bibliographic attributes, title and title page construction, pagination, and headings;
2. A history of the project that produced the MRDF, describing the evolution of the data from the point of collection until conversion to machine-readable form;
3. A summary of the MRDF's data-processing history;
4. A dictionary listing of the data items in the file (the "codebook" proper);
5. A set of appendices containing glossaries, error listings, bibliographies, and other information.

While there is much to be said about the preparation of each of the chapters in the user's guide, most of chapters 2-5 will be familiar to data processing professionals. The bibliographic information that constitutes the first section of a user's guide is essential to a file's successful dissemination, and is probably less familiar to the computing professional.

Bibliographic Identities

Both the public-use file and its documentation are designed to be library materials. While the data file itself is an unfamiliar object to most librarians, the user's guide is a publication and is the mainstay of most technical libraries. As a publication, the user's guide has a bibliographic identity. One of the main uses of the user's guide is to provide a bibliographic identity for its data file, thus converting the data file into a library item.

A machine-readable data file's bibliographic identity is provided by six kinds of information:

1. Information which *identifies* the MRDF and prepares it for integration into existing manual or automated information systems. This information consists of bibliographic elements that can be processed by librarians, converted to catalog records, and integrated into bibliographic storage and retrieval systems.
2. Information that *describes* the contents of a MRDF. This information is contained in a data abstract, which also can be automated and integrated into existing information systems.
3. Information that *classifies* a MRDF is contained in a set of descriptors or keywords that will facilitate the retrieval of a group of MRDFs on the same or similar subjects.
4. Information required to *access* a MRDF includes a description of the physical characteristics of the file and its relation to the computer hardware and software.
5. Information necessary to *analyze* the MRDF describes the data items in the file, the methods used to create the file, and the file's linkage to methodologically similar data files. This information is usually presented in a "data dictionary listing" or "codebook."
6. Information necessary to *accession* or *archive* the MRDF, to evaluate its quality, and to prepare it for future use.

The bibliographic identity of a MRDF is composed of

1. title,
2. subtitle,
3. authorship,
4. author responsibility statement,
5. edition,
6. edition responsibility statement,
7. producer, and
8. distributor.

The bibliographic identity of a user's guide, while parallel to that of its MRDF, is not the same as that of the MRDF. A terse example is shown in Figure 1, which is a sample title page from a user's guide. The title of the MRDF is "Juvenile Detention and Correctional Facility Census of 1971." The title of the user's guide is "Juvenile Detention and Correctional Facility Census of 1971: User's Guide to the Machine-Readable Data File."

The bibliographic elements in Figure 1 are

1. title,
2. subtitle (if appropriate),
3. special sponsorship or funding source,
4. producer's name,
5. date of production or collection,
6. authorship,
7. address,
8. edition statement (if appropriate) for data,
9. distributor's name, address, and telephone number,
10. acknowledgement of organization/funding source responsible for publishing the related documentation (only if different from the producer or distributor of the data file),
11. date of the documentation's publication,
12. edition statement (if appropriate) for documentation.

Bibliographic Citation

Many MRDF that have value for secondary analysis are eventually turned over to data archives, but before this happens such data files are first cited in the published literature. Certain data producers rely primarily on the practice of scholars citing their data in the various research journals as the means of publicizing the existence of such MRDF. Consequently, the initial access route for many MRDF is via the

JUVENILE DETENTION AND CORRECTIONAL FACILITY OF 1971 (1)

User's Guide for the Machine-Readable Data File (2)

Produced by

U.S. Bureau of the Census (4)
Washington, D.C.
1971 (5)

for

National Criminal Justice Information and Statistics Service (6)
Law Enforcement Assistance Administration
U.S. Department of Justice
Washington, D.C. 20537 (7)

Rev. LEAA 1975 ed. (8)

Revised by LEAA Data Archive and Research Support Center (9)
Center for Advanced Computation
University of Illinois
Urbana, IL 61801
(217) 333-3234

User's Guide Prepared by
LEAA Data Archive and Research Support Center (10)
(Under LEAA Grant 77-SS-99-6003)
December 1978 (11)

LEAA User's Guide 4th ed. (12)

Figure 1—A hypothetical title page for a user's guide

cited reference in the research literature. The bibliographic elements outlined above for the title page are also used to create a citation or end-of-work reference. For example:

1. author's full name,
2. title of data file, subtitle (if appropriate), and [material designator],*
3. statements of responsibility (if appropriate),
4. city and state (abbreviated) of the data producer,
5. name of production organization [producer],
6. date of production,
7. city and state (abbreviated) of data distributor (if appropriate),
8. name of distribution organization [distributor], and
9. notes (optional).

Item numbers refer to the bracketed numbers in the following example:

<1> U. S. Law Enforcement Assistance Administration. <2> *Juvenile Detention and Correctional Facility Census of 1971* [machine-readable data file]. <3> Conducted by the U. S. Bureau of the Census for the National Criminal Justice Information and Statistics Service, LEAA, U. S. Department of Justice. LEAA rev. 1975 ed. Revised by LEAA Data Archive and Research Support Center, University of Illinois at Urbana. <4> Washington, D.C.: <5> U. S. Bureau of the Census [producer], <6> 1971. <7> Urbana, Ill: <8> LEAA Data Archive and Research Support Center [distributor].

The Abstract

A bibliographic citation may also be used as a heading for the abstract. An abstract is an abbreviated and informative representation of the file being described. It is not intended to give information on a question-by-question level, but rather is a summary of the major subject content. Its purpose is to tell the reader whether the file might be of interest and what is involved in obtaining it or securing more information.

Components of an abstract should include

1. Unique identification numbers either for the abstract or study, if appropriate;
2. Type of file (e.g., text, numerical, graphic, program source, etc.);
3. Bibliographic citation;
4. Methodology—
 - a. source(s) of information,
 - b. chronological coverage,
 - c. universe description of target population,
 - d. type of sample,

* The material designator is used to denote the generic form or type of material being referenced and to distinguish one type of medium from another. It is always enclosed in brackets and follows immediately after the title. Brackets are used to enclose the material designator, producer, and distributor statements.

- e. instrumentation characteristics (e.g., telephone interview or mail questionnaire, and
- f. dates of data collection;
5. Summary of the major subject content—
 - a. purpose or scope of study,
 - b. special characteristics of the study,
 - c. subject matter, and
 - d. number of variables, observations, and records;
6. Geographic coverage;
7. Descriptors that express an idea or concept or phenomenon *not* covered in the body of the abstract (using terms that summarize the underlying conceptual framework of the study);
8. Technical notes—
 - a. file structure (rectangular, hierarchical, etc.),
 - b. file size,
 - c. special formats (SPSS, SAS, etc.), and
 - d. computer or software dependence;
9. Terms of availability—
 - a. condition of data (e.g., statements that edit checks have been made),
 - b. restrictions on access, if any, and
 - c. contact person or organization (full address and telephone number); and
10. Cited references to any written or published reports that were based on these data and might provide additional information for the potential user.

The following is an example of an abstract for an MRDF.

Unique identification number(s): Accession number QP-003-004-USA-1957.

Citation: *American Family Growth, 1957-1967*. [Machine-readable data file]. Principal investigators, Charles F. Westoff *et al.* DPLS ed. Edition prepared by Mary Ann Hanson, under the direction of Larry Bumpass, Center for Demography and Ecology, University of Wisconsin-Madison. Madison, Wis.: University of Wisconsin Center for Demography and Ecology [producer], 1978. Madison, Wis.: University of Wisconsin Data and Program Library Service [distributor].

Methodology. The target population was urban, native-born white couples with two children, couples whose marriages so far had been uncomplicated by death, divorce, separation, or extensive pregnancy wastage, with the second birth to have occurred during September 1956 for every couple. A probability sample, stratified by metropolitan area, was drawn from 7 SMSAs with population over 2 million (exclusive of Boston). Couples were interviewed three times in February–March 1957, 1960, and between 1963 and 1967 to determine eligibility and to complete questionnaires. Data checks and full-scale processing were run on the public use version. The final sample size is 1,165 couples; 814 couples completed all three interviews.

Summary of contents. *American Family Growth, 1957-1967* is a longitudinal study which examines the fertility history of American couples in metropolitan America and the motivational connections between the environment and fertility decisions and behavior. Phase I looks at the

social and psychological factors thought to relate to differences in fertility. Phase II focuses on why some couples stopped at two children while others had a third or fourth child during the first and second phase. Phase III examines how well attitudes and events of the early marriage determined the record of the later years of childbearing. The data file contains over 1000 variables.

Geographic coverage. United States SMSAs (New York, Indianapolis, Chicago, Los Angeles, Milwaukee, Cleveland, Minneapolis).

Descriptors. Fertility, family planning, family composition, socioeconomic status, work satisfaction, contraceptive practices, religiosity.

Technical notes. Rectangular file with 1,165 observations.

Terms of availability. Data checks and full scale processing have been performed on the public-use file. There are no restrictions on access to the public-use file. Copies of the data and documentation can be obtained by writing to the Data and Program Library Service, 4452 Social Science Building, University of Wisconsin-Madison, Madison, Wisconsin 53706 USA; telephone number: (608)262-7962.

Cited references. Principal monographs include *Family Growth in Metropolitan America* by Charles F. Westoff, Robert G. Potter, Jr., Philip C. Sagi, and Elliot G. Mishler (Princeton, NJ: Princeton University Press, 1961); *The Third Child: A Study in the Prediction of Fertility* by Charles F. Westoff, Robert G. Potter, Jr., and Philip C. Sagi (Princeton, NJ: Princeton University Press, 1963); and *The Later Years of Childbearing* by Larry Bumpass and Charles F. Westoff (Princeton, NJ: Princeton University Press, 1970).

POLITICS

By producing a user's guide, the data center has created a library item describing a library non-item. Most data centers are not data archives, nor do they wish to become data archives. One possible way to relieve a data center of much of the burden of serving as an archive is to form an alliance with the organization's technical library. Copies of the user's guide should be sent to the technical library to be accessioned as part of the technical collection. (A university library's reluctance to deal with MRDFs and user's guides was overcome when an instructor placed a number of user's guides on his course reserve list.)

The librarians should be asked to follow the catalog entry for the user's guide with a catalog entry for the data file, listing the data center rack number (or other data center locator) as the file's location. (The latest edition of the "Anglo-American Cataloging Rules" (AACR-II) contain detailed procedures for the cataloging of machine readable data files.) The paperwork for filling external requests can then be delegated to the library as part of its interlibrary loan function. Where a data center must recover costs from its users, it may be easier to establish a single internal account with the library than to initiate billing arrangements with external clients who wish to obtain copies of archival data files.

By producing comprehensive documentation, the data center can comply with data access policies without incurring a continuing burden of data consulting. By producing standard bibliographic identity information for archived data files, the data center facilitates the establishment of cooperative relations with a technical library, which is equipped to handle customers for archival materials. Such a collaboration yields

advantages for the data center, the technical library, and the data user.

REFERENCES

1. Roistacher, R.C. *A Style Manual for Machine Readable Data Files and Their Documentation*. Washington, DC: U. S. Government Printing Office, 1980.

Computerized weighted voting reapportionment

by L. PAPAYANOPOULOS

Rutgers University
Newark, New Jersey

ABSTRACT

A general model of a weighted voting legislature is described. The voting power of each of the N members is measured using the Banzhaf index. The voting power vector is a function of the vector of voting weights. An optimal weighted voting plan is one in which the voting powers are proportional to the respective constituencies. The optimization procedure is described in general terms and is demonstrated through real examples.

INTRODUCTION

Computer methods of legislative apportionment have been attended to since the "one man-one vote" Supreme Court decisions were handed down in the Sixties.¹ However, difficulties of various kinds have plagued the efforts to computerize. Single-member districting, by far the best-known means to attain constitutional representation, is subject to great combinatorial complexity. Even the best heuristics do not guarantee near-optimality or even feasibility.² Interactive districting techniques, on the other hand, create horrendous problems with data generation and graphical display, both essential elements of this approach.

The computerization of multimember districting is even further behind because this form of apportionment lacks quantitative definition. The legal and political criteria for evaluating it have not been established. However, some studies have explored the use of criteria and computational techniques that are multimember district generalizations of the weighted voting model.^{3,4}

In this paper we focus on the third major form of representation, weighted voting, which is widely used in counties of New York State. This type of reapportionment is also computationally complex because of the combinatorial measures it employs. A subtask that involves astronomically large numbers is that of finding the "voting power" of each of N members of a legislature. It requires the examination, perhaps implicitly, of all possible voting combinations that may arise and the determination of those that permit each member to "exercise power." This relatively expensive calculation represents an iteration that may be repeated until the power is equitably distributed according to established criteria.

Here we describe the axiomatic models and mathematical *a priori* voting-power measures which have come to characterize this type of analysis. The fairness of a legislative decision game, like that of an ordinary card game, is assessed on the basis of its rules. This paper is about reapportionment and "reapportionment is about fair rules, not fair play."²

A central ingredient in weighted voting is the notion of voting power. Given N decisionmakers with diverse voting weights $w_i > 0$, $i = 1, \dots, N$, Shapley and Shubik⁵ proposed a measure for each voter's relative *a priori* strength which has been studied for some time and which has merits of its own. However, here we employ the mathematical model implied by Banzhaf⁶ because his index of voting power was accepted by the New York courts^{7,8,9} and has been used in numerous reapportionments in the state. It is elegant by virtue of its simplicity and its computational and mathematical properties.

After reviewing this model and the associated voting power index, we shall see, through real examples, how these were implemented in various instances. In these and other cases, an iterative search procedure is involved. It terminates with an optimal (or adjusted) weighted voting plan that, when adopted, allocates a voting weight to each legislator. This plan remains in effect until a new census shows a population shift.

Although traditionally weighted voting is cast in the political setting it must be remembered that it also arises in the corporate environment. It is usually employed in stockholder meetings.

THE A PRIORI MODEL OF SIMPLE WEIGHTED VOTING

The model can be described in terms of statutory rules normally found in the weighted voting environment.

1. The legislature consists of N members.
2. Each member represents a well-defined district with known population.
3. Each district is represented by a single member.
4. The legislature adopts or rejects resolutions through a voting procedure.
5. A majority rule (such as simple majority, two-thirds majority, etc.) decides the outcome.
6. Each legislator votes Yes or No only.
7. Each legislator holds several votes (these are his *voting*

TABLE I—Analysis of voting power under weighted voting for Putnam County, N. Y.
(proportional plan; simple majority rule; 1970 census; Putnam-74-1)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Carmel	21,639	22	26.0	38.596	54.167	38.167	41.921
2. Southeast	9,901	10	6.0	17.544	12.500	17.463	-28.421
3. Kent	8,106	8	6.0	14.035	12.500	14.297	-12.571
4. Phillipstown	7,717	8	6.0	14.035	12.500	13.611	-8.164
5. Ptnm Valley	5,209	5	2.0	8.772	4.167	9.188	-54.649
6. Patterson	4,124	4	2.0	7.018	4.167	7.274	-42.717
Totals	56,696	57	48.0	100.000	100.000	100.000	
Majority	29 Votes						
41.921		-54.649					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

weight denoted by w_i , which may or may not be equal to the voting weight of some other legislator).

8. A legislator casts his assigned votes as a block, either all Yes or all No.
9. There are no vetoes or other special privileges.

Furthermore, it is reasonable to make the following assumptions about voting behavior in view of the *a priori* nature of this model.

10. A legislator may vote Yes or No with equal likelihood and independently of his colleagues.

The model thus defined is sufficiently complete to support the quantitatively precise Banzhaf power index.

A MEASURE OF VOTING POWER¹⁰

A 20-member legislature of the above type may vote in more than a million different ways. The number of Yes/No combinations for a 32-member body exceeds four billion. The number of possibilities rises to about one trillion in an assembly of 40 legislators. In general, a N -member legislature is capable of 2^N voting combinations.

Consider some specific voting combination. If the total weighted Yes vote is not vastly different from the total weighted No vote, then it is possible for some members to reverse the outcome by reversing their own votes.

For example, if a legislature consists of five members A, B, C, D, and E who hold, respectively, 3, 4, 2, 1, and 9 votes, then the voting combination (Yes, Yes, Yes, No, No) results in 9 weighted Yes votes and 10 weighted No votes. Under a simple majority rule this combination defeats the resolution being voted on. However, if D were to reverse his vote (10 Yes, 9 No) the resolution would pass. The (Yes, Yes, Yes, No, No) combination is said to be *critical* (or decisive) to member D.

The definition of voting power is based on the notion of decisive voting combinations. "... In a case in which there are N legislators ... the ratio of the power of legislator X to the

power of legislator Y is the same as the ratio of the number of possible voting combinations of the entire legislature in which X can alter the outcome by changing his vote to the number of combinations in which Y can alter the outcome by changing his vote."⁶ This suggests a method for computing voting power.

- a. List all 2^N voting combinations.
- b. Count those which are critical to the first member; repeat for the second member, the 3rd, etc. The numbers obtained are shown under column D of the accompanying tables.
- c. Add all counts just obtained in (b).
- d. Divide each number in (b) by the sum (c) and multiply by 100. This is a percentage of the relative voting power of each legislator (shown as column F in tables).

This explicit method of calculating voting power works well for small problems such as that of Putnam County shown in Table I but becomes infeasible when N exceeds 20. Calculations for the larger bodies such as Oswego County (Table II) or the Electoral College¹¹ require special mathematical shortcuts, discussed elsewhere.^{12,13} Indeed, our computerized weighted voting methods, used since 1967, have been exclusively of the implicit type.

PROPORTIONAL AND ADJUSTED WEIGHTED VOTING¹⁰

The simplest way to design a weighted voting plan is to allocate votes in proportion to each member's constituency. Thus, if legislator X represents 5% of the total number of persons in the polity, he is assigned 5% of the votes. However, this proportional approach does not guarantee that X 's voting power will necessarily equal 5%.

According to Imrie, "An illustration of the... 'simplistic' approach to weighted voting is provided by assuming a four-member legislative body made up of members representing 20,000, 20,000, 20,000 and 10,000 people, respectively. If each representative is given one vote for each 10,000 people he represents, three of the representatives will have two votes

TABLE II—Analysis of voting power under weighted voting for Oswego County, N.Y.
(This is a simple majority plan. ID: Oswego-73-1.5 A)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Albion	1,452	147	2046266024.0	1.457	1.437	1.439	-0.13
2. Amboy	557	56	778329728.0	0.555	0.547	0.552	-0.98
3. Boylston	276	28	389087906.0	0.277	0.273	0.274	-0.10
4. Constantia	3,547	356	5000637790.0	3.528	3.512	3.515	-0.09
5. Fulton 1	2,252	227	3167871266.0	2.249	2.225	2.232	-0.31
6. Fulton 2	3,830	384	5403902986.0	3.805	3.795	3.796	-0.01
7. Fulton 3	2,230	225	3139722112.0	2.229	2.205	2.210	-0.23
8. Fulton 4	2,106	213	2970941262.0	2.111	2.087	2.087	-0.03
9. Fulton 5	1,666	168	2339891050.0	1.665	1.643	1.651	-0.47
10. Fulton 6	1,919	194	2704157880.0	1.922	1.899	1.902	-0.14
11. Granby	4,718	471	6673496764.0	4.667	4.687	4.676	0.23
12. Hannibal	3,165	318	4456816724.0	3.151	3.130	3.137	-0.21
13. Hastings	6,042	596	8555394018.0	5.906	6.009	5.988	0.34
14. Mexico	4,174	418	5896868950.0	4.142	4.141	4.137	0.11
15. Minetto	1,688	171	2381875488.0	1.694	1.673	1.673	-0.00
16. New Haven	1,845	187	2606000458.0	1.853	1.830	1.829	0.09
17. Orwell	836	85	1181794162.0	0.842	0.830	0.829	0.17
18. Oswego Town	6,514	639	9223672398.0	6.332	6.478	6.456	0.33
19. Oswego 1	3,645	366	5144398558.0	3.627	3.613	3.613	0.01
20. Oswego 2	1,419	144	2004361312.0	1.427	1.408	1.406	0.09
21. Oswego 3	3,405	342	4799841400.0	3.389	3.371	3.375	-0.11
22. Oswego 4	2,170	219	3055302808.0	2.170	2.146	2.151	-0.22
23. Oswego 5	2,402	242	3379228570.0	2.398	2.373	2.381	-0.30
24. Oswego 6	2,528	255	3562719466.0	2.527	2.502	2.506	-0.13
25. Oswego 7	2,168	219	3055302808.0	2.170	2.146	2.149	-0.13
26. Oswego 8	3,176	320	4485351170.0	3.171	3.150	3.148	0.07
27. Palermo	2,321	234	3266457910.0	2.319	2.294	2.300	-0.27
28. Parish	1,782	180	2507901732.0	1.784	1.761	1.766	-0.27
29. Redfield	386	39	541980046.0	0.386	0.381	0.383	-0.50
30. Richland	5,324	528	7521798554.0	5.232	5.283	5.277	0.11
31. Sandy Creek	2,644	266	3718239378.0	2.636	2.611	2.620	-0.34
32. Schroepfel	7,153	696	10131234510.0	6.897	7.115	7.089	0.36
33. Scriba	3,619	363	5101242068.0	3.597	3.583	3.587	-0.11
34. Volney	4,520	451	6379167888.0	4.469	4.480	4.480	0.00
35. West Monroe	2,535	256	3576850952.0	2.537	2.512	2.512	-0.01
36. Williamstown	883	89	1237479890.0	0.382	0.869	0.875	-0.69
Totals	100,897	10,092	142384955392.0	100.000	100.000	100.000	
Majority	5,047 Votes						
0.366		-0.980					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

each and the fourth will have one vote...It will be seen that [under a simple majority rule] the representative with but one vote can never, by changing his vote, make any difference in the outcome of any issue before the legislative body. Hence, his constituency has no real representation on the legislative body and this type of weighted voting does not cure inequality in representation."¹⁴

It is this simplistic approach of *proportional* weights that prompted the Banzhaf statement: Weighted Voting Doesn't Work!¹⁶ The New York Supreme Court demonstrated that weighted voting can indeed be made to work.⁸ The court adopted a logical criterion for determining the quality of a plan:

"Ideally, in any weighted voting plan, it should be mathematically possible for every member of the legislative body to cast the decisive vote on legislation in the same ratio which the population of the constituency bears to the total population... This is what is meant by the one man-one vote principle as applied to weighted voting plans for municipal governments..."¹⁷

This sets the required standard. A weighted voting plan must be assessed in terms of the discrepancies between voting power and population. A plan is acceptable if the discrepancies are as small as possible.

If the discrepancies are not small, the voting weights are adjusted so as to reduce them. This involves a combinatorial

TABLE III—Analysis of voting power under weighted voting for Schoharie County, N.Y.
(Simple majority decisions SCHO-10.5 P)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Cobleskill	4,573	588	18561.0	19.600	22.500	19.606	14.760
2. Schoharie	3,088	397	10733.0	13.233	13.011	13.240	-1.727
3. Middleburgh	2,486	320	8633.0	10.667	10.465	10.659	-1.813
4. Richmondville	1,903	245	6513.0	8.167	7.895	8.159	-3.232
5. Esperance	1,567	202	5327.0	6.733	6.458	6.718	-3.882
6. Sharon	1,566	201	5301.0	6.700	6.426	6.714	-4.290
7. Seward	1,271	163	4287.0	5.433	5.197	5.449	-4.633
8. Wright	1,086	140	3665.0	4.667	4.443	4.656	-4.581
9. Fulton	1,060	136	3569.0	4.533	4.326	4.545	-4.801
10. Carlisle	1,040	134	3515.0	4.467	4.261	4.459	-4.438
11. Gilboa	854	110	2877.0	3.667	3.488	3.661	-4.748
12. Jefferson	840	108	2833.0	3.600	3.434	3.601	-4.642
13. Summit	690	89	2331.0	2.967	2.826	2.958	-4.482
14. Broome	551	71	1859.0	2.367	2.254	2.362	-4.606
15. Conesville	489	63	1643.0	2.100	1.992	2.097	-5.001
16. Blenheim	260	33	845.0	1.100	1.024	1.115	-8.109
Totals	23,324	3,000	82492.0	100.000	100.000	100.000	
Majority	1,501 Votes						
14.760		-8.109					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

optimization procedure. Voting assignments derived in this manner are called *adjusted plans* and satisfy the one man-one vote principle as required by the Court.

In the accompanying tables, Column H measures the *discrepancy* between voting power and population and is obtained via the following formula:

$$100 \frac{\text{Power} - \text{Population}}{\text{Population}}$$

This can be regarded as the percent difference between what a legislator's voting strength is and what it should be.

In summary, the computational procedure consists of (a) an *iteration*, which given a set of voting weights computes the number of critical combinations for each legislator and (b) a *search*, which repetitively adjusts the voting weights until the relative discrepancies are minimized.

Table III shows a proportional plan* for Schoharie County. Under this plan the legislator from Cobleskill enjoys excessive

power, which is not so surprising in view of the fact that he has nearly one fifth of all the votes. His ability to control such a large block of votes may suggest a disproportionate amount of power. This is actually borne out by the numbers of this example. With 588 votes, Cobleskill commands 22.5% of the power. The towns of Esperance, Sharon, Summit, Conesville, and Blenheim, on the other hand, have a total of 18.7% of the power even though their aggregate vote is also 588.

This relative dictatorial effect is common under weighted voting. It tends to be amplified as a single member's voting weight is enlarged, relative to the weights of the other members. In fact, past the majority point, it renders him an absolute dictator. We can therefore conclude that the whole voting weight is not necessarily equal to the sum of its parts. The inequities in a plan can be reduced by decreasing the voting weights of overrepresented towns while at the same time increasing those of the underrepresented. Such an adjusted plan for Schoharie County is shown in Table IV. It contains substantially smaller discrepancies than its proportional counterpart. This plan was enacted by the Schoharie legislature in 1975.

SPECIAL MAJORITIES¹⁰

Voting power, as defined here, is explicitly tied to the effective majority rule. This means that under a specified apportion-

*This plan was obtained by distributing 3000 votes among the 16 members in proportion to their respective constituencies. The total vote figure (3000) was arbitrarily selected. It provides simple and two-thirds quotas that are mnemonically convenient—1501 and 2000 (or 2001), respectively. The choice of such a high total vote serves two purposes. On one hand, it minimizes the round-off error that occurs when integral votes are assigned, and on the other hand, facilitates the adjustment phase of the computation.

TABLE IV—Analysis of voting power under weighted voting for Schoharie County, N.Y.
(Simple majority decisions. SCHO-10.5 A)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Cobleskill	4,573	523	16542.0	17.439	19.280	19.606	-1.666
2. Schoharie	3,088	400	11534.0	13.338	13.443	13.240	1.536
3. Middleburgh	2,486	323	9198.0	10.770	10.720	10.659	0.579
4. Richmondville	1,903	251	7010.0	8.369	8.170	8.159	0.137
5. Esperance	1,567	208	5798.0	6.936	6.758	6.718	0.583
6. Sharon	1,566	208	5798.0	6.936	6.758	6.714	0.647
7. Seward	1,271	169	4698.0	5.635	5.476	5.449	0.481
8. Wright	1,086	145	3994.0	4.835	4.655	4.656	-0.024
9. Fulton	1,060	141	3882.0	4.702	4.524	4.545	-0.444
10. Carlisle	1,040	139	3830.0	4.635	4.464	4.459	0.111
11. Gilboa	854	114	3118.0	3.801	3.634	3.661	-0.749
12. Jefferson	840	112	3062.0	3.735	3.569	3.601	-0.907
13. Summit	690	92	2574.0	3.068	3.000	2.958	1.409
14. Broome	551	74	2010.0	2.467	2.343	2.362	-0.835
15. Conesville	489	65	1798.0	2.167	2.096	2.097	-0.047
16. Blenheim	260	35	954.0	1.167	1.112	1.115	-0.255
Totals	23,324	2,999	85800.0	100.000	100.000	100.000	
Majority	1,500 Votes						
1.536		-1.666					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

TABLE V—Analysis of voting power under weighted voting for Schoharie County, N.Y.
(For a two-thirds majority rule only. ID: SCHOHARIE-75-10.67 B)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Cobleskill	4,573	523	8192.0	17.439	16.691	19.606	-14.869
2. Schoharie	3,088	400	6652.0	13.338	13.553	13.240	2.370
3. Middleburgh	2,486	323	5326.0	10.770	10.852	10.659	1.812
4. Richmondville	1,903	251	4126.0	8.369	8.407	8.159	3.036
5. Esperance	1,567	208	3436.0	6.936	7.001	6.718	4.204
6. Sharon	1,566	208	3436.0	6.936	7.001	6.714	4.270
7. Seward	1,271	169	2794.0	5.635	5.693	5.449	4.467
8. Wright	1,086	145	2384.0	4.835	4.857	4.656	4.322
9. Fulton	1,060	141	2318.0	4.702	4.723	4.545	3.922
10. Carlisle	1,040	139	2288.0	4.635	4.662	4.459	4.549
11. Gilboa	854	114	1866.0	3.801	3.802	3.661	3.837
12. Jefferson	840	112	1846.0	3.735	3.761	3.601	4.436
13. Summit	690	92	1550.0	3.068	3.158	2.958	6.753
14. Broome	551	74	1214.0	2.467	2.474	2.362	4.705
15. Conesville	489	65	1074.0	2.167	2.188	2.097	4.374
16. Blenheim	260	35	578.0	1.167	1.178	1.115	5.646
Totals	23,324	2,999	49080.0	100.000	100.000	100.000	
Majority	2,000 Votes						
6.753		-14.869					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

TABLE VI—Analysis of voting power under weighted voting for Schoharie County, N.Y.
(For a two-thirds majority rule only. ID: SCHOHARIE-75-10.67 A)

A Town	B Population	C Vote	D # Decisive Combinations	E ** % Vote	F *** Voting Power	G * % Population	H Discrepancy
1. Cobleskill	4,573	633	9871.0	21.114	19.393	19.606	-1.089
2. Schoharie	3,088	379	6771.0	12.642	13.303	13.240	0.476
3. Middleburgh	2,486	312	5449.0	10.407	10.705	10.659	0.439
4. Richmondville	1,903	242	4173.0	8.072	8.198	8.159	0.484
5. Esperance	1,567	199	3423.0	6.638	6.725	6.718	0.098
6. Sharon	1,566	199	3423.0	6.638	6.725	6.714	0.161
7. Seward	1,271	162	2789.0	5.404	5.479	5.449	0.551
8. Wright	1,086	138	2365.0	4.603	4.646	4.656	-0.210
9. Fulton	1,060	134	2301.0	4.470	4.521	4.545	-0.529
10. Carlisle	1,040	132	2275.0	4.403	4.470	4.459	0.238
11. Gilboa	854	109	1861.0	3.636	3.656	3.661	-0.144
12. Jefferson	840	107	1823.0	3.569	3.582	3.601	-0.553
13. Summit	690	87	1533.0	2.902	3.012	2.958	1.807
14. Broome	551	70	1211.0	2.335	2.379	2.362	0.711
15. Conesville	489	62	1067.0	2.068	2.096	2.097	-0.014
16. Blenheim	260	33	565.0	1.101	1.110	1.115	-0.423
Totals	23,324	2,998	50900.0	100.000	100.000	100.000	
Majority	1,999 Votes						
1.807		-1.089					Maximum Discrepancies

* Percentage ratio of column B to sum of column B.

** Percentage ratio of column C to sum of column C.

***Percentage ratio of column D to sum of column D.

ment of voting weights the voting power of member X will be different under different majority rules.

In *Slater versus Board of Supervisors of Cortland County* the court states: "Ordinarily, a weighted voting plan applicable to a simple majority vote of the County Legislature will not comply with acceptable standards when matters need a two-thirds or three-fifths vote for affirmative action. A legislator's voting power will differ when the votes needed for affirmative action are increased above a simple majority."

The very identical vote assignment of Table IV can be re-evaluated under a two-thirds rule to yield discrepancies of up to 15% as shown in Table V. Such deviations render this plan unacceptable under the one man-one vote criterion cited above.

However, it is possible once again to obtain an adjusted plan with substantially smaller discrepancies. Such a plan is shown in Table VI.

One must conclude, therefore, that weighted voting plans designed for a given majority rule must be used under that majority rule only. If the legislature must decide certain matters under two-thirds, three-fifths, or other special majority formula then separate weighted voting plans must be devised for use under such circumstances.

COMPUTATIONAL CONSIDERATIONS

An explicit enumeration of all voting combinations obviously is exponential in N . Banzhaf's original program required

about two hours of IBM/7094 time for a single iteration with $N=20$. This time would double for $N=21$ and again for each unit increase in the size of the legislature.

Through implicit enumeration the same iteration can be solved in less than one second of IBM/360-91 time. The complexity of the problem is linear in N .

REFERENCES

1. *Baker v. Carr* 369 U.S. 186, 1962.
2. *Gray v. Sanders* 372 U.S. 368, 1963.
3. Papayanopoulos, L., "Quantitative Principles Underlying Apportionment Methods," *Annals of the N. Y. Acad. of Sci.*, Vol. 219, 1973, pp. 183-191.
4. Papayanopoulos, L., "Voting Power and Multimember District Reapportionment in New York's Municipal Legislatures," Paper given at the Public Choice Society Meeting, Chicago, April 3-5, 1975.
5. Banzhaf, J. F., III, "Multimember Electoral Districts—Do They Violate the 'One Man, One Vote' Principle," *Yale Law Journal*, Vol. 75, 1966, pp. 1309-1338.
6. Shapley, L. S. and Martin Shubik, "A Method for Evaluating the Distribution of Power in a Committee System," *American Pol. Sci. Review*, Vol. 48, 1954.
7. Banzhaf, J. F., III, "Weighted Voting Doesn't Work: A Mathematical Analysis" *Rutgers Law Review*, Vol. 19, 1965.
8. *Iannucci vs. Board of Supervisors of Washington County and Saratogian, Inc., vs. Board of Supervisors of Saratoga County*, 20 N.Y. 2d 244, 299 NE 2d 195, 282 NYS 2d 502, 1967.
9. *Dobish vs. Board of Supervisors of Wayne County*, 279 NYS 2d 565, 282 NYS 2d 791, 1967.
10. *Slater vs. Board of Supervisors of Cortland County*, 330 NYS 2d 947, 346 NYS 2d 185, 42 A.D. 2d 795, 1972-73.

-
10. Papayanopoulos, L. Reapportionment reports to N.Y. municipalities, 1967—present.
 11. Banzhaf, J. F., III, "One Man, 3,312 Votes: A Mathematical Analysis of the Electoral College," *Villanova Law Review*, Vol. 13, 1968.
 12. Papayanopoulos, L. "Power Indices and Combinatorial Frequency Functions," TIMS/ORSA Conference paper, May 1978.
 13. Papayanopoulos, L., "Power Computations for Large Weighted Voting Bodies," ORSA/TIMS Conference Paper, November 1978.
 14. Imrie, R. W., "The Impact of the Weighted Vote on Representation in Municipal Governing Bodies of New York State," In L. Papayanopoulos (ed.), *Democratic Representation and Apportionment: Quantitative Methods, Measures, and Criteria*. New York: The New York Academy of Sciences, 1973.

Hospital information systems tutorial: a guide for computer scientists and practitioners

by DAVID J. MISHELEVICH

*Dallas County Hospital District and The University of Texas Health Science Center at Dallas
Dallas, Texas*

ABSTRACT

With the United States spending in excess of two hundred billion dollars per year on health care, it is clear that only the small percentage of that likely to be dedicated to computer-based hospital information systems offers both a tremendous opportunity and a responsibility for those in computing. This is particularly true because both software and hardware technologies have evolved to the point where comprehensive on-line systems are now functionally and financially a practical reality. This tutorial provides an overview of the software and hardware approaches that have been successfully applied, the delineation of success factors in the realm of human engineering, a cost per incident of service distribution methodology, and a review of cost/benefit considerations. The session is directed to those in the computing arena as opposed to those in health care, but the items covered will be of direct application to those on the medical side as well.

INTRODUCTION

The U.S. health industry as a whole accounts for approximately 9% of the Gross National Product. Even just 3% of the \$200 billion (a minimal projection) to be spent in health in 1980 amounts to \$6.6 billion. It is highly likely that we will evolve to spending at least this percentage on computing within the health industry. Thus there is economic impetus for a further thrust in medical computing, particularly in hospitals, which represent more than 40% of the health industry funds to be spent.

A critical ingredient is outside the realm of the economic. That factor is the desire of those in data processing to participate in systems that offer direct benefit to one's fellow person. Hospital information systems represent such an opportunity.

I have developed this tutorial as a self-described hard-core realist with respect to hospital information systems (HIS). My experience has been in the implementation of POIS, the Parkland On-Line Information System, at the Dallas County Hospital District (DCHD), otherwise known as the Parkland Memorial Hospital. This activity has been documented in various sources.¹⁻¹¹ In addition, current or past HIS consulting engagements include the American Hospital Association, the East Texas Chest Hospital (now The University of Texas Health Center at Tyler), the Harper-Grace Hospitals, the

National Center for Health Services Research, the Northwestern Memorial Hospital, the St. Joseph Hospital (Mt. Clemens, Michigan), St. Thomas Hospital (Nashville), Sisters of Charity of the Incarnate Word Hospital System (headquartered in Houston), the University of Cincinnati General Hospital, the University of Florida (Gainesville) Shands Teaching Hospital, the University of Mississippi Medical Center, and the University of Texas Medical Branch (Galveston).

Although my major personal experience has been with the IBM patient care system, the principles outlined in this paper are applicable to essentially any hospital information system.

BASIC PHILOSOPHY

The fundamental objective for a Hospital Information System is to install a health care delivery system with charge capturing as a logical by-product rather than an end. Thus health care and administrative/financial applications represent concurrent rather than conflicting priorities. It is appropriate to note that a comprehensive HIS will not just deal with order entry and results (and administrative) reporting, but will also play an active role in the work management aspects of all components, including ancillary departments such as the clinical laboratories and radiology.

BASIC STRATEGY

The fundamental HIS strategy is to put the computing power where it belongs: in the hands of the user. Since the ultimate user for much of both order entry and inquiry is the physician, it is logical to have a human-engineered system that can easily and effectively be used by the physician. A major component thus is the use of cathode ray tube (CRT) display terminals with the capability for the user to point to the screen to select a desired item. The menu selection process, via light pen or other mechanism, as opposed to keyboard-only operation, is a critical necessity for other users as well as physicians to promote easy use as well as user training.

HIS CLASSIFICATION

In the past decade various classifications have been put forth for defining the levels of functionality of hospital information

scheduled as individual resources. In April the clinics on the ground floor—ophthalmology; ear, nose, and throat; and Oral Surgery—were added. Psychiatry was also put into production. In March of 1980, the ancillary department of physical medicine and rehabilitation was added.

Production in our first laboratory with automated instruments, the special chemistry laboratory, began in September 1980. Two pairs of blood-gas instruments were interfaced to POIS on the IBM 370/168 via an IBM Series/1 minicomputer.

Current development includes the rest of the clinical laboratories, a new patient accounting system and a new payroll/personnel system. The status projected for mid-1981 is shown in Figure 2. Examples of screen flows and reports and a description of benefits are beyond the scope of this article but are covered elsewhere¹⁻⁸. Cost distribution on an incident-of-service basis^{1,9} and cost/benefit^{10,11} for POIS have also been covered.

HIS ALTERNATIVES

The three basic alternative possibilities are (1) a completely do-it-yourself implementation, (2) a turnkey system, and (3) installation using a package system approach, such as the IBM health care support/patient care system (PCS).

The Do-It-Yourself Approach

To develop a hospital information system on a develop-it-yourself basis, is rarely if ever justified, particularly since viable alternatives are now available in the other two categories. Organizations that have produced effective hospital

information systems have put a significant amount of time and money into that activity. That a vast quantity of resources were put into such projects does not mean the organizations were always incompetent or unknowing; the development of a comprehensive HIS from scratch is a large undertaking in any circumstances and should not be taken lightly. The probability of doing so functionally and cost-effectively is so low as to be essentially dismissed out of hand.

Turnkey Systems

Philosophically, it is easy to fall into the trap of feeling that just because one has a contract with a turnkey vendor, automatically all the problems will be solved and all the deadlines met. While certainly the use of a turnkey vendor is a potentially viable option and should be seriously considered, it is still necessary to be vigilant and to participate fully in the design and the implementation of the specific system installed in your hospital.

The Package/System Approach

The IBM Health Care Support/Patient Care System¹⁻¹¹ offers an integrated approach. The IBM patient care system (PCS) provides the installation of the basic system as it was developed as the Duke Hospital Information System (DHIS) at the Duke University Medical Center. It offers the possibility of the addition of other packages as well. The additional package currently available in applications arena is Patient Care System—Radiology, which was developed at the Dallas County Hospital District; other products are being made ready or being planned as well. The radiology package¹²⁻¹⁴ is available as an installed user program (IUP). Other IUPs currently available have to do with architectural enhancements to the patient care system: Patient Care System-Edit,^{15,16} which was developed at the Sisters of Charity of the Incarnate Word Hospital System headquartered in Houston; and the Patient Care System-Data Manager,^{17,18} which was developed at the Cedars-Sinai Hospital in Los Angeles.

It is possible to get the IBM patient care system partially installed, and probably in the future fully installed, by another organization; one will still have to give a significant amount of effort to specify the user requirements. Of course, what the user wants must be balanced by what is practical; a reasonable equilibrium must be maintained.

No matter which system is being considered, those to whom marketing is directed need to be able to touch and feel the system, not just deal with artist's conceptions.

SUCCESS FACTORS

The following are major success factors that are generally applicable and have been found to be extremely helpful.

Administrative Mandate

A constant theme in this tutorial is the hospital information system being a hospital project rather than a data processing

POIS PROJECTED IN MID 1981

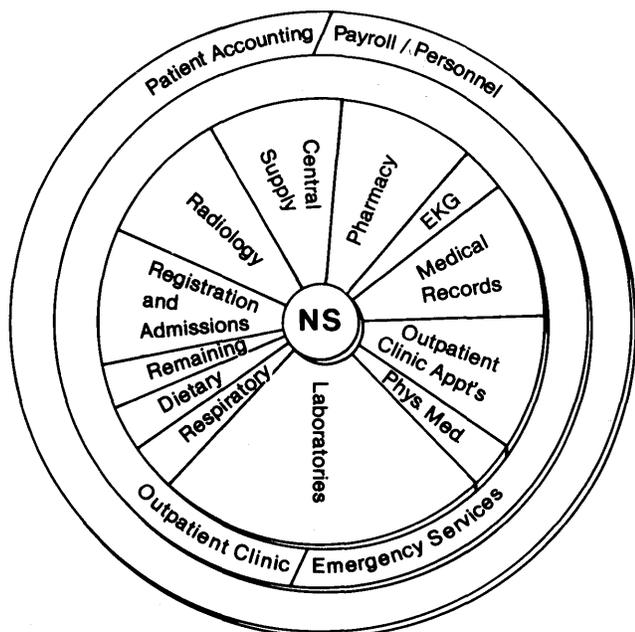


Figure 2—Status of POIS projected for mid-1981

project. This attitude must emanate from the top, and all concerned must have a self-fulfilling prophecy of success. The installation of an HIS is second only to major capital construction of a new building or a major renovation in its impact on the hospital. Clearly much compromise will be involved because of the hospitalwide aspects of the project. In addition, the significant change represented will often be greeted either with some inertia or an impetus to do the job in some other way that will be more effective for an individual department—at least as that department perceives it.

Strong administrative leadership is needed, and it must be consistent. Progress will be markedly slow or will be stopped if an HIS is installed in an environment of noncooperative leaders, each with his or her own fiefdom.

Quantitative Objectives

As part of the planning process it is terribly important for the chief executive officer to set quantitative objectives. How many hours of waiting time are to be reduced? What should be the time cut for the delivery results back to the nursing station from the laboratories? How many days in receivables shall be deleted in the accounts/receivable function? How many fewer days should discharge or final discharge bills be issued after the patient leaves the hospital? All these and other questions make legitimate and important goals, so that one can tell if one has been successful or not. Unfortunately such projections or goals are rarely stated. Thus the evaluation of hospital information system implementation deals frequently with complaints about the system, from either users or external political forces. One needs to bring the process out of the realm of the anecdotal into the realm of the quantitative.

Interaction with Hospital Board

One wishes to avoid board politics if at all possible. It is tempting for the vendor to elicit as much support for a given position during the selection process as it can. Members of the hospital board who have had experience directly or indirectly with systems from that firm are occasionally viewed as important potential allies. It is a dangerous game at best. Frequently board members do become involved in the decision of which HIS is to be installed. They should work at a policy-making level in general, and certainly, because of the magnitude of the project, they will be interested in following it. Board support of an HIS project is a strong part of the administrative mandate.

Directions Committee

Continuing on the theme of stressing that HIS implementation is a hospital project rather than a data processing project, the directions committee or steering committee must represent multiple sources, namely all the interest groups involved. It operates in the realm of priority resolution, information dissemination, education, and provision of a forum for intergroup interaction. We have representatives from nursing ser-

vices, the medical staff, the ancillary departments involved in the implementation, the hospital administration, the hospital engineering group, the financial and internal audit representatives, the vendor, and information systems.

At our institution the committee is chaired at the level of an associate administrator, at least in part to emphasize the committee's importance and overall hospital participation in the project.

In some institutions there will be an additional committee operating at a higher level administratively than the directions committee. It might be composed of the chief executive officer and individuals at the associate administrator or vice-presidential level within that organization. If this higher level committee exists, that is fine, and it can work effectively; but it should not substitute for the directions committee, which tends to be inclusive in its membership. The directions committee of the DCHD meets biweekly for one-half to one hour.

In order to continue its work, the directions committee at DCHD has three standing committees, which in turn report to it. These will be described:

Fiscal committee

The fiscal committee has representatives from patient accounting, general accounting, and internal audits. Its responsibility is to insure system financial and statistical integrity. On large implementations a member of the committee will work with the task force doing that implementation. The committee will also get involved with special projects (which might have cross-application aspects). Inherent in this approach is the auditing concept, in which one endeavors to check out applications and test them appropriately to get rid of problems before they actually affect performance and production. The fiscal committee also provides a link to the patient accounting task force, active at DCHD, which is a group responsible for the fiscal integrity of the hospital district. This group performs direction committee functions, in some cases, with respect to some of the financial applications, such as patient accounting.

PIPOIS committee

The physicians' interaction with POIS (PIPOIS) committee provides the mechanism for medical staff input to system development as well as ongoing activities. PIPOIS has representatives from the faculty attending staff, the house staff, nursing services, information systems, and two POIS associates. This group is becoming more active as the clinical laboratories come on stream.

Laboratory executive committee

Because of the magnitude (in depth, breadth, and length of implementation time) of the clinical laboratory application, we also organized the laboratory executive committee (LEC). The LEC has representatives from administration, pathology, nursing services, two POIS associates, the vendor (IBM), and information systems.

Each individual HIS will have its own character and will have different committees constituted to meet specific implementation needs. Thus the three committees reporting to the directions committee are presented for illustrative purposes.

Associates

The associates are the users, who are health and other hospital professionals assigned full time to what has previously been considered data processing jobs. We currently have six, two from nursing services, one from education and training, one from radiology (representing also the non-laboratory ancillary departments), one from the clinical laboratories, and one from the business office. As shown in Figure 3, they report to their individual departments as well as to the Chief POIS associate. The latter individual, reporting to nursing services, has a dotted-line relationship to the information systems department project manager for medical applications.

The responsibilities of the associates are wide-ranging as to type of function—e.g., analyze old work flows, design new work flows, design and code CRT screens, design and code printer formats, participate in logic implementation, create user manuals and a portion of the systems documentation, prepare (and actually perform some of) the education and training programs, and trouble shoot (including ongoing maintenance). When a new application comes up, we have at least one POIS associate in the hospital for at least the first 72 hours. As to geographic considerations, while they have primary responsibilities, the associates work in any area—nursing, financial application, ancillary department—as needed, regardless of the particular associate title.

While in our hospital the associates hold full-time positions, they may be (at least in some cases) part time. This is fine as long as their basic associate function is not adversely affected by the press of other duties.

Besides the definite user orientation (because the associates as users are designing a user-oriented system), there is the benefit of getting individuals who, as they are performing what would often have traditionally been a data processing function, are deeply committed to the given application and in fact likely to that given institution. This makes attrition less likely. This is important in the face of the highly competitive and highly mobile data processing market.

Task Forces

A good deal of time is spent by POIS associates participating in the various implementation task forces, which get deeply into the realm of individual application design, implementation, documentation, and training. The task forces come and go as needed as specific areas are addressed and may well lie dormant for long periods.

Payback for Users for Participation

A critical element of ease of system implementation, as transmitted through the directions committee, the associates, and the task forces, is to give advantage to the users for being

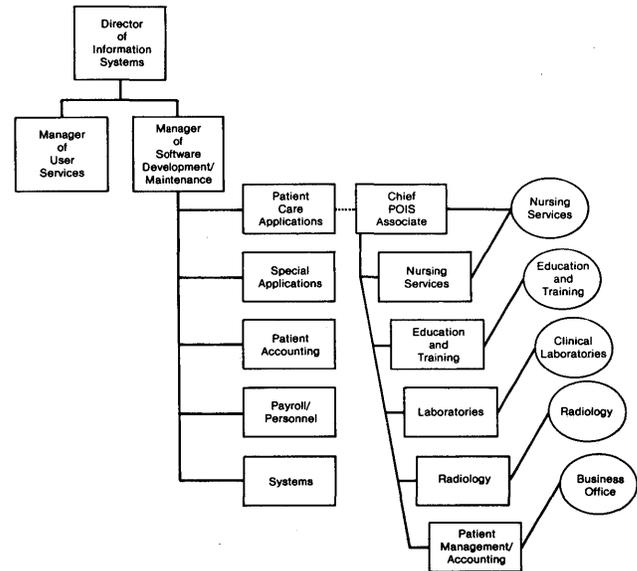


Figure 3—Relationship of POIS Associates to information systems

involved. It is important to find out what the users want out of a given application and work hard to give it to them.

Productivity Enabling

Any hospital information system is going to require at least some tailoring to individual hospital requirements. Thus easily tailored systems, such as the IBM Patient Care System, are important. Factors inherent in a productivity-enabling or application development system are

1. Data independence
2. Logic independence
3. Ease of logic implementation
4. Ease of CRT screen design and coding
5. Ease of printer format design and coding
6. Ability of non-data-processing personnel to do 3, 4, 5
7. Extensibility
8. User-Friendly Production System

The IBM patient care system has these features, as discussed by Mishevich and Van Slyke.^{19,20}

Modular, Phased-In Implementation

Because of the magnitude of the education and training effort involved in getting the large number of users comfortable with the system used, usually a modular phased-in implementation is preferable. This is run more functionally than geographically. Successful implementations have been accomplished on a whole-house basis, however.

Reporting Structure

The most effective structure is to have the data processing director report directly to the chief executive officer. This

emphasizes the overall importance of and commitment to the HIS and does not get projects sidetracked. If the DP director reports to a financial officer, the generally existent suspicion of priority only for financial and administrative applications can markedly undermine HIS efforts.

Sharing

Fortunately, for historical reasons, hospitals are much more likely to share software, approaches, manuals, etc., than other organizations. For example, the DCHD has had productive interactions with the Cedars-Sinai Hospital, Duke, Forsyth Memorial Hospital, Samaritan Health Services, the Sisters of Charity Hospital System, University of Iowa, University of Michigan, and The University of Texas Medical Branch (Galveston).

COST DISTRIBUTION

An important question is how much a given system will cost and/or should cost. There is not an easy answer; it will clearly depend on the size and functionality of the system. It is clear, however, that just taking the total cost and dividing by the number of inpatient days to get a cost per patient day is not appropriate if there is a significant ambulatory care component in terms of emergency room and/or outpatient visits. Thus one must resist doing so. There is a simple methodology available, based on volume of charges generated in given areas, for proportioning the cost for the hospital information system, on a cost-per-incident-of-service basis, among the three components of cost per inpatient day, cost per emergency room visit, and cost per outpatient visit⁹. This methodology also appears in a publication of the Texas Hospital Association¹. It can deal with multiple inpatient sources and/or combinations of ambulatory care sources.

For the DCHD, the 1979 figures using this methodology were \$5.81 per inpatient day, \$3.05 per outpatient visit, and \$3.64 per emergency room visit. This covers all data processing costs, whether they be in patient care functions or related to patient accounting, payroll/personnel, etc. Such figures are less useful for comparisons between hospitals because of the difference of functionality and volumes and more useful in a single environment to track from year to year. The DCHD figures are relatively somewhat higher (factoring out inflation) than they will be in the future because of the large amount of development activity ongoing.

It is interesting to note that in the early to mid-1970's—even prior to the great inflationary spiral, which brought escalation of costs for hospitals as well as the other sectors of the economy—costs of \$10.00 per patient-day were frequently stated as typical for a hospital information system. Thus economies have clearly been achieved in hospital information systems. This is due to the availability of better software and the marked decrease in the cost of hardware.

It is appropriate to note that when a number (often hundreds) of terminal devices are to be involved in a hospital information system, the cost of terminals becomes a significant part of the system. The use of nonspecialized terminals,

which frequently are about half the cost or less of specialized terminals (a standard IBM 3278 terminal with a light pen costs about \$3,500 per CRT terminal as opposed to CRT terminals from Technicon or Datacare, which would be \$7,000–\$8,000 each), makes quite a difference indeed. At DCHD, there are currently more than 350 CRT devices (mainly IBM 3278) installed, in addition to some 100 printers (IBM 3287). By the end of 1981 over 400 CRT devices will be installed. At perhaps \$3,500 difference per terminal times 400 terminals, there is a difference of \$1.4 million dollars for this single set of components for the system alone.

There is no magic in the reason for the difference in terminal costs. It is a matter of the volume of production. The IBM terminals are sold to cover a variety of applications in all industries, and many tens of thousands will be sold. Clearly the market for a specialized HIS terminal is much, much smaller. Amortizing the development and some of the production costs over a much smaller quantity of terminals is clearly going to increase the unit cost markedly.

COST BENEFIT

Cost-benefit methodologies are currently rudimentary. The reader is referred to a recent A.D. Little study¹¹ for a review. In practical terms, personnel are rarely reduced (although avoidance of adding personnel as hospital functional volumes grow appears to be a true factor), mainly because of the difficulty of displacing partial full-time equivalents.

OTHER CONSIDERATIONS

Many other factors are relevant for consideration. I have selected a number for discussion, as follows:

Advocate Role

The data processing function must play an active advocate role rather than just wait and see what is asked for. This is particularly true because in most hospitals the practical realities of computerized systems are either over- or underinterpreted a great deal, and data processing is frequently used in an attempt to solve problems that are not data processing problems.

Database Approach

It is interesting to note that development of software and hardware technology adequate to support on-line hospital information systems arose at a point when database management systems became both effective and popular. As a result of on-line systems installed, it is more likely that database management systems will be used in the health industry as opposed to other industries. This is at least in part because applications are delivered to customers, rather than just the tools to produce them. Thus in the IBM world the DL/I database management system has a higher penetration in the health industry than in any other segment of IBM marketing.

Not Inpatient Care Alone; Ambulatory Care As Well

A hospital information system is much easier to develop and install in an environment that involves the inpatient service alone. The inpatient arena is more controlled and may have lower service volumes. Thus it is important to consider whether a system being evaluated is designed for ambulatory care and has demonstrated its ability to operate effectively in this domain. This is particularly true because there is now an evolutionary thrust toward preventive medicine in the context of ambulatory care.

Multiple Hospital Systems

Not all hospital environments are single entities. Some hospital systems have two units close together or far away, and some systems may have a number of units, which may be widely distributed (e.g., in multiple states). In some cases the patient care applications will be distributed while financial/administrative functions, such as patient accounting or payroll/personnel, are handled centrally. Having standards and consistency while allowing individual tailoring for individual facilities is a constant challenge.

Education and Training

For both initial installation of applications and ongoing activities, education and training are vital functions. This is particularly true because of the high turnover that frequently exists in a hospital. Our initial education and training program for each application is developed, including manuals, by the POIS associates. The POIS associates also participate in the initial round of training for the application. Thereafter the education and training department takes over completely. The training itself takes place in (1) a training room within the education and training department which has six CRTs and two printers, (2) a training room within the clinical laboratories which has two CRTs and one printer, and (3) overflow areas as needed. Education and training is serious business, especially since it is likely that thousands of person-hours of training will be involved per year.

Role of Consultants

The responsible and effective use of consultants is much too broad and deep a topic to be dealt with reasonably here. It is important to note that knowledge and experience in hospitals is mandatory. The magnitude and complexities are such that an HIS is by no means just another on-line system. In addition, the 24-hour-a-day, 7-day-a-week functionality must be appreciated.

Interface to Laboratory Systems

While an increasing number of computer-based clinical laboratory systems will be done as integral components of the

central HIS, many hospitals have installed or will install stand-alone laboratory systems which will have to be interfaced. This may not always be trivial, and the work involved is likely to be underestimated. Do not take such projects lightly. A key factor is to insure that the results are sent where the patients are at the time the results are available, not where the orders were placed.

Interface to Patient Accounting

As stated under "Basic Philosophy," near the beginning of this paper, charge capturing and other administrative/financial functions should be transparent to the user. Thus an effective interface must be provided.

Reliability

As more and more applications are brought on stream, particularly in the medical care arena, reliability becomes a greater concern. While a single CPU can adequately serve (we average approximately 97% up-time of the 164 hours of planned availability per week on the IBM 370/168 at DCHD), the decrease in the cost of hardware will soon permit both redundancy and volume considerations to be routinely addressed in a multiple-CPU environment.

Technological Evolution

We are clearly in a time of rapid technological change, with respect to both software and hardware. This has implications in procurement, as discussed in the following two sections. It is important to note here that in on-line systems such as an HIS the projected transaction volumes are *almost always underestimated*.

Certificate of Need

Responding to changing needs is especially difficult when additional layers of control, with associated time delays, are present. One such factor to be reckoned with is the certificate of need process mandated by Federal laws and regulations having to do with Medicare/Medicaid portions of the Social Security Act. Procurements involving \$150,000 or more of a capital expenditure, or individual related units adding up to that amount, or the rental, lease, lease-purchase, or installment-purchase of combined items whose value is that amount or more, must have an approved certificate of need granted by the individual state through a health systems agency (HSA) and/or health facilities commission (HFC). This process can take four to ten months and can make life very difficult. Advance planning becomes especially important. An unusual factor is that technically even the acquisition of a replacement item at less cost also requires an approved certificate of need if the dollar amount falls within the monetary guidelines.

Procurement by Bid

Depending on the type of hospital or hospital system involved, hardware and/or software may by policy have to be procured via bidding. This is another important factor, with its potential for attendant delays and its additional paperwork, with appropriately-drawn specifications which must be dealt with.

Scheduling and Target Dates

Setting of target dates and/or deadlines is a tricky area. There is unfortunately a tendency for such dates to be set for geopolitical reasons, frequently external, rather than to be based on a careful analysis of the time and resources, considering all relevant areas, not just data processing, required to accomplish a given task. One must watch being railroaded. A well-documented, well-tracked project is a particularly good vehicle for defense and maintaining credibility.

Extension to the Physician's Office

The physician is the one who brings patients to the hospital, not the hospital itself. Thus good service to the medical staff may well give a particular hospital a competitive marketing edge. A potentially powerful approach is to offer to provide CRT terminals in physicians' offices to permit them to directly check status, place orders, or inquire as appropriate into the system. If there is a charge for this service, it would be possible to at least break even or perhaps underwrite partially the cost of the HIS itself.

Local Vendor Personnel Make the Difference

Wherever a vendor is involved, the corporate commitment, however great, must be translated into reality by the individuals actually doing the work. Thus excellence in the local vendor staff is important to success. Adopting them as part of the HIS team and as full responsible and accountable partners is an important process.

Love and the "Family" Orientation

Implementing an HIS is a difficult and complicated task. It requires a lot of patience and cooperation. The whole hospital and related components must act as a family to maximize responsible service to and support of the hospital. Love is a key ingredient.

SOURCES OF INFORMATION

There is fortunately an increasing number of materials available in the area of hospital information systems. The American Hospital Association released in summer 1980 a document, "How to Request for Proposal for the Acquisition of a

Hospital Information System"²¹. The National Center for Health Services Research (NCHSR), which is the Federal organization primarily responsible for spreading the word about and encouraging the development of hospital information systems, is currently sponsoring, via a contract being performed by the University of Southern California, a project for which the product will be an automated hospital information systems (AHIS) workbook. The AHIS workbook will deal with the process of deciding when to acquire a hospital information system, including what background studies are desirable; how to go through the acquisition process, including the release of a request for proposal (RFP) and system selection; and how to carry on through the installation process. The AHIS workbook will also present a survey of applications and potential vendors. This NCHSR-sponsored study is scheduled for completion in 1981.

Though it is impossible to be in any way exhaustive about the documentation available, the journal *Computers in Hospitals* has come on the scene, the publication *National Report on Computers and Health* has been initiated, and additional special reports are coming out, such as the NCHSR-sponsored study by A.D. Little on cost-benefit methodologies applicable to hospital information systems¹¹. It is important to note here that there are user groups for various systems, such as the Electronic Computing Health Oriented (the IBM health industry users group), the Technicon MIS users group, and others. They are a valuable source of information about what is currently going on and should definitely not be overlooked. Too often those who are actually doing the work and producing effective systems are exactly not those likely to produce papers in journals and/or write books. The user groups—which in some cases, like other sources, produce what has been termed "fugitive literature"—are key information resources with respect to knowledge in the field.

CONCLUSION

The simple concepts covered here, which were usually adapted from the work of other groups, do work. Thus they should not be taken lightly. It takes great effort to have a system as complex and far-reaching as a hospital information system woven into the fabric of an institution. It is, however, one of life's great, exciting, and personally rewarding experiences. The challenge is great; the time is now!

REFERENCES

1. Mischelevich, D.J., and L.D. Cranfill, "On-line Hospital Information System," *THISS* (Texas Hospital Information Systems Society) *Installation Planning Guidelines*, pp 43-54, 1978.
2. Mischelevich, D.J., Borden, Ruby and J.F. Stay. "Implementation Factors in Hospital Information Systems," In press, *National Computer Conference 1979 Proceedings on Computers in Health Care*, 1981.
3. Hudson, Betty G., Mischelevich, D.J., Mize, Elaine I., and J.R. Roberts, Jr. "POIS—the Parkland On-line Information System," *Electronic Computing Health Oriented Boondocks*, 8:65-116, 1979.
4. Mischelevich, D.J. "Installation of the IBM Patient Care System at the Parkland Memorial Hospital," *Electronic Computing Health Oriented Boondocks*, 7:36-56, 1978.
5. Mischelevich, D.J., Hudson, Betty, G., Van Slyke, D., Cranfill, L.D., Mize, Elaine, Robinson, Anna L., Brieden, Helen C., Atkinson, J., Willis, J.R.

- and J. Robertson. "The Parkland Order Information System (POIS): Installation of the IBM Health Care Support/Patient Care System at the Parkland Memorial Hospital," In press, *National Computer Conference 1979 Proceedings on Computers in Health Care*, 1981.
6. Mishelevich, D.J., Hudson, B.G., and E.I. Mize. "Parkland System: POIS (Parkland On-line Information System) Update," *Electronic Computing Health Oriented Boondocks*, 9:17-58, 1980.
 7. Mishelevich, D.J., Hudson, B.G., Van Slyke, D., Mize, E.I., Robinson, A.L., Brieden, H.C., Atkinson, J. and J.G. Robertson, Jr. "The POIS (Parkland On-line Information System) Implementation of the IBM Health Care Support/Patient Care System," *Proceedings of the Fourth Annual Symposium on Computer Applications in Medical Care*, 4:19-33, 1980.
 8. Mishelevich, D.J., Hudson, B.G., Van Slyke, D., Mize, E.I., Robinson, A.L., Atkinson, J., Robertson, Jr., J.G. and R.G. Newman. "Success Factors in the Implementation of a Comprehensive Hospital Information System: POIS, the Parkland On-line Information System," *Computers in Hospitals*, 2:26-36, 1980.
 9. Mishelevich, D.J., Day, M.W., Gipe, W.G., and L.D. Cranfill. "Distribution of Data Processing Costs for a Hospital Information System on a Cost-Per-Incident-of-Service Basis," *Proceedings of the Fourth Annual Symposium on Computer Applications in Medical Care*, 4:658-664, 1980.
 10. Mishelevich, D.J., Gipe, W.G., Roberts, Jr., J.R., Denney, C., Stem, A.D., and M.W. Day. "Cost-Benefit Analysis in a Computer-Based Hospital Information System," *Proceedings of the Third Annual Symposium on Computer Applications in Medical Care*, 3, 339-347, 1979.
 11. Arthur D. Little, Inc. "Methods for Evaluating Costs of Automated Hospital Information Systems," Report of National Center for Health Services Research Contract 233-79-3000, 192 pages, March 14, 1980.
 12. IBM Corporation. Availability Notice for "Patient Care System-Radiology," G320-6092-0, 1979.
 13. IBM Corporation. Program Description/Operations Manual, "Patient Care System-Radiology," SH20-2159, 1979.
 14. IBM Corporation. Terminal Operators Guide, "Patient Care System-Radiology," SH20-2160, 1979.
 15. IBM Corporation. *Patient Care System/Edit Availability Notice*, G320-6344-0 (Unlicensed Material), Program Number 5796-AYR, 1980.
 16. IBM Corporation. *Patient Care System/Edit Program Description/Operations Manual*, SH20-6143-0. (Unlicensed Material), Program Number 5796-AYR, 1980.
 17. IBM Corporation. *Patient Care System/Data Manager Availability Notice*, G320-6343-0 (Unlicensed Material), Program Number 5796-AYQ, 1980.
 18. IBM Corporation. *Patient Care System/Data Manager Program Description/Operations Manual*, SH20-6142-0 (Unlicensed Material), Program Number 5796-AYQ, 1980.
 19. Mishelevich, D.J., and D. Van Slyke. "An Overview of the Software Architecture of the IBM Health Care Support/Patient Care System," *Proceedings of the 1980 Conference on Application Development Systems*, Special Issue of the ACM Special Interest Group on Business Data Processing *DATA BASE*, 11:64-75, 1980.
 20. Mishelevich, D.J., and D. Van Slyke. "Application Development System: The Software Architecture of the IBM Health Care Support/Patient Care System," *IBM Systems Journal*, 19:478-504, 1980.
 21. American Hospital Association. "Hospital Computer Systems Planning: Preparation of Request for Proposal," AHA Catalog No. 1445, 117 pages, 1980.

**VISUALS,
NATURAL LANGUAGE PROCESSING,
AND ARTIFICIAL INTELLIGENCE**

Issues in the development of natural language front-ends

by JAMES HENDLER, THOMAS P. KEHLER, PAUL ROLLER MICHAELIS,
BRIAN PHILLIPS, KENNETH M. ROSS, and HARRY R. TENNANT

Texas Instruments
Dallas, Texas

ABSTRACT

This paper will discuss some issues we believe to be important to the design of a natural language front-end. These are divided into three categories: conceptual coverage, linguistic coverage, and implementation issues. The section on conceptual coverage discusses the use of a domain expert, which understands what the user is saying even though the system to which the front-end is interfaced might not be able to properly do what the user wants. The section on linguistic coverage discusses attempts to allow a natural language interface to handle natural, interactive human communication. Two solutions are explored: First, the design of a robust natural-language-understanding system, composed of many experts that know about some aspect of the organization of language, is considered; second, because the design of a robust system is a large task, the intermediate goal of limiting the vocabulary and constructions that can be used while retaining all the user-oriented benefits of natural language is considered. The implementation issues considered are the design of a system in which the grammar and the domain of discourse can be easily extended and which can be used for more than one domain without extensive rewrite.

INTRODUCTION

In this paper we will present some of the ongoing research from the Texas Instruments Intelligent Interactive Systems (IIS) Branch. Specifically, we will be discussing research in natural-language processing. Some of the goals of the IIS group are to develop advanced educational software, expert systems in various fields, hardware systems compatible with artificial intelligence technologies, and personal computer applications.

One of the problems faced by the group when designing these systems is making them accessible to users who may not have the time or motivation to learn (or frequently relearn) a complex, esoteric computer interaction language. We hope to develop the natural-language-processing technology that will give users access to sophisticated software packages.

We are simultaneously investigating the three main areas of interest in natural-language processing: conceptual coverage,

linguistic coverage, and implementation issues. We feel that before a truly useful natural-language-understanding system can be delivered to a large user population, advances must be made in all three of these areas.

In the following sections we will consider each of these areas of interest. We will indicate some of the limitations of the current technology in these areas and present our solutions for overcoming these limitations.

CONCEPTUAL COVERAGE

One area in which we see a potential for near-term application of natural-language-processing technology is access to database systems. One of the ways in which our approach differs from others is in our intention to provide the system with expanded conceptual coverage.

Most natural-language question-answering systems rely on the premise that users will use a natural-language interface to access data in a database and nothing more. Dialogues collected from actual problem-solving situations illustrate that this is an oversimplified view. In actuality a significant portion of the interaction is devoted to the asker and the answerer coming to a mutual understanding of what the interaction can accomplish and what was meant by utterances from each side.

This process of coming to a mutual understanding is uniquely characteristic of natural-language communication—one that sets it above more constrained forms of communication, such as menu selection or formal query languages. Examples of several classes of these utterances are shown below. Only a few examples will be given in each category, but many more were found. These examples were gathered from a study done on the PLANES system. (See Tennant¹¹ for a description of the study and more examples.)

1. What parts or system caused the NOR or RMC time in 1971?

The example above illustrates that the user's incomplete knowledge of the database system sometimes encourages him to ask poorly formed or unexpectedly expensive queries. The retrieval for this question would have taken about 30 minutes. If the user was simply asking for data in order to gain insight

about another process, a more efficient way of gaining the insight might be found.

Even after getting the results of an extensive search, the user might find them of little value. The results of the above search would consist of thousands of elements. In the actual situation from which this example comes, such a long list would have been worthless to the user.

2. Were more landings made with arresting equipment in 1971 than in other years?

The query above could not be answered because there did not happen to be any data in the database on arresting equipment. If a natural-language interface to a database will allow casual use of the database, one must expect that the users will not have detailed knowledge of the contents of the database.

3. How far from a supply base is PUC 38 and 306?

This question is an attempt by the user to clarify his understanding of the general domain of discourse. He knew that the answer to this question was not in the database, but he needed this information to pursue this theory for possible causes of increased down time. Once again, the conceptual coverage of the system must exceed the scope of the data in the database in order to understand (let alone answer) questions like this.

4. Were the aircraft equipped differently during 71?

Questions in this category require judgmental interpretations. "Equipped differently" implies significantly different equipment. The interpretation of this phrase depends upon understanding what the norms are for the domain of discourse and what have been established as norms in the course of the conversation.

The solution that we propose to the problem areas listed above is that a knowledge base be maintained within the natural-language interface that describes the database, including its contents and search sizes, describes the domain of discourse, giving it knowledge about concepts that are not described in the database, and describes the system itself along with the course of the current interaction. This knowledge allows a change in the accepted paradigm of question answering. Where in the past each utterance was interpreted as a database query, then evaluated, now the utterances would be understood prior to forming a query. A query would be formed and evaluated only if the system had determined that that would most efficiently help the user achieve his goals.

More broadly, what we are proposing is to build a domain expert as an integral part of the natural-language-processing system. Its task is to try to understand what the user wants, then determine how or whether it can provide it. The idea of a domain expert acting as an intermediary between the user and computing facilities can be applied in a variety of situations, which we are considering.

The difference between the approach to natural-language processing presented here and the approaches that have been generally taken in the past primarily hinge on what is to be expected of the users. Few assumptions about the users of natural-language processors have been explicitly described by

system designers. One can infer, however, that users are expected to be familiar with the domain of discourse, that they will restrict themselves to questions that can be interpreted as database queries, and that they will generally understand the dialogue as it progresses. Our assumptions are much more sympathetic to the user. He is still expected to be very familiar with the domain of discourse and the specialized nomenclature of the domain. We do not expect him to know (or necessarily want to learn) anything of the structure of the database. We do not expect him to remember much about the system's capabilities or limitations between sessions. In other words, the user should be able to forget everything about the interaction from session to session and retain only his knowledge of the domain. But he should still be able to use the system effectively in spite of his forgetfulness.

LINGUISTIC COVERAGE

The most fundamental human-oriented question in the field of natural language processing is this: What is natural language? Specifically, when people spontaneously communicate in an electronic medium, such as telephone or teletypewriter, what does their communication look like? It has been demonstrated by several researchers² that natural, uninhibited interactive human communication is characterized by extreme errors in grammar and spelling. It is clear that natural language and immaculate English are two very different styles of communication. We are interested in natural language.

As an initial step, we are examining person-to-person dialogue. In a study by Michaelis,⁶ two-person teams worked together to assemble an abstract model made of colored wooden dowel rods and wheel-shaped connectors. The two team members were in different rooms, and all communication between them was via teletypewriter. One team member, the source (or SO), was given a completely assembled model and was required to assist the other member, the seeker (or SK), who had to build an identical model from the separate parts. It should be noted at this point that all the teams in this study were able to assemble the model correctly. Below is a portion of one of the protocols from the study. The dialogue begins at a point where the seeker has assembled what he believes is a correct model, only to recheck it and find otherwise.

SK: i think i have it let's check it over

SO: describe

SK: yellows form tri.s and the greens form 1 tri. 2 blues stick straight up while the other blue connects the two large wheels of which the green goes through one of the center holes?

SO: bzzzzz wrongo completely

SK: let me review clues

SO: looking from end to end: tri w/2 sides yellow and one side blue green st

SK: i got it!

SO: what?

SK: does the top green go through the big hole of sliding wheel?

SO: define TOP green—

SK: from side to side view the top?

Although the task performed by the subjects in this study was abstract, the nature of the dialogue is essentially what we would expect if a person could interact with a true natural-language-understanding system. Notice how many of the statements are ambiguous and how few of them are grammatically correct. There are presently no natural language systems capable of understanding such dialogue. However, we hope to build one. As a first step we are creating a system that will be forgiving of minor grammatical errors.

The common approach of completely separating syntax and semantics is too rigid for tackling the problems in designing a robust language understander. We view language as composed of many "experts" that know about some aspect of the organization of language.⁷ Each expert has something to contribute to the process of understanding, but it can offer several different views of the data. The process of understanding is one, then, of the subsystems "negotiating" among themselves to achieve a consistent view of the data, out of which comes the understanding of the text. Our approach to creating such a model is to use the notion of a society of communicating, knowledge-based, problem-solving experts, called actors.³ These experts will be "objects"¹ that can communicate by passing messages to any other expert in the system. Thus a flexible control structure is achieved—one that allows experts at any level of the analysis to talk to experts at other levels.

We are designing a system that is basically quite normative, and hence aware of errors, which then explicitly takes action in the context to accommodate anomalies. This contrasts with other approaches, where designers effectively anticipate ill-formedness by weakening the constraints built into the basic analysis scheme,^{9, 13, 15} thus achieving robustness (though this may not be an explicit purpose of the systems).

A robust system must first exhaustively explore the domain of well-formed input. Otherwise it could blunder through the analysis, forcing an erroneous analysis of a well-formed string.

An error can be detected only at an equal or higher phase in the analysis, assuming there is a hierarchy from word, to syntax, to semantics, to pragmatics. Thus for example, the semantic error in "The stone ate the rose" cannot be detected by dictionary lookup or by syntax.

We also need to establish how a sentence is judged acceptable. The goal must be pragmatic acceptance, which, given the earlier statement on the hierarchy of phases, implies that all lower phases have accepted it.

Once an anomaly is detected, one possibility is to have the system cycle through a set of metarules that designate certain constraints that can be relaxed to find one that could produce an analyzable string.¹⁰ But a blind search ignores the possibility that the error can give some insight into the problem.

When analysis is blocked, there are likely to be several points of blockage; which is taken to be the one to be reworked? Weischedel and Black¹⁴ suggest a hypothesis in which the analysis that has consumed most of the input string is taken. Even after the string to be reworked is chosen, the error will not necessarily be located at the point where its existence was detected.

We are proposing that the actor that determines an anomaly

will dispatch messages to the actors that proposed the information that led to the impasse. These actors then function to see if they can propose some way of modifying the input so that analysis can proceed. Rather than designating a subset of the rules of the system as relaxable, we are allowing an actor to examine all the rules available to it. Of course, this may be too generous; it may be the case that errors are clustered around the inapplicability of certain rules. In this case the remedial tactics of actors will have to be modified. When several proposals are offered, the problem of choosing among them has to be faced.

Consider a dialogue between an automated tutor and a student. Semantic and pragmatic errors are explicitly shown to the student to correct his misunderstanding of the activity, in this case changing a car tire.

STUDENT: With a hammer. . .

TUTOR: You mean a wrench. You have to undo some bolts.

A script expert is tracking progress. The script shows that the next action expected is the removal of bolts. The prediction includes the kind of tool to be used. A syntactic expert finds the prepositional phrase, which is interpreted as being a match for the predicted tool. But it is not of the bolt-removing kind, hence the interruption of the student by the system.

Because of the enormous processing burden required of a fully robust parser, it will be many years before such a parser is completed. In the meantime, we are pursuing the intermediate goal of designing a limited language that would retain all the user-oriented benefits of natural language while simultaneously reducing the processor's workload to currently attainable levels. In one study⁶ 24 teams solved the model assembly problem without any restrictions. The portion of the protocol shown previously is a sample of their dialogue. On the average, these teams used 545.5 word tokens to solve their problems and took 27.6 minutes. An additional 24 teams solved the same problem but were instructed to use as few words as possible. These teams used an average of only 128.6 word tokens, but, what is more interesting, they took significantly less time to solve their problems, an average of only 20.5 minutes. A complete protocol from one of these teams is shown below.

SO: 3 round, 2 yellow, 1 blue form right triangle
green through each middle
slide rounds on greens, blue between
same triangle at other end

SK: which green for 2nd line

SO: long

SK: on right triangle, which slide on green

SO: same size as other rounds: 9 holes

SK: slide on green with round on two yellow?

SO: no, slide 2 round with one blue onto 2 greens

SK: 3 parallel blues?

SO: right, also three parallel green perpendicular to blues

SK: down?

SO: yes

It should be stressed, once again, that this team, as well as all the others in this study, correctly assembled the model. Although the subjects on this team were not conversing in grammatical English, the manner in which they did converse nevertheless had all the user-oriented benefits of natural-language dialogue. For example, these subjects had no prior formal experience with this type of dialogue. Yet it is clear that the restricted subjects adapted very quickly, since they, on the average, solved their problems in significantly less time than did the unrestricted subjects. Further, none of the restricted subjects in the study complained about having to communicate in this manner. (By contrast, in a similar experiment,⁵ some subjects were required to work with a restricted vocabulary; although the restricted subjects in this study were just as fast as their unrestricted counterparts, they voiced frequent complaints about the awkwardness of their communication.)

We are presently examining, from a natural-language-processing viewpoint, the potential benefits of asking users to be concise. It is apparently a restriction on natural language that people can adapt to easily, and it may also reduce the computer's processing burden. However, even if this particular user-oriented heuristic fails to decrease the computer's burden, we are confident that we can discover and develop other user-acceptable dialogue modifications that will bear fruit.

IMPLEMENTATION ISSUES

Developing natural language front-ends for extended use in a variety of domains raises a number of implementation issues. Of these, three will be considered: Extendability of the grammar, extendability of the database, and transportability of the system to different domains.

To achieve grammar extendability, the grammar description and semantic base should be as easy to understand as possible. In addition, user interfaces to the language model in the form of model-oriented editors and a supportive executive environment are essential to grammar extendability. A well-designed language-modeling system has an additional benefit in aiding the original language model developer in the task of creation and debugging of the language model.

Current natural language systems focus on the research environment for the specialist in artificial intelligence. In most cases one must have a strong background in Lisp programming, techniques of AI programming, and linguistics. Grammars are often not directly transportable among specialists because of special features of the local programming environment. To obtain transportability and extendability for language-modeling systems there is a need to design systems that focus on language-modeling skills exclusively. Some considerations for a language modeling system are as follows:

1. Ease of use for both novice and expert modes of operation
2. Perspicuity of model representation
3. Support for a variety of linguistic theories
4. Transportability to a variety of systems and domains

Most of the time in the grammar development process is spent cycling between editing and debugging tasks. Thus an easy-to-use editor for the grammar and knowledge represen-

tation is of importance. Syntactic knowledge of grammar and of lexical and knowledge representation forms, when incorporated into the editor as special modes, can be used to enhance ease of use and efficiency of development for the model grammar.

For large systems, comprehensive utilities are needed within the system to obtain an overview of rule types or summaries of lexical structure. For example, one would like to determine "which rules reference rule NP1?" or to check to see that syntactic categories used in rules match the lexical representation. Queries to the knowledge database are also important.

A second aspect of grammar development as a process is debugging the language model. Clear displays of parsed structures, tracing facilities, and perspicuous error messages are required in this phase of the language model development. A supportive executive that handles grammar, lexicon, and knowledge database files; provides error diagnostics; and permits grammar efficiency studies through performance tools is an alternative.

An approach to developing a grammar editor for network-based models (e.g. ATNs)⁴ introduces the following functions:

- Network creation
- Arc deletion or editing
- Arc insertion
- Arc reordering
- State insertion and deletion

Illustrative of the kind of supportive functions provided by the editor are diagnosis of network integrity on creation—for example, looking for arcs that point to nonexistent nodes. Knowledge of the form of a network permits diagnosis of improperly inserted states. Warnings to the user may be given when a state is deleted that is pointed to by other nodes.

Given a specific linguistic model for developing the computational model, syntactic restrictions at all levels of design can be used in special editor modes to maintain the integrity of the implementation. By placing a series of checking functions in an editor, it is possible to filter out many potential errors before a grammar is tested. The user is able to focus on the grammar model and not on the specific programming requirements.

A more difficult problem is that of detecting execution errors in the grammar. Incorrectly parsed structures, references to non-existent lexical items, or incorrect reference to the knowledge database are but a few types of errors that are detected dynamically during interpretation of a structure. Facilities for tracing and inserting breakpoints are useful aids in solving some of the problems with debugging. A supportive executive should be used to incorporate as comprehensive and clear a set of diagnostic tools as possible. Performance monitoring is an additional facility that can be provided through the executive.

Development of a relatively easy-to-use, transportable grammar design system can make possible more extensive use of grammar modeling in education, the applied linguistics environment, and linguistics research, in addition to providing features that support extendability of computational models of language for the end user. The issue of developing interfaces for user extension of grammars in natural-language-processing systems can be more effectively handled within a sys-

tem that focuses on a model environment with an associated editor and supportive executive.

Another issue of extendability is extending the database for the system. Some databases, say an employee data collection, are very neatly structured, and adding new facts is simple. In the case of the employee database we could just add a new record for a new employee. However, many databases are not structured in any neat way and are constantly changing. An example of this is the database in the CYRUS program written by Janet Kolodner at Yale University.⁹ In this program the database to be addressed is knowledge about Cyrus Vance, then Secretary of State. This program was given information about Vance's travels and duties and added these into its database for later question-answering retrieval. This sort of database is needed in some of the applications the TI group is looking into.

One of the major problems in dealing with these sorts of databases is the knowledge representation used. The representation must be flexible enough to allow the database to be extended, but fixed enough to let one access the data already stored. The data must be parsed into this knowledge representation, and the memory must be updated accordingly.

The key to accessing this information is a good question-answering system. This must be able to determine the intent of the user's question and find the appropriate information in the database. Imagine a library system with information about medical books and papers. The user may ask, "What books deal with disease treatment?" If the system is not able to access information about cancer, diabetes, etc., as "disease," the question will not be answerable. Similarly, if the user asks about cancer, the system must be able to examine information about disease.

Another feature of this sort of system is that the user must be able to add to the information about a domain. In the medical example, a user may want to add the information that, for example, "Hypoglycemia is a blood-sugar-related disease." The system must be able to add this new information to its database. In a more complicated example, say a generalized library system, the user may wish to add a whole new domain to the systems knowledge base (say, science fiction books). An editor, similar to that described earlier for adding grammatical information, is needed to add to the semantic knowledge of this system. The user must be able to relate the new knowledge to the existing knowledge in the representation. He must also be able to add new word definitions, new word senses of existing words, and new semantic categories to existing word senses.

Another important implementation issue is the study of transportability. By *transportability* we mean the ability to move from one domain to another with the same front-end program. The most important questions here are the following:

- What parts of the program have to be changed when moving to a new domain?
- How much of the program has to be changed when moving to a new domain?

Clearly the best case would be to implement a natural-language front-end that would work for any domain without

change. Given current technology, this is not possible. Nonetheless, we do not think the idea of developing a transportable natural language front-end should be abandoned. Even if it is not possible to write a completely transportable system, a large portion of the system should in theory be able to remain the same. We propose to isolate the portions of the system where changes will be required and investigate what sorts of changes are required to each of these pieces. This investigation will provide crucial information necessary for the design of a natural-language front-end which is transportable without making massive unspecified changes. We contend that a natural-language front-end which meets the criteria of being transportable must be modular and that the choices of how to segment the system into modules must be strongly influenced by the kinds of changes required to the modules when moving from domain to domain. Thus each module will require either very specific changes or no changes when changing domains.

For the most part, existing systems are not at all transportable. Moving from one domain to another usually requires massive changes, and these changes are generally not isolated to specific pieces of the system. They must be scattered throughout the code. Implementers of natural-language front-ends often make off-the-cuff remarks about their systems being transportable, but no one has yet demonstrated that a system is indeed transportable. Furthermore, there have been no discussions of what sort of work it would take or has taken to move from one domain to another. If natural-language front-ends are ever to be useful tools, they must exist for a wide range of domains. If we had to start from scratch and generate a totally new system for each domain, the job would be insurmountable. New domains that required natural language would be generated far faster than new natural-language front-ends. If we are unable to use one system without change for all domains, then we must at least understand the process of going from one domain to another well enough to do the conversion in reasonable time. Furthermore, if this process were reasonably well understood, we could consider the problem of automating the change as much as possible. Thus, the change could be accomplished by the computer after consulting with experts on the domain in question. This latter project is clearly quite difficult; however, we see the investigation of transportability that we are proposing to be a necessary predecessor to the design of this automated system.

CONCLUSION

The lines of research outlined in this paper are certainly not the only ones that need to be pursued. However, we believe that the research we are doing is extremely important to the ultimate success of natural-language-understanding systems. Of course, our beliefs can be confirmed only after our work is completed.

REFERENCES

1. Birtwistle, Graham M., Dahl, Ole-Johan, Myhrhaug, Bjorn, and Nygaard, Kristen. *Simula Begin*. Lund, Sweden: Studentlitteratur. 1973.

2. Chapanis, A., Parrish, R.N., Ochsman, R.B. and Weeks, G.D. Studies in interactive communication: II. The effects of four communication modes on the linguistic performance of teams during cooperative problem-solving. *Human Factors*, 1977, 19, 101-126.
3. Hewitt, Carl. Viewing control structures as patterns of passing messages. A.I. Memo 410. Cambridge, MA: MIT AI Laboratory, 1976.
4. Kehler, T.P. and Woods, C.A. ATN Grammar Modelling in Applied Linguistics. *ACL 1980 Conference Proceedings*.
5. Kelly, M.J. and Chapanis, A. Limited vocabulary natural language dialogue. *International Journal of Man-Machine Studies*, 1977, 9, 479-501.
6. Michaelis, P.R. Cooperative problem solving by like- and mixed-sex teams in a teletypewriter mode with unlimited, self-limited, introduced and anonymous conditions. *JSAS Catalog of Selected Documents in Psychology*, 1980, 10, 35-36. (Ms. No. 2066)
7. Phillips, Brian, and Hendler, James A. The impatient tutor: an integrated language understanding system. *Proceedings of the International Conference on Computational Linguistics*, Tokyo, 1980, pp. 480-486.
8. Schank, Roger C. *Conceptual Information Processing*. NY: American Elsevier. 1975.
9. Schank, Roger C. and Kolodner, Janet. Retrieving Information from an Episodic Memory, or, Why Computers' Memories Should Be More Like Peoples. Yale University Depart. of Computer Science Research Rpt. 159 1979.
10. Sondheimer, Norman K., and Weischedel, Ralph M. A rule based approach to ill-formed input. *Proceedings of the International Conference on Computational Linguistics*, Tokyo, 1980, pp. 46-53.
11. Tennant, Harry. Evaluation of natural language processors. Ph.D. Thesis, University of Illinois, 1981.
12. Thompson, B.H. Linguistic analysis of natural language communication with computers. *Proceedings of the International Conference on Computational Linguistics*, Tokyo 1980.
13. Waltz, David L. An English language question answering system for a large relational data-base. *Comm. ACM* 21, 1978, 526-539.
14. Weischedel, Ralph M., and Black, John E. Responding intelligently to unparsable inputs. *American Journal of Computational Linguistics* 6, 1980, 97-109.
15. Wilks, Y. A preferential, pattern-seeking, semantics for natural language inference. *Artificial Intelligence* 6, 1975, 53-74.

Text-critiquing with the EPISTLE system: an author's aid to better syntax

by LANCE A. MILLER, GEORGE E. HEIDORN, and KAREN JENSEN

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT

The experimental EPISTLE system is ultimately intended to provide office workers with intelligent applications for the processing of natural language text, particularly business correspondence. A variety of possible critiques of textual material are identified in this paper, but the discussion focuses on the system's capability to detect several classes of grammatical errors, such as disagreement in number between the subject and the verb. The system's error-detection performance relies critically on its *parsing* component which determines the syntactic structure of each sentence and the grammatical functions fulfilled by various phrases. Details of the system's operations are provided, and some of the future critiquing objectives are outlined.

A. INTRODUCTION

EPISTLE is the name of both a research project and a software system—a system in which the project goals are being implemented. The acronym is to be read as “Evaluation, Preparation, and Interpretation System for Text and Language Entities.” As a project, the long-term objectives are to provide office workers—particularly mid-level-management principals—a variety of applications to assist in their interaction with natural language texts. We are focusing initially on business correspondence and are working toward the eventual development of two classes of applications. The first is concerned with the processing of incoming texts, providing for the principal—qua administrator—such services as synthesizing letter contents, highlighting portions known to be of individual interest, and automatically generating document index terms based on conceptual or thematic characteristics rather than “key words.” The second class of applications would provide services for the principal—qua author—primarily in terms of a series of increasingly sophisticated critiques of the latest draft of a letter or other text (see Miller 1980). EPISTLE as a system addresses only this second kind of application at the present time. It is this working—but highly dynamic and experimental—system that is the primary focus of the paper.

We first provide a perspective on possible text critiques (Section B) and then discuss the two very limited areas in

which we are presently making some headway: the assessment of features violating some sense of “good style”, and the quite definite detection of true grammatical errors. Although we describe the characteristics of the former, almost all of our actual implementation has been in the second of these areas; and we characterize in detail the 14 classes of textual phenomena which constitute our grammatical critique targets (Section C). The following two sections discuss our implementation for these target classes with an overview of the EPISTLE architecture and our parsing philosophy (Section D), followed by an outline of our approach to syntactic error detection, with two detailed examples of specific detection rules (Section E). Finally, we conclude with a brief discussion of our longer-range objectives (Section F).

B. SOME CRITIQUE POSSIBILITIES

Our classification of critiques involves three separate factors: (1) the different hierarchical levels in text, (2) the perspective from which units at a particular level are examined, and (3) the nature of this examination. Concerning the first factor, we list the five commonly accepted levels: character, morpheme (minimal “meaning unit”), word, sentence, and paragraph; for completeness, we would like to add two further levels—those of “chapter” and “book,” the latter being the unit for which some long-term unified communication purpose has been achieved. Concerning perspectives, we feel it is useful to consider the examination of texts from three different viewpoints: those of syntax, semantics, and pragmatics (the triumvirate concepts of “semiotic” theory; cf. Morris 1946). Finally, we distinguish between two types of examinations of texts: that which examines a text unit (from some perspective) and pronounces it correct or incorrect—Type I—and that which makes a much more graded judgment of acceptability, based on comparison of the features of a unit to some set of “standards” (not necessarily well-formed)—Type II.

These factors combine to form a total of 42 types of critiques, and for each type of examination there may be a number of separate judgments (as the present EPISTLE system makes 14 different Type I judgments of sentence grammaticality). We recognize, however, that for levels above the sentence, or for perspectives other than that of syntax (the “meaning” or the “purpose” of the text), the standards for

making Type I judgments are much less than universal. To retain the taxonomy, in view of the unquestioned appropriateness of the Type I-Type II distinction for syntax, we suggest that when there is less than universal agreement about standards of correct-incorrect, the reference or source of the authority be characterized as to the model which enables the judgments of correct or not and also as to the specific application domain of the model and of the text. Thus, to make a Type I judgment of a text from a pragmatic perspective, we would first identify the domain involved: say, that of business correspondence and, even more particularly, the type of letter one writes when applying for a job. Second, we would identify the reference model for the Type I judgment: in this case, that skeleton provided by Wilkinson et al. (1980, p. 356) detailing the ideal composition and sequence of points to be made for such a situation. Thus, if one such letter does not "generate interest from the start"—contrary to Wilkinson et al.'s dictum—one may then quite firmly make the Type I judgment of incorrect.

C. CRITIQUE OUTPUTS FROM THE EPISTLE SYSTEM

With respect to the three factors of text level, perspective, and judgment type just defined, our present work can be described as (1) focusing on the sentence level, (2) concerning only the syntactic perspective, and (3) emphasizing Type I critique-judgments over those of Type II. At the time of this writing, only Type I critiques have been programmed and tested. Nevertheless, the software capability to provide Type II acceptability judgments has at least been examined for several factors. Our plans and current work for the latter type of critique are briefly described first, and then the remainder of this paper is concerned only with the Type I critiques.

1. *Type II Acceptability Critiques*

This is the type of critique most emphasized by those very many "how to" books concerned with writing of all kinds, and it is particularly emphasized by those more specialized works intended specifically for the writing of business letters (e.g., Bates 1978; Cloke & Wallace 1969; Prentice-Hall 1980; Wilkinson et al., 1980). Typically, for every page of discussion on the importance of Type I syntactic critiques such as subject-verb agreement these must be 50 pages on the Type II nuances of good and bad "style." To some extent, however, this outpouring of advice and counsel is symptomatic of the complexity of these types of judgments: whether something is an instance of good or bad style seems to depend on a large variety of interacting factors, and the books are long on examples but almost mute concerning concrete feature lists or objective evaluation procedures.

Two basic implementation strategies for detecting such stylistic aspects are either to follow a simple brute-force storage and recognition of all of the examples of a particular Type II rating dimension, or else to distill from the many examples those key features whose weighted Boolean combination can be tuned to predict "expert" human judgments. An example

of a critique that would seem to require this latter approach is the very complex and many-faceted notion of readability. Although simple sentence-length and word-length factors underlie most quantitative measures of what is called "readability" (e.g., the Gunning Fog index; see Dyer 1962), these are intended to be very general indicators of the overall readability of a long text, and there is no evidence that they are at all indicative of individual sentence understandability. Aspects of unclear writing—such as ambiguity of reference or of scope, excessive modification, extreme abstraction, and awkward, deeply embedded, or unbalanced syntactic constructions—can occur for even shorter sentences, independently of word length. Interestingly enough, however, many of the features that appear to underlie the examples of "poor style" appear to be syntactic rather than semantic in nature. The syntactic features our initial analyses suggest as being useful for evaluating individual sentence readability all appear to be determinable from the approximate parses of sentences returned as one of the outputs from the syntactic component of our system (see Section D). We therefore plan to implement the detection of sentence readability and other problems using a (weighted) combination of syntactic features derived from the parse tree; we will then evaluate the utility of this syntax-only detection approach by attempting to predict subjects' judgments of readability collected in behavioral experiments. Even if minimal or no semantic information is required to detect style problems, we expect that the needed iterations between reprogramming and behavioral testing will probably delay the implementation in EPISTLE of this more complex type of critique for some time.

Concerning the former "brute-force" implementation possibility, there is actually one Type II situation for which it would appear to be a reasonable and workable solution. This is discussed in every book and concerns the use of so-called objectionable phrases—objectionable in the sense of being overworked, outdated, stilted, unnecessarily lengthy, excessively formal, or obscure (e.g., "along the same lines as," "due to unforeseen circumstances," "effect an alternative procedure"). We estimate that these phrases number around 600–1200, or even fewer if highly similar patterns are combined (e.g., singular-plural differences), and we believe we have collected most of the major ones. Detection of these phrases will therefore be one of the first implemented Type II critique aspects.

Some of our intended system extensions, such as that for questionable phrases, are already implemented in another text-critiquing project at Bell Laboratories which has developed a set of programs collectively known as the "Writer's Workbench." These programs provide a variety of what we would call Type II syntactic evaluations including text-level summaries—e.g., readability, sentence type and length distributions, breakdown of the parts of speech of sentence openers—as well as specific sentence-level critiques, e.g., identification of sentences having excessive length or having questionable punctuation or word choice (Cherry 1978, 1980; Macdonald 1979). These programs are intended for internal use and have apparently been received quite warmly. In contrast to our parser-based approach, the Writer's Workbench programs rely on morphological and word-pattern analyses to achieve rather remarkable levels of accuracy in the part-of-

speech assignments that appear to underlie many of their diagnostics.

2. Type I Syntactic Error Detection

Errors of this yes/no grammatical/ungrammatical type should theoretically have been prevented by the public grade school educational process which, for most citizens, was supposed to have dealt once and for all with these issues. Nonetheless, such errors do appear with alarming regularity in even the most prestigious scientific journals, in the most carefully written letters, and out of the most learned mouths. These errors have no gradations of incorrectness as there are for the aspects of poor style; a grammatical error is either present or it is not. There is, therefore, something of a "clean-cut" aspect to the detection of grammatical errors, each revealed typically by simultaneously occurring disallowed feature combinations. We add, however, that the rules required to achieve the error detections are dependent upon a wide variety of factors, just as we expect the Type II evaluations to be.

The 14 classes of syntactic errors now detected by the EPISTLE system, with examples, are as follows:

1. *Disagreement in number between subject and verb*—Your blueprint and statement of correction does not contain . . .
2. *Disagreement in person between subject and verb*—I almost always knows . . . /He never write . . .
3. *Disagreement in number between determiner and noun*—This letters . . . /These letter . . . /A meetings . . . /
4. *Disagreement in number between quantifier and noun*—Both letter . . . /Each men . . . /Several of the book . . .
5. *Disagreement in number between relative clause and head noun-phrase*—The men who writes the letter . . .
6. *Use of object pronoun in predicate nominative position*—If I were him . . . /I know that it was her . . .
7. *Use of object pronoun in subject position*—Sally and me wrote the letter . . . /Her writes well . . .
8. *Use of subject pronoun in direct object position*—I know he who writes . . . /They saw Mike and I . . .
9. *Use of subject pronoun in indirect object position*—I gave she the letter /They told Sam and I the story . . .
10. *Use of subject pronoun in prepositional phrase*—Between Eva and I, we wrote the letter . . .
11. *Use of "who" ("whoever") for "whom" ("whomever")*—Who did you give it to? /I know who you mean . . .
12. *Use of "whom" ("whomever") for "who" ("whoever")*—Whomever wishes to go now may do so . . . /I'll give this to whomever asks for it . . . /Whom did you suppose was coming?
13. *Use of "of" for "have" in auxiliary position*—You could of gone . . . /I should of written . . .
14. *Use of improper verb form*—The letter was wrote by him . . . /I have write the letter . . .

Detection of each of these types of errors has been implemented for a wide set of circumstances, but testing of the system by users instructed to attempt to "fool" it (by having

it classify correct instances as errors, or tricky errors as correct) has not yet been done.

For demonstration purposes we created a contrived (but representative) letter, which illustrates how various errors might occur in coherent text. This letter contains seven syntactic errors, of the types described above. In addition, the letter contains five errors of two further types, for which the EPISTLE system now has limited detection capability. The first is

Incorrect use of a homonym (which may be perceived by the author, when detected, as a misspelling)—e.g., "This is to much . . . /That film has terrific lighting affects . . . /I except your apology and offer you mine in return . . .

and the second is what we call

Incorrect word choice—e.g., "We are anxious to receive your answer" (should be "eager to")/"The book will be discussed first among the two of us and then between the three of you" (prepositions should be reversed).

Both of these types probably will require, for generalized detection, some level of semantic information, which is not now implemented.

There is another very general type of grammatical error which is, in one way or another, involved in all of the other errors and may be characterized as

Unallowed part-of-speech sequence—e.g., "The the . . . / Our slowly car . . . /In all respectfully we . . . /He sang all through the through the night . . .

We expect to be able to detect most errors of this type because our grammar is tightly tuned to reject all but grammatical sentences (but we do not yet have techniques for determining just which words caused the problem in this more general class).

Dear Mr. Jones:

We received your note of August 6, 1980, that asks for confirmation of your recent application.

Although we appreciate your desire for a prompt reply now, we cannot /1/except/ your forms at this point. Unfortunately, your /2/statement/ of deficiencies /2/have/ not been completed for /3/several reason./

First, your /2/blueprint and your plan/ of building correction /2/does/ not show enough detail. Second, /3/both/ of the /3/variance/ must be requested from the Industrial Board. Third, the Fire and Panic Regulations will have an /1/affect/ on /4/these plan./

If you wish to complete /4/a new plans/ of correction, please contact Jim Brown of our department. The person in charge of such applications is /5/him./

We are happy /1/too/ have received your correspondence and are /6/anxious/ to complete this matter /1/to./ Thank you for your interest.

Sincerely,
Thomas Brown

The preceding constructed letter aggregates and illustrates some of the syntactic errors detected by the EPISTLE system, in the context in which they might plausibly occur. The letter contains 12 separate errors of the six categories given below. The words signifying a particular error are bounded by slashes, with the error identification NUMBER shown on the left (i.e., the form “/NUM/... WORDS...”).

The syntactic-error categories are (1) use of incorrect homonym, (2) disagreement in number between subject and verb, (3) disagreement in number between quantifier and noun, (4) disagreement in number between determiner and noun, (5) use of object pronoun in predicate nominative position, and (6) inappropriate word choice.

D. DETAILS OF SYNTACTIC PROCESSING IN THE EPISTLE SYSTEM

In this section we first discuss the general processing characteristics of the EPISTLE system, particularly the syntactic aspects, and then describe our parsing philosophy as it is represented in the grammar and as it underlies our syntactic-error detection.

1. General Description

EPISTLE as a system designed to accomplish a variety of text-critiquing activities is built upon a general-purpose natural language processing system called NLP (cf. Heidorn 1972, 1975). NLP itself is embedded in LISP and comprises, in addition to the control supervisor, three major components: a dictionary, an input decoder, and an output encoder. The dictionary contains stems of words and associated part of speech and inflection information. Along with the dictionary there is a set of morphological rules that permit recognition of the word stem as it occurs with prefixes and suffixes to form a complete word. The output encoder permits the translation of internal representations into natural language output whose syntax or grammatical structure is controlled by the encoding rules. While this component will figure importantly in the longest-range goals of the project, it is secondary to the present critiquing capabilities of the system.

The input decoder is the most important component of the system for text-critiquing applications. It translates input text into internal representations consisting essentially of lists of attribute-value pairs. The two major aspects of the decoder are a set of grammar rules and a parsing algorithm that determines how the rules are applied and how intermediate and final results of rule applications are represented and used. The parser involves a strictly left-to-right, bottom-up, parallel-processing algorithm in which all rules that can be applied to the input string at a particular time are applied (subject to a system variable that controls which decoding rules are “visible” to the parser at any time).

The collection of decoding rules forms the system’s grammar, known as an Augmented Phrase Structure Grammar (APSG) because it consists of essentially context-free phrase structure rules augmented by arbitrary conditions and structure-building actions. The schema of an NLP decoding

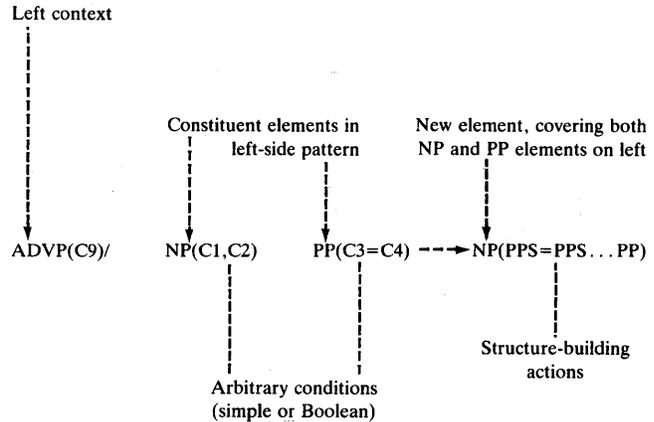


Figure 1—Example of an NLP Augmented Phrase Structure Grammar decoding rule used in the parsing component of the EPISTLE system

rule shown in Figure 1 illustrates some of the rule features. The structure of the rule is best seen by mentally masking out all components contained in parentheses; doing so, a context-free type of rule is apparent. The left side pattern in this example can be seen to be an adverb phrase providing the left context for a two-element pattern composed of a noun phrase and a prepositional phrase (the slash “/” is the character that separates the context part); the right side consists of a single noun phrase, indicating that the prepositional phrase is to be incorporated into the noun phrase. The elements within the parentheses provide, for the left side pattern, the conditions that must hold for each pattern element if complete recognition is to occur; in the example these conditions are expressed abstractly (e.g., as “C1, C2”) but stand for any test involving attribute values, or any Boolean combination of such tests. On the right side, the information within parentheses concerns structure-building actions, with the example illustrating the addition of the new prepositional phrase to a list of any previous such phrases.

At this time our grammar consists of about 200 such decoding rules (although they are much more complex than the example). In addition, there are about 200 simpler decoding rules concerned with morphology—the recognition of words, based on a character-by-character processing of the input. Finally, there are some 50 decoding rules for handling the detection and correction of syntactic errors.

Concerning performance, we correctly parse a wide variety of English syntactic structures, including a major portion of those in our database of 400 business letters (which include examples of as tortured a sentence syntax as one is likely to find outside of Faulkner and Joyce). A representative example of the somewhat longer sentences is the following: “How nice it was to receive your letter complimenting our manager, Bob Halby, on his service and courtesy to you while you were shopping in our Buy-Now store in Harrison.”

2. Parsing Approach

We have held to three strong views in developing our grammar. The first, rather unorthodox, position is that we should

avoid using semantic information in the parsing of sentences. That is, the grammar rules should be principally based on the part of speech and inflectional characteristics of words, and should not depend on what a word means. This view is based on four presumptions: (1) that the transferability of a critiquing system among different applications is maximized when parsing does not involve semantics (because word meanings differ so much from application to application); (2) that there is no principled way of limiting the extent to which semantic processing may be required; (3) using just a moderate degree of semantics for syntactic processing is not a good guarantee that the sentence will be parsed correctly; in many cases—as in “I hit the man in the street”—all the lexical semantics in the world would be insufficient to resolve whether “in the street” should be viewed as a description of the man or a locative concerning the hitting; and (4) given that most parsing projects have employed semantics to one degree or another, perhaps the opportunities for squeezing more resolution out of purely syntactic aspects have not yet been completely exhausted.

Our second parsing tenet, held generally, is that the number of separate parses obtained for a single sentence should be minimized—at least to only those that correspond to different legitimate interpretations. We go further, however, and hold that we ought to get no more than one parse per sentence, no matter how many structural or constituent ambiguities. This is not to say that we ignore or are otherwise uninterested in significantly differing parses; it is just that we prefer to have such information as an attribute of a single parse (so that we can obtain the alternatives later if necessary).

The third tenet is that, for the highly purposive activities we intend to support, the theoretical distinctions of competence vs. performance or surface vs. deep structure have little utility. Rather, we take the very direct view that every communication act is actively designed, however unconsciously, to fulfill specific pragmatic objectives; further, the level of design includes the selection of the sequence of syntactic structures believed best fitted to achieve these objectives. Thus, the actual surface structure of a sentence—not any presumed deep progenitor—is the key syntactic entity, the structure chosen deterministically by the communicator to best represent the thought.

It is one thing to hold such strong views, and it is quite another to realize them in practice. As might be expected, something had to give, and we abided by these views at the potential expense of the accuracy of our parses. Rather than seeking to achieve whatever linguists might be able to agree upon as a highly accurate parse of the *surface structure* of a sentence, we decided to make somewhat arbitrary decisions in situations of ambiguity and achieve what we call an “approximate surface parse.” Two of the most important of such decisions are (1) to attach structurally uncertain constituents (like post-object prepositional phrases) to the next higher verb phrase, and (2) to arbitrarily specify a preference order, for cases of constituent rather than structural ambiguity (when a text unit can be assigned to different grammatical categories), so that certain constituent assignments will be the first attempted (e.g., for “I hit the man in the street,” we would force the parse in which the prepositional phrase serves as the location of the hitting). However questionable these views

might be on paper, in practice things have turned out rather well: we seldom have more than one parse, and that parse is a reasonable approximation of what might be considered the ideal parse.

While we have accepted an approximate surface parse as a compromise for the moment, our longer-range plans include rectification of any parse inaccuracies. We plan to pass the parse information from the syntactic component to the to-be-developed semantic unit to determine the best “fit” of meaning to the sentence, as a function of the syntactic structure, the lexical semantics, and the semantic information obtained from the previous text. If the semantic information ultimately suggests that some part of the assigned syntactic structure is erroneous, alternative parses may have to be obtained or, more simply, syntactic substructures can simply be reattached to their proper places.

E. DETAILS OF ERROR DETECTION IN THE EPISTLE SYSTEM

In this section we first discuss the general approach used for doing syntactic error detection and then describe two example error detection rules written in the NLP APSG rule language.

1. *The General Approach*

The general approach used for doing syntactic error detection consists of basically three steps:

1. We attempt a parse of a given English sentence, using fully grammatical syntax rules (where “fully grammatical” includes restrictions on, for example, number agreement between subject and verb). Only sentences that fit the constituent class patterns and obey all restrictions on the patterns will parse successfully.
2. If the program is unable to assign structure to the sentence, we then try to parse again—this time with the restrictions relaxed, and with the help of some additional rules.
3. If the sentence parses this second time, then the program will produce a tree, diagnose the error, and display the sentence twice, once brightening the focus of error, and the second time substituting the correct form in its place.

The general approach stated here is quite similar to techniques described in Weischedel and Black (1980) and in Kwasy and Sondheimer (1979), although in each of those papers the implementation described is in terms of augmented transition network grammars (ATNs, e.g., Woods 1970), rather than APSGs.

2. *Detecting Use of “who” for “whom”*

The following NLP decoding rule comes into play in parsing a sentence that contains a relative clause beginning with a relative pronoun which is to be considered to be the object of

the verb of the relative clause, e.g. "The man *whom I know* lives here":

```
PRON(REL,-,"WHO") VP(TRANS,SUBJECT,-OBJECT)→
  VP(RELCL = εVP,PRMODS = PRON ... PRMODS,
    OBJECT = PRON)
```

This rule says that if a relative pronoun other than "who" is followed by a transitive verb phrase (actually a clause) that already has a subject but no direct object, a new verb phrase can be formed to cover these two segments. This new VP is marked as a relative clause, and the relative pronoun is picked up as a premodifier of the head verb and is also called the direct object. (The equal sign is the assignment operator, and the cent sign means to make a copy of a record.)

If the sentence being processed had erroneously been written with "who" instead of "whom", e.g. "The man *who I know* lives here", the above rule would not be applicable and the sentence would not be parsed during the first step of the processing described above. However, during the second step the following rule is added to the set of rules to be considered, and in fact would be applicable, resulting in a parse:

```
PRON("WHO") VP(TRANS,SUBJECT,-OBJECT)→
  VP(RELCL = εVP,PRMODS = PRON ...
  PRMODS,OBJECT = PRON,
  ERRORS = ERRORS ... <"ERROR",TYPE =
  "PNCASE2",
  EHEADB = PRON,CSEGS = <εPRON,'WHOM' > >)
```

This rule is similar to the one above, but requires that the relative pronoun be "who." When this rule is applied, in addition to describing the new segment formed with the same information as above, it is also given another attribute (ERRORS) with information about the error made by the writer. This additional information is itself in the form of a record with attributes to specify the type of error, where it occurs in the sentence, and what correction should be made. If the segment record created by applying this rule ends up as a node in the parse tree, this information will be used to produce a diagnostic message to the writer.

We consider the idea of having separate error-detection rules of this sort as temporary in the current implementation, and intend to replace them with a scheme that would simply allow restrictions in the original rules to be relaxed semi-automatically, much in the manner described in the papers by Weischedel and Black and by Kwasny and Sondheimer, cited above.

3. Detecting Use of "Of" for "Have" After a Modal

An error that occurs commonly in spoken English and sometimes in written English is to use "of" instead of the contracted form of "have," because they sound alike, e.g., "I should *of* gone" instead of "I should've gone." Such a sentence would fail to parse during the first step of processing with our grammar. However, during the second step the following rule is added to the set of rules to be considered and would be applied in such a case:

```
VERB(MODAL)/ PREP('OF')→
  VERB('HAV',-INDIC,INF,ERRORS = <'ERROR',
  TYPE = 'AUX1',EHEADB = PREP,
  CSEGS = <εPREP,'HAV',INF,SEGTYPE = 'VERB' > >)
```

This rule says that if the preposition "of" is preceded by a modal, treat it as if the verb "have" had appeared there instead. (Note the use of the modal as a left-context element.) As with the error detection rule shown above, this rule creates a record describing the segment as it should be and also associates with it information about the error made.

F. FUTURE DIRECTIONS

In the near future we expect to implement a variety of Type II syntactically-based stylistic evaluations, such as those provided by the Writer's Workbench. In our very long range objectives, however, we hope to shift at least two of the three critique factors, and move from the sentence to the paragraph level, as well as change from a syntactic to a semantic perspective. For example, we would like to be able to represent the meaning of a sentence as a series of related propositions, and then assess the degree of continuity between these propositions and previous ones, in particular being guided by a few pattern models typifying acceptable exposition principles. Two such models that seem to characterize some of our letters are a parallel pattern in which the initial one or two sentences establish a variety of propositions to be subsequently dealt with in turn, and a sequential narrative-type model in which the previous sentence's propositions are those to which the next sentence's propositions most closely relate. Thus, paragraphs having irregular, nonmodel associations among successive sentence propositions could be judged as much less connected or cohesive than those that conform well to a model.

REFERENCES

1. Bates, J.D., *Writing with Precision*, Washington, D.C.: Acropolis Books Ltd., 1978.
2. Cherry, L.L., "PARTS—A System for Assigning Word Classes to English Text," *Bell Laboratories Computing Science Technical Report*, No. 81, 1978.
3. Cherry, L.L., "Writing Tools—The STYLE and DICTON Programs," Unpublished Bell Laboratories Report, 1980.
4. Cloke, M. and Wallace, R., *The Modern Business Letter Writer's Manual*, New York: Doubleday and Co., Inc., 1969.
5. Dyer, F.C., *Executive's Guide to Effective Speaking and Writing*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1962.
6. Heidorn, G.E., "Natural Language Inputs to a Simulation Programming System," *Naval Postgraduate School Technical Report*, No. NPS-55HD72101A, 1972. (Copies are available from the author at IBM Research).
7. Heidorn, G.E., "Augmented Phrase Structure Grammars". In *Theoretical Issues in Natural Language Processing*, B.L. Nash-Webber and R.C. Schank (Eds.), Association for Computational Linguistics, 1975.
8. Kwasny, S.C. and Sondheimer, N.K., "Ungrammaticality and Extra-Grammaticality in Natural Language Understanding Systems," *Proceedings of the 17th Annual Meeting of the Association for Computational Linguistics*, La Jolla, Calif., 1979, pp. 19-23.
9. Macdonald, Nina, "Pattern Matching and Language Analysis as Editing Support." Paper presented at the American Educational Research Association meeting, Boston, April 1979.
10. Miller, L.A., "A system for the Automatic Analysis of Business Correspondence."

-
- dence," In *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford University, 1980, pp. 280-282.
11. Morris, C., *Signs, Language, and Behavior*, New York: Prentice Hall, Inc., 1946.
 12. Prentice-Hall, Inc., us. *Director's and Officer's Complete Letter Book* (prepared by the editorial staff; 20th printing), Englewood Cliffs, N.J. 1980.
 13. Weischedel, R.M. and Black, J.E., "Responding Intelligently to Unparsable Inputs," *American Journal of Computational Linguistics*, Vol. 6, No. 2, 1980, pp. 97-109.
 14. Wilkinson, C.W., Clarke, P.B. and Wilkinson, Dorothy C.M., *Communicating through Letters and Reports*, 7th edition, Homewood, Ill.: Richard D. Irwin, Inc., 1980.
 15. Woods, W.A. "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, Vol. 13, 1970, pp. 591-606.

Shifting to a higher gear in a natural language system

by BOZENA HENISZ THOMPSON

and

FREDERICK B. THOMPSON

California Institute of Technology
Pasadena, California

ABSTRACT

We have completed the development of the REL System, a system for communicating with the computer in natural language concerning a relational database. We have been using that system in a series of experiments on how people actually do communicate in solving an intellectual task. These experiments, together with our general experience with REL, and related work elsewhere, have led us to the specification and development of a new system, the POL (Problem Oriented Language) System. POL is an evolutionary extension of REL, preserving what has worked, and extending and adding new capabilities to meet observed needs. These improvements include more responsive diagnostics, handling of sentence fragments, inter knowledge base communications, and new facilities for building and extending the knowledge bases of users. This paper introduces POL.

INTRODUCTION

We have completed the development of the REL System, a system for communicating with the computer in natural language concerning a relational data base. We have been using that system in a series of experiments on how people actually do communicate in solving an intellectual task. These experiments, together with our general experience with REL, and related work elsewhere, have led us to the specification and development of a new system, the POL (Problem Oriented Language) System. POL is an evolutionary extension of REL, preserving what has worked, and extending and adding new capabilities to meet observed needs. This paper introduces POL.

KNOWLEDGE BASED SYSTEMS IN THE RAPIDLY CHANGING ENVIRONMENT

The continuing rapid increase in both the capability and availability of computers has raised the expectations of what they can do for us. In the near future they should be able to re-

spond intelligently to directions we give to them in our own natural language. Intelligent response to natural language implies more than the understanding of the structure of language. It also implies knowledge of the meanings of the technical terms we use in dealing with the complex problems of our work domains, and the ability to use that knowledge in formulating responses. In communicating about technical matters, we make use of many facts and relationships that are tacitly understood. Thus the computer must not only have available to it a body of facts, a database, but knowledge of the relationships that tie that data together. We will refer to this wider body of knowledge as a "knowledge base."

A knowledge base concerning a given subject area contains the relevant data concerning that area; the notion of knowledge base incorporates and extends the notion of database. When one queries a database, one expects to get back only the raw data one asks for. Most database systems go somewhat beyond simple recitation of data that has been put into them, providing, for example, statistical reduction of that data. A knowledge base goes well beyond this, incorporating knowledge of the domain that it can use in digesting the query in conjunction with its data.

A simple example may be useful. Consider a system concerned with the loading of cargo ships. One can instruct it to put various items into the ship's cargo spaces. If the system is knowledgeable, it will be able to answer a query concerning the remaining area available in a given cargo space, for it will know that the remaining area is the total area less that occupied by shipments and that it can compute the area occupied by a shipment from the dimensions of each particular type of shipment, dimensions contained in its data.

Knowledge is closely associated with language. Certainly it is knowledge of the language that gives a system the capability to respond to natural language queries. In the above example it was knowledge of the term "remaining area" that allowed the system to give a useful reply concerning the remaining area of a cargo space after a variety of shipments had been placed there. Thus a knowledge base is a language-database package where the language component includes the semantics of the domain of application.

Knowledge base systems cover a wide range. We first characterize this range and then identify that area within this range where our interests lie. At the low end of this spectrum are the programming languages, such as Fortran, Pascal, and Coniver; they know nothing about the domain of the user. The high end of the spectrum is open ended, as there is no "most knowledgeable" system.

The most intelligent systems now are typified by the medical diagnostic systems such as MYCIN. In these systems, the physician who is not a specialist in a particular field of medicine can call up the computer, hold a highly technical dialogue concerning the history and symptoms of his patient, and expect to get diagnostic information reflecting the knowledge of the best specialists in the field. We point out several properties of such systems. The task of putting into a system the best knowledge available in the field is expensive in both time and resources. Further, the source of this knowledge is not the user, but experts removed in time and place. To make such systems economically viable, the domain must be stable, the technical terms of the domain widely known and unambiguous, the application must be widespread and important, and the expectation must be that the knowledge base will change only slowly with time.

In contrast to this type of knowledge domain, there is the knowledge base of the typical research team, management staff, or administrative office. A research team may be designing, constructing a prototype and testing a new device; the inventory control staff of a firm may be keeping track of and reordering a large variety of parts; a government agency may be administering contracts and evaluating proposals. In each of these cases the user is intimately involved in the maintenance of his or her knowledge base. It may appear that the structures of these respective knowledge bases are rather static; however, this is not the case. Indeed it is the constant shifting in structure of the knowledge base and its associated vocabulary that mark these organizations that must operate in a constantly changing environment. Not the least of these changes is in the personnel themselves, each with their own ways of doing things, ways which must be reflected in the base. These knowledge base systems are the properties of their users, and the principal tasks of their development and maintenance lies with their users.

REL and POL are designed as knowledge base systems for these rapidly changing environments.

In these systems there are two levels of change that are important. First, there is change by the users themselves. They must be able not only to modify the various data items in the knowledge base, but to extend and modify the structure of the base itself. They must be able to add definitions and other abbreviated means for extracting and manipulating the data, and by these means add their own tacit knowledge and expectations so that the computer will respond meaningfully and succinctly to their further queries.

The second level of change is at the application programmers' level. Preparing the knowledge base for a particular using community constitutes a major task. For a system like MYCIN, the clerical aspects of this task are dwarfed by the time resources of true experts. The mundane aspects of actually putting their knowledge into the computer can be relegated to far less costly resources. However, in the case of

systems on which we are focusing, the "experts" are more local to the using group and are called upon far more frequently to prepare knowledgeable working environments to fit the newly arising tasks or circumstances confronting their user clients. The systems on which we are working must support both levels of change.

THE REL/POL KNOWLEDGE BASE SYSTEMS

In designing the REL and POL Systems, we sought to build into the system all aspects that would be common to all or most rapidly changing knowledge base environments. These include managing the input and output, data storage and retrieval, and processing of the language—parsing and semantic interpretation. They also provide facilities for creating and for extending and modifying a knowledge base, for adding and changing data and vocabulary, and for handling definitions. These capabilities are essentially identical in REL and POL. Since they are adequately covered in the REL documentation,^{11,13} we will cover them only briefly here. Here are some of the main features of POL.

The core of a knowledge base system is its language processor. The POL language processor consists of four parts:

- the Preparser, which takes care of such tasks as line continuation, multiple blanks, recognizing whole numbers, and looking up all words in the lexicon;
- the Parser, which develops the parsing tree for the input by using a pattern matching algorithm in conjunction with the particular grammar table for the specific language involved;
- the Semantic Processor, which uses the parsing tree to compose the interpretive routines that are associated with the grammar rules;
- the Output Processor, which does a variety of final editing on the results of semantic processing to prepare the lines to be output.

Following is some information about the various techniques we use in each of these four steps of message processing.

The most interesting aspect of the Preparser is the lexicon processing method. We use a triple hashing technique. Given an identifier, or word, to look up in the lexicon, we hash it one character at a time into the first hash bit table of 2^{14} bits. If the corresponding bit in this table is 0, then the segment so far hashed can not be the initial segment of a word in the lexicon, and we can stop. Otherwise we refer to the second hash bit table, again of 2^{14} bits. If this corresponding bit is 0, then the segment hashed so far is not itself in the lexicon. Only on success here do we finally hash down to 64 hash buckets and search the appropriate bucket for identical key. This technique is in general very fast, since the hash bit tables can be kept in highspeed memory. Moreover, it results in a particularly fast, lexicon-driven spelling corrector.

There are today two parsing algorithms being used for grammar-driven natural language processing, the top down ATN parser, developed by Woods,¹⁴ and the bottom up Powerful Parser of Kay.⁶ Both are chart parsers in the sense of R. Kaplan,⁴ and it is now known that their basic algorithms are

essentially identical (as shown by Papachristidis⁷). REL and POL use the Kay parser. We feel that the bottom up approach provides the basis for more useful diagnostics and for more useful information in handling sentence fragments such as ellipses, false starts and added information.

We do not order our grammar rules or multiple parsings, thus we get all possible parses of the input message. A distinctive feature is that ambiguities are handled internal to the semantic processor routine, and thus interpretive routines, associated with the grammar rules, do not have to be aware of the possibility of ambiguity. The semantic processor also expands definitions, including instantiation of variables. This again leaves the interpretive routines to deal only with local problems of the immediate phrase and its immediate constituents.

An interesting aspect of the Output Processor is that it distinguishes between diagnostic messages and substantive answers. Any interpretive routine, looking at its local context, can output a message and may mark it as a diagnostic. At the end of output processing, the Output Processor considers all messages, diagnostic and substantive alike. If there is at least one substantive response, all diagnostics are repressed. Remaining ambiguous messages are edited for redundancy. We give two illustrations of the use of this mechanism.

First, suppose the input message was "What is the rank of the radiation officer of the Alamo?" In the processing of this message, an interpretive routine will be called to determine the radiation officer of the Alamo; suppose it finds by reference to the data base that the Alamo does not have a radiation officer. This interpretive routine then issues the diagnostic message: "There is no radiation officer of the Alamo." and otherwise signals that it was unsuccessful, aborting the remainder of semantic processing. This message would then be output.

As a second example, suppose that the term "gross sales" was multiply defined, having two entries in the lexicon; one that resulted in summing the sales for a given product, the other summing the sales made by a given sales person. In answering the question "What is the gross sales of diodes?" the interpretive routine interpreting the construction "gross sales of diodes" would be called twice by the semantic processor, once for each definition of "gross sales." In the first case, it would issue a substantive response; in the second case it would issue a diagnostic: "Diodes is not a sales person." The Output Processor, seeing that a substantive response is forthcoming, would repress the diagnostic, and the user would get only the response he or she desired.

The data base organization underlying both REL and POL is the relational one. It is complete, in the technical sense of that term as used in relational data base theory. We have devoted a great deal of attention to optimization of data base algorithms, particularly in regard to access to peripheral storage. Indeed, efficiency of processing was a paramount concern throughout the development of REL. A principal objective of REL was to establish that fully operational, natural language, relational data base systems could be realized in the near term. A primary requirement therefore was good response time. Total throughput message processing time, on an IBM370/3032 computer and using a data base developed by others for testing just such systems (the Navy "blue" file), is

averaging about four seconds. The total response time from input to output of the answer for the query

What is the destination and cargo type of each ship whose port of departure was some Soviet port?

is less than 10 seconds.

THE REL EXPERIMENTS

We have learned a great deal from REL and, in particular, from the series of experiments on human-to-human and human-to-computer communication. The majority of these experiments involved a real life task of loading Navy cargo ships. The total time spent by subjects was over 50 hours, which yielded for final comparisons 20 face-to-face protocols, 11 terminal-to-terminal protocols, and 21 human-to-computer protocols, containing over 80,000 words. (See Thompson¹² for a complete report.)

It was found that in task-oriented situations the syntax of interaction is influenced in all modes by this context in the direction of simplification, resulting in short, simple sentences (averaging in all three modes about seven words). Users seek to maximize efficiency in solving the problem. When given a chance, in the human-to-computer mode, to use special devices facilitating the solution of the problem, they all resort to them.

In reporting on the analysis of these protocols, the term "message" refers to an utterance of one speaker; a "sentence" was required to contain both a noun phrase and a verb phrase, be confined to a single message and have substantial semantic cohesiveness. Parts of messages not also parts of sentences were identified as either "phatics" or "fragments." A phatic is any string whose function was to keep the channels of communication open, e.g., "okay," "I see," and—to the computer—"You lie." The following table presents some of the results of analysis (F-F = face to face, T-T = terminal to terminal, H-C = human to computer).

	F-F	T-T	H-C
Sentence length	6.8	6.1	7.8
Message length	9.5	10.3	7.
Fragment length	2.7	2.8	2.8
% of words in sentences	68.8	72.8	89.3
% of words in fragments	17.2	21.1	10.7
sentences per message	.96	1.22	.81
fragments per message	.59	.74	.19
phatics per message	1.1	.59	.04

The types of sentences used in human to computer communication is of considerable interest:

	Total	Percent
All sentences	882	100
Simple sentences, e.g., "List the decks of the Alamo."	651	73.8
Wh-type questions, e.g., "What are ships?"	658	75.0
Sentences with pronouns, e.g., "What is its length?"	30	3.4

	<u>Total</u>	<u>Percent</u>
Sentences with quantifiers, e.g., "List the class of each cargo."	101	11.4
Sentences with conjunctions, e.g., "List hatch width and hatch length of each deck of Alamo."	113	12.8
Sentences with relative clause, e.g., "List the ships that have water."	16	1.9

The dominance of simple sentences is striking. The reason is certainly not the lack of availability of complex sentences. We think that several factors account for this. The problem-solving situation influences the subject to work in a simple manner, often employing what we have termed special strategies, e.g., repetition of the same type of requests. Another reason is definitions. Once subjects introduce a definition whose right hand side is complex, they use it in subsequent messages, which are therefore short and simple. Another reason may be that subjects tend to be more formal in conversation with a computer. On the whole, one is forced to conclude that monotony of structure is the rule rather than the exception in human-computer communication.

Fragments compose a significant part of communication in all three modes. In an earlier paper on human dialogue in problem-solving situations, it was noted by Chapanis¹ on the basis of extensive experiments that "people do not naturally speak in sentences" and that "in general great unruliness characterizes communication." At first sight of the protocols one tends to confirm the impression. But a closer look and careful analysis reveals a considerable orderliness. Over the course of analysis of these protocols we have been led to classify fragments into eleven categories,¹² each suggesting corresponding procedures for processing them. We will briefly comment on three of these categories here.

Terse Question: e.g., "How about pyrotechnics?" "How many?", "Which ones?" Elliptical questions of this type should be handleable by computational means. Note that in handling pronouns, one looks in the preceding part of the protocol for a referent that can be substituted for the pronoun. The same techniques appear applicable in this case, when one seeks a referent that can be replaced or modified by the body of the elliptical expression.

Added Information: e.g., "What are the destinations of ships? ... Soviet ships," "It doesn't say anything about weight. ... Except for the crushables." The frequency of such added information suggests that the terminal keyboard should be kept open and additional input before start of output should be incorporated into message processing.

False Start: e.g., "Do ships What Ships carry ammunition?" Although computer terminals usually provide a convenient means for deleting an input and starting over, they are not always used. Ways of intercepting these occurrences should not be difficult to incorporate.

Important insights into ways to improve the habitability of human-computer systems was gained by analysis of the errors that occurred in the protocols. In the human to computer protocols there were 446 errors. A breakdown into eight categories is given by the following table:

	<u>Total</u>	<u>Percent</u>
Vocabulary	161	36.1
Punctuation	72	16.1
Syntax	62	13.9
Spelling	61	13.6
Transmission	32	7.2
Definition format	30	6.7
Improper response to prompt	16	3.6
"Bug," error in system	12	2.7

THE EXTENDED CAPABILITIES OF POL

Although the core of the POL System is similar to REL, it also embodies several significant extensions. In the first place, in the development of REL, very little attention was given to problems of habitability, a property of human-computer systems sometimes referred to as "friendliness." For example, there was no spelling corrector and substantive diagnostics were particularly weak. Analysis of the experimental protocols has indicated concrete directions for improvements. To illustrate, you will notice in the table immediately above that the greatest source of errors was the use of words that were not in the vocabulary of the particular application. REL did not identify those words nor even indicate the nature of the problem. The user would often try paraphrases using the same missing word before sensing the source of his or her difficulty. POL immediately identifies such words, as in the following example:

User: What is the usual use of the super deck of the Alamo?

REL: Input error: please re-enter request.

POL: The following word is not in the vocabulary: usual

Other illustrations will be given below.

In REL, we were preoccupied with the core of the system and with REL English and its relational database system. Capabilities beyond these core concerns are, however, also important, in particular those facilitating inter database communication, the distributed database problem, and also capabilities to facilitate the building of specialized user applications. We have had two fine doctoral dissertations which have introduced significant improvements in these two areas. These have added new dimensions to POL beyond REL, as described below. Finally, we are incorporating several methods for augmenting natural language for more efficient human-computer communications. These topics will be discussed in the following four subsections.

All of these topics, however, share a common theme. Problems of natural language processing and efficient database processing are now well in hand and quite adequate for implementation of practical systems with good response times. The REL System gives quite convincing evidence of this. The next stage in improving human-computer communication will be through a better understanding of how users will actually behave as they productively interact with the computer in ways that are natural for them. The systems that we can now provide are unfriendly, they are too sensitive to trivial errors, too pedantic in the messages they are willing to understand, they make inadequate provision for building the user's knowledge base and vocabulary into the system, they do not offer ade-

quate protection from catastrophic mishap, and they do not provide for normal kinds of inter knowledge base communication. The user is in a position where s/he realizes s/he is doing something wrong but has no idea how to proceed. The thrust of our work in extending REL to POL is to relieve these unfriendly aspects of using the computer and to replace them by a habitable working environment.

Diagnostics

The user inadvertently makes mistakes. Even when the user's message is quite correct and the computer formulates a correct response, that response may still be confusing. In these cases, the computer needs to provide a more useful response. Diagnostics are of two kinds: semantic, which must be accounted for in the interpretive routines of the specific language involved; and syntactic, which can largely be taken care of in the language processor. We give several illustrations of POL diagnostic techniques.

The first step in correcting the diagnostic deficiencies in REL was to maintain with each phrase recognized by the parser its underlying literal string so that this string is available to both the system and the interpretive routines for framing meaningful responses. Thus for example: "San Diego ships" may be found to be ambiguous, referring both to those ships located in San Diego or to those ships whose home port is San Diego. REL provides these two interpretations but does not provide the user with any indication of the source of the ambiguity. In POL, the interpretation routine that is looking for San Diego ships discovers the ambiguity and, having the string "San Diego ships" available, is able to tag them. Thus:

›What are the destinations of San Diego ships?

Ambiguous:

- (1) San Diego (location) ships
New York
Tokyo
- (2) San Diego (home port) ships
Naples
San Diego

Using this technique we have been able to incorporate many of the ideas concerning diagnostics expressed by S.J. Kaplan,⁵ as illustrated by the following example:

›What is the square foot capacity of the super deck of the Alamo?

The Alamo does not have a super deck.

A technique we are widely employing in POL, facilitated by an "evaluate" procedure that can be called from an interpretive routine, is to check out possible corrections by calling the parser and semantic processor, and using these evaluations in deciding on a response. When corrections are made, we signal the interpretation by echoing. If the number of potential corrections exceeds some small number at any given point, then any attempt at correction is discontinued and a less informative diagnostic is given. Thus the query:

›Who is the commander of the Alemo?

might, as the case may be, yield any one of the following responses:

- Capt H. Smith
- The Alemo does not have a commander.
- Spelling corrected: "Alamo" for "Alemo" Capt H. Smith
- Ambiguous:
 - (1) Spelling corrected: "Alamo" for "Alemo" Capt H. Smith
 - (2) Spelling corrected: "Alimo" for "Alemo" Capt K. Jones
- The following word is not in the vocabulary: Alemo

We are augmenting the POL English grammar with many rules which recognize what are, strictly speaking, ungrammatical forms. The interpretive routines associated with these rules will have checks and safeguards built into them, again with use of evaluate, and will echo the corrected form before giving the response. For example:

›width, length of M74 truck

Width and length of M74 truck?

96 180

›What is the destination of the Alamo.

What is the destination of the Alamo?

London

These methods do not, however, handle the problem where the input message does not completely parse because it does not strictly adhere to the grammar. How to frame truly helpful responses in these cases is a difficult research question. The work of Sondheimer and Weischedel¹⁰ provides important directions.

Fragments and Pronouns

There has been some excellent work by Grosz on identifying and following the focus of a dialogue² and by Sidner on using this notion of focus in identifying the referent for pronouns and anaphoric expressions.⁹ Using these and other linguistic analyses of pronouns, Roach has developed a considerably improved method for handling pronouns in POL.⁸ We hope to be able to apply these same techniques to the processing of terse questions and added information. The discussion above of the experimental protocols indicates other areas where we plan to strengthen POL's ability to handle a variety of sentence fragments.

Inter-Knowledge-Base Relationships

A research or management staff has not one but many knowledge bases. For example, in a manufacturing firm there would be a personnel base, an inventory control base, a purchasing base, a customer base, etc. Each of these has its own practices for updating and deleting, and each is the responsibility of a different part of the firm. In the past, attempts have been made to consolidate all of these into a single corporate database, but this has been found to be unsatisfactory. Similarly, in a research staff or project team, there may be a variety of knowledge bases, each documenting a

variation of a basic experimental design. In these rapidly changing environments, where members of the staff are involved in modifying and extending their knowledge bases, backup copies must be maintained, contingencies examined, alternative plans evaluated—requiring several working copies of a common knowledge base.

POL provides the capability for creating and maintaining many knowledge bases within the same knowledge base system. Further, these several knowledge bases may be inter-related in a variety of different ways. One such relationship is “basing.” One knowledge base, say B, can be “based” upon another, say A. Once this basing operation has taken place, all of the information available in A is automatically available in B, and any subsequent changes in A are automatically reflected in B. Changes in B, on the other hand, will not affect A at all. A knowledge base may be based upon several other knowledge bases, and many knowledge bases may be based upon a single one.

To see how these capabilities might be used, suppose the accounts in a firm were divided into three groups, aa, bb and cc, each the responsibility of a separate desk in the accounting section. Then three knowledge bases, AA, BB and CC would be created, each owned by its respective desk which would preserve the rights to modify it. The general accounting base, say GG, would be based on all three. The higher management base MM and the home office base HH would be based on GG. A staff office, studying the effects of a change in pricing policy, could also base their study base SS on GG, making what ever changes they were interested in in SS but not affecting GG at all. These capabilities reflect the doctoral dissertation of Yu.¹⁵

Metalinguage

POL incorporates a significant new notion of metalanguage, the knowledge-base environment for the application programmer. The bottom “knowledge base” of the POL System is POL English, containing the syntax of a subset of natural English, a mathematics package and the function words of English, e.g., “have,” “and,” “which” etc. All other knowledge bases for users are based upon POL English in the sense described in the last paragraph. There is another basic “knowledge base,” namely MetaEnglish. It contains a variety of constructions, including (1) all of Pascal, (2) the capability of writing in succinct form new grammar rules for a given target language and their associated interpretive routines, (3) a similar capability to extend the metalanguage itself with either new procedures or macros, (4) a variety of useful utility procedures for dealing with the relational database, effectively handling input and output, literals, and the evaluate function, and (5) a compiler/linker that is able to relate this self-extended Pascal and its associated syntax. POL English and MetaEnglish are associated with each other.

When a knowledge base, say AA, is based upon POL English, a parallel knowledge base, MetaAA is also created, based on MetaEnglish; AA and MetaAA are associated with each other. An application programmer using MetaAA adds grammar rules and associated interpretive routines to her user's knowledge base AA. In doing this, she may expeditiously add new utilities to her own “knowledge base,” namely Meta-

AA. Obviously she can use the utilities and procedures of MetaEnglish, since her MetaAA is based on MetaEnglish. In fact the whole basing structure of hierarchically related knowledge bases is strictly paralleled by the associated Meta-basing structure.

This base/meta-base apparatus is designed to facilitate the building of specialized knowledge bases that extend and specialize more general capabilities to the needs of users. This aspect of POL reflects the doctoral thesis of Hess.³

IMPLEMENTATION

REL was implemented on a large IBM-370/3032 computer. POL, on the other hand, is being implemented on a desk top computer, the Hewlett-Packard HP-9845B with a 50 megabyte disk. POL is written in Pascal.

At the present time all of the central parts of the system—preparser, parser, semantic processor, output processor, definition handling, paging, list processor—have been completed. Basing has been completed and we are in the last stages of completion of the metalanguage. Much of POL English has been completed, including the optimized utilities for managing the relational data base.

Of course there is much left to be done. But the most important work on POL will start when the system is completed, response times have been brought under control and the first applications implemented. For then we can observe, in an environment much closer to being friendly and conducive to natural propensities, just how professionals with a job to do will communicate with computers. That is the exciting moment, for as we know from our REL experience, reality reveals the directions to truly more responsive systems.

REFERENCES

1. Chapanis, A., “Interactive Human Communication,” *Scientific American*, April 1975, pp. 39-42.
2. Grosz, B.J., *The Representation and Use of Focus in Dialogue Understanding*, SRI International, Tech. Note 151, 1977.
3. Hess, G.D., *A Software Design System*, Ph.D. Dissertation, Calif. Inst. of Tech., 1980.
4. Kaplan, R.M., “A General Syntactic Processor,” Rustin, R. (ed.), *Natural Language Processing*, Algorithmics Press, New York, 1973, pp. 193-241.
5. Kaplan, S.J., *Cooperative Responses from a Portable Natural Language Data Base Query System*, Ph.D. Dissertation, Univ. of Penn., 1979.
6. Kay, M., *Experiments with a Powerful Parser*, The Rand Corp., Santa Monica, Calif., RM-5452-PR, 1967.
7. Papachristidis, A.C., *Comparison of ATN, Kay and Related Parsing Algorithms*, Master's Thesis, Calif. Inst. of Tech., 1980.
8. Roach, K., *Pronouns*, Master's Thesis, Calif. Inst. of Tech., 1980.
9. Sidner, C., *Towards a Computational Theory of Definite Anaphor Comprehension in English Discourse*, Mass. Inst. of Tech., Tech. Report 537, 1979.
10. Sondheimer, N. and R.M. Weischedel, “A Rule-Based Approach to Ill-Formed Input,” *Proc. 8th Intern. Conf. on Comp. Ling.*, Tokyo, 1980, pp. 46-53.
11. Thompson, B.H., *REL English for the User*, Calif. Inst. of Tech., 1978.
12. Thompson, B.H., “Linguistic Analysis of Natural Language Communication with Computers,” *Proc. 8th Intern. Conf. on Comp. Ling.*, Tokyo, 1980, pp. 190-201.
13. Thompson, B.H. and F.B. Thompson, “Rapidly Extendable Natural Language,” *Proc. 1978 Nat. Conf. of ACM*, pp. 173-182.
14. Woods, W.A., “Transition Network Grammars for Natural Language Analysis,” *Comm. of ACM*, vol. 13, (1970), pp. 591-606.
15. Yu, K.I., *Communicative Databases*, Ph.D. Dissertation, Calif. Inst. of Tech., 1980.

Computer speech for people with cerebral palsy

by JAY HEWITT

*University of Missouri at Kansas City
Kansas City, Missouri*

ABSTRACT

Using a grant from the Apple Educational Foundation, a speech system was constructed using an Apple micro-computer, a Corvus hard disk drive, and the Mountain Hardware Supertalker. Since digitized rather than synthesized speech is employed, speech quality is extremely high. Use of an 8M byte hard disk drive permitted the system to contain approximately 5000 words. In normal operation, user types in the first two letters of a desired word. A list of words beginning with those two letters than appears on a TV monitor. To select a word, user enters the number which is beside each word. At the end of a sentence, user types a period and the computer speaks the n words in the sentence in approximately $2n$ seconds. For individuals unable to operate a keyboard, two alternative means of input exist. The paper describes the nature of the words in the system, the computer program, and certain characteristics of the hardware.

INTRODUCTION

Certain individuals are above average in I.Q. and are capable of complex communication but lack sufficient control over throat and mouth muscles to generate speech. Most of these individuals would have cerebral palsy but the condition may also be found in those with muscular dystrophy and, on rare occasions, in those who have suffered from a stroke. When these individuals have reliable control over some other muscles (e.g. hand or foot), it is possible to provide them with microprocessor controlled artificial speech.

Two of these speech units, all based on the principle of synthesized speech, have recently been developed. One is an inexpensive toy—the Texas Instruments Speak and Spell. Individuals who can press the keys on this device with either hand or toes can use it to spell out the letters of a word. There is one key for each letter of the alphabet and a keypress generates the sound of that letter. The other device is called the Handi-Voice. The unit contains the Votrax speech synthesizer and is capable of producing either 500 or 1000 words depending on which model is selected. In the model with 500 words, the words are written on the top of the device and the individual touches a particular square to generate a word. In the unit with 1000 words, user enters a three digit code and the

device then speaks the word. Although the Handi-Voice is a major contribution to the area of artificial communication, there are certain built in disadvantages. It has been estimated that a child of 5 already has a vocabulary of over 2500 words; 500 words is far too few for adequate communication and the unit with 1000 words requires the user to learn approximately 1000 3-digit codes. Speech quality is also a problem. The voice output has an extremely artificial “robot-like” quality and the words are sometimes hard to understand.

The current paper describes an alternate communication system that was designed to overcome the problems of limited vocabulary and poor speech quality. The system uses an Apple computer, a high quality speech reproduction system sold by Mt. Hardware called the Supertalker, and a Corvus hard disk drive. Although it is not portable, the unit contains approximately 5000 words with speech quality similar to that obtained on a tape recorder.

SYSTEM OPERATION

When using the computer keyboard to input words, user types in the first two letters of the desired word. A list of words beginning with those two letters is then shown on a TV screen. If the desired word is not on the list, user can press a particular key and another list of words beginning with those two letters will be presented. Beside each word is a number. When the desired word is located, user types in the corresponding number and the word is then printed at the top of the TV screen. Subsequent words in the sentence are printed to the right of the first word so that an entire sentence can be generated and shown on the TV screen. At the end of the sentence, user types a period and the computer then speaks the words in the sentence. A sentence of n words is spoken in approximately $2n$ seconds.

ADDITIONAL WORDS IN SYSTEM

In addition to the usual words found in a dictionary and used by a college graduate, several additional sets of words are available. One is a set of 67 common first names (e.g. Jane, Jack). Another is a set of 82 food items. A third is a list which makes it possible for user to generate any number. A fourth

is a set of 75 locations—primarily names of states and large U.S. cities. Finally, one Corvus volume has been set aside for idiosyncratic words desired by a given user. A friend or parent can record up to 80 of these words and make them available to the user.

USE AND ADVANTAGES OF SYSTEM

The system makes use of a Corvus hard disk drive which makes the unit non-portable and rather expensive. The cost of a Corvus is approximately \$5000. It is estimated that a school system would be the best location for the unit. If there is more than one child needing to make use of the unit and both are located in the same class, a multiplexer can be purchased which will allow multiple users to access the speech system simultaneously. For individuals without manual control, a five-pedal foot switch system is available. When running in this mode, the computer asks for the first two letters of a word and then presents, on the TV screen, a matrix consisting of letters and computer commands. One switch scans up the columns, one down the columns, one scans to the right on a given row, one scans to the left on a given row, and one selects, as input, the letter or command on which the cursor is located. A second alternative means of input is a joy-stick. Using a joy-stick, user can move directly to the desired row and column for a letter or command. These alternative means of input together with high voice quality and large vocabulary are the central advantages of the current system. Another advantage, at least as far as the listener is concerned, is that the message is not delivered until it is complete. An individual with severe motor handicap could take up to three minutes to construct a sentence. It is far easier on the listener to attend for a single 20 second period at the end than to attend to the user during the entire three minute period of message construction.

DETAILS ON THE SUPERTALKER COMPONENT

The Mt. Hardware Supertalker device consists of a card that fits in the back of an Apple Computer, a loudspeaker, plus some software. It is normally used with a single Apple disk. In the initial stage, user creates something called a phrase table containing up to 18 spoken words. Each word is spoken into a microphone, the hardware digitizes the word and then stores it in memory. When the phrase table is full, user stores the phrase table on disk. Between 80 and 90 words can be normally be stored on one Apple diskette. When it is desired to speak a word during program execution, user gives a command to load a given phrase table into the computer and then gives another command to send a particular word to the loudspeaker.

The current system uses not an Apple disk but a Corvus hard disk drive. Sixty-four Corvus volumes were used to store approximately 500 phrase tables. During program execution,

when it comes time to speak a given word, the appropriate Corvus volume is made the default volume, the desired phrase table is loaded into the Apple, and the desired work within the phrase table is sent to the loudspeaker.

DETAILS ON PROGRAM OPERATION

A phrase table containing only one spoken word would take up at least 3k. Storing only one word in each phrase table would not be an efficient utilization of the space on the Corvus and it would only be possible to store, under this procedure, about 2400 words. Furthermore, there would not be sufficient memory available in the Apple—about 32K under this procedure—to store information about the location of each word on the disk. As a result, it was necessary to store several words in each phrase table. With up to 20K being taken up by a phrase table, only 15K remained in order to store information about the location of words on the Corvus. An initial attempt was made to determine the exact phrase table a given word would be in should it be on the disk. There was not sufficient computer memory available to do this. As a consequence, the program operates by determining the first of several phrase tables a given word would be in if it should be on the disk. A list of these words on the first phrase table is presented to the user. If the desired word has not yet appeared—in alphabetical order—user can bring in and examine subsequent lists with a single keyboard stroke.

For every phrase table located on the disk there is a corresponding text file containing a list of the spoken words in the phrase table. The text file, which is only 2K in length, is much faster to load into the computer than is a phrase table which can be up to 19K in length. When a list of words is shown on the TV screen, it is the list from the text file that is being presented. The name of the text file indicates the location of the corresponding phrase table. For example, a given text file might have the name Corvus Volume = 32 --- phrase table number = 5. If the individual were to select word 7 from this list, the program would then have information that the spoken word was located on Corvus volume 32 in phrase table 5 and that the desired word was word 7. As each word was selected, the program would keep track of the location of each spoken word. At the end of the sentence, the program would use this information to load in and present each word. Each text file also contains both a forward and a backward pointer. These pointers contain information about the name of the preceding and the name of the subsequent text file that should be loaded in and presented should a given key be pressed. With the use of text files and pointers, it was thus possible for the user to gain quick access to each of 5000 words, at the same time preserving the rather scarce RAM memory.

REFERENCE

1. Eulenberg, John, ed. *Proceedings of the VOCA Conference*, Berkeley, California, May, 1980. Artificial Language Laboratory, Computer Science Department, Michigan State University, East Lansing, Michigan.

GRASS3, a language for interactive graphics

by NOLA DONATO

Wizard Software
Chicago, Illinois

ABSTRACT

With the advance of technology, graphics devices are becoming more powerful and less expensive, making interactive graphics increasingly popular as a method of man-machine communication. Often nonprogrammers play a principal role in the design and implementation of graphics applications. Interactive graphics requires a high level of feedback both with the user and with the hardware. For these reasons, conventional programming languages are not well suited for such applications.

This paper describes GRASS3, an interpretive language designed as a base for interactive graphics systems. The work derives from the author's thesis at the University of Illinois at Chicago Circle (UICC)¹ and similar work done by the author for the Bally Manufacturing Corp.² Design rationale for the language is given, followed by an overview including examples and a description of a specific real-time graphics system based on GRASS3.

DESIGN PHILOSOPHY

The GRASS3 language (GRASS3 stands for GRaphics Sym-biosis System version 3) was designed as a base language for development of interactive graphics systems. Although GRASS3 bears very little resemblance to its predecessor, GRASS2,³ much of the interactivity and simplicity which made old GRASS so powerful have been preserved in GRASS3. The language also borrows heavily from C and SNOBOL^{4,5} for language design and internal structure.

One of the most serious drawbacks of conventional programming languages in the graphics environment is the difficulty of tailoring them to a particular device. In most of the higher level languages, subroutines are the only feasible way to add new features. Consequently, it is almost impossible to achieve the communication between hardware and software needed to support a real-time application. Even the recent efforts at graphics standardization such as the Core System are aimed primarily at static devices such as plotters.

So far, the standard way of solving this problem has been to design a special purpose language revolving around a particular device or hardware system. This approach was taken by the designers of SMALLTALK (which depends heavily on the

Interim Dynabook⁶). GRASS2, a language used by artists at UICC,³ revolves around the Vector General refresh CRT.

GRASS3 is designed to interface easily with specialized hardware and software. Depending on what devices it must talk to, GRASS3 may require a set of special commands, device-dependent variables or even new datatypes. A refresh Cathode Ray Tube, for example, needs a "picture" datatype. Creation and manipulation of display lists require a special set of functions. Device variables are also needed if the system has dials or joysticks.

The GRASS3 language is designed to make such internal rearrangements simple and straightforward. To a large extent the language is table driven. Because of this, it is not difficult to add commands, datatypes, or even new operators. One can define new conversions rules or redefine old ones. Many existing features (such as floating point support, interactive debugging, etc.) can be eliminated simply by recompiling the source. Almost all of the system commands can be made to dynamically swap in and out when needed. (One can also do this with user functions). Thus, GRASS3 can be easily configured to meet user specifications.

Another important feature of GRASS3 is that it is interactive. Almost anything allowed in a program may also be typed directly on the terminal. A user may "try out" a statement, display a picture or inspect her variables all without having to write a program.

This kind of interaction is necessary for interactive graphics. The feedback provided by such a system speeds up program development and the evolution of a graphics application. One should not have to go through the whole cycle of updating a source file, compiling, loading, initializing and then setting up the proper environment in order to determine the implications of a trivial change.

Interactivity is also essential when a human must be in the loop to supply decisions about how the animation is to proceed. Much of computer graphics is visual—the machine cannot predict whether one will be excited or bored by a particular effect and it is not capable of making artistic judgements. Conversational graphics systems are structured to permit just this sort of thing. Thomas Standish comes to the same conclusion in his paper on computer animation.⁷

Recent trends in home computing show that interactive systems are better for beginning programmers. Most commercially available home computers use some derivative of BA-

SIC.⁸ Even experimental home computers, such as the Interim Dynabook, rely heavily on interactive feedback.⁶

Interactivity can help overcome the qualities of computer languages that are unnatural to the novice user. The immediate response available in an interactive system can surmount barriers that make a system difficult to program. For instance, both LISP and APL^{9,10} are cryptic languages, yet they are very popular in interactive environments. LOGO, a derivative of LISP designed especially for naive programmers, was used as the base language for a graphics system developed by Abelson.^{11,12}

The GRASS3 language is high level, but easy to learn and understand as well. Novices do not have to become super programmers to try out their ideas and experiment with the system. But as they gain experience they are able to expand their use of the more general features. GRASS3 can be useful to the naive user with minimal learning and, as she demands more powerful capabilities, they can be easily absorbed in small increments. This is important because problems in computer graphics are often tackled by nonprogrammers like educators and engineers. The designers of GLIDE, a language developed for CAD applications, discuss this in their book.¹³

If the graphics language is easy to learn and use, small projects can be done without hiring a professional programmer. For large projects, a readable language can allow a greater level of understanding and communication between designers and programmers. By reducing the gap between these two classes, systems can be tailored closely to the requirements of individual designers.

Much of learning involves making generalizations upon what one already knows. In learning a new programming language, one will often look at examples already known to work and modify them to suit a new purpose. If the semantics of a language are consistent (that is, operators in expressions always behave the same way, expressions are allowed whenever constants of the same type can be used, etc.) the learning process will be faster because the user's generalizations will be correct more often.

Consider, for example, the calculation of a subscript in FORTRAN. There are explicit rules governing the form such an expression may take, which may be found in any FORTRAN manual. Yet many FORTRAN programs are full of statements such as

```
K = I - 5
B = ARRAY(K)
```

when it is perfectly legal to combine the two ($B = \text{ARRAY}(I - 5)$). One may argue that, since the restrictions on subscripts are documented and consistent, cases like those above are programming errors and not limitations in the FORTRAN language. But constructions such as the above are rarely seen in C programs. Because there are no restrictions at all on subscripts in C, it does not occur to programmers to worry about whether a particular expression is permitted or not. Similar views are expressed by Weinberg in his book on the psychology of computer programming.¹⁴

In addition to being interactive, a graphics language must do a certain amount of housekeeping for the user. Most special purpose languages have a set of high level, nonprocedural

primitives that free the user from the burden of managing details and allow her to concentrate on the real problem. ORACLE,¹⁵ a relational database system, can do very complex queries in one or two statements. SIMULA,¹⁶ an ALGOL¹⁷ derivative designed for simulation, supports sophisticated multi-tasking capabilities. This "behind-the-scenes" management is especially important in graphics where data must be displayed as well as generated.

Part of system housekeeping includes maintaining datatypes. High-level datatypes such as strings, arrays, pictures, and list structures can be very useful for managing and organizing information. Consider the task of comparing two strings, something done often in programming. The C language does not have a string datatype.⁴ A string is considered as a collection of characters. The following C program will return TRUE if the two given strings are the same:

```
index = 0;
while (string1[index] == string2[index])
  {if(string1[index] == END) return(TRUE);
   index = index + 1;}
return(FALSE);
```

GRASS3 allows the user to directly compare strings. Since strings are datatypes, the routine can be reduced to a single statement.

The same idea can be applied to computer graphics. A graphics programmer is often faced with displaying a series of pictures consecutively on the screen. If she can manipulate a picture as a single entity and group it with other pictures in an array or list she can trivially solve this problem. But if she must first create her own mechanism for dealing with pictures as whole objects, the simple display problem becomes a time-consuming programming task.

Another job the system can take over is memory management. All languages do this to some degree. Many, like FORTRAN and BASIC, have only static allocation. A program cannot reclaim memory used by arrays for other purposes (not even for different arrays). Others, like PL/1 and C, have primitives that will parcel out chunks of a dynamic memory area and reclaim them again. The programmer is responsible for maintaining the integrity of this area. Finally, there are languages like ALGOL and SNOBOL⁴ that manage all memory automatically. The user thinks only in terms of the logical datatypes. To delete a list of items in PL/1, a subroutine is needed:

```
PTR = LIST;
WHILE PTR != NULL;
  DO
    TEMP = PTR -> LINK;
    FREE PTR;
    PTR = TEMP;
  END;
LIST = NULL;
RETURN;
```

In GRASS3 a single statement suffices:

```
list = null;
```

There are other housekeeping burdens the system assumes. Conversions between datatypes are done automatically whenever possible. The system provides simple mechanisms to input datatypes from the terminal or disk. User functions can be easily designed to accept arguments if supplied and prompt for them if omitted. Such things allow a programmer to describe her problem in terms which are closer to her logical conception of it.

GRASS3 is extensible and allows the user to program her own commands and configure environments easily and quickly. She can create independent subroutines and pass information between them. A logically clear method of passing parameters and returning values permits her to make extensions to the system. Local variables ensure that these extensions will be independent of one another.

User-extensible datatypes like structures and arrays help the user build complex constructions from simpler ones. Consider the implementation of an animation system. An artist typically creates a number of separate frames and displays them in a fixed order. The individual animation sequence determines how many frames there are and how long each one is displayed. If the programmer can associate a display time and duration with each frame and then group the frames together in a list, implementing a simple animation system becomes much easier.

String manipulation facilities also help the user configure environments. String manipulation is especially important for communication with the user on a terminal. If capabilities exist in the language to facilitate parsing, a programmer can develop a tailored sub-language whose syntax need not be a derivative of the syntax of the base system.

GRASS3 also has many easy-to-use debugging aids. Debugging tools include the ability to set breakpoints, examine variables, patch code, and trace a program's flow. Clear and plentiful error messages are part of this, too. Most programmers, especially novices, spend the majority of their time debugging. GRASS3 debugging features make programming less painful and can significantly decrease the time spent developing an application.

LANGUAGE OVERVIEW

The main way of communicating with GRASS3 is to type to it on the terminal. You can ask it to print information, create and run programs, or read files off the disk. Many statements are commands requesting the system to do something. For example, to print something on the terminal, you can type

```
print "The print command prints"
print "things on the terminal."
print 1,2,3
```

Other statements ask GRASS3 to evaluate an expression and perhaps save its value.

```
a = 2 + 3
```

And, of course, GRASS3 can evaluate expressions and use their results with a command.

```
print "The sum of 2 and 3 is", 2 + 3
print "The value of a is", a
print "The average of 1 thru 5 is", (1 + 2 + 3 + 4 + 5)/5
```

Expressions need not involve only numbers. There are several other datatypes which can also be used in expressions.

integer	16 bit integers
float	32 bit floating point
string	variable length strings
array	<i>N</i> -dimensional arrays
node	programmer-defined datatypes
picture	device dependent
process	program which is scheduled

Most of the operators (like “+” and “-”) operate on numbers. A few, like “\$” (concatenation) or “@” (indirection), need one of the other datatypes to operate on. In general, GRASS3 will attempt to convert whatever it is given to the type it wants.

```
print 1 + 2, '1' + 2, '1 + 2'
```

The statement above prints “3 3 1 + 2” on the terminal. In the first case, 1 and 2 are added and the result (integer 3) is converted to a string and printed. The second case requires the string ‘1’ to be converted to integer 1 before the addition. The last case is already a string and is printed as is.

Most of the commands need their arguments to be of a certain type, too. For example, the print command will only print strings. Since numbers (integer and floating) can be converted to strings, it can print numbers or the results of expressions involving numbers, too. But the print command cannot print arrays, nodes or pictures.

To do complicated things you have to write a program. Programs are really the same as strings. To create a program, you just define a string containing the commands GRASS3 must execute. To run it, simply type its name.

```
hello = [print "howdy"]
hello
```

The example above creates a program called *hello* with a single print command in it. When executed (by the second statement above) “howdy” is printed on the terminal. There are four sets of string delimiters—single quotes, double quotes, square brackets and curly brackets. The quotes (“ and ’) may not be nested. The brackets ([] and { }) may be nested so long as they are paired.

You pass arguments to programs the same way you do to system commands. A program gets its arguments by using the *input* command.

```
max = [;return the maximum of 2 arguments
input int,A,B
if A > B, return A
return B]
```

In the example above, the *input* command fetches the next two arguments to the program, converts them both to type *integer* and stores them in the variables A and B. The *return*

command returns a single value (A or B) depending on their relative magnitudes.

The *prompt* command can be used in conjunction with input to provide the program with a means of prompting for arguments which were not supplied.

```
max = [;return the maximum of 2 arguments
prompt "enter first value"
input int,A
prompt "enter second value"
input int,B
return A * (A > B) + B * (A <= B) ]
```

The alternate version of *max* above will prompt the user for any argument that is not supplied. She may then enter it on the terminal and the program will proceed. Note that the relational operators (< > <= >= == !=) can be used in expressions with other operators. Any line starting with a semicolon is considered a comment and ignored.

There are two types of names in GRASS3—dynamic and fixed. Fixed names are one or two characters long and may only have one kind of datatype associated with them. For example, fixed names a,b,...,z can only have integer values. Fixed names fa,fb,...,fz can only be floating point. Depending on your system there may be names d0,d1,...for dials and jx,jy,jz for joysticks as well.

Dynamic names can be up to seven characters long and can be assigned any kind of data. No declarations are needed in GRASS3. One simply assigns a value or expression to a name as needed. When a new value is assigned, the old value is deleted.

Dynamic names that begin with a lower case letter are known throughout the system to all programs. Those that begin with an upper case letter are local. If a local name is used in a program, it is deleted when that program exits. This allows programs to use the same names without confusion.

Transfer of control in GRASS3 can be done with the *goto* command and labels, or by the more elegant structured constructs like *while* and *do*. The following program prints the values of an array.

```
prompt "Which array"
input array,ARRAY
I = -1
S = size(ARRAY)
while ++I < S, print I,ARRAY(I)
```

Statements may also be grouped within brackets, as illustrated in the following loop, which prints the types of its arguments.

```
do[prompt "enter argument"
input value,ARG
if ARG = "",break
TYPE = type(ARG)
if TYPE = "array",TYPE = $" of " $ type(ARG(0))
print "Type is",TYPE
]
```

Some explanation is in order here. The *do* command is like *while* except that the test (if any) is done at the end of the loop. Using *value* in the input command allows any type of

argument to be fetched. The *type* command returns a string giving the type of its argument. Note that array arguments are further inspected as to the type of their elements. When a null argument is gotten, *break* is used to exit the current loop.

For beginning programmers, GRASS3 has some nice features to make programming easier (and more enjoyable). First, there are interactive helps. If a user forgets the syntax or arguments of a command, she simply types *help* followed by the command name and GRASS3 responds with a description of the command and examples of how to use it. You don't have to look in the manual if you forget what a command does or how to call it.

Second comes interactive debugging. When GRASS3 finds something wrong, (it is requested, say, to do something it can't do or the system runs out of some resource), an error message is printed on the terminal. If this error occurred inside a program, GRASS3 puts that program into *debug* mode. When in debug mode, the normal "*" prompt is replaced by "#" to let the user know she can issue debugging commands. With the debugger, the user can set breakpoints, single step, trace program execution, and even make simple patches. The *edit* command (which invokes the GRASS3 text editor) can also be used in debug mode. In addition to debug directives, the user can still issue any other GRASS3 commands, too. If one is not absolutely sure a program is correct, the *debug* command can be used to place the program in debug mode before an error actually occurs.

GRASS3 can be configured for small systems where memory is tight using the automatic swap feature. Some of the GRASS3 commands are not resident—they live on the disk. When a nonresident command is requested, GRASS3 will automatically read it off the disk and then delete it when it has finished execution. The user can request some of her own programs to automatically swap off the disk, too. The *keep* command allows a nonresident module to remain in memory after it has finished execution. *Keep* can speed up programs where a swapping command is used repeatedly or in a loop.

One of the most powerful features of the GRASS3 language is that the user can run two or more independent programs at the same time. For example, suppose we have already written a program called *walk*, which makes a little person walk across the screen. It accepts arguments telling it where to start the person and which way she is to walk. On most systems the program would have to be completely rewritten if you wanted to have two people walking at the same time. In GRASS3 you would use the *schedule* command as follows:

```
sched walk,100,10,left
sched walk,10,10,right
```

The *walk* program can be scheduled twice with different arguments to show two people walking. GRASS3 will execute one line from the first scheduled program, one line from the next, etc. to give the illusion that all scheduled programs are running at the same time.

THE VISION II INTERFACE

Although the GRASS3 language definition does not include graphics primitives, the system is designed to make the addi-

tion of new commands and datatypes as easy as possible. The VISION II raster graphics system¹⁸ is the most recent device to be interfaced to GRASS3. The screen is a 256 x 256 array of pixels, directly addressable by their X,Y coordinates. Commands exist in GRASS3 to draw lines, boxes, and points, display text, save areas of the screen on the disk or in memory and display them again, etc. A *picture* datatype and utilities to create and manipulate pictures are also part of the system.

Suppose we have some function describing a particular sequence of X,Y coordinates. It could be algorithmic and coded as a program or it could describe some inputs from the outside world (joysticks, perhaps). Let us assume the GRASS3 variables *x* and *y* are being updated (in real time) according to this function.

If we have a previously created picture, CAR, and we want it to move around on the screen according to the path specified by *x* and *y* we would code

```

sched [plot CAR,x,y,erase
      repeat ]

```

This would schedule a program to continuously move CAR as directed by the variables *x* and *y*. Using this method, any number of pictures may be moved simultaneously on the screen.

CONCLUSIONS

Experiments with VISION II and other systems have shown GRASS3 to be very powerful in putting together complex graphics applications quickly. (The VISION II picture editor described by Rocchetti¹⁸ was implemented by the author in a single evening). The language has proven to be easy to learn by a variety of nonprogrammers (several of them children). A subset of GRASS3, called ZGRASS, was used by Bally in their *FUN 'N BRAINS* home computer system.² About half of the programs used to demonstrate the above unit were written by nonprogrammers (advertising executives) over a period of several weeks. The other half were written by the developers within the span of a few days. Had the same applications been implemented the conventional way (in assembly language), the combined effort would have exceeded many man-months.

Isolation of operating system interface code made it trivial to port GRASS3 from UNIX¹⁹ to the DEC operating systems. This was particularly desirable, since at the time GRASS3 was developed UNIX had no real-time primitives. An experienced assembly language programmer coded and debugged the RT-11 operating system interface in under a week. It was running under RSX-11M several days after that. Note that the above times represent only coding of language features (like file I/O, panic traps, etc.). The author does not mean to imply that device- or hardware-dependent applications can be ported to a new operating system nontrivially. (It would be impossible,

for example, to fully support a refresh CRT under UNIX without making operating system modifications).

When GRASS3 was born (1976), the only implementation language that spanned all PDP-11 operating systems was MACRO-11 (PDP-11 assembler). Since then, the C language has grown in popularity and has been implemented on many different machines and operating systems. A portable version of GRASS3 (coded in C) is currently being written. The new version will have more powerful string manipulation primitives and enhanced multitasking capabilities (similar to those in the ADA language²⁰). We hope that these efforts will also yield a GRASS3 compiler, which will produce C or some sort of portable macroassembler source.

Another feature in the works is a *picture compiler*, which will compile a subset of GRASS3 into a form that can be loaded and executed by the VISION II graphics processor. Thus, picture programs could be created and debugged interactively with GRASS3 and finally executed by one or more VISION II processors.

REFERENCES

1. Donato, Nola, "GRASS3—A Base System For Interactive Graphics," Masters Thesis. University of Illinois, 1978.
2. DeFanti, T. A.; Jay Fenton; and Nola Donato, "Basic Zgrass—A Sophisticated Graphics Language for the Bally Home Library Computer," *Computer Graphics*, Vol. 12, No. 3 (August 1978), pp. 33-37.
3. DeFanti, T. A. Dissertation. Ohio State University, 1973.
4. Ritchie, D. M. *C Reference Manual*. Bell Telephone Laboratories, Murray Hill, 1974.
5. Griswold, R. E.; J. F. Poage; and I. P. Polonsky. *The SNOBOL4 Programming Language*. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1968.
6. Kay, Alan and Adele Goldberg. *SMALLTALK-72 Instruction Manual*. Xerox Corporation, March 1976.
7. Standish, Thomas A., "Remarks on Interactive Computer Mediated Animation," *Proceedings of the Ninth Annual UAIDE Meeting*, 1970.
8. Kemeny, John G. and Thomas E. Kurtz. *BASIC Programming*. New York: John Wiley & Sons, Inc., 1971.
9. Howard, Forrest William. *LISP Programmer's Manual*. HRSTS Science Center, September 1975.
10. Freeman, Peter. *Software Systems Principles*. Chicago: Science Research Associates, Inc. 1975.
11. Abelson, Hal; Nat Goodman; and Lee Rudolph. *Logo Manual*. Massachusetts Institute of Technology, 1974.
12. Goldstein, Iran, and others. *LLOGO: An Implementation of LOGO in LISP*. Massachusetts Institute of Technology, June 1974.
13. Eastman, Charles and Max Henrion. "GLIDE: A Language for Design Information Systems," *Computer Graphics*. Vol. 11, No. 2, Summer 1977.
14. Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold Company, 1971.
15. Preger, R.L. "ORACLE User's Guide." Relational Software Inc., Menlo Park, California, 1980.
16. Birtwistle, G.M., et al. *SIMULA begin*. Auerbach Publishing Co., 1973.
17. Tanenbaum, A. S., "A Tutorial on ALGOL 68," *ACM Computing Surveys*. Vol. 8, No. 2, June 1976.
18. Rocchetti, R. J., "VISION II—A Small Scale Expandable Graphics System," Masters Thesis. University of Illinois, 1978.
19. Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*. Vol. 57, No. 6, July-August 1978.
20. Ichbiah, J. D. et al., "Rationale for the Design of the ADA Programming Language," *SIGPLAN Notices*. Vol. 14, No. 6, June 1979.

VISION II: A dynamic raster-scan display

by ROBERT ROCCHETTI

*Wizard Software
Chicago, Illinois*

ABSTRACT

The analog graphics display is one of the most fascinating devices that computer technology has produced. The fluidity and complexity of motion possible with line-drawing displays is quite captivating and has initiated many interesting and fruitful software endeavors.

Unlike their analog counterparts, the majority of raster-scan display systems have been capable of producing only text or static imagery. Dynamic graphics on raster displays has yet to be explored thoroughly in either the hardware or software area. This paper describes a digital video display system architecture with dynamic graphics capabilities. The hardware design and implementation derives from the author's thesis work at the University of Illinois.¹

CURRENT DIGITAL VIDEO SYSTEM ARCHITECTURES

A digital video display system maintains a digital representation of the image in the computer memory, or frame buffer, and displays it on a raster-scan cathode ray tube. The frame buffer may contain the actual screen pixels or an encoded representation of the picture. Displaying (or decoding) the final image from the frame buffer representation is called scan-conversion.

With a digital display, a picture need not be made up only of lines and points. Areas can be shaded, lines can be thickened and gray scale or color can be employed. It is precisely this rich set of capabilities and potential for realism that allows the digital video display to be attractive for dynamic graphics.

When raster-scan displays first emerged, they had significant problems to overcome. Screen resolution was limited due to the high cost of electronic memory. They were slow because of the processing required for scan conversion. Since then, memory has become much cheaper, making feasible systems that were impossible a few years ago.

Scan conversion, however, is still a time-consuming computation. It can be carried out in different ways depending upon the representation of the picture before and after the conversion process and the amount of available memory. Various scan conversion algorithms may be found in articles by New-

man and Sproull, Metzger, and Barret and Jordan.^{2,3,4,5,6} Some of the more popular scan conversion techniques are described below.

Character mapping is widely used in CRT terminals and home computers. It is simple, inexpensive and supports dynamic motion in a somewhat limited framework. The screen is broken up into M by N pixel rectangles, each of which is assigned a pointer. The pointer for a position refers to the particular member of the character set that will be displayed in that position.

Character mapping is common among alphanumeric video terminals because it is ideally suited for displaying text.⁷ By generalizing the characters, the class of possible images can be greatly enriched.^{8,9} Systems exist that can display text lines or binary pictures.¹⁰ Many home computers provide graphics-oriented character sets. Two examples of this are the Commodore PET and the Radio Shack TRS-80. The Exidy SORCERER and the ISC Intecolor have programmable character generators with which the user can define her/his own characters.

Even though they are not strictly limited to text, most character mapping systems do not have enough memory to assign a unique pointer to every character position on the screen. With these systems, it is possible to define static scenes that are too complex to display. Since the display may only be changed by updating character pointers, motion is not fluid—it is limited to character boundaries.

One way to enhance character mapping and provide fluid motion is to predefine a set of hardware registers that may contain objects and their X,Y coordinates. This scheme allows motion on bit boundaries but restricts the number of objects. A general implementation of this scheme with a reasonable number of objects of medium complexity would be very expensive.

Another alternative to character mapping refreshes the screen directly from a coded picture definition. This architecture is very similar to the use of structured picture definitions with vector displays.

Several recent systems have been based on run-length encoding, which provides an efficient storage technique for simple graphic displays and is easily decoded by a raster-scanning refresh processor.^{11,12,13} There are also high-performance systems which use pipelined processors to produce real-time moving images from three-dimensional edge and surface

simulation can even remove hidden surfaces and do smooth shading.¹⁴ The Interim Dynabook^{15,16,17} uses a double buffering architecture to separate the image expansion and raster-scan refresh processes. Although pictures are stored in highly compact form, the system has an expensive scan converter and a large amount of memory devoted to storing three representations of the picture. Because the special purpose hardware needed to decode the condensed digital representation is very expensive, general implementations of the coded picture scheme are either severely limited or still in the experimental stage.

The last scan conversion scheme we will discuss is bit mapping. In a bit mapping system, scan conversion is done in software. The picture is displayed directly from the frame buffer and requires minimal hardware. Arbitrary shapes may be easily defined and displayed at any X,Y position. Although it is very flexible, bit mapping has major drawbacks when portraying real-time motion. The amount of data to be processed is much greater than with any of the previously discussed methods. Consequently, only small objects may be moved quickly if current processor speeds are considered.

VISION II ARCHITECTURE

VISION II is a multiprocessor raster graphics display capable of supporting a high-level programming language. It is designed to be inexpensive but able to operate in real-time for a significant range of dynamic graphics applications.

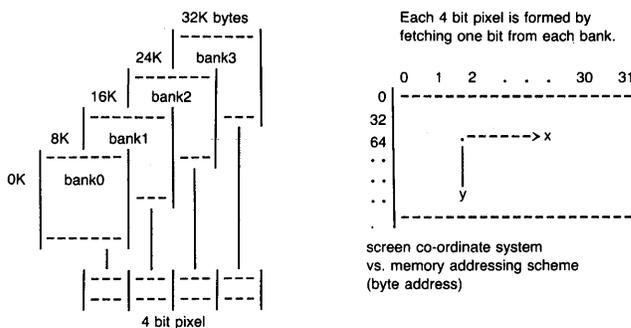
The current system uses two processors linked by a 9600 baud serial port. A DEC PDP-11/03 functions as a language processor to provide communication between the user and the display. It is responsible for terminal I/O, disk I/O and running applications programs. It also supports an interactive graphics language called GRASS3. The PDP-11 family of computers was chosen because of the wide range of computing power and the hardware/software compatibility between all of its members.¹⁸

Software scan conversions (plotting boxes, lines, etc.) is performed by a Z80 microprocessor.¹⁹ Simple programs involving display functions are also handled by the Z80. Although the most powerful configuration uses both processors, the two components may operate independently. For example, the Z80 by itself allows the graphics processor hardware to emulate devices such as those discussed in the first section.

The frame buffer is a 256x256 pixel array. The number of bits per pixel is determined by how many planes of video memory are installed in the graphics unit. Each pixel is formed by fetching one bit from each bank. Thus the N th bit of a pixel can be found in the N th memory bank. The diagram below shows a 4 bit per pixel system.

The frame buffer can be accessed by the graphics processor as normal memory. Each bank is 8K bytes. Changing a byte will modify one bit of 8 consecutive pixels. The first bank contains the least significant bit of all pixels. Successive banks are higher order pixel bits.

Pixels can also be accessed by specifying their X,Y coordinates. A single pixel can be modified independently of other pixels on the screen. This access method touches one bit of each bank simultaneously.



The bit-map technique was chosen because it is the most flexible display mechanism. Because the X,Y pixel mapping is done in hardware, the graphic processor is not bogged down with isolating individual bits in memory. Consequently, displays and transformations are faster. A similar technique was used at Bell Telephone Laboratories in Murray Hill, New Jersey.^{20, 21, 22}

In addition to the X,Y pixel mapping, VISION II has another important hardware assist called *pixel mapping* to further reduce processor overhead. Each pixel value undergoes a state transformation to another pixel value when the screen is refreshed by the hardware. This refresh rate may be set by the user. If the new state of a pixel is mapped to its former value, all of the pixels with that value will remain unchanged on the screen. If it is different, all such pixels assume the new state during the next refresh.

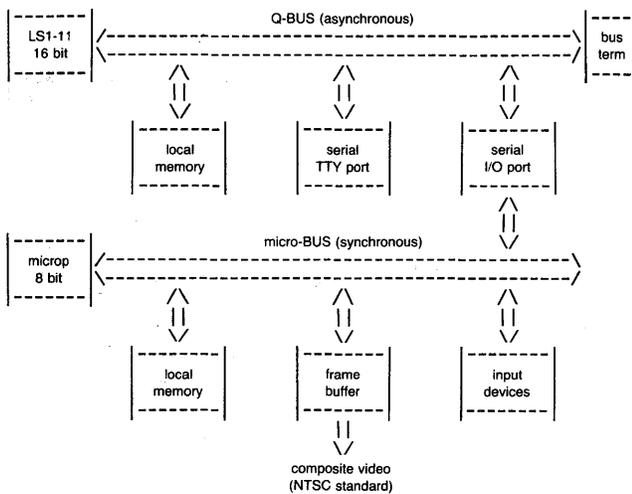
By mapping pixels to successively dimmer colors or gray levels until the background is reached, an object will appear to *decay* without processor intervention. Unlike coded picture definitions described earlier, the pixel-mapping mapping technique does not restrict picture definition or movement. It is also simple and inexpensive to implement in hardware. Pixel mapping can be used to emulate a refresh CRT (albeit a slow one) such as a Vector General.²³ It is also flexible enough to produce a variety of other effects which would otherwise consume considerable processor time.

THE GRAPHICS PROCESSOR

The VISION II graphics processor consists of a Z80 CPU, ROM (permanent programs), RAM (scratch-pad area), a frame buffer and a serial I/O port for communication with the language processor. Ports to peripherals such as joysticks, dials, a data tablet and buttons may be added easily. Below is a block diagram of the VISION II system.

In the current configuration, ROM contains the SDB-80 operating system and debugger. The graphics processor software and data areas (character sets, variables) reside in RAM; it will eventually be moved to PROMS. The frame buffer, of course, contains the objects being displayed. Communication with the language processor is via a full-duplex asynchronous serial port.

The graphics processor scan conversion software has the following capabilities:



1. Plot a point (at arbitrary X,Y) of a given color;
2. Draw an arbitrary rectangle in any color anywhere on the screen;
3. Draw a line of any color between any two points;
4. Save an arbitrary rectangle on the screen in local memory;
5. Plot a previously saved rectangle anywhere on the screen; and
6. Display a text string anywhere on the screen.

There are several plot modes possible. A pixel can be changed to a given color. It is also possible to conditionally modify a pixel depending on the color it contains. For example, one can tell the graphics software to only plot a pixel if the new color is greater (numerically) than the color that was there before. This can be used to have one object pass over another. One can exchange a pixel or rectangle on the screen with one in scratch memory.

The basic primitives above can be combined in pictures. The graphics processor can be asked to display a list of such pictures continuously, i.e. it can *refresh* the VISION II screen. Associated with each picture is a plot mode and X,Y position. The graphics processor has commands to change the attributes of a particular picture. For example, changing the X,Y position of a picture will cause it to be displayed at a different spot when it is refreshed. If pixel mapping is enabled, the picture will be automatically erased.

THE LANGUAGE PROCESSOR

The VISION II language processor is a DEC PDP-11. Choosing a PDP-11 gives the widest range of computing power possible while still permitting hardware and software compatibility. All members of the PDP-11 family have the same instruction set. Consequently, software that runs on a small, single user LSI-11 will also run on a multi-user VAX-11/780.

GRASS3, the language interpreter, is implemented for most of the PDP-11 operating systems.²⁴ The language is easy to learn and use, yet powerful enough to do high-level applications. Although the system is interpretive, a compiler can be invoked to speed up individual programs. A user may

schedule two or more routines to run in parallel. In this case, GRASS3 executes one line from each in a round-robin fashion. Other features include interactive debugging, file I/O, error trapping and on-line helps.

The GRASS3 base system is device-independent, but a software interface for the VISION II graphics unit has been implemented. There are commands to draw boxes, lines, points and text strings. A user may save an arbitrary rectangle on the screen and display it at any X,Y coordinate. Pictures composed of the basic primitives can be linked to GRASS3 variables to control their screen position. Special variables allow access to joysticks, dials, etc. Because it is easy to add commands to the language, one can write a specialized function in assembler and call it as system directive.

Communication with the Z80 graphics processor is via a serial port. When a user issues a graphics command, information is sent to the Z80 telling it what to do. When the graphics processor has finished its task, it sends confirmation back to GRASS3. GRASS3 will wait for confirmation before sending another display directive.

Updates on the analog input devices (joysticks, dials, etc.) are sent to the language processor at intervals. The graphics processor uses one of its internal timers to determine when to send the information. Updates will not be sent if none of the devices have changed since last transmission.

The VISION II interface consists of several new GRASS3 commands (written in assembly language) and a set of utility programs (written in GRASS3). The new commands allow a user to directly manipulate the display:

point X,Y,M	plot or read a point
box S,Y,XL,YL,M	draw a box
line X,Y,M	draw a line
text X,Y,M,S	display a string of text
snap X,Y,XL,YL	save a picture
display X,Y,M	display a picture
xchg X,Y,M	exchange pictures
move pix,X,Y,M	move picture dynamically

Data may be displayed in a variety of modes (the *M* argument above). In addition to direct plotting, many bit operations such as exclusive or, complement, etc. are provided. For example, the following program will exclusive-or random boxes on the screen forever.

```
box ran(-128, 128),ran(-128,128),ran(10,40),ran(10,40)
,5
repeat
```

The *move* command links a picture with GRASS3 variables that, when updated, control its movement. A picture can be a simple screen rectangle, or a collection of basic primitives such as boxes, lines and text. Associated with each picture is an X,Y position and a display mode. If we wanted to move a previously created picture called *cursor* based on the joystick variables we would type:

```
move cursor,jx,jy,5
```

Whenever the joystick was moved, the picture would follow its movement on the screen.

A set of utility programs which use the VISION II commands have been developed to facilitate the creation and manipulation of pictures as objects. Because they are written in GRASS3, these programs are easy to understand and modify. Although they are useful as tools, they also make excellent tutorial examples for learning VISION II.

zinit	initialize Z80 memory
zalloc	allocate block of Z80 memory
zfree	free previously allocated block
zmap	print Z80 memory map
save	save picture from screen in Z80 memory
putdisk	save picture on disk
getdisk	get picture from disk
scale	make picture bigger, smaller
reflec	reflect picture about axes
draw	picture "editor" to create pictures

Pictures (rectangular screen areas) may be saved in the Z80 memory (graphics processor) or in the PDP-11 memory (language processor). If a picture resides in Z80 memory, displaying it is fast but manipulating it from GRASS3 is slow. This is because the picture does not have to be sent to the graphics processor before it is displayed—it is already there. To get at it from GRASS3, the pixels must be accessed individually with the *point* command. Keeping a picture in the PDP-11 memory requires more space but allows straightforward manipulation of individual pixels without changes to the screen. Displaying such information, however, is very slow because the pixels must be plotted one at a time. The utility programs can deal with pictures in either place and can transfer pictures back and forth between the Z80 and the PDP-11.

To create pictures of any complexity, the *draw* program can be used. With *draw* one can move a cursor of arbitrary size around the screen using the keyboard or joystick. By combining *draw* and *scale*, a picture can be created at low resolution with a larger cursor and scaled down to a higher resolution. This feature is useful for creating small, detailed objects such as members of a character set.

After a picture is created, it is easy to manipulate it with GRASS3. For example, to move a picture of a car across the screen one would write:

```

draw                ;draw the car
... draw the car ...
car = save(x,y,10,5) ;save 10 by 5 car
do {
  displ y car,x,y,1 ;display it
  box car,x,y,10,5,0 ;erase it
  x = +10           ;move it
  repeat           ;keep doing it
}

```

By using arrays, it is possible to move the car along a predetermined path. In the program below, successive coordinates are sorted in a two dimensional array and used to move the car.

```

next = array(integer,99,1) ;array of X,Ys
... store X,Ys in array ...
S = size (next)           ;how many?

```

```

{I = - 1                ;start at 0th pair
while ++I<S,{disply next
(I,0),next(I,1),1,car box x,y,10,5,0}
}
repeat                  ;wrap around path
}

```

One of the most powerful features of GRASS3 is that two or more independent programs can run at the same time. For example, suppose we want to further develop the program above to accept the picture of display and the array of points as arguments. Let us define a picture of a Volkswagen Beetle and an array of points describing a figure 8. Thus we can move the *beetle* picture in a figure 8 with the command below.

```
drive beetle, fig8
```

To move a pinto in a figure 8, we would type

```
drive pinto, fig8
```

On most systems, our *drive* program would have to be completely rewritten if we wanted to move both the Beetle and the Pinto at the same time. In GRASS3 one would use the *schedule* command:

```

sched drive,,beetle,fig8
sched drive,,pinto,fig8

```

The *drive* program can be scheduled twice (or more) with different arguments to show two cars moving. GRASS3 will execute one line from the first scheduled program, one line from the next, etc. to give the illusion that all scheduled programs are running at the same time. Note that if you wanted to make the seat on the Beetle eject and the Pinto explode whenever they collided it would require extra programming.

DISTRIBUTED DESIGN

If you were to poll a group of computer hobbyists on how to increase the power of a particular microcomputer system, you would most likely get two suggestions, namely "add memory and I/O devices" and "substitute a faster processor with a more powerful instruction set." If someone suggested "add a few more processors," he/she might draw stares of disapproval.

There are few examples of distributed processing as applied to digital display devices. All too often the solution to scan conversion speed problems has been to increase the complexity (thereby reducing the generality) of the hardware or impose limitations on the pictures or transformations possible on the system.

With the VISION II system, not only can a graphics processor support several frame buffers, but multiple graphics processors can be attached to a single language processor. Connecting the various modules in different configurations allows the VISION II system to be tailored to a specific application. Graphics processors can be added to control different groups of objects, thereby alleviating speed problems. The

generality of pictures and motions is not affected because scan conversion is still done in software.

For example, consider the problem of displaying geographic maps and panning over landscapes. Each feature (e.g., rivers, cities) could be assigned to a graphics processor. The output of the processors could then be merged to display the whole map. With this approach, a feature can be enhanced or eliminated independently of the others. Synchronization would be controlled by the language processor.

By using a larger language processor in a timesharing environment, many graphics units can be handled simultaneously. This approach would be useful for an interactive educational system. Lessons could be written in GRASS3. Each student would have his/her own terminal and graphics unit but would be able to communicate with others through a common database.

Because scan conversion is in software, VISION II can easily be programmed to emulate other systems. By changing the graphics processor software, the display unit can behave like a vector plotter, incremental plotter, plasma panel, CRT terminal, etc. The VISION II display can thus be used with existing software packages. This allows users to run software already written with little or no modification.

CONCLUSIONS

Experience with VISION II has shown that dynamic graphics is possible on a raster-scan display with relatively simple hardware assist. Several small objects can be moved simultaneously with somewhat fluid motion on the current system. Increasing the speed of the graphics processor CPU could improve this. The Z80 currently runs at 2 MHz and could be replaced with a 4 MHz version. It could also be replaced with one of the newer, faster 16 bit microprocessors which have recently been developed.

Software on the VISION II system evolved gradually. Initially the software was very primitive. The graphics processor could plot points, lines, and boxes, and the GRASS3 interface merely provided these capabilities to the user. The next phase was to allow the user to define arbitrary screen rectangles, save them, and display them. This was still essentially static graphics, since movement could be achieved only in a very procedural way and was not at all fluid.

The most recent software development allows the graphics processor to emulate a display list processor. It maintains a display list in local memory and constantly refreshes screen memory. Each picture has an X,Y position associated with it, which may be updated by the language processor. If the picture's position has changed since it was last refreshed, it is redisplayed by the display list processor. Note that it does not have to be erased first—the pixel mapping feature takes care of this. In this way, the graphics processor looks like a (slow) refresh CRT. Admittedly, the number and complexity of the objects that may be moved in this way is not comparable to a vector display, but neither is the cost.

It is worth mentioning here that the generality of bit mapping more than offset the increased processing time necessary for movement of objects. Because the system is bit mapped,

a user can define an arbitrary picture on the screen easily and move it to any position. Pictures can be scaled, rotated and otherwise transformed. There is no limit to the size of a picture (although defining pictures larger than the screen is difficult) or to its complexity.

VISION II has proven to be easy to configure and expand. The first version had 1 bit per pixel (one plane of memory) and a resolution of 256 by 256. Two other memory planes were added to the system—it now has 3 bits per pixel. The next version will have 4 bits per pixel and a resolution of 512 by 512.

Experience with naive users has shown that more communication between the language processor and graphics processor is needed. The graphics primitives are nonprocedural and do much of the behind-the-scenes work necessary to refresh the display. For this reason, it is not straightforward to synchronize the movements of one object with the movements of another. To do this currently places the burden on the programmer and is beyond the ability of most beginners.

Future plans for the software include a GRASS3 *picture compiler* which will translate a subset of the GRASS3 language into a form executable by the graphics processor. The user will then be able to use GRASS3 control structures in pictures to initiate or synchronize movement. GRASS3 multi-tasking primitives are also being enhanced. Process control capabilities such as those of the ADA language²⁵ are being considered, along with a general message-sending facility to communicate between GRASS3, multiple graphics processors and the host operating system.

REFERENCES

1. Rocchetti, Robert, "VISION II: A Small Scale Expandable Graphics System," Master's Thesis. University of Illinois, 1980.
2. Newman, William M., and Robert F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1973.
3. Metzger, Richard A., "Computer Generated Graphics Segments in a Raster Display," *Proceedings of the SJCC*. 1969.
4. Jordan, B. W. Jr., and R. C. Barret, "A Scan Conversion Algorithm with Reduced Storage Requirements," *Communications of the ACM*. Vol. 16, No. 11 (November 1973).
5. Barret, R. C., and B. W. Jordan, Jr., "Scan Conversion Algorithms for a Cell Organized Raster Display," *Communications of the ACM*. Vol. 17, No. 3 (March 1974).
6. Thornhill, D. E., and T. B. Cheek, "Raster-Scan Tube Adds to Flexibility and Lower Cost of Graphic Terminal," *Electronics*. February 1974.
7. Articles on the Model 2640A Interactive Display Terminal Family. *Hewlett-Packard Journal*. June 1975.
8. Jordan, B. W. Jr., and R. C. Barret, "A Cell Organized Raster Display for Line Drawings," *Communications of the ACM*. Vol. 17, No. 2 (February 1974).
9. Articles on the Tektronix 4025 Computer Display Terminal. *Tekscope*. Vol. 10, No. 1, 1978.
10. Baskett, Forest, and Leonard Sustek, "The Design of a Low Cost Video Graphics Terminal," *Computer Graphics*. Vol. 10, No. 2 (Summer 1976).
11. Hunter, Gregory M., "Full-Colour Television from the Computer, Refreshed by Run-Length Codes in Main Memory," Technical Report Number 182, Computer Science Laboratory, Princeton University, April 21, 1975.
12. Laws, A., and W. M. Newman, "A Gray-Scale Graphic Processor Using Run-Length Coding," *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structure, IEEE Computer Society*. May 14-16, 1975.

13. Myers, Allan J., "A Digital Video Information Storage and Retrieval System," *Computer Graphics*. Vol. 10, No. 2 (Summer 1976).
14. Rougelot, R.S., "The General Electric Color TV Display," *Pertinent Concepts in Computer Graphics*. University of Illinois Press, 1969.
15. Kay, Alan. Presentation at the First Annual Conference on Computer Graphics and Interactive Techniques, Boulder, Colorado, July 1974.
16. Learning Research Group. *Personal Dynamic Media*. Xerox Palo Alto Research Center. 1976.
17. Baecker, Ronald M., "A Conversational Extensible System for the Animation of Shaded Images," *Computer Graphics*. Vol. 10, No. 2 (Summer 1976).
18. *Digital Microcomputer Processors*. Digital Equipment Corp., 1979.
19. *SDB-80 Software Development Board, Operations Manual*. Mostek Inc., May 1977.
20. Noll, Michael A., "Scanned-Display Computer Graphics," *Communications of the ACM*. Vol. 14, No. 3, March 1971.
21. Denes, Peter B., "Computer Graphics in Colour," *Bell Laboratories Record*. Vol. 52, May 1974.
22. Denes, Peter B., "A Scan-type Graphics System for Interactive Computing," *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structure, IEEE Computer Society*. May 14-16, 1975.
23. "Vector General, System 3 Reference Manual." Vector General Inc., 1974.
24. Donato, Nola. "GRASS3—A Base System for Interactive Graphics." Masters Thesis. University of Illinois, 1978.
25. Ichbiah, J. D., et. al., "Rationale for the Design of the ADA Programming Language," *SIGPLAN Notices*. Vol. 14, No. 6 (June 1979).

The development of the reactor safety film

by NANCY N. SHEHEEN and PATRICK J. HODSON

*Los Alamos Scientific Laboratory**

Los Alamos, New Mexico

ABSTRACT

The first computer-generated film of LASL's Reactor Safety efforts was developed using the ANIMATE framework, a program that adds visual capabilities to MAPPER. Numerous software limitations had to be overcome within a very limited production schedule. A significant achievement was the 15,000-vector-per-frame sequence depicting a pressurized water reactor core with parts flashing while pumps circulate fluid through the system.

INTRODUCTION

In August of 1978, the Energy Division of the Los Alamos Scientific Laboratory (LASL) was asked to participate in the Seventh Annual Energy Technology Conference to be held in Washington, D.C., in March of 1979. Subsequent meetings regarding our contribution crystallized the idea of presenting an overview of LASL's Reactor Safety efforts using the very computers on which we develop large and sophisticated codes to evaluate postulated accidents as well as actual occurrences in nuclear reactors.

Visual sequences are often used by the Reactor Safety analysts for presentation of technical data; however, these usually present only singular concepts and are limited to highly technical data. Before the present work, no attempts had been made to develop computer-generated technical material into educational form. Using the software packages MAPPER, ANIMATE, and ANIPLT developed here at LASL, we created a film about our Reactor Safety efforts.

The following hardware was used in the development of the film: a Tektronix model 4014-1 graphics terminal, a Tektronix model 4954 tablet, a CDC-7600 computer, and an FR-80 microfilm recorder. The personnel involved included a senior programmer, a data analyst, a technical writer, and a graphic artist in addition to program sponsors who served as technical advisors.

*The Los Alamos Scientific Laboratory was renamed the Los Alamos National Laboratory on January 1, 1981.

PROGRAM DESCRIPTIONS

MAPPER is a program created by David A. Dahl, of the LASL Environmental Surveillance group. MAPPER reads English language-based commands from a file that has been generated by the user. These commands tell MAPPER to draw boxes, circles, ellipses, characters, complex line segments, etc. on a specified graphics device in a wide variety of line format characteristics. The following is a list of some of the options available in MAPPER:

- six types of label commands,
- twelve different character styles,
- color controls,
- shading controls,
- projection ports,
- symbol generation and distortion, and
- movie controls

ANIMATE is a program created by David A. Dahl and Kenneth H. Rea, also with LASL. ANIMATE reads a specially constructed movie file and produces an input file. The user starts with a MAPPER command file that defines everything that needs to be drawn in one frame. Then the user modifies this command file using the ANIMATE commands and syntax to define how things should be changed from frame to frame during a movie sequence. ANIMATE allows the user the following capabilities:

- to switch on or off specified sections of the command file,
- to overlap switches,
- to specify how much and in what way specified values in the command file should vary from frame to frame, and
- to specify the number of frames for each sequence

PROGRAM DEVELOPMENT

Two new commands were added to the MAPPER program to aid in the production of the Reactor Safety movie. The first command assists in the drawing of pipes. The user simply specifies where the pipes in the reactor are to go and MAPPER inserts them with speed and precision. The second addi-

tion is a command to assist in drawing wavy lines for the simulation of water level and movement. The user specifies the X-Y start and stop points, the number of points to use in drawing the curve, and the type of curve desired. MAPPER then draws the wavy line as specified.

The ANIMATE program required two modifications. The first changes the way ANIMATE handles the frame-to-frame variations for specified values inside a MAPPER command file. The change makes the frame-to-frame variations based on a circular function. This fix makes it easier to do the same series of frame-to-frame variation repeatedly without having to specify duplicate variations for different frame periods. The second modification to ANIMATE dumps to a file the values for each switch and variable on a frame-by-frame basis. This provides an accurate record of what happened during a sequence of frames so that the user can, with greater ease, match soundtrack to action.

To get a better idea of what happens to certain values during a movie sequence, a program called ANIPLT was created. ANIPLT reads the ANIMATE dump file containing the frame-by-frame values and then makes two-dimensional plots of each variable vs. frame number. The frame number is the X axis and the switch or variable is the Y axis. This makes it very easy to see the approximate condition of a particular switch or variable at any frame number in a sequence.

PROGRAMMING THE FILM'S MAJOR SEQUENCE

The Reactor Safety film's major sequence starts with a basic drawing of a nuclear reactor power plant. The drawing was made on a finely meshed sheet of graph paper. Actual X-Y locations for each line drawn were manually read from the drawing and typed into the computer in a format compatible with the MAPPER program. Then MAPPER shading and color commands were used to set up the different color tones for the various reactor components. The command file was then modified for the ANIMATE program. These modifications included the following:

- setting up the variables to illustrate pump action,
- setting up the variables to show water level and wave action,
- setting up a variable to handle the turbine blade rotation at a constant rate,
- setting up the variables to control the projection port capabilities in MAPPER so that we could zoom in on the various reactor components,
- setting up the switches and variables for ANIMATE to control the pump actions,
- setting up a complex switching and shading scheme in the command file to give the illusion of water flow in the pipes and the reactor vessel, and
- setting up the switches to control the flashing of the various components as they were being described.

After the MAPPER command file had been prepared, it was then processed by ANIMATE to generate a command file for MAPPER and a file to be postprocessed by ANIPLT. When the results from the running of ANIPLT revealed that

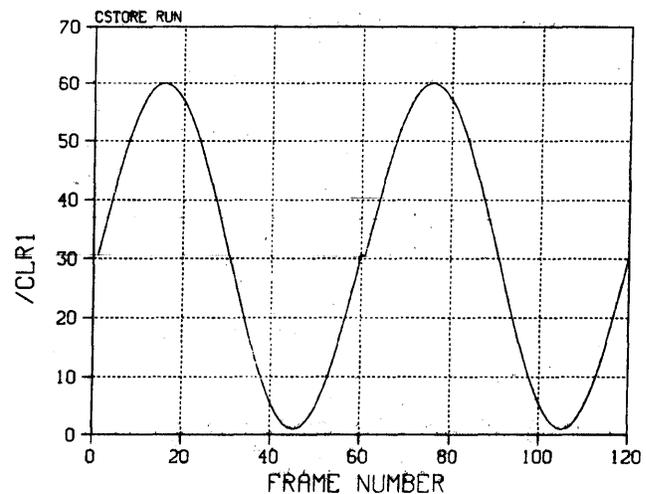


FIGURE 1—ANIPLT plot indicating variance of one color. This shows the user how ANIMATE changed the variable CLR1 through a range of 60 different colors starting at 30 on frame 1.

all of the variables and switches were performing correctly, the command file generated by ANIMATE was run through MAPPER to generate a graphics file to be processed by the FR-80 microfilm recorder. After viewing the film and making some corrections and modifications the command file was again run through ANIMATE, ANIPLT, MAPPER, and the FR-80 microfilm recorder to obtain the final copy of the sequence. It was then spliced in with many other sequences to make the entire movie.

PRODUCTION COSTS

The original 11-1/2-minute film required 70 hours on the Laboratory's CDC-7600s; of this time, 40% was spent on software development and on development of visual concepts. In addition, every sequence required several remakes to obtain acceptable footage. Additional expense was incurred because of substantial overtime charges, some priority use of the 7600s, the taping of the narrative, adding a music track, and having the film commercially finalized.

Approximately eleven man months were required; when all relevant charges were factored in, the production costs represented \$3400/min. Had animation artists developed the film, charges would have ranged from \$4000-\$7500/min. We estimate that succeeding projects of this nature would represent a 50% decrease from the initial expenditure because of the experience gained and the now significantly improved software. In addition, future Reactor Safety film efforts will employ the Energy-Division VAX computers, which may be even more cost effective for this kind of work.

CONCLUSIONS

Our computer-generated film evolved from a simple outline into a sophisticated and complex medium. It required large amounts of computer resources and extension of the available graphics software capabilities.

Impending deadlines required the abandonment of several sequences and concomitant deletions from the narrative. Thus, the finalized version is not as originally envisioned. Nonetheless, the film was received favorably and we have subsequently had numerous requests for copies. This clearly reflects the impact of computer-generated animation upon viewing audiences. It has shown itself to be a highly effective means of communicating technical information to varied audiences. We also recognize its potential as a technical tool for translating complex calculations into tangible form.

ACKNOWLEDGMENTS

Special thanks are due Michelle E. Schirru and Margaret M. Scott for their programming support, Vicki Hartford for her

creative contributions, and Jerry V. Valdez for his work at the graphics table.

We appreciate the assistance of James H. Scott, Michael G. Stevenson, James F. Jackson, and John C. Vigil who provided both technical reviews and management support.

BIBLIOGRAPHY

No formal document describes the ANIPLT program. It was developed solely to assure the user that switch commands were implemented correctly for our ANIMATE film. A sample plot of ANIPLT is attached that shows an input variable vs frame number. For information on MAPPER and ANIMATE, see David A. Dahl, "MAPPER," Los Alamos National Laboratory report PIM-2-J5AJ (March 1979).

The MODEL/IMAGES2 system: An application of computer graphics and three-dimensional geometric modeling to the jet impingement problem

by W.R. WINFREY and S.R. RICKETTS

Babcock & Wilcox
Lynchburg, Virginia

ABSTRACT

The design of a nuclear power plant requires the consideration of many abnormal conditions, including the effects of a fluid jet emerging from a postulated break in a high-pressure pipe and striking other plant components. This paper describes the computer graphics and three-dimensional modeling system, MODEL/IMAGES2, which has been used successfully to automate the solution of the jet impingement problem. The MODEL program is used to construct the targets, using modeling techniques such as stacking of graphic primitives, set operations, and coordinate operations. The IMAGES2 program constructs a model of the fluid jet, determines the target struck, and computes the impinged areas. The MODEL/IMAGES2 system has reduced the man-hours required for analysis and improved the accuracy and reliability of the results.

INTRODUCTION

The design of a nuclear power plant requires careful consideration of many abnormal conditions. An important calculation, the jet impingement problem, considers the effects of a fluid jet emerging from a postulated break in a high-pressure pipe and striking plant components. The portion of the analysis of particular interest here is the calculation of the loads exerted on the plant components by the jet. The analysis is broken down into two parts—an engineering problem and a geometric problem.

- Engineering problem—Computation of break size, fluid properties at the break, jet expansion characteristics and dynamic pressures within the jet
- Geometric problem—Description of target geometry and computation of effective cross-sectional areas

This paper describes a computer graphics system, MODEL/IMAGES2, which has successfully automated the solution of the geometric jet impingement problem. This automation has reduced the man-hours required for the analysis and improved the accuracy and reliability of the results.

PROBLEM DESCRIPTION

The basic geometric problem of the jet impingement analysis will be described with the aid of Figure 1. This figure shows a typical reactor coolant pump, its supports and restraints, and two postulated breaks in the horizontal pipe connected to it.

Consideration of the break on top of the pipe reveals two problems immediately: (1) The geometry of the jet, the pump, and the supports and restraints must be described. The ideal situation would be to actually construct (three-dimensional) mathematical models of these objects and perform the calculations on them. (2) The portion of the target actually lying within the boundary of the jet must be determined. The parts of the target that lie outside the jet boundary are not of interest, since they could not be struck by the jet.

Consideration of the lower break reveals another problem: (3) The restraints just below the break are potentially shielding the supports on the base of the pump. The computation of this shielding requires that the target surfaces be projected along stream lines to a projection plane and the projected surfaces clipped against one another to account for shielding and thus determine the parts of the surfaces actually exposed to the fluid.

Once these problems are resolved, one must compute the effective cross-sectional areas of the exposed surfaces, which—in combination with the fluid pressures—give the jet impingement loads on the targets. Five steps are involved in the jet impingement analysis:

1. Model the targets that are nuclear steam system components, supports, and restraints.
2. Model the jet and intersect the jet and targets to determine the portion of the target lying within the jet boundary.
3. Project target surface along stream lines to the projection plane.
4. Compute the shielding of one surface by another.
5. Compute the projection plane areas of exposed surfaces.

Each of these steps involves a geometric analysis; and, because the problem is three-dimensional, each is nontrivial.

The major accomplishment of the MODEL/IMAGES2 sys-

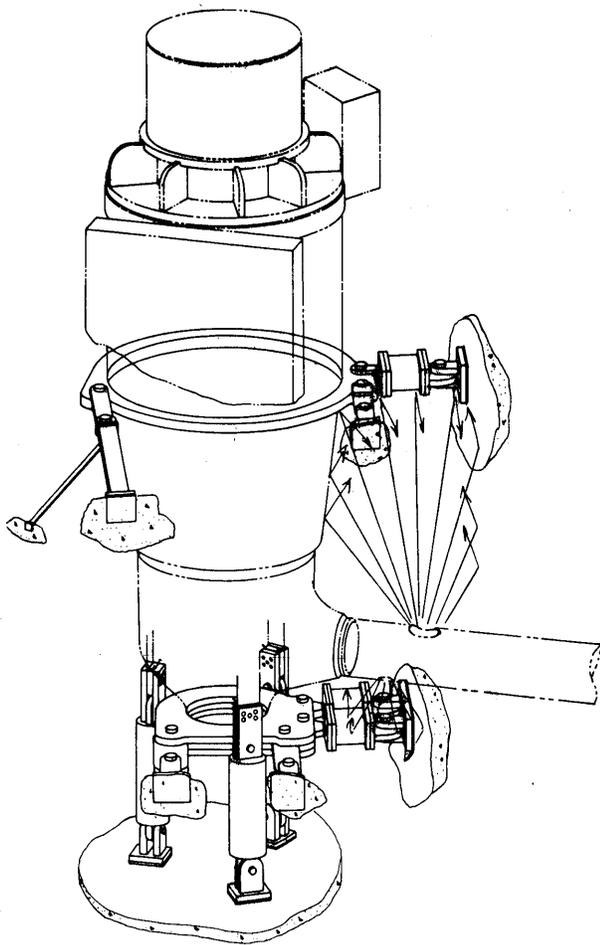


Figure 1—Reactor coolant pump with postulated pipe breaks

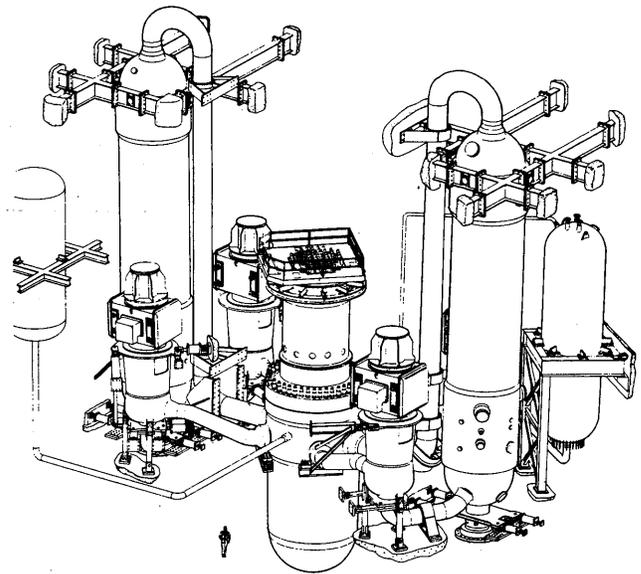


Figure 2—Nuclear steam system component supports and restraints

tem is the ability to do the difficult three-dimensional geometry and thus perform Steps 1 through 5 straightforwardly.

Historical Remarks

Prior to the development of the MODEL/IMAGES2 system, computation of jet impingement target areas was done by hand. In the hand computational procedure, the targets were approximated by simple shapes, such as cylinders and cubes. In the simplest case, the jet would be approximated as a right circular cone. The analyst would derive the equations of the curves of intersection (of the cone and the targets), project these curves back to a projection plane, and compute their areas.

As an analytical technique, this procedure has a number of drawbacks: (1) Many power plant components cannot be well approximated by simple shapes such as cylinders and cubes. However, in order to do the intersection described above, the analyst is restricted to geometrically simple shapes. (2) Computation of the shielding of one component by another is typically quite difficult. (3) Next-of-a-kind calculation, involving a change in the break location or the introduction of

a new target requiring shielding, are typically as expensive as the original calculation. Little is gained by having done the original calculation. (4) The computed loads are often artificially high and must often be multiplied by very conservative safety factors, since only very simple geometries can be used in the hand calculation.

Overall, area computation by hand is a tedious, time-consuming, and inaccurate process.

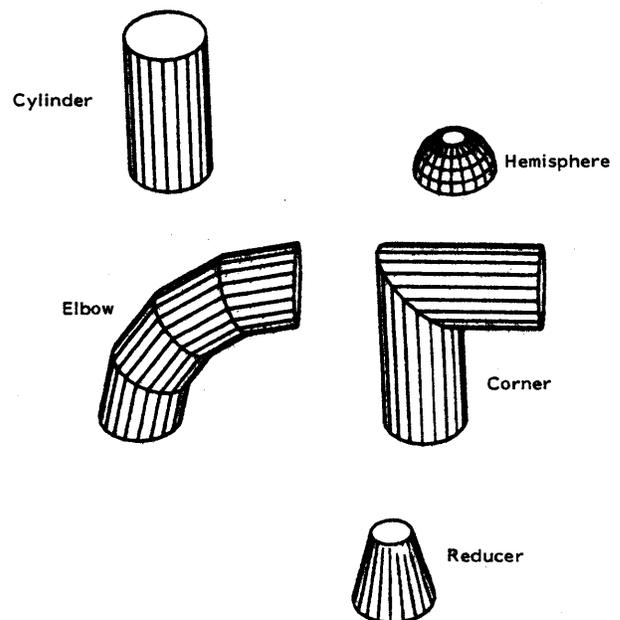


Figure 3—Piping primitives

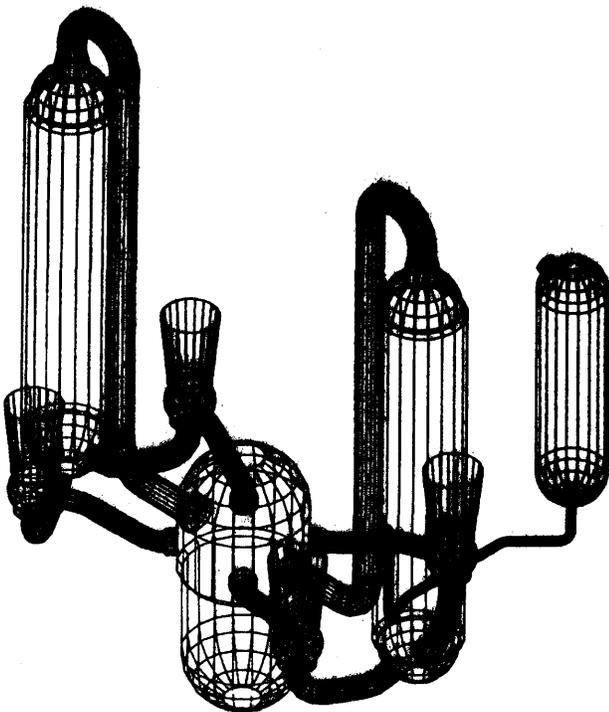


Figure 4—Model for nuclear steam system constructed from piping primitives using MODEL

METHOD OF SOLUTION

The solution of the problem posed in the previous section is achieved by two computer programs, MODEL and IMAGES2. The modeling of power system components, supports, and restraints is performed by the MODEL program. The models, along with certain engineering data, form the input for IMAGES2, the program that actually does the area calculations.

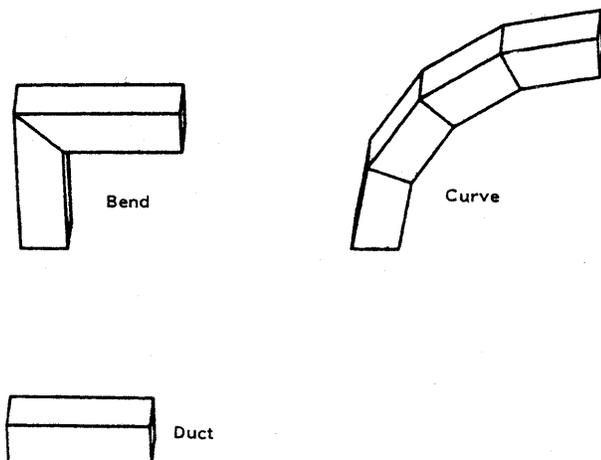


Figure 5—Ductwork primitives

The MODEL Program¹

The scope of the modeling problem can be appreciated by referring to Figure 2.

The plant components are a mixture of regular shapes (such as pipes) and irregular shapes (supports and restraints). Since each object in this figure is a potential jet impingement target, the modeling program must be capable of representing all these shapes. The overall goal then is to construct mathematical models of three-dimensional objects. The solution to this problem is subject to a number of constraints. These constraints essentially guarantee the utility of the models constructed:

- The models must be fully three-dimensional. It is *not* sufficient simply to construct different views of an object, as is done in various computer-aided drafting systems.
- The models must be amenable to the ready computation of basic quantities such as interferences, areas, volumes, and centroids.
- The models must be readily displayed as wire frame drawings, hidden line drawings, and cross-sectional views.
- The models must be easy to manipulate by coordinate operations, such as rotation, translation, scaling, and reflection.
- The models must be easily constructed from such basic information as position, orientation, and dimensions. That is, one should *not* transform a tedious manual calculation into tedious data preparation.

The solution to the above problem that was chosen is the polyhedral approximation of solids. All models built by MODEL are combinations of polyhedra. The choice of a polyhedral approximation instead of an exact curvilinear surface representation involves a number of considerations:

- Polyhedra are among the simplest three-dimensional solids to work with.
- Surface area, volume, and centroid of a polyhedron are straightforward to compute.

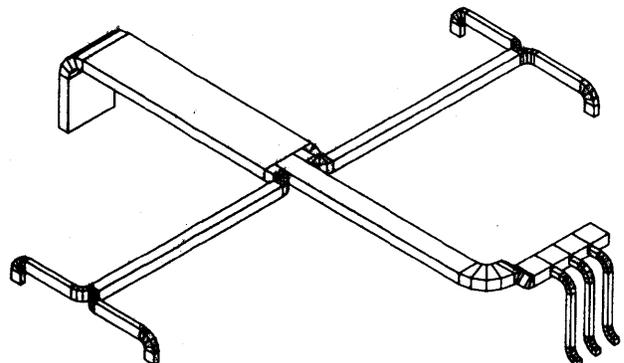


Figure 6—Model of heating system ductwork constructed from ductwork primitives using IMAGES2

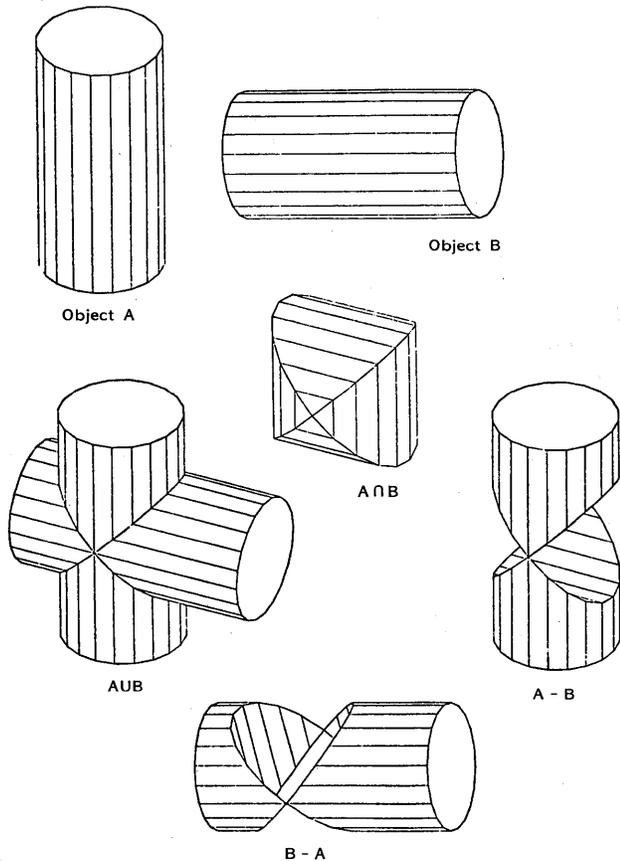


Figure 7—Illustration of set operations involving simple cylinders

- Computation of the intersection (interference) of two polyhedra is straightforward. The intersection of two polyhedral surfaces reduces to the intersection of planes, a computationally simple process. Intersection of two curvilinear surfaces requires the solution of systems of nonlinear equations, a computationally more complex process.
- Polyhedra are easily displayed as wire frame drawings. Computation of hidden surface and cross-sectional views is straightforward and can be automated.
- Polyhedra can be easily rotated, translated, scaled, and reflected.
- The overall modeling process using polyhedra is straightforward.

In short, tradeoff of polyhedra versus curvilinear surfaces is one of ease of construction and computational power versus exact representation.

The modeling process

Three basic modeling techniques are implemented in MODEL:

- Graphic primitives
- Set operations
- Coordinate operations

Graphics primitives

Two basic classes of primitives are supported in the MODEL program: the piping primitives and the duct primitives. The piping primitives are illustrated in Figure 3 and a model of the primary components of a nuclear steam system, built with these primitives, is shown in Figure 4. The duct primitives are illustrated in Figure 5; and a model of heating system ductwork, built with these primitives, is shown in Figure 6.

For both classes of primitives the basic modeling technique is one of stacking. The user specifies a starting point, starting direction, and starting dimensions, such as radius for the pipe and width and height for the duct. Then the particular primitive is specified, along with any additional parameters, such

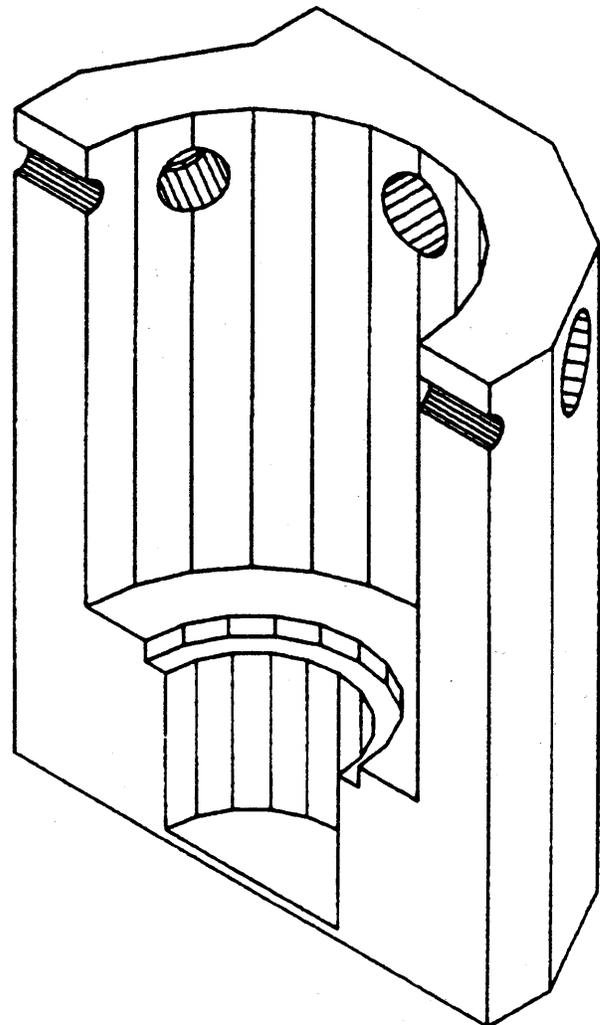


Figure 8—Model of concrete shielding built using set operations in MODEL

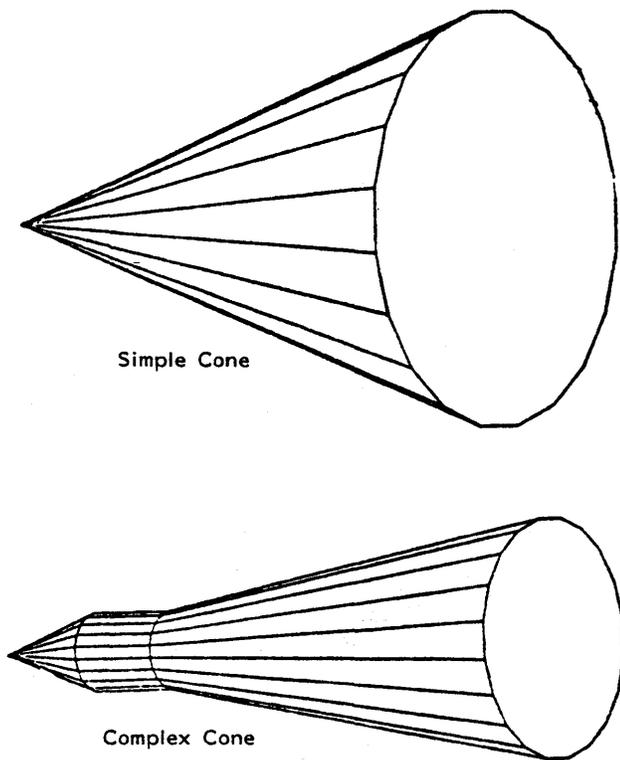


Figure 9—Models of conical jets

as length for a cylinder and radius of curvature for an elbow. The program automatically constructs the data structure for the primitive and moves the user to the other end, where the user specifies the next primitive and its associated parameters, such as a cylinder and its length. The program then constructs that primitive and puts it in place.

Set operations

The set operations used in modeling are union, intersection, and difference, as illustrated in Figure 7. These operations may be thought of, respectively, as welding two objects together, computing the overlap or interference of two objects, and cutting one object with another. These operations supplement the graphic primitives by permitting the user to construct odd-shaped objects by welding and cutting primitives.

An example of the application of the set operations is shown in Figure 8. The model was made by first unioning a block to a large cylinder, then subtracting a number of smaller cylinders to make the cuts shown.

Coordinate operations

The coordinate operations—rotation, translation, scaling, and reflection—are used to change the location and shape of

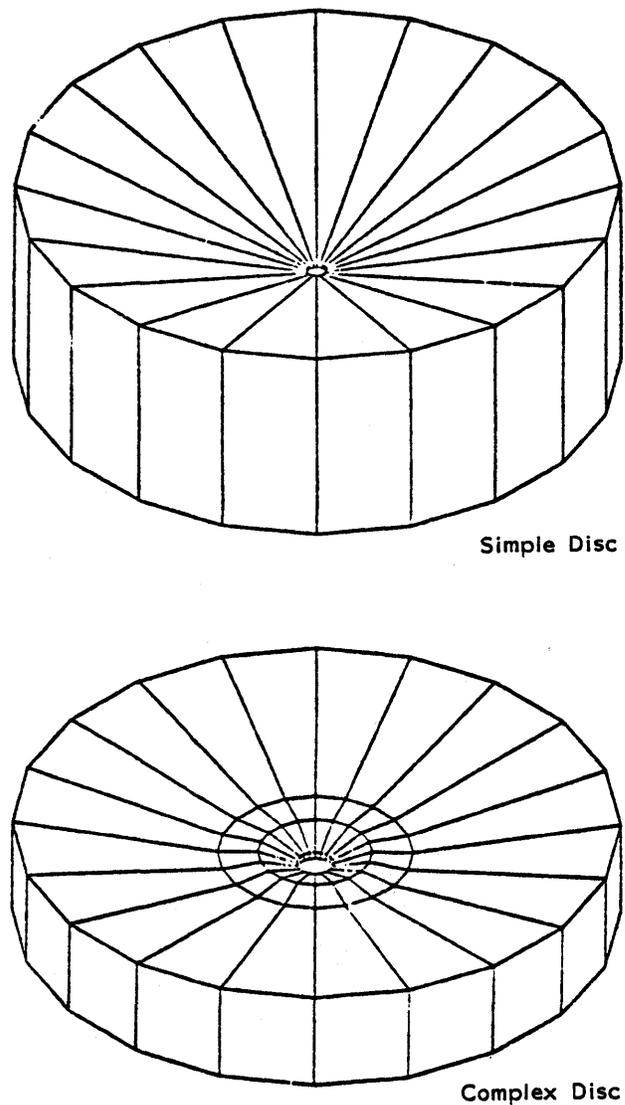


Figure 10—Models of disc jets

a model and to build models by exploitation of symmetry. For example, the model in Figure 4 was constructed by building essentially one-fourth of it directly from piping primitives and generating the remainder by reflection.

The IMAGES2 Program²

The actual computation of jet impingement areas is performed by the IMAGES2 program. It receives input from the user in the form of models constructed by the MODEL program and data files describing the break and the jet characteristics. The output of the program is a set of effective cross-sectional areas and related geometrical information about the targets.

Two basic types of jets are analyzed: the conical jet (Figure 9) emerging from a split/guillotine break and disc jet (Figure

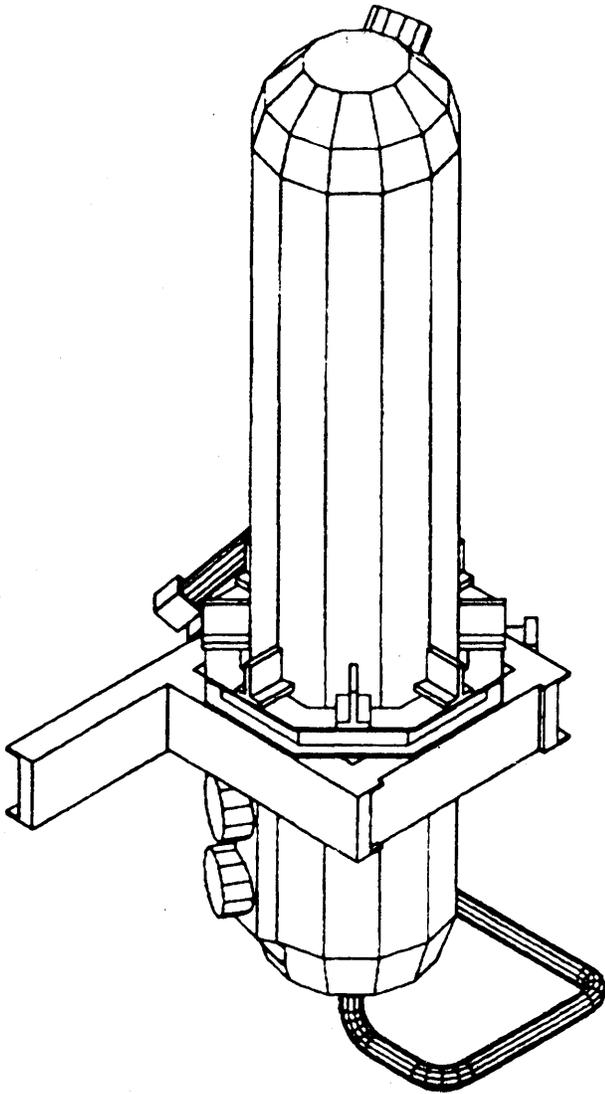


Figure 11—Models of pressurizer and support structures

10) emerging from a limited displacement rupture (circumferential pipe break). The axis of the disc jet lie in a plane perpendicular to the pipe axis at the break. The axis of the conical jet is perpendicular to the plane of the break. The cross-section of the conical jet is assumed to be circular. The jet models are constructed by IMAGES2 from information about the break location, size, and jet expansion angle.

Sample Jet Impingement Calculation

Figures 11 through 13 show a sample jet impingement case. The targets, a pressurizer, and its support framing are shown in Figure 11. The targets were built by the techniques outlined in the previous section. Figure 12 shows the targets again, viewed from the back side, with the jet superimposed. The jet is a simple conical jet emerging from a postulated break in the upper cold leg of the reactor coolant system. Finally, Figure 13 shows the pictorial output of the analysis. The picture

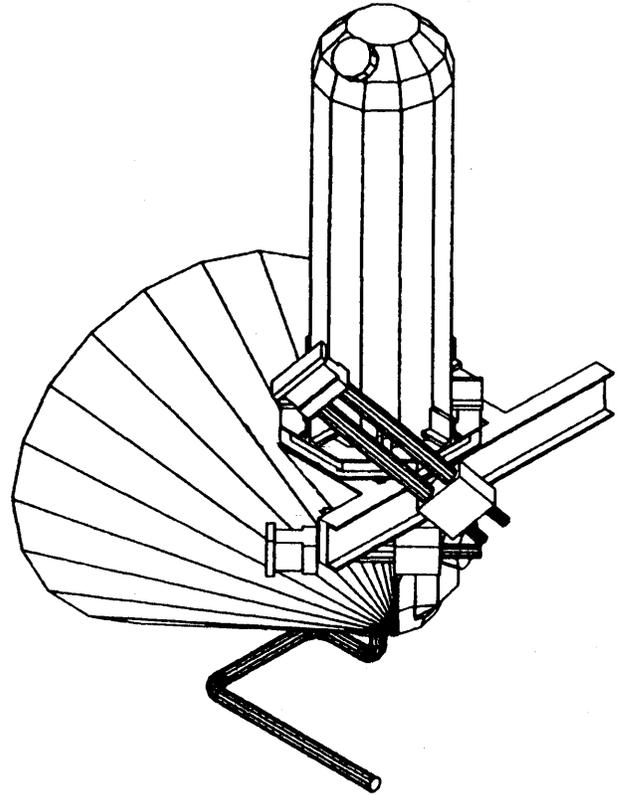


Figure 12—Models of pressurizer and support structures with fluid jet model superimposed (different viewing point from that of Figure 11)

shows the portions of the targets actually struck by the fluid. To understand this picture, one should think of the stream lines radiating from the apex of the cone as lines of sight for an eye located at the apex. Then this fluid cone is equivalent

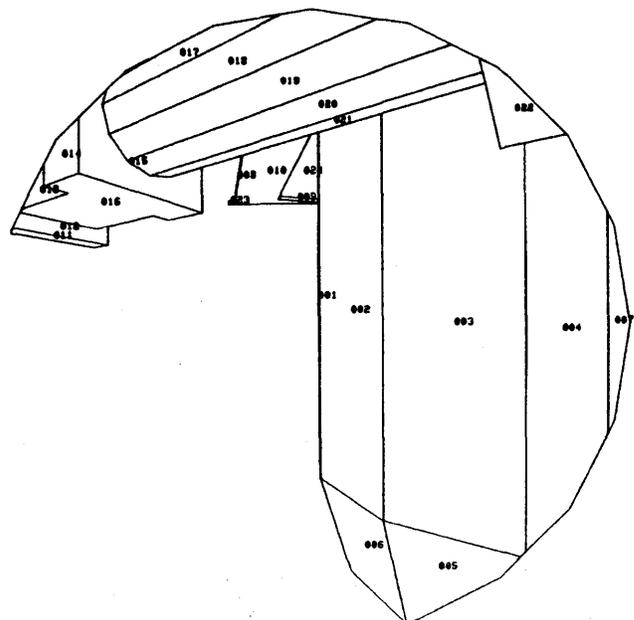


Figure 13—Pictorial output from jet impingement analysis showing impinging surfaces

to a cone of vision, and the visible surfaces are in fact those struck by the fluid. Finally, the projected areas are computed for the IMAGES2 output.

SUMMARY

The MODEL/IMAGES2 system transforms a costly and time-consuming manual calculation into an efficient and inexpensive computer calculation. The human error and approximations associated with manual calculations have been eliminated. Next-of-kind-analyses are inexpensive, since models and data files can be retrieved from backup and used for later calculations.

Extensions

An active effort is under way to create additional engineering applications programs that will use the models created by MODEL. Two particular areas of interest are piping system layout and design and automatic interference determination. The goal in both cases is to automate these calculations in the same fashion that jet impingement has been automated.

ACKNOWLEDGMENTS

Parts of the MODEL/IMAGES2 system are based on an earlier system, IMAGES, developed in a joint project involving the authors and H.J. Borkin and J.A. Turner of the Architectural Research Laboratory of the University of Michigan. That system relied heavily on the ARCH software packages developed at the laboratory.^{3,4}

REFERENCES

1. Ricketts, S.R., and W.R. Winfrey. "MODEL—An Interactive Computer Graphics System for Three-Dimensional Geometric Modeling and Computation." *NPGD-TM-550*, Babcock & Wilcox, Lynchburg, Virginia, May 1980.
2. Winfrey, W.R., and S.R. Ricketts. "IMAGES2—Computer Graphics System for Three-Dimensional Geometric Jet Impingement Analysis." *NPGD-TM-552*, Babcock & Wilcox, Lynchburg, Virginia, June 1980.
3. Borkin, H.J., J.F. McIntosh, and J.A. Turner. "The Development of Three-Dimensional Spatial Modeling Techniques for the Construction Planning of Nuclear Power Plants." *SIGGRAPH Quarterly*, 12 (1978).
4. Turner, J.A. "An Efficient Algorithm for Doing Set-Operations on Two- and Three-Dimensional Spatial Objects." Architectural Research Laboratory, The University of Michigan, Ann Arbor, Michigan, 1978.

The applications of artificial intelligence to law: A survey of six current projects

by SANDRA COOK
London School of Economics
London, England

CAROLE D. HAFNER
University of Michigan
Ann Arbor, Michigan

L. THORNE McCARTY
State University of New York at Buffalo
Buffalo, New York

JEFFREY A. MELDMAN
Massachusetts Institute of Technology
Cambridge, Massachusetts

MARK PETERSON
The Rand Corporation
Los Angeles, California

JAMES A. SPROWL
American Bar Foundation
Chicago, Illinois

N. S. SRIDHARAN
Rutgers University
New Brunswick, New Jersey

D. A. WATERMAN
The Rand Corporation
Los Angeles, California

INTRODUCTION

The literature on computer-based consultation systems has often suggested the possibility of building an expert system in the field of law, but it is only recently that several researchers have begun to explore this possibility seriously. For this session, we have assembled summaries of six major projects on the applications of artificial intelligence to legal problem domains, and we have invited representatives of these six projects to participate in a panel discussion of their work.

The work is quite diverse. Two of the projects are concerned primarily with practical applications: Hafner, represented in Section I below, has explored the use of a conceptual knowledge-base in an enhanced legal retrieval system, and Sprowl, in Section II, has developed and tested a system which assists an attorney in the drafting of routine legal documents. Several of the projects have considered the general problem

of designing a language in which legal rules and legal concepts might be easily expressed: the LEGOL project (Section IV) and the TAXMAN project (Section V) fit within this category, as does the work of Meldman (Section III) on the design of a system for computer-aided legal analysis. Finally, two of the projects are exploring some general theoretical issues about the legal process: McCarty and Sridharan (Section V) are attempting to understand the patterns of argument that appear in a contested corporate tax case, and Waterman and Peterson (Section VI) are attempting to understand the decisionmaking procedures of attorneys and claims adjusters in product liability cases.

Despite these diverse objectives, there appears to be a basic paradigm underlying all of the attempts to apply artificial intelligence techniques to legal problem domains. In one way or another, a legal consultation system must represent the "facts" of a case at a comfortable level of abstraction, and it

must represent the "law" in the chosen area of application, where the "law" consists of a system of "rules" and "concepts" which specify the rights and obligations of the parties in the case. Legal analysis, in its simplest form, is then a process of applying the "law" to the "facts." However, this formulation is deceptively simple, since it masks the real difficulties of the representation problem. The facts of a legal case typically involve all the complexities of daily life: human actions, beliefs, intentions, motivations, etc., in a world of ordinary objects like houses and automobiles, and complex institutions like businesses and courts. And even if the facts of a particular case could be represented in a computer system, the legal rules themselves would often be problematical. Faced with these difficulties, the designer of an artificial intelligence system in a legal problem domain must make several strategic choices, and several compromises, as illustrated in the project summaries printed below.

One strategic choice is the decision to provide either a shallow coverage of a broad area of law, or a deeper coverage of a narrow area of law. Another strategic choice, but one clearly related to the first, is the decision to work with either a syntactically simple, or a semantically rich, representation language. In general, the projects which are most clearly directed towards a practical application have tended to follow the first option: broad but shallow coverage, using a simple and uniform language. Thus Hafner⁴ develops a very sketchy model of negotiable instruments law, but demonstrates that a shallow conceptual model of this sort is adequate for the purposes of document retrieval. Sprowl's approach^{17, 18} is even simpler: his system provides some limited syntactic processing, but leaves the semantics of the legal document entirely in the hands of the user. The rule-based system of Waterman and Peterson²⁴ takes an intermediate position on this issue: the "antecedent-consequent" rules are themselves quite simple in structure, but they can be assembled into "rule sets" in order to model a more complicated legal concept. On the other hand, there are several projects in this collection which build semantic structures of varying degrees of complexity directly into the representation language itself: for example, the representation of "time" in LEGOL; or the representation of "situations" in Meldman's prototype system;¹⁵ or the representation of "rights" and "obligations" in McCarty and Sridharan's TAXMAN project.¹³ It remains to be seen which of these approaches will turn out to be the most satisfactory.

The knowledge representation problem is related to another problem which faces the designer of an artificial intelligence system in a legal problem domain: how can we build up a realistic legal data base? Even assuming that the legal conceptual model has been adequately defined, there remains the task of acquiring a large number of case descriptions and classificatory rules. Although Hafner was able to code her collection of 200 cases and 200 statutory provisions by hand, an automated knowledge-acquisition system would clearly be necessary to extend this data base into the thousands, or, realistically, the tens of thousands. More difficult still is the task of modifying the legal conceptual model itself, when the legal rules change and an entirely new factor is deemed relevant to the analysis of a class of cases. One approach here is to insist on the clarity and the semantic integrity of the repre-

sentation language: thus the LEGOL project has sought to develop a high-level legally oriented language which is independent of its lower-level implementations; and Waterman and Peterson have emphasized the modularity, and hence the modifiability, of their antecedent-consequent rules. Another approach, exemplified by the work of McCarty and Sridharan on the TAXMAN project, is to study how legal rules and legal concepts have actually been modified, over time, in the course of deciding contested cases. The expectation here is that the mechanisms which provide an adequate historical account of the structure and dynamics of legal concepts will also turn out to be useful in building and modifying a contemporary legal data base.

As this survey suggests, the application of artificial intelligence techniques to legal problem domains raises a host of difficult issues. But these issues need not all be addressed at once. By selecting a particular area of the law, or a particular kind of application, we can simplify some of these issues temporarily and focus our attention on others. In this way, too, we might hope to provide some useful tools for the practicing attorney, long before the more difficult theoretical issues are fully resolved. The diversity of strategic choices in this field helps to explain some of the diversity of the projects which are described below.

- I. *A Knowledge-Based Approach to Legal Document Retrieval*, by Carole D. Hafner, University of Michigan, Ann Arbor, Michigan. Current address: General Motors Research Laboratories, Warren, Michigan.

The Legal Research System (LRS) is a knowledge-based information retrieval system, intended to be used by lawyers and legal assistants to retrieve documents based on their relevance to specific legal problems. The subject of the system's knowledge is Negotiable Instruments Law, an area of Commercial Law that deals with checks and promissory notes. The current implementation, described in Hafner,⁴ has a database of about 200 statutes from the Uniform Commercial Code²⁶ and 200 related cases.

In LRS, a semantic network is used to represent knowledge about legal concepts and relationships, enabling the system to interpret and compare descriptions of complex situations. (For other examples of semantic networks, see Findler).³ The legal knowledge encoded in the semantic network is accessed by general-purpose rules for "descriptive inference," which determine whether a document in the database "satisfies" a description entered by the user. Thus, documents are retrieved which are semantically implied by the user's query, even if none of the terms in the query appear in the document.

The semantic network model consists of six types of links, corresponding to general knowledge-structuring functions. The descriptive inference rules operate on these network structures regardless of the subject-area concepts that are encoded; thus, the system could be adapted to other legal or non-legal databases. The link types are described below:

1. *Set/member* links represent individual concepts belonging to a class; for instance, a set/member link encodes the fact that U.S.A. is a member of the class

- GOVERNMENT. If a user query asks for cases where the plaintiff is a government, those cases where the plaintiff was "U.S.A." will be retrieved by the system.
2. *Constituent* links represent the fact that one object is a constituent of another object; for example, a legal instrument such as a check has two kinds of constituents: signatures and terms.
 3. *Property* links represent attributes of objects, e.g., a signature can have the property "forged."
 4. *Subclass/superclass* links encode the hierarchical relationships among types of objects. For example, a "check" is a subclass of "draft"; thus, if a user query asks for cases involving a draft, the system will retrieve cases dealing with checks also.
 5. *Role* links are used to describe concepts that are relational in nature, such as "payee," "signer," and "purchaser." These concepts are an important part of the legal vocabulary. A "signer," for example, is described as a role whose *role filler* is a person, and whose *role object* is a legal instrument.
 6. *Event-condition* links are used to describe the things that can happen to an object, e.g., "ratified" is an event-condition of an unauthorized signature, which changes its legal meaning.

In LRS, knowledge about properties, roles, and event-conditions is combined with knowledge about superclass/subclass relationships to interpret and compare descriptions of complex situations. For example, the description "an unauthorized signature on a draft" is satisfied by the description "a forged endorsement on a check" since forged is a subclass of unauthorized, an endorsement is a role whose filler is a signature, and a check is a subclass of draft.

Keyword retrieval systems such as LEXIS²⁵ are already providing valuable aid to lawyers in searching the large and rapidly expanding database of legal rules and precedents. However, a user of LEXIS must generate many different combinations of words to describe the concepts he is interested in, in order to retrieve the relevant documents. Techniques for representing conceptual knowledge, which are being developed by workers in artificial intelligence, offer great promise for transferring some of this burden to the computer.

II. *Automating the Delivery of Instruction And the Preparation of Legal Documents*, by James A. Sprowl, American Bar Foundation, Chicago, Illinois.

Attorneys are quite used to working with books of legal forms and with published compilations of statutes. We have constructed a prototype computer system that is "programmed" by feeding into it document descriptions that resemble published legal forms and computational procedures that resemble statutes. Using these elements as a guide, the prototype system can teach, ask questions, accept answers, draw conclusions, and assemble client-customized versions of the documents ready for court filing.^{17, 18}

The language used in drafting the document descriptions and procedures is an ALGOL-68-like language in which the operators are special symbols ("+", "-", etc.) and capital-

ized words ("IF," "REPEAT," etc.). The variables are long, meaningful strings of lower-case letters, commas, apostrophes, quotation marks, periods, and spaces ("the name of the client," "the client's gross income," etc.). No declarations or data structure specifications are required since space for data storage is allocated dynamically at run time.

To allow document descriptions to resemble legal forms, "stand-alone" variables and expressions are permitted that would have no meaning in a conventional programming language. Textual comments enclosed by outward-facing brackets ("...TEXT...") are also permitted. The run-time processor evaluates the stand-alone variables and expressions and returns them, along with the intervening textual comments, as output. Undefined variables are parsed into questions. Accordingly, the following form legal document:

CONTRACT

The contractor, [the contractor's name], and the contractee, [the contractee's name], do hereby agree as follows:

The contractor agrees to buy [the number of items] [the type of items] costing \$ [the cost of each item] for a total price of \$ [the cost of each item * the number of items].

[IF implied warranties ARE to be waived INSERT]

All warranties of merchantability and suitability for a particular purpose are hereby expressly waived, and the above items are sold AS IS.

[ENDIF]

causes the computer to conduct the following interview, where the questions are automatically derived from the names of any variables that are in an "undefined" state:

What is the contractor's name?

>John Jackson

What is the contractee's name?

>Adams Supply Company

What is the number of items?

>27

What is the type of items?

>Octagonal Outlet Boxes (Catalog No. 33,274)

What is the cost of each item?

>\$1.50

Are implied warranties to be waived?

>No

Following this interview, the computer assembles and returns the following "client-customized" version of the above form contract:

Contract

The contractor, John Jackson, and the contractee, Adams Supply Company, do hereby agree as follows:

The contractor agrees to buy 27 Octagonal Outlet Boxes (Catalog No. 33,274) costing \$1.50 for a total price of \$40.50.

Computational procedures may be supplied to compute the value of any variable. For example, the procedure

IF the salesman IS on commission

THEN the cost of each item = 1.3 * the catalog price of each item
 OTHERWISE the cost of each item = the catalog price of each item

when stored in the same "library" with the above contract suppresses the question asking for "the cost of each item" and substitutes the following two questions:

Is the salesman on commission?
 >No
 What is the catalog price of each item?
 >\$1.50

If a question is poorly formulated, a "new question" document may be added to a library to serve as a replacement. For example, the questions asking for the number and price of the items may be replaced with the following "new question" documents:

How many [the type of items] are being purchased?
 How much does each [the type of items] cost?

The computer can then ask:

What is the type of items?
 >Octagonal Outlet Boxes (Catalog No. 33,274)
 How many Octagonal Outlet Boxes (Catalog No. 33,274) are being purchased?
 >27
 How much does each Octagonal Outlet Boxes (Catalog No. 33,274) cost?
 >\$1.50

A "library" may contain any number of documents, procedures, and new questions. The documents within a library may also incorporate one another by reference. A document may contain a variable defined by a first procedure; the first procedure may contain a variable defined by a second procedure; and the second procedure may contain a variable that corresponds to a new-question document that itself contains variables defined by yet other procedures. Undefined variables thus implicitly produce calls for the execution of procedures, the assembly and display of new questions, or the formulation of a question from the name of a variable. The documents, procedures, and new-question documents stored together in a library thus implicitly define a tree-structured algorithm that the computer can follow by "branching" down each time an undefined variable is encountered.

This system differs from conventional systems in permitting "stand-alone" expressions, variables, and comments to be assembled into output and in providing a "multiple-valued" calculus that permits any variable to occupy either of two undefined states. A variable may be "not yet defined," in which case the computer must search for a procedure or new question document or must convert the variable's name into a question; and if the user refuses to supply an answer to such a question, the variable enters a "never-to-be-defined" state. Logical (true-false) variables may thus enter any one of four states (not yet defined, true, false, or never to be defined),

and logical operators such as "AND" and "OR" which must operate upon these four-state variables can produce any one of sixteen possible results.

In field tests, the prototype system has proved to be an excellent vehicle for automating the assembly of wills, trusts, real estate closing agreements, divorce petitions and decrees, and other similar form legal documents.

It generates a client data file which may be used to control the assembly of other forms without a further interview, and a revised client file may control the automated assembly of a revised set of forms. The ABF language also shows considerable promise as a tool for use in creating computer-aided instructional lessons. Using the ABF system, an attorney specialist can construct a combined document-assembly and instructional system that can then be shared by large numbers of non-specialist attorneys in their individual practices.

III. *A Preliminary Study in Computer-Aided Legal Analysis*, by Jeffrey A. Meldman, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts.

This project has designed a prototype for a computer system that can perform a simple kind of legal analysis.^{15,16} The system user, who is presumed to be a lawyer, describes to the system a hypothetical set of facts. The system determines the extent to which these facts fall within certain legal doctrines (by syllogism), or near to these doctrines (by analogy). During this process, the system may ask the user for additional facts. The system then tells the user of its determinations and of the logic behind its conclusions, supporting these conclusions with reference to judicial decisions and other legal authority. The prototype system communicates with the user in a computer language (called Preliminary Study Language) designed to be translatable into and out of English by natural-language processing techniques, based on case grammar, that are currently being developed in other research.

As the basis for this analysis, structural machine models are built to represent legally-relevant human activity and doctrines of law. The primitive components in these structures represent simple *things* and *relations* (like persons, firearms, hitting, near, etc.) in the everyday world of human affairs. These things and relations are classified hierarchically into categories. They are assembled into *facts* comprising two things and the relation between them. Facts, in turn, are assembled into more complicated structures called *situations*, which are represented in terms of component *elements*, or in terms of alternative *types*, or both. These situational structures are used to represent the hypothetical facts being analyzed as well as the factual content of legal doctrines. The factual situations of specific cases provide *examples* and *counter-examples* that behave as alternative types of the situational components of more general legal doctrine. The prototype system contains representations for doctrine involving civil battery and assault.

Analysis is performed by decomposing the situations that represent legal doctrines according to their elements and their types. When this decomposition reaches the level of things and relations, these things and relations, together with their

situational structure, are matched against the things and relations contained in the hypothetical facts. The matching of individual things and relations is accomplished by reference to their hierarchical categorization.

IV. *The LEGOL Project: An Abstract*, by Sandra Cook, London School of Economics, London, England. Current address: SRI International, Menlo Park, California.

The LEGOL (*legally oriented language*) Project is developing a language, computer system, and technique of analysis to enable legislation to be expressed in a form which may be interpreted by computer. Described as such, the work can be viewed as research in legal informatics, or computers and law. But this was not its original motivation. The LEGOL Project was launched to develop improved tools for systems specification. One of the prime requisites of a very high level systems specification language is that it provides a means of stating what actions must be performed under what circumstances, without going immediately into detailed questions of implementation methods. That is, it must make it possible to say what has to be done without going into details of how to do it. As this is exactly the feature of legislation, say, statute law, the designers of LEGOL have used legislation as their experimental material. Legislation states general principles to be applied in a precise, formal and yet general way. It also provides the basis for many actual financial and administrative systems, where the provisions of the law must be translated in terms of information processing tasks to be performed by persons and/or machines.

If a computer system for administering a tax is taken as an example, then one sees that the data processing rules embodied in the programs are a logical consequence of the legislative rules which, in effect, specify what the resulting organization should achieve. Legislation for a new tax will embody, in all its essentials, the work of a department (organization) to collect the tax. The legislation will not say how the tax is to be collected, but it will specify the real world circumstances in which tax will be liable. Later, the executive branch of the government will create a department for this work, and translate the legislation into a set of procedures which can be performed by clerks and/or computers. Thus, implicit in the legislation is an information system. A formalism which is capable of expressing complex rules with the richness and precision seen in well-drafted legislation could easily cope with other application areas where information systems must be defined.² This is the characteristic of the LEGOL language, and helps to explain why we have entered the field of computers and law.

The tangible outputs of the research are specific versions of the LEGOL language (a specification language with time handling capabilities) and a computer system to interpret it. Given a blueprint for an information system specified in LEGOL, the accompanying interpreter enables a working model of the system to be generated automatically. That is, the translation from the legal prescriptions or high level rules (expressed in the LEGOL formalism) into computational algorithms is performed automatically, using the LEGOL sys-

tem. Thus, rules written in LEGOL are interpreted automatically and can be tested to discover whether they will have the desired effect. As a consequence, draft legislation may be tested by computer before it is put on the statute book. Analogously, the systems analyst can specify a data processing system in LEGOL, such as one to give effect to tax legislation. A model of this system can be run directly from this proposed specification without the intervention of programming in a conventional language. Using the simulator which the LEGOL system provides, the analyst can explore the functioning of the proposed system. Reducing the gap between definition of the system and a working prototype shortens the development life cycle dramatically.⁸

One very necessary feature of a legally oriented language is the ability to deal consistently and automatically with time, and this is a unique feature of LEGOL. The correct application of a law may require access to facts about events taking place over a long time span; past history can never be definitively deemed irrelevant. This perspective differs from that adopted in many conventional data processing applications where decisions to destroy quite recent information are commonplace, and where items such as dates are recorded and manipulated in the same way as any other pieces of data. LEGOL is based on the philosophy that information about time provides a framework which defines the validity of all other information stored in a database, and that it must be handled in a special way. Only then will a very high level systems specification have the necessary degree of generality, and decisions to "forget" the past information be seen as what they are—part of the lower level process of system design and implementation.⁶

LEGOL entity representations are based on the relational model,¹ but with additional structuring and constraints.⁷ These constraints have a basis in an "epistemological semantic model."²⁰ These built-in constraints ensure a sensible correspondence between data structures and what they represent, and make it very difficult to write expressions in the language which are not meaningful in a strict sense. This contributes toward drawing up a correct and complete specification.

The current version of the LEGOL language is capable of handling routine administrative legislation, and has been applied to a large number of legislative problems, among them, interstate succession.⁵ The Project is now looking at legislation of increased complexity and at an extension to the language (LEGOL-X) which would enable it to handle notions such as purpose, right, duty, judgment, privilege, liability, etc.²¹ This would greatly increase the scope of legislation amenable to a LEGOL analysis.

The interpreter for the current version of the language (LEGOL-2.1) is programmed in POP-2 and runs on a DEC-system 10 at the Edinburgh Regional Computing Centre. In the next phase, an interpreter for LEGOL-X will be developed on a 380Z microcomputer.

The LEGOL Project is supported by the British Science Research Council, with additional support from IBM and the British Social Science Research Council, during earlier phases. The Principal Investigator is Ronald K. Stamper. More information may be obtained by writing to the Secretary of the LEGOL Project, S110, London School of Economics, Houghton Street, London WC2A 2AE, England.

V. *The TAXMAN Project: A Summary*, by L. Thorne McCarty, Faculty of Law and Jurisprudence, State University of New York at Buffalo; and N. S. Sridharan, Department of Computer Science, Rutgers University, New Brunswick, New Jersey.

The TAXMAN project is an experiment in the application of artificial intelligence techniques to the study of legal reasoning and legal argumentation, using corporate tax law as an experimental problem domain. In our earlier work,⁹ in a system called TAXMAN I, we were able to construct computer models of the *facts* of corporate tax cases and the *concepts* of the United States Internal Revenue Code, so that the system could produce an "analysis" of the tax consequences of a given corporate transaction. Our current research is concerned with some theoretical questions which were left open in the earlier study. It is clear that the TAXMAN I system is inadequate as a model of legal reasoning and legal argumentation, since it provides no facilities for representing the "open-texture" of most legal concepts, and no facilities for modeling the "construction" and "modification" of legal concepts that occurs during the analysis of a difficult case.¹⁴ In the system we are now developing, called TAXMAN II, we are attempting to remedy these deficiencies, and attempting at the same time to develop a cognitive theory of the patterns of argument adopted by lawyers and judges in the early years of the corporate tax code.¹⁰ We are currently testing the TAXMAN II model on several stock dividend and corporate reorganization cases decided by the Supreme Court in the 1920's and 1930's, and the results so far are encouraging, although still quite incomplete.

The TAXMAN system is implemented at present in the AIMDS representation language,¹⁹ which is one of a family of frame-based languages now under development in several centers of artificial intelligence research. To set up a domain of discourse in AIMDS, we first construct a system of *templates* to describe the classes of objects in the domain, and a system of *relations* to express the possible relationships between these objects. To describe the facts of a particular case, we generate *instances* of the templates and their associated relations in a particular *context*. To represent the concepts and rules which are potentially applicable to a set of facts, we use either a semantic *description*, which is basically a network of instantiated relations in which the instance names have been replaced by variable names, or a *production*, which is a linked pair of descriptions. The descriptions and productions are then arranged into a hierarchy of *abstractions* and *expansions*, and a recursive pattern-matching procedure is used to *generate* the expansions when given the abstractions, and to *recognize* the abstractions when given the expansions. For a detailed exposition, see McCarty and Sridharan.¹³

Given these representational mechanisms, the TAXMAN system is constructed by encoding the various concepts of corporate tax law into the framework of the abstraction-expansion hierarchy. The basic "facts" of a corporate tax case can be represented in a relatively straightforward manner: corporations issue securities, transfer property, distribute dividends, etc. Below this level there is an expanded representation of the meaning of a security interest in terms of its component rights and obligations: the owners of the shares of

a common stock, for example, have certain rights to the "earnings," the "assets," and the "control" of the corporation. Above this level there is the "law": the statutory rules which classify transactions as taxable or nontaxable, ordinary income or capital gains, dividend distributions or stock redemptions, etc. We have demonstrated that the TAXMAN I system is capable of representing the full set of facts of an actual corporate tax case, such as *United States v. Phellis*, 257 U.S. 156 (1921), and capable also of representing the statutory rules and concepts which classify such cases as tax-free reorganizations under Sections 368(a)(1)(B), (C) and (D) of the Internal Revenue Code. (See, for example, McCarty⁹ and McCarty and Sridharan.¹³) Furthermore, as long as we confine our efforts to the general areas of corporate and commercial law, we believe that there are numerous practical applications for a system of this sort. We have discussed these possibilities in an earlier paper.¹¹

However, the main goal of our present research is to move beyond the limitations of the original TAXMAN paradigm, and to develop a more realistic model of the structure and dynamics of legal concepts. In the TAXMAN II system, as in the TAXMAN I system, the precise statutory rules are represented as *logical templates*, a term intended to suggest the way in which a "logical" pattern is "matched" to a lower-level factual network during the analysis of a corporate tax case. But the more amorphous concepts of corporate tax law, the concepts typically constructed and reconstructed in the process of a judicial decision, are represented in the TAXMAN II system by a *prototype* and a sequence of *deformations* of the prototype.^{10,12} The prototype is a relatively concrete description selected from the lower-level factual network itself, and the deformations are selected from among the possible *mappings* of one concrete description into another. We have argued that these "prototype-plus-deformation" structures play a crucial role in the process of legal argument, and that they contribute a degree of stability and flexibility to an emerging system of concepts that would not exist with the template structure alone. We have illustrated these ideas with a detailed analysis of *Eisner v. Macomber*, 252 U.S. 189 (1920), the early stock dividend case, and our analysis is now undergoing a full-scale implementation.¹²

The TAXMAN project has been supported by the National Science Foundation under Grant SOC-78-11408 and Grant MCS-79-21471.

VI. *Rule-Based Models of Legal Expertise*, by D. A. Waterman and Mark Peterson, The Rand Corporation, Santa Monica, California.

We are currently engaged in designing and building models of legal expertise. Some progress has already been made in developing computer systems to perform legal analysis, as described in the previous sections of this paper. Our system differs from these efforts in that it is a rule-based model of expertise, i.e., a computer program organized as a collection of antecedent-consequent rules²² that embodies the skills and knowledge of an expert in some domain. The primary goal of our work is to develop rule-based models of the decisionmaking processes of attorneys and claims adjusters involved in product liability litigation. We will use these models

to study the effect of changes in legal doctrine on settlement strategies and practices.

Our legal decisionmaking system (LDS) is being implemented in ROSIE, a rule-oriented language designed to facilitate the development of large expert systems.²³ The models created in ROSIE are rule-based, have an English-like syntax, and use special language primitives and pattern matching routines that facilitate interaction with external computer systems. The ROSIE design also supports hierarchical data structures and rulesets that can be called as subroutines, functions or predicates. The typical Rosie rule has the form IF <situation> THEN <action>, where the situation is a description of a possible data base configuration and the action describes how the data base is to be modified when that configuration is detected. The syntax of ROSIE is more English-like than that of any other programming language to date. It is intended to facilitate model creation, modification and explanation. In Section A our approach to modeling legal expertise is discussed and the operation of our prototype version of LDS is described. The conclusions are presented in Section B.

A. Legal Model. The model of legal decisionmaking we are building will contain five basic types of rules: those based on formal doctrine, informal principles, strategies, subjective considerations and secondary effects. These terms are defined below.

- **FORMAL DOCTRINE:** rules used as the basis for legal judgements such as legislation and common law.
- **INFORMAL PRINCIPLES:** rules that don't carry the weight of formal law but are generally agreed upon by legal practitioners. This includes ambiguous concepts (e.g., reasonable and proper) and generally accepted practices (e.g., pain and suffering = 3 * medical expenses).
- **STRATEGIES:** methods used by legal practitioners to accomplish a goal, e.g., proving a product defective.
- **SUBJECTIVE CONSIDERATIONS:** rules that anticipate the subjective responses of people involved in legal interactions, e.g., the effect of plaintiff attractiveness on the amount of money awarded, or the effects of extreme injuries on liability decisions.
- **SECONDARY EFFECTS:** rules that describe the interactions between rules, e.g., a change in the law from contributory negligence to comparative negligence may change other rules such as strategies for settlement or anticipated behavior of juries.

The formal doctrine evolves from court decisions and statutes, while the informal principles, strategies, etc. are shaped by example and experience. Sources for these rules include legal literature, case histories and interviews with experts. By separating the rules as described we can study both the relevant inference mechanisms and the influence of each type of knowledge on the decisionmaking process.

We are using our model of legal decisionmaking to systematically describe how legal practitioners reach settlement decisions and to test the effect of changes in the legal system on these decisions. Individual cases are analyzed by comparing the chains of reasoning (the chains of rules) that lead to the

outcomes to similar chains in prototypical cases. This helps clarify the relationships existing between the formal doctrine, informal practices and strategies used in the decisionmaking. We are examining the effects of changes in legal doctrine, procedures and strategies on the processing of cases by modifying appropriate rules in the model and noting the effect on the operation of the model when applied to a body of selected test cases. This can provide insights that will suggest useful changes in legal doctrine and practices.

Our current implementation of LDS is a small prototype model of legal decisionmaking containing rules representing negligence and liability laws. This prototype contains rules describing formal doctrine and informal principles in product liability. Future versions of the system will incorporate the other rule types discussed earlier. Given a description of a product liability case the model attempts to determine what theory of liability applies, whether or not the defendant is liable, how much the case is worth, and what an equitable value for settlement would be. Once a decision is reached the user may ask for an explanation in terms of the rules used to reach the decision.

We will now describe the use of LDS to test the effect of a legislative change on a case outcome. The case is briefly summarized as follows:

The plaintiff was cleaning a bathtub drain with a liquid cleaner when the cleaner exploded out of the drain causing severe burns and permanent scarring to his left arm. Medical expenses for the plaintiff were \$6000, and he was unable to work for 200 working days, during which time his rate of pay was \$47 per day. The cleaner was manufactured and sold by the defendant, and experts judged it not to be defective. The product's label did not give a satisfactory description of means to avoid chemical reactions. The plaintiff was familiar with the product but did not flush out the drain before using the cleaner. The amount of the claim was \$40,000.

The system was first applied using a definition of strict liability that did not involve the product being unreasonably dangerous. (See Waterman and Peterson,²⁴ a longer version of this paper that describes this definition.) It was determined that the defendant was partially liable for damages under the theory of comparative negligence, with the amount of liability lying somewhere between \$21,000 and \$29,000. The case was valued between \$35,000 and \$41,000. After the definition of strict liability was modified to state that the product must be unreasonably dangerous for strict liability to apply, the defendant was found to be not liable, since the product had been judged not unreasonably dangerous (see Figure 1).

B. Conclusions. Our preliminary work with LDS has demonstrated the feasibility of applying rule-based modeling techniques to the product liability area. In spite of the inherent complexity of product liability law, the number of basic concepts manipulated by the rules is easily handled (in the hundreds), while the number of rules required to adequately represent legal doctrine and strategies is manageable (in the thousands).

The rules that represent legal doctrine in this area are basically declarative in nature. Most of them are easily represented as definitions with complex antecedents and simple consequents that name the concept being defined. Rules of this sort can be organized as relatively unordered sets that are

EFFECT OF LAW CHANGE

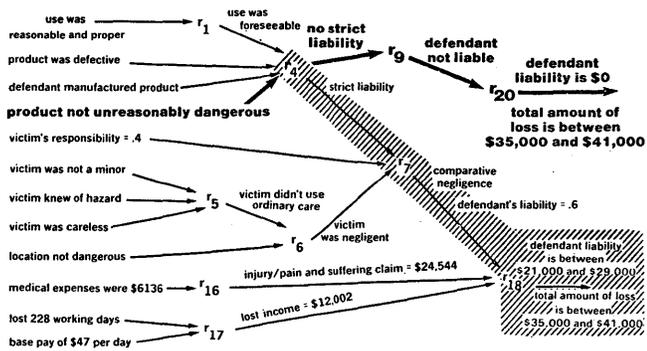


Figure 1—Inference process for drain cleaner case (crosshatched area shows inference before law change)

processed with a simple control scheme. Most of the action takes place in calls to other rule sets representing definitions of terms used by the initial set. This simple control structure facilitates rule modification and explanation.

REFERENCES

- Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," 13 *Communications of the ACM*, No. 6 (June, 1970).
- Cook, S., and Stamper, R., "LEGOL as a Tool for the Study of Bureaucracy," in H. Lucas, ed., *The Information Systems Environment* (North Holland, Amsterdam, 1980).
- Findler, N.V., ed., *Associative Networks: The Representation and Use of Knowledge in Computers* (Academic Press, New York, 1978).
- Hafner, C. D., *An Information Retrieval System Based on a Computer Model of Legal Knowledge*, Ph.D. Thesis, The University of Michigan, Ann Arbor (1978).
- Jones, S., "Control Structures in Legislation," in B. Niblett, ed., *Computer Science and Law: An Advanced Course* 157-69 (Cambridge University Press, Cambridge, 1980).
- Jones, S., and Mason, P., "Handling Time in a Data Base," in *Proceedings of the International Conference on Data Bases*, Aberdeen, Scotland (July, 1980).
- Jones, S., Mason, P., and Stamper, R., "LEGOL-2.0: A Relational Specification Language for Complex Rules," *Information Systems*, Vol. 4, No. 4 (1979).
- Mason, P., "A Systems Analysis Workbench," *Computer Bulletin*, Vol. 2, No. 23 (March, 1980).
- McCarty, L.T., "Reflections on TAXMAN: An Experiment in Artificial Intelligence and Legal Reasoning," 90 *Harvard Law Review* 837-93 (1977).
- McCarty, L.T., "The TAXMAN Project: Towards a Cognitive Theory of Legal Argument," in B. Niblett, ed., *Computer Science and Law: An Advanced Course* 23-43 (Cambridge University Press, 1980a).
- McCarty, L.T., "Some Requirements for a Computer-Based Legal Consultant," in *Proceedings of the First Annual National Conference on Artificial Intelligence* 298-300 (Stanford University, 1980b).
- McCarty, L.T., "A Computational Theory of Eisner v. Macomber," in *Proceedings of the International Study Congress on Logic, Informatics, Law* (Florence, Italy, forthcoming 1981).
- McCarty, L.T., and Sridharan, N.S., "The Representation of an Evolving System of Legal Concepts: I. Logical Templates," in *Proceedings of the Third Biennial Conference of the Canadian Society for Computational Studies of Intelligence* 304-311 (Victoria, British Columbia, 1980).
- McCarty, L.T., Sridharan, N.S., and Sangster, B.C., "The Implementation of TAXMAN II: An Experiment in Artificial Intelligence and Legal Reasoning," Report LRP-TR-2, Laboratory for Computer Science Research, Rutgers University (1979).
- Meldman, J.A., "A Preliminary Study in Computer-Aided Legal Analysis," Ph.D. Dissertation, Massachusetts Institute of Technology, Technical Report No. MAC-TR-157 (November, 1975).
- Meldman, J.A., "A Structural Model for Computer-Aided Legal Analysis," 6 *Rutgers Journal of Computers and Law* 27-71 (1977).
- Sprowl, J.A., "Automating the Legal Reasoning Process: A Computer that Uses Regulations and Statutes to Draft Legal Documents," 1979 *American Bar Foundation Research Journal* 1-81.
- Sprowl, J.A., and Staudt, R., "Computerizing the Law School Teaching Clinic: An Experiment in Law Office Automation" (forthcoming, 1981).
- Sridharan, N.S., ed., "AIMDS User Manual, Version 2," Report CBM-TR-89, Department of Computer Science, Rutgers University (1978).
- Stamper, R., "Towards a Semantic Normal Form," in G. Bracchi and G.M. Nijssen, eds., *Data Base Architecture* (North Holland, Amsterdam, 1979).
- Stamper, R., "LEGOL: Modelling Legal Rules by Computer," in B. Niblett, ed., *Computer Science and Law: An Advanced Course* 45-71 (Cambridge University Press, Cambridge, 1980).
- Waterman, D.A., "User-Oriented Systems for Capturing Expertise: A Rule-Based Approach," in D. Michie, ed., *Expert Systems in the Micro Electronic Age* (Edinburgh University Press, 1979).
- Waterman, D. A., Anderson, R. H., Hayes-Roth, F., Klahr, P., Martins, G., and Rosenschein, S. J., "Design of a Rule-Oriented System for Implementing Expertise," Report N-1158-ARPA, The Rand Corporation (Santa Monica, California, 1979).
- Waterman, D. A., and Peterson, M., "Rule-Based Models of Legal Expertise," in *Proceedings of the First Annual National Conference on Artificial Intelligence* 272-75 (Stanford University, 1980).
- LEXIS: A Primer*, Mead Data Central, Inc. (New York, 1975).
- Uniform Commercial Code*, American Law Institute and National Conference of Commissioners on Uniform State Laws (Philadelphia, 1972).

An automated reasoning system

by L. WOS and S. K. WINKER

Argonne National Laboratory
Argonne, Illinois

and

E. L. LUSK

Northern Illinois University
DeKalb, Illinois

ABSTRACT

This paper is an introduction to an automated reasoning program developed at Northern Illinois University and Argonne National Laboratory over the past nine years. Recently the program has reached the stage where it can be considered a useful research tool in a variety of disciplines. It has solved open problems in mathematics and participated in the design of new electronic circuits. Here we describe the general types of capabilities provided to the user by the program and give examples of how they are currently being used in diverse areas of investigation.

INTRODUCTION

In this paper we survey various uses of an existing automated reasoning system. Rather than describe its internal organization or theoretical foundation, which has been presented elsewhere, we concentrate here on how it appears to a user who has no need to understand the internals of the system.

The program, developed over the past nine years at Northern Illinois University and Argonne National Laboratory, has gradually grown from a tool for studying the automated reasoning process itself into a program which is being used as an intelligent colleague on research projects in several areas.

We begin by giving in the next two sections an overview of the system and those features of the control language by which the user tailors the program to the problem he wishes to study. Then we give some examples of problems successfully solved using the system. These are presented not so much for their content, which has in some cases been published elsewhere, as to demonstrate what *kinds* of reasoning tasks the program is capable of, so that the reader may determine whether or not he faces tasks that are similar enough that a program such as this one may be of genuine use.

We present examples of how the program has been used to approach problems in circuit design, program verification and

debugging, formal logic, and abstract algebra. These are intended to illustrate concrete instances of general reasoning tasks, such as conjecture formulation and testing, automatic generation of counterexamples, formula classification, case analysis, and finding alternate solutions to problems.

Finally we speculate on how the system and its relatives and descendants might be used in the future.

OVERVIEW OF THE SYSTEM

In this section we present an informal summary of some of the system's salient features as they appear to the user.

History

The system being described here was originally developed for the purpose of investigating algorithms for automated deduction. The classical test problems for such programs are known mathematical theorems, and programs like ours are often called automated theorem provers. It is the contention of this paper that such a label is far too narrow.

Known mathematical theorems provided a good starting point for development of the program. Their proofs were well understood and could be used to pinpoint methods of reasoning to be incorporated into the system. The techniques so developed, however, have become powerful enough to be of real use in attacking a wide variety of problems, both in and out of mathematics, a number of whose solutions were not known ahead of time. For example, some open problems in mathematics have been solved, and some original research in circuit design has been completed using the system. It has thus become a genuine reasoning *tool* for research in several areas, some of which are described in detail below.

A user community is just beginning to develop, consisting of people who use the program in research investigations, without knowledge of or even interest in the internal workings

of the program. The developers hope that this community will continue to expand to include an even greater number of researchers.

Representation of Information

Like most computer programs, this system has a rather formal input language. Preprocessors exist for converting a variety of other forms into the format suitable for input to the system. A logician would say that the input language has the expressive power of the first-order predicate calculus with equality. What this means to the user is that most problems can be readily represented.

Rules of Inference

There are a number of ways in which the program can be instructed to reason. They can be specified by the user either singly or in various combinations to suit the problem at hand. We give some examples.

One rule concentrates on "if-then" statements. It derives the conclusion of such a statement once all of the conditions are established.

Two other rules involve a preference for simple facts. For example, the statement that A is a larger number than B would be treated by the system as a simple item of information, whereas the statement that if A is odd and B is even then their product is even is a complex one. The first of these inference rules reasons *from* simple facts, in the sense that at each step at least one of the facts used in the derivation must be simple. The other rule reasons *toward* simple facts, in the sense that the conclusion of each inference must be simple, no matter how complex the antecedents might be.

Another inference rule, which is useful in the presence of many facts that assert the equality of certain expressions, either conditionally or unconditionally, is a generalized form of equality substitution.

Finally, the program can solve a problem by case analysis, which means that the problem can be broken into subproblems, each of which is attacked with a possibly different combination of inference rules.

Strategies of Attack

One can think of the inference rules as small-scale strategies. There are also a variety of mechanisms for specifying large-scale strategies. These guide the overall attack on the problem by specifying such things as the order in which certain facts are examined for consequences, which inference rules will be applied to which combinations of facts, and what derived information will be retained for further use or discarded as useless.

For example, most problems consist of general background information about the field being studied, together with some specific statements that describe the particular problem or situation being investigated. Sometimes it is worthwhile to reason from the background information to derive still more

general information; at other times it is best to focus entirely on the problem at hand. The user can specify that new information is to be derived only when it is at least in part based on a selected subset of the input statements.

There is much flexibility in information management. In particular, the user may define various pools of facts which can be used as he specifies. For example, facts from one pool can be selected to see if they generate any new information when used with facts from another pool, and the results placed into a third pool. The input language is used to control the use of each of these pools during the course of the run.

A third component of an attack strategy is the determination of what sorts of information to keep and what sorts to discard. The number of facts that can be deduced in a given situation is potentially quite large and can contain many irrelevant and redundant facts. Therefore some filtering mechanism is essential, both as a criterion for forgetting a fact altogether, and for remembering it but using it only as a last resort. The mechanism used in our system provides considerable flexibility in this regard. For example, one can penalize or prefer a class of objects or even statements having a very particular form. This mechanism will be discussed in detail in the next section.

EVALUATION AND TRANSFORMATION OF INFORMATION

In this section we discuss two of the main subsystems of the reasoning system. The first is a technique for specifying the value of various items of information. The second is a mechanism for transforming derived facts in various ways prior to measuring their value. These two features used together are surprisingly flexible and powerful, and have been used in many ways that were unforeseen at the time of their implementation. Some of these specific applications will be given in detail below.

Assigning Values to Terms

The language used for both input to and output from the program consists entirely of user-defined expressions that contain function and predicate symbols, variables, and constants. A flexible format exists for assigning values to various patterns of expressions. Thus each term has an integer associated with it, called its *weight*.

The simplest valuation scheme gives each symbol the same value. This makes complex expressions "heavier" than simple ones. In several different ways (discussed below), the program prefers lighter terms and facts about them to heavier ones. Thus the scheme of assigning to each symbol the same value causes the program to prefer facts about simple expressions to facts about complex ones. This is often a useful way to begin, since it corresponds to our intuition that we are on the right track in working on a problem when it seems to be getting simpler instead of more complicated.

However, the valuation scheme chosen can be used to prefer or penalize expressions according to far different measures

than symbolic complexity. For example, one could prefer facts about multiplication over those about addition, causing the product of four operands to be "lighter" than the sum of three operands. In circuit design, specific gates or combinations of gates can be preferred. In algebra problems, a standard direction of association can be specified. The weighting mechanism can arbitrarily recognize complex patterns of symbols. For example, logical expressions whose rightmost major subexpression contains no repeated variables can be preferred over those not having this property.

There are three major ways in which the program prefers light terms to heavy ones. First, expressions and facts about them that are deemed too heavy are simply discarded. One must be careful, of course, not to set the weight limit for retained facts too low, or valuable information will be lost. Second, the program uses weight to select at each step which facts to derive further information from, using light ones before heavy ones. This has the effect of causing the program to press forward along lines of inquiry that it can measure as valuable, ignoring at least temporarily facts that it considers less valuable. Thus selection of a weighting scheme supplies the program with a type of intuition about promising lines of reasoning. Third, when two expressions carry the same information, their weights are used to determine which representation of the information they contain should be used, as we describe in the next section. Thus the simplest weighting scheme causes the program to simplify formulas, while more elaborate weighting schemes trigger more subtle transformations of information.

Transforming Information

Whenever a fact of the form that one expression is equal to another is either input or derived during the run, the two expressions are weighed according to the user's weighting scheme. If one term is preferable to the other by a wide enough margin (where "wide enough" is user-supplied), then this fact assumes a special status, that of *demodulator*. What this means is that all instances of the less preferred form of the expression are immediately replaced by the more preferred form, and from that point on only the preferred version of derived facts is kept.

This dynamic rewriting process, called *demodulation*, has a surprising variety of uses. Perhaps its most common use is in simplifying expressions, but it can also be used to analyze and classify formulas, to rewrite terms in a canonical form, to transform information from one notation to another, and to cause information that has served its purpose to be deleted. Some illustrations of its use are given in the next few sections.

EXAMPLES OF CURRENT USE OF THE SYSTEM

In this section we describe some of the specific ways in which the system is currently being used. It is hoped that these cases will illustrate the general usefulness of the program and suggest to the reader how it might be applicable to his own current problems, whatever they may be.

Circuit Design

Multi-valued logics present an interesting class of circuit design problems. A particular group of researchers was recently interested in designing circuits for a four-valued logic (inputs and outputs of circuits could take on four different values) and wanted to utilize T-gates in the construction of these circuits. A T-gate in four-valued logic is a five-input, one-output gate in which one of the inputs is used to select which of the other four inputs will be used as the output value. It was known how to design such circuits, but more efficient circuits than those then known were being sought.

Our system was used to investigate this problem. It must be emphasized that no new programming was done to obtain the solution. Rather, the researchers involved learned how to describe the design process as a set of reasoning tasks expressed in the input language of the system.

First, the table of input-output conditions of the desired circuit had to be described. This was relatively straightforward. Next, this table had to be decomposed to produce the various tables that result from choosing each of the target circuit's inputs as selector input for T-gates. This task was carried out by describing the decomposition rules to the system as "if-then" statements, so that it would "infer" the decomposition tables from the original specification. The system was directed to generate and examine alternate decompositions through its mechanism for case analysis. Elementary subcircuits (such as constant or identity functions) were recognized through the use of demodulation. Finally, a weighting scheme was specified which directed the system to prefer T-gate circuits meeting certain efficiency criteria.

This project originated as an experiment in using a general-purpose reasoning system rather than a special-purpose program for carrying out real circuit design tasks. The experiment was a success in that new circuits were discovered that utilized T-gates and had efficiency properties superior to those appearing at that time in the literature.

Program Verification

Program verification is the process of proving that a program is correct by a logical argument instead of by exhaustive testing. There is currently much research in this area, and a number of experimental systems are in operation. We describe here some of the applications of our system to this area.

Our experiments have been with programs written in FORTRAN, accompanied by statements (expressed in first-order logic) about the program. We have available to us a program transformation system that converts the FORTRAN code, together with assertions bearing on how the program is expected to behave, into a collection of statements that says in our system's input language that the FORTRAN code does what the assertions say it does. The system is then asked to prove that this collection of statements is true. Alternatively, the system can find a counterexample, that is, can pinpoint the case in which the program fails to meet its specifications.

The application of our system to program verification is still in experimental stages in the sense that the complexity level of the programs that can be verified directly from the FOR-

TRAN code, without hints from the user, is very low. For example, the system can verify a FORTRAN program to find the maximum element of a vector. The conditional and unconditional transitions within the program are represented as statements of the form "If one is at statement N in the program and such and such a condition holds, then this action is taken on the program variables and control is passed to statement M." A case analysis mechanism explores the various paths through the program.

A related application of the same reasoning system is symbolic execution of programs. The system can be made to generate a collection of test data for the program of whatever kind desired and automatically step through the program with each set of data. The results can be automatically compared with the expected results supplied by the user, and exceptions noted. This too is in the early development stage. The following is a very simple example that is quite easy for both the system and for a human, but which illustrates the kind of activity which can be carried out using the system.

Consider the following algorithm for sorting the integers from 1 to N. Suppose A is the array of integers to be sorted. Then for I from 1 to N, if A(I) is not equal to I, exchange A(I) and A(A(I)). If one tries this algorithm on the vector (2,4,1,3), one might get the mistaken impression that it is correct. The reasoning system examined the algorithm, coded in FORTRAN, and correctly discovered that it does not correctly sort the vector (3,4,2,1).

Formal Logic

A logician introduced us to the field of equivalential calculus, a logical system in which the "truth" of formulas can be determined syntactically, and for which there is a single inference rule, called condensed detachment, for deriving one formula from two others. A family of interesting questions arises in considering which of the true formulas are "single axioms" in the sense that all other true formulas are derivable from them by the rule of condensed detachment.

The first task we undertook was to shorten a known proof that a particular formula implied a known single axiom and thus was itself a single axiom. Our reasoning system was used to do a number of subtasks associated with trying to find a shorter proof. First, condensed detachment had to be simulated. Since it is a special form of one of the system's built-in inference rules, this was easy. Next we studied the structure of the existing proof. One could see that terms with a specific structure were used predominantly but not exclusively in the proof. The demodulation and weighting mechanisms described above allowed the system to classify formulas and prefer those of certain classes. The control we had over various pools of facts allowed us to bring other terms into the proof search periodically. The result was a proof of twenty-one steps as opposed to the forty-four step proof appearing in the literature.

Since the system had proved so useful in approaching this problem, we were inspired to try some of the open questions in the area. There were seven formulas about which it was unknown whether or not they were single axioms. These formulas were relatively complex, and applying the rule of con-

densed detachment to them by hand was extremely difficult. (We were not at all familiar with equivalential calculus and so had to make the system into a substitute for intuition.) We selected one of the seven formulas and attempted to prove or disprove that it was a single axiom for the equivalential calculus.

What made this endeavor significant as an application of our automated reasoning system was the irregular, changing set of uses we found for it. We made various conjectures along the way and used the system to prove or disprove them. The conjectures themselves were suggested to us by the output of the system.

Examination of the formulas derived from another single axiom candidate led us first to conjecture that the formulas derivable from it would all follow a certain pattern. The system quickly proved this conjecture false.

Several iterations of this type were performed. As the patterns being investigated became more complex, demodulators were introduced to have the system itself examine its own output for these patterns. Eventually, a new notation was introduced for expressing derived formulas, and the demodulation feature of the system was used to write the formulas in this notation. More than once, a disproof of a conjecture by the system was accompanied by the exact information needed to make the next, better, conjecture. This activity eventually led to a proof that all formulas derivable from the single axiom candidate contain a certain subtle pattern. Since there are "true" formulas in the equivalential calculus that do not contain this pattern, the given formula cannot be a single axiom. Several more open problems of this nature were done, each in a slightly different way but in all cases relying heavily on the automated reasoning system for generating and testing conjectures.

Mathematics

The system has been used to solve several open problems in mathematics. We give here an example from abstract algebra. We were asked whether our system might be of use in answering the following question: Does there exist a finite semigroup which supports an antiautomorphism but no involution? This question was answered positively by the system, which exhibited a semigroup of order eighty-three with the desired property. An exhaustive search through all semigroups, starting with those of small order, would have been inordinately expensive, and the system did not do this. Rather, it examined the consequences of adding relations to a given semigroup, testing whether a given antiautomorphism remained well defined and whether any involutions could be found. There was thus substantial interaction between the researchers, who proposed the relations, and the system, which checked the effects of adding them to the semigroup being constructed. With much interaction of this kind with the system, it was eventually proved that the smallest semigroup with this property had order seven, and that there were exactly four semigroups of order seven with this property. Thus the system behaved like a mathematical colleague in eventually helping the researchers arrive at a complete answer to the question.

The model generation facility of the system was also used to solve an open question in the theory of ternary Boolean algebra. Since this capability can be used to find counterexamples to conjectures, it forms a valuable complement to the system's ability to prove theorems.

FUTURE USES OF AN AUTOMATED REASONING SYSTEM

Here we consider some of the areas in which we expect that continued research and development in automated reasoning will have a significant impact.

Program Verification

Perhaps the most significant area, and the one currently attracting the most research effort, is program verification. An ultimate goal would be to have significant programs certified correct without human intervention (except in supplying the statements that assert the purpose of the program). This is a very difficult problem, and several more modest goals are being pursued. One is to allow a user to guide the reasoning system through a proof of correctness, decomposing the task and having the system verify a series of small steps of the program. Another is to verify properties of the program that are easier to prove than its overall correctness. An example of such a property might be: "The variable I only takes on values between 1 and 100." Yet another goal short of the ultimate one is to have the system verify an abstract form of the program or a form written in some special-purpose language that is easier to work with than a production language.

On the other hand, the model generation facility of the system described here can be used to find a bug in a program when a correctness proof is eluding the user. Often a counterexample is just what is needed to pinpoint the error.

Automated Expression Generalization

One of the traditional components of the theorem-proving system is a mechanism for finding the most general common instance of two expressions. By varying the input language we have directed our system on occasion to do the opposite; that is, to find the least general expression of which two given expressions are instances. This ability to generalize could be used in the case where the expressions represented programs to find an algorithm that generalizes specific known cases. Results of such a technique could be used in code optimization and modularization.

Mathematics

The system we describe here has the potential of becoming a genuine mathematical calculator. Numerical calculators are of course now familiar, but many problems in mathematics involve extensive calculations with non-numerical objects. This is particularly true in abstract algebra and topology. A

mathematician may describe as a "calculation" any sequence of inferences that can be defined by an algorithm. Of course a special purpose program can be written to carry out such a computation, but this is time-consuming. By contrast, the allowable inferences and their sequencing can be described to a general reasoning system, and the desired calculations carried out without any new programming. It was in this way that the semigroups described above were discovered.

More generally, the system provides a way to easily examine the consequences of adding or removing facts to those defining a situation being studied. Examination of such consequences by the researcher may be of use in forming conjectures to be tested by further use of the system.

Circuit Design

Circuit design promises to be a fertile field for application of automated reasoning systems, using techniques originally developed for applications in abstract algebra. It is possible to weight various alternative constructions and thus cause the system to evaluate circuits according to very flexible user-supplied criteria as it generates a family of designs that meets a certain specification.

Process Control

One potential area in which very little has been done is the application of general reasoning systems to the control of complex physical processes. Again, the key step is to abstract the control logic to an inference system; that is, physical consequences of situations are specified by modeling them as logical consequences. Programming these systems from scratch can be very difficult, since a very complex system with many interacting components can hide the interdependencies among certain actions and results. An automated reasoning system can be used to define very clearly the actions to be taken when certain conditions arise and to apply those rules in unforeseen situations, all without any new programming.

CONCLUSION

We have tried to convey, principally by example, the usefulness of a system such as ours, in which reasoning tasks, rather than numerical computations, are automated. In order to use the system for these tasks one need not understand the internals of the system, only learn the input language. By means of this language, our general-purpose program can be tailored to a specific problem area so that it behaves much like a special purpose program. It is this ability to tune the system that has made possible the success of the program as a tool in such diverse areas.

REFERENCES

1. McCharen, J. D., R. A. Overbeek, and L. Wos. "Problems and Experiments for and with Automated Theorem-Proving Programs." *IEEE Transactions on Computers*, Vol. C-25, No. 8, August 1976, pp. 773-782.

2. McCharen, J. D., R. A. Overbeek, and L. Wos. "Complexity and Related Enhancements for Automated Theorem-Proving Programs." *Computers and Mathematics with Applications* 2 (1976), pp. 1-16.
3. Overbeek, Ross A. "An Implementation of Hyper-Resolution." *Computers and Mathematics with Applications*, Vol. 1 (1975), Pergamon Press, pp. 201-214.
4. Overbeek, R. A., and E. L. Lusk. "Data Structures and Control Architecture for Implementation of Theorem-Proving Programs." In W. Bibel and R. Kowalski (Ed.) *5th Conference on Automated Deduction*, Berlin: Springer-Verlag, 1980, pp. 232-249.
5. Winker, S. K., and L. Wos. "Automated Generation of Models and Counterexamples and its Application of Open Questions in Ternary Boolean Algebra." *Proceedings of the Eighth International Symposium on Multiple-Valued Logics*, Rosemont, Illinois, pp. 251-256.
6. Winker, S. K. "Generation and Verification of Finite Models and Counterexamples Using an Automated Theorem Prover Answering Two Open Questions." *Proceedings of the Fourth Annual Workshop on Automated Deduction*, Austin, Texas, pp. 7-13.
7. Winker, S. K., and J. Berman. "Finite Basis and Related Results for an Upper Bound Algebra" (Abstract), *Notices of the American Mathematical Society*, vol. 26 (August 1979), p. A-427.
8. Winker, S. K., L. Wos, and E. L. Lusk. "Semigroups, Antiautomorphisms, and Involutions: A Computer Solution to an Open Problem, I." (to appear)
9. Wojciechowski, W., and A. S. Wojcik. "Multiple Valued Logic Design by Theorem Proving." *Proceedings of the Ninth International Symposium on Multiple-Valued Logic*, Bath, England, 1979.
10. Boyle, J. "Program Adaptation and Program Transformation." In R. Ebert, J. Lugger, L. Goedcke (Ed.) *Practice in Software Adaptation and Maintenance*. Amsterdam: North Holland Publishing Company, 1980.
11. Manna, Z. *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

1980 NATIONAL COMPUTER CONFERENCE COMMITTEES

PROGRAM COMMITTEE

Chairman

Alex Orden
University of Chicago, Graduate
School of Business
Chicago, IL

General Advisor for Program and Proceedings

Don Medley
US Department of Agriculture,
Science and Education
Administration
Beltsville, MD

Computers and Productivity

Robert Benson
Washington University, Center for the
Study of Data Processing
St. Louis, MO

Programming Languages

Peter Buneman
University of Pennsylvania, The
Moore School
Philadelphia, PA

Database Management Software

Robert Carlson
Bell Telephone Laboratories
Warrenville, IL

Software Quality Assurance

Ned Chapin
Info Sci, Inc.
Menlo Park, CA

Software Engineering

Eric K. Clemons
University of Pennsylvania, The
Wharton School
Philadelphia, PA

Vice Chairman, Information Processing Management

Raymond Dash
Benefit Trust Life Insurance Company
Chicago, IL

Computers and Productivity

Ali Dogramaci
Rutgers University, Graduate School
of Management
Newark, NJ

Health Information Systems

Karen Duncan
Health Information Systems
Palo Alto, CA

Computational Linguistics

Martha Evens
Illinois Institute of Technology
Chicago, IL

Vice Chairman, Applications

Roger Firestone
Sperry Univac
Blue Bell, PA

Legal Applications

Haley Fromholz
Morrison & Forester
Los Angeles, CA

Computers and Productivity

John Fulkerson
Chicago Bridge & Iron Co.
Oak Brook, IL

Senses Technology Software

Adele Goldberg
Xerox Corporation, Palo Alto
Research Center
Palo Alto, CA

Vice Chairman, Applications

Michael Grigoriadis
Rutgers University
New Brunswick, NJ

Computer Science Theory

S. Louis Hakimi
Northwestern University
Evanston, IL

Information Systems Management

Gerald Hoffman
Standard Oil Company (Indiana)
Chicago, IL

Microprogramming

Samir Husson
IBM Corporation
White Plains, NY

Personnel; Planning and Managing Transition

Mary Karon
835 Ridge Avenue
Evanston, IL

Vice-Chairman, Social and Economic Implications

Robert Korfhage
Southern Methodist University
Dallas, TX

Networks and Communication

George D. Kraft
Illinois Institute of Technology
Chicago, IL

Auditability and Controls

James Krause
Pullman, Inc.
Chicago, IL

Simulation

Lance Leventhal
Emulative Systems Company
San Diego, CA

Networks and Communication

William Lidinsky
International Harvester Co.
Burr Ridge, IL

Fault-Tolerant Computing

Gerald Masson
Johns Hopkins University
Baltimore, MD

Data Processing Production Process

Russell Melton
Inland Steel Company
East Chicago, IN

Systems Development Process

Clifton Merry
Harris Trust and Savings Bank
Chicago, IL

Vice-Chairman, Software

Howard Morgan
University of Pennsylvania,
The Wharton School
Philadelphia, PA

Capacity Planning

Arnold Ockene
IBM World Trade Europe/Mid
East/Africa Corp.
White Plains, NY

*Computer Literacy; Coordinator of
Sessions Dealing with Education*

David Rine
Western Illinois University
Macomb, IL

Personnel

David Risku
Inland Steel Company
East Chicago, IN

Energy Applications

Patsy Rivera
Los Alamos National Laboratory
Los Alamos, NM

Office Automation

Janette Rose
NBI, Inc.
San Francisco, CA

Database Software Applications

Susan Rosenbaum
AT&T
New Brunswick, NJ

Personnel

Robert Rouse
Washington University, Center For
The Study of Data Processing
St. Louis, MO

*Information Systems for IS
Management, Planning, and
Managing Transition*

Robert Scheer
International Harvester Co.
Chicago, IL

Applications of Artificial Intelligence

N. S. Sridharan
Rutgers University
New Brunswick, NJ

*Strategy and Structure; Production
Process*

George Tutt
Urban Investment and Development
Company
Chicago, IL

*Vice-Chairman, Computer Architecture
and Hardware*

Anthony S. Wojcik
Illinois Institute of Technology
Chicago, IL

CONFERENCE STEERING COMMITTEE

Conference Chairman
Albert K. Hawkes
Sargent & Lundy Engineers
Chicago, IL

Program Chairman
Alex Orden
University of Chicago, Graduate
School of Business
Chicago, IL

*Vice-Chairmen, Program Committee,
Chicago Area*
Raymond Dash
Benefit Trust Life Insurance Co.
Chicago, IL

Anthony S. Wojcik
Illinois Institute of Technology
Chicago, IL

Director of Operations
Richard B. Wise
UOP, Inc.
Des Plaines, IL
(formerly ITT Research Institute,
Chicago, IL)

Pioneer Day Chairman
Carl Hammer
Sperry Univac
Washington, DC

*Steering Committee Secretary/Special
Projects Manager*
Marvin W. Ehlers
Square D Company
Palatine, IL

Professional Development Chairman
George R. Eggert
U.S. Dept. of Defense
Chicago, IL

Personal Computing Chairman
Sam Papa
Data Forms, Inc.
Chicago, IL

Special Activities Chairman
Fred H. Harris
University of Chicago
Chicago, IL

*Communications and Promotion
Chairman*
M. Mildred Wyatt
Wyatt Communications
Chicago, IL

Fiscal Officer
Charles W. Schmidt
LIFT, Inc.
Northbrook, IL

VIP Visitors Chairman
Stephen S. Yau
Northwestern University Technological
Institute
Evanston, IL

NCCC Liaison
Robert C. Spieker
AT&T
New Brunswick, NJ

AFIPS Representative
James A. Kroell
AFIPS
Arlington, VA

AFIPS Liaison
Sam Lippman
AFIPS
Arlington, VA

COMMUNICATIONS COMMITTEE

Chairman
M. Mildred Wyatt
Wyatt Communications
Chicago, IL

Vice-Chairman
Perry J. Eli
IBM Corporation
Chicago, IL

Members
Doug Blackwood
Hewlett-Packard
Ft. Collins, CO

Carl L. Blesch
Bell Telephone Laboratories, Inc.
Naperville, IL

Victor J. Danilov
Museum of Science & Industry
Chicago, IL

Donald G. Dowd
A.B. Dick Company
Niles, IL

Richard Hunter
NCR Corporation
Dayton, OH

Jerry Kalman
Informatics, Inc.
Woodland Hills, CA

Martin Kantor
Northwestern University
Evanston, IL

Nancy Moss
Illinois Institute of Technology
Chicago, IL

Gordon Smith
Memorex Corporation
Santa Clara, CA

Regional Promotions
Evelyn M. Bonney
Pacific Northwest Bell
Seattle, WA

E. Z. Million,
Consultant
Norman, OK

Kent Nichols
Control Data Corporation
Minneapolis, MN

Ex-Officio
Albert K. Hawkes, Conference
Chairman
Sargent & Lundy Engineers
Chicago, IL

Ted E. Lorber
C. Itoh Electronics, Inc.
Los Angeles, CA

Arnold P. Smith
IBM Corporation
White Plains, NY

Betty Lou Cooke
AFIPS
Arlington, VA

OPERATIONS COMMITTEE

Chairman

Richard B. Wise
UOP, Inc.
Des Plaines, IL
(formerly IIT Research Institute,
Chicago, IL)

Members

*Guest Room Assignments, Buses, and
Catering*

Jack Biddison
Capitol Construction Division of
Capitol Companies, Inc.
Arlington Heights, IL

Registration Chairman

Joe Leubitz
Checkers, Simon, and Rosner
Chicago, IL

Manager, Conference Operations

Sam Lippman
AFIPS
Arlington, VA

Room Setup, Facilities, and Supplies

Forest Mayberry
TAB Products
Chicago, IL

Signs

Marjorie McCarthy
FMC Corporate MIS
Chicago, IL

Personnel and Human Resources

Mary W. Owen
Leo Burnett Company
Chicago, IL

Operations

Mary Rich
ICS Group, Inc.
Torrance, CA

Information Booth and Message Center

Earl Calkins
Sargent & Lundy Engineers
Chicago, IL

Exhibits Coordinator

Roger Trenkle
Excellent Software Inc.
Chicago, IL

PERSONAL COMPUTING STEERING COMMITTEE

Chairman

Sam Papa
Data Forms, Inc.
Chicago, IL

Vice Chairman and Program

James Gerdes
Argonne National Laboratory
Argonne, IL

Demonstrations

Rex Burton
Thermark-Avery International
Scherverville, IN

Joel Zygmunt

U.S. Steel Supply
Chicago, IL

Publicity

Victor Humphrey
Victor Accounting Service
Homewood, IL

Operations

Edward Staros
Software Design Corporation
Lansing, IL

Digest

Joseph S. Jerbich
Humphreys Leather Goods, Inc.
Chicago, IL

Advisors

Portia Isaacson
Richard Kuzmack

PC Digest Track Captains

Robert Judd
Governors State University
Park Forest South, IL

Art Lindeman

Indiana University
Bloomington, IN

Frank Dougherty

Blackhawk Bit Burners
Belvidere, IL

PIONEER DAY COMMITTEE

Chairman

Carl Hammer
Sperry Univac
Washington, DC

Members

Donald G. Dowd
A.B. Dick Company
Chicago, IL

Nancy Stern
Hofstra University
Hempstead, NY

Henry P. Stevenson

AT&T
New Brunswick, NJ

Henry S. Tropp
135 Red Rock Way
San Francisco, CA

PROFESSIONAL DEVELOPMENT COMMITTEE

Chairman

George R. Eggert
DCASR, Chicago, U.S. Department
of Defense
Chicago, IL

Members

Shirley Baird
Milestone Systems Development
Downers Grove, IL

Rudolph E. Hirsch
Center for Continuing Education
Chicago, IL

Dr. Barbara A. Pletcher
Creative Sales Careers, Inc.
Sacramento, CA

Warren J. Simpson
U.S. Office of Personnel Management
Chicago, IL

Kathy Srednicki
U.S. Department of Defense
Chicago, IL

SPECIAL ACTIVITIES COMMITTEE

Chairman

Fred H. Harris
University of Chicago Computation
Center
Chicago, IL

Film Forum

Jeff Benchley
First National Bank of Lake Forest
Lake Forest, IL

Art Beck

Northwestern University
Evanston, IL

Rus Becker

NMDS
Des Plaines, IL

Willett Pierce

University of Illinois
Chicago, IL

Diane Ptasienski

First National Bank of Lake Forest
Lake Forest, IL

Cecil Suarez

First National Bank of Lake Forest
Lake Forest, IL

International Visitors Center

Milton D. Shulman
DePaul University
Chicago, IL

Joseph G. Arnold

Urban Investment and Development
Company
Chicago, IL

John A. Driscoll

United Equitable Insurance Company
Skokie, IL

David A. Roitman

MAS Consultants, Inc.
Chicago, IL

Handicapped Services

Marjorie M. Benson
University of Chicago Computation
Center
Chicago, IL

Barbara Herbster

Computer Corporation of America
Chicago, IL

Carole Herbster

2748 Linneman
Glenview, IL

Receptions

Joe McGrath
Pryor Corporation
Addison, IL

Eugene Karczewski

Sexton Data
Chicago, IL

John Eide

Market Facts
Chicago, IL

Fred Hoffman

Signod Corporation
Glenview, IL

Public Events

Roz Silverman
200 E. Delaware, Suite 11A
Chicago, IL

VIP VISITORS COMMITTEE

Chairman

Stephen S. Yau
Northwestern University
Technological Institute
Evanston, IL

Members

H. Lee
Northwestern University
Evanston, IL

C.V. Ramamoorthy
University of California
Berkeley, CA

NCC '81 SESSION LEADERS

Gene Altshuler
Pete, Marwick, Mitchell & Co.
New York, NY

Paul Armer
University of Minnesota
Minneapolis, MN

Marion Ball
Temple University
Philadelphia, PA

Bharat Bhargava
University of Pittsburgh
Pittsburgh, PA

Meera M. Blattner
University of California, Davis/Livermore
Livermore, California

Jeffrey A. Bloom
Network Analysis Corporation
Vienna, VA

Louis J. Brocato
U.S. Department of Agriculture
Beltsville, MD

Thomas A. Browdy
Washington University
St. Louis, MO

A. Winsor Brown
Point 4 Data Corporation
Irvine, CA

James H. Carlisle
Office of the Future, Inc.
Guttenberg, N.J.

C. R. Carlson
Bell Laboratories
Naperville, IL

William C. Carter
IBM Research Laboratory
Yorktown Heights, NY

Ned Chapin
InfoSci Inc.
Menlo Park, CA

David Clapp
U.S. Dept. of Transportation
Cambridge, MA

Danny Cohen
USC/Information Sciences Institute
Marina del Rey, CA

James C. Cooper
IBM Corporation
Gaithersburg, MD

Ira W. Cotton
Booz, Allen & Hamilton, Inc.
Bethesda, MD

J. Daniel Couger
University of Colorado
Colorado Spring, CO

Cory Devor
Honeywell
Bloomington, MN

John Donovan
Aetna Life and Casualty Co.
Hartford, CT

Karen Duncan
670 Antonio Rd., #2
Menlo Park, CA

Robert J. Elliott
Arthur Andersen & Co.
San Francisco, CA

Gerald Estrin
University of California at Los Angeles
Los Angeles, CA

Martha Evens
Illinois Institute of Technology
Chicago, IL

William E. Farley
U. S. Department of Agriculture
Beltsville, MD

Richard Federico
Occidental Petroleum—Hooker
Chemical Company
Niagara Falls, NY

Lawrence Feidelman
Management Information Corporation
Cherry Hill, NJ

Roger M. Firestone
Sperry Univac
Blue Bell, PA

Mark Fox
Carnegie-Mellon University
Pittsburgh, PA

Haley J. Fromholz
Morrison & Foerster
Los Angeles, CA

Harvey A. Freeman
Sperry Univac
St. Paul, MN

K. S. Fu
Purdue University
West Lafayette, IN

R. Stockton Gaines
Sierra Information Machines
Marina del Ray, CA

Leonard B. Gardner
Consultant
Spring Valley, CA

Frederick L. Goodman
University of Michigan
Ann Arbor, MI

Harvey J. Greenberg
Energy Information Administration
Washington, DC

Carl Hammer
Sperry Univac
Washington, DC

Thomas I. M. Ho
Purdue University
West Lafayette, IN

A. A. J. Hoffman
Consultant
Fort Worth, TX

John T. Hogan
Oak Ridge National Laboratory
Oak Ridge, TN

Gregory T. Hopkins
The MITRE Corporation
Bedford, MA

Ray Houghton
National Bureau of Standards
Washington, DC

Pei Hsia
University of Alabama in Huntsville
Huntsville, AL

S. S. Husson
IBM Corporation
White Plains, NY

Tadao Ichikawa
Hiroshima University
Hiroshima, Japan

Daniel Ingalls
Xerox Corporation
Palo Alto, CA

Suresh K. Jain
Chesebrough-Pond's, Inc.
Clinton, CT

Svetlana P. Kartashev
University of Nebraska-Lincoln
Lincoln, Nebraska

Steven I. Kartashev
Dynamic Computer Architecture, Inc.
Lincoln, Nebraska

Walter J. Karplus
University of California, Los Angeles
Los Angeles, CA

Krishna M. Kavi
University of Southwestern Louisiana
Lafayette, LA

Benn Konsynski
University of Arizona
Tucson, AZ

Eugene Kozik
Pennsylvania State University
Radnor, PA

George Kraft
Illinois Institute of Technology
Chicago, IL

James Krause
MCC Powers
Skokie, IL

Samuel C. Lee
University of Oklahoma
Norman, OK

Ellen M. Leonard
Los Alamos National Laboratory
Lost Alamos, NM

Belkis Leong-Hong
National Bureau of Standards
Washington, DC

Joseph Y-T. Leung
Northwestern University
Evanston, IL

Leon S. Levy
Bell Laboratories
Whippany, NJ

William P. Lidinsky
International Harvester
Burr Ridge, IL

Leonard D. Lipner
BGS Systems, Inc.
Waltham, MA

Peter Lykos
Illinois Institute of Technology
Chicago, IL

Gerald M. Masson
Johns Hopkins University
Baltimore, MD

L. Thorne McCarty
State University of New York at Buffalo
Buffalo, NY

Carma L. McClure
Northwestern University
Evanston, IL

Robert J. McGlenn
Southern Illinois University
Carbondale, IL

Arthur Melmed
U. S. Department of Education
Washington, DC

W. Russell Melton
Inland Steel Company
Chicago, IL

Clifton Merry
Harris Trust & Savings Bank
Chicago, IL

Diana Merry
Xerox Corporation
Palo Alto, CA

Leslie Jill Miller
Xerox Corporation
Rochester, NY

Thomas Mitchell
Rutgers University
New Brunswick, NJ

David J. Mishelevich
University of Texas
Dallas, TX

Howard Lee Morgan
The Wharton School
Philadelphia, PA

Abbe Mowshowitz
Croton Research Group, Inc.
Croton-on-Hudson, NY

Susan H. Nycum
Gaston Snow & Ely Bartlett
Palo Alto, CA

K. S. Padda
Texas Instruments
Houston, TX

Lee Papayanopoulos
Rutgers University
Newark, NJ

Fred E. Petry
Tulane University
New Orleans, LA

Susan Rosenbaum
AT&T
New Brunswick, NJ

Steven J. Ross
The Plagman Group
New York, NY

Robert A. Rouse
Washington University
St. Louis, MO

John Van Savage
Army Armament R&D Command
Edison, NJ

Robert H. Scheer
International Harvester Company
Chicago, IL

Norman S. Schneidewind
Naval Postgraduate School
Monterey, CA

Brad Schultz
Computerworld
Framingham, MA

Robert Seidel
Human Resources Research Organization
Alexandria, VA

Bruce D. Shriver
University of Southwestern Louisiana
Lafayette, LA

Thomas Sinopoli
Algorithmics, Inc.
Wellesley, MA

Norman K. Sondheimer
Sperry Univac
Blue Bell, PA

N. S. Sridharan
Rutgers University
New Brunswick, NJ

James R. Swager
Honeywell Information Systems
McLean, VA

Keith A. Taggart
Los Alamos National Laboratory
Los Alamos, NM

Linda Taylor
System Development Corporation
Santa Monica, CA

Michael Tempel
New York Academy of Sciences
New York, NY

Satish K. Tripathi
University of Maryland
College Park, MD

George L. Tutt
Urban Investment and Development Company
Chicago, IL

Harold S. Uhrbach
DBD Systems, Inc.
Rockville Centre, NY

Walter Ulrich
Walter E. Ulrich Consulting
Houston, TX

K. Vairavan
University of Wisconsin-Milwaukee
Milwaukee, WI

Denny O. Wallace
Illinois Tool Works, Inc.
Chicago, IL

Conrad H. Weisert
Information Disciplines, Inc.
Chicago, IL

Stephen S. Yau
Northwestern University
Evanston, IL

Raymond T. Yeh
University of Maryland
College Park, MD

Dan Zatyko
Zatyko Associates
Santa Ana, CA

Ken Zoline
Continental Bank
Chicago, IL

Mitch L. Zolliker
IBM Corporation
San Jose, CA

NCC '81 REFEREES

Adams, R. E.
 Agrawal, Dharma
 Ahuja, Pratap
 Ahuja, Sanjiv
 Al-Fedaghi, Sabah S.
 Amenta, Joyce
 Ames, Stanley R., Jr.
 Andrisan, John V.
 Antal, J. R.
 Archibald, J. A., Jr.
 Aylor, James H.

Baer, Jean-Loup
 Bail, William
 Baird, George N.
 Baker, F. Terry
 Barnes, Bruce H.
 Bartlett, Frederick
 Bateman, Barry
 Bauer, M. A.
 Baxter, Brent
 Belford, Geneva
 Bering, Doug
 Bhargava, Bharat
 Bise, Robert G.
 Blomgren, George H.
 Bork, Alfred
 Borko, Harold
 Bracey, Randolph D.
 Brocato, Louis J.
 Brown, Nander
 Brown, Russell K.
 Burton, William D., Jr.

Campbell, R. H.
 Cannon, George R., Jr.
 Capraro, Gerard T.
 Carey, Bernard J.
 Carroll, B. D.
 Carter, William J.
 Chapin, Ned
 Charney, R. B.
 Charp, S.
 Chiang, Paul
 Chow, Yuan-Chieh
 Christopher, Thomas
 Cieslowski, Richard J.
 Clema, J. K.
 Clemons, Eric K.
 Cobb, Gary W.
 Coopridger, Lee W.
 Cowan, George
 Crenshaw, Edsel G.

Daniels, Walter E.
 Danner, Lee
 Davis, Alan

Day, William H. E.
 De Jong, Kenneth
 DeKock, Arlan R.
 Denenberg, Stewart A.
 Dixon, Louis F.
 Donato, Nola
 Dutton, Ron
 Dwyer, Samuel J., III

Eastman, Caroline M.
 Eccles, William J.
 Eger, John
 Ein-Gal, Moshe
 Ernst, Ronald L.
 Evens, Martha

Feldman, Michael B.
 Finfer, Marcia
 Fishbeck, Ronald
 Flinchbaugh, B. E.
 Flynn, Robert J.
 Fong, Elizabeth
 Friedman, Lee A.
 Fu, K. S.

Galkowski, J. T.
 Gannon, T. F.
 Gehani, Narain
 Gips, James
 Goel, Amrit L.
 Goldman, Neil
 Gonzalez, Mario J., Jr.
 Gottlieb, Allan
 Granlund, Goesta
 Green, Teresa O.
 Grosch, Audrey N.
 Gross, Arthur G.
 Gupta, Ram K.

Hakozaki, Katsuya
 Hall, Ernest L.
 Hamblen, John W.
 Hanna, William E., Jr.
 Hart, Peter E.
 Hecht, H.
 Hopper, Grace M.
 Hua, Cecil T.
 Hurley, Paul

Ichikawa, Tadao

Jensen, E. Douglas
 Jette, Christina L.
 Johnson, L. Arnold
 Johnson, Mark Scott
 Jordan, Harry F.

Kahn, Kevin C.
 Kaufman, Arie E.
 Koory, Jerry L.
 Kornfield, N. R.
 Kronman, J. A.
 Krulee, Gilbert
 Kubitz, W. J.
 Kulaga, Joseph

Landis, Carolyn P.
 Lee, Mary Jane
 Lee, Theodore M. P.
 Lennon, William
 Lien, David A.
 Lint, B. T.
 Little, Joyce Currie
 Lockett, JoAnn
 Long, Harvey S.

Machover, Carl
 Madrigal, Orlando S.
 Maekawa, Mamoru
 Magel, Kenneth
 Magnuson, Waldo G.
 Maher, Austin J.
 Mallett, Patrick
 Mander, K. C.
 Maniotes, John
 Matshushita, Yutaka
 McAllister, David F.
 McCrea, Donald R.
 McDonald, Nancy H.
 McMahan, Edith M.
 Metzner, John R.
 Miller, Charles E.
 Miller, Mark Leslie
 Modesitt, Kenneth L.
 Mounce, John
 Muka, Stephen
 Murphy, Robert E.

Naqvi, S. A.
 Navlakha, Jainendra
 Nelson, Victor P.
 Nestman, Chadwick H.
 Nielsen, Norman R.
 Nilsson, Arne A.
 Nutt, Gary J.

O'Kane, Kevin C.
 O'Neal, Beverly

Peralta, L. A.
 Perry, J. M.
 Pfaltz, John
 Pottinger, Hardy J.

Powell, John E.
Prewitt, Judith

Reho, Andy
Riddle, William E.
Rocchetti, Robert
Rosenbaum, Susan L.
Rosin, Bob
Roth, R. Waldo
Rulifson, J. F.
Ruschitzka, Manfred

Sankar, P. V.
Savas, E. S.
Scheuermann, Peter
Schneider, G. Michael
Schultz, David J.
Segal, Ronald
Shapiro, Michael D.
Shelter, Toni

Simmons, Dick B.
Sitkin, Irwin J.
Smith, Eugene B.
Smith, Raoul
Smoot, Oliver R.
Sondheimer, Norman K.
Spaniol, Roland
Stavely, Allan M.
Stevens, D. F.
Stevens, W. Richard
Stuck, B. W.
Sunshine, Carl
Svigals, J.
Szolovits, Peter

Tai, K. C.
Tausner, Miriam R.
Taylor, Linda T.
Taylor, Robert W.
Teng, Albert

Thurber, Ken
Tinaztepe, Cihan
Tomaru, Keisuke
Tucker, Edwin K.
Turn, Rein

Van Name, Mark L.

Waterman, David J.
Weiss, Stephen F.
Wesselkamper, T. C.
Whiting-O'Keefe, Patricia M.
Wolfson, Seymour J.
Worrest, Ralph W.
Wynne, A. James

Yamamoto, Masahiro
Yasnoff, William A.
Yen, W. C.

NCC '81 SPEAKERS AND PANELISTS

Adam, Robert
 Agrawala, A. K.
 Aiken, Robert M.
 Alexander, Roger
 Alguire, Robert
 Aiso, Hideo
 Anastasi, Larry
 Anderson, Ronald
 Andrews, Dorothy M.
 Arora, Adarsh K.
 Aspray, William F.
 Athey, Thomas H.
 Avizienis, Algirdas
 Atkins, Cal
 Aylor, J. H.

Ballard, Stony
 Barrett, Ernie
 Bass, Charlie
 Bates, Madeline
 Bebel, Don
 Belady, Les A.
 Belknap, John H.
 Benenati, J. David
 Berger, Carl
 Berry, R.
 Bickel, Rudolf G.
 Bigelow, Robert
 Blackman, Jim
 Blazie, Deane
 Bolek, Raymond W.
 Brackett, C. A.
 Bright, Herbert S.
 Bronner, Lee Roy
 Browdy, T. A.
 Brown, Harold
 Brown, John Seely
 Brown, Mary Lou
 Brown, Tom
 Bucy, Richard
 Bulen, Robert A.
 Bush, James Wilson
 Buzen, J. P.

Campbell-Kelly, Martin
 Carlson, Eric
 Carlucci, Carl P.
 Carmony, Lowell A.
 Carter, William C.
 Cerf, Vinton G.
 Ceruzzi, Paul
 Chandy, K. M.
 Chang, N. S.
 Chang, S. K.
 Chereb, David
 Chien, Y. T.

Cichelli, Richard
 Cobb, Thomas
 Clary, James
 Clauss, Karl
 Clement, Andrew
 Cohen, Danny
 Cohen, Leo
 Colestock, H.
 Collins, Alan
 Cook, Sandra
 Cougar, J. Daniel
 Cragon, Harvey G.
 Crowley, Charles
 Curtiss, Philip F.

Dasgupta, Subrata
 Daverio, Paul
 Davis, Randy
 Day, John
 De Jong, Peter
 DeLine, James R.
 Denning, Peter J.
 Dowdy, L. W.
 Dreifus, Henry
 Duke, Chris

Earnest, E. Dean
 Edwards, Paul L.
 Eichberger, Joe

Fisher, David
 Fuire, Robert
 Fletcher, John G.
 Fox, Mark S.
 Franta, William R.
 Frase, Lawrence T.
 Freiwald, Joyce
 Friend, David
 Froese, Robert
 Fronk, William
 Fu, K. S.
 Fulmer, Ken

Gajnak, George
 Galitz, W. O.
 Gehani, Narain
 Gemignani, Michael
 Gewirtz, Bill
 Giammo, Tom
 Giannini, Margaret
 Gilmer, George H.
 Gnanamgari, Sakunthala
 Gold, Charles L.
 Goldberg, Jack
 Goodman, Frederick L.
 Graham, James W.

Graham, Robert L.
 Greenberg, Harvey J.
 Groves, Stan

Hadcock, Walter J.
 Hafner, Carole D.
 Hancock, Jack
 Handy, Jim
 Hanson, Robert C.
 Haralick, R. M.
 Harris, Daniel K.
 Harris, Fred H.
 Harris, Larry R.
 Harris, Richard
 Hart, Peter E.
 Harvey, Samuel B.
 Hayn, John
 Held, Gerald
 Heller, Andrew R.
 Hellprin, Lawrence B.
 Herman, Alvin J.
 Hess, Matt
 Hicks, H. Richard
 Higgins, Ruth
 Holland, L. Donald
 Hollingsworth, Marion
 Hopkins, Albert
 Hopkins, Gregory T.
 Howe, Robert
 Hua, Cecil
 Huckaby, Don
 Hughes, Charles E.
 Hughes, Herman D.
 Hunter, Beverly

Ingalls, Daniel

Jackson, Annette
 Jain, Suresh K.
 Johnson, Anthony W.

Kaehler, Ted
 Kahn, Robert E.
 Kehler, Tom
 Kim, K. H.
 Kirkley, John L.
 Klassen, Daniel
 Kling, Rob
 Kolstad, Charles
 Krasner, Glenn
 Kurator, William

LaRue, Richard
 Lashof, Joyce
 Lee, J. A. N.
 Lefkovits, Henry C.

Leneway, Robert J.
Leonard, Carl A.
Leonard, Ellen
Leliotis, Ted
Lenat, Douglas
Leopold, Harry
Levinthal, Cyrus
Levy, Leon S.
Linebarger, Robert
Lipner, L.
Lynch, Mary Jo

Major, Joseph B.
Marcus, Aaron
Matthews, Gordon
MacDonald, Nina
McClure, Carma L.
Mallie, Tony
Mathews, Gordon
Meldman, Jeffrey A.
Merwin, Richard E.
Miller, Mark
Millman, Ann Miller
Mills, Harlan
Mischevich, David J.
Misra, J.
Munson, Jack
Montgomery, Christine A.
Musselman, Francis H.

Nakamoto, Robert
Neighborgall, Roger
Newkirk, M. Glenn
Nierstrasz, Oscar M.
Nolte, Sid
Nunamaker, Jay F., Jr.
Nycum, Susan H.
Nye, J. Michael

Ogorchock, James
Orr, Joel N.
Owens, Donald P.

Parikh, Girish
Parr, M. R.
Pavlidis, T.
Peercy, David E.
Perry, Seymour
Peterson, Mark
Pollock, Kenneth G.
Potter, David
Post, Douglas E.
Poston, Bob

Prewitt, J. M. S.
Price, Camille
Prince, Warren
Printis, Robert
Probst, Jerry
Provan, Scott

Rabenhorst, James F.
Rahimi, M. A.
Ramamoorthy, C. V.
Ramellini, Joseph
Rather, Elizabeth
Rattner, Justine
Ravenel, Bruce
Reenskaug, Trygve
Reggio, Patrick
Roberts, Stephen
Rolland, Ron
Robson, David
Robinson, Lee
Rose, Alan
Rosenbaum, Richard
Rosenfeld, A.
Rosenthal, Gerald
Rotolo, Elio
Ryburg, Jon

Saal, Harry J.
Sayani, Hasan
Schneyman, A. H.
Schultz, Brad
Seidel, Robert
Seigle, Dave
Selinger, Patricia
Shaffer, Hy
Shapiro, Linda
Sharpe, Richard
Sheil, B. A.
Sheppard, Sallie
Shoch, John
Shriver, Bruce D.
Shumway, Dick
Slavitz, Jeffrey M.
Sleeman, Derek
Smith, John
Smith, Robert Ellis
Smith, Wayne
Snow, Andrew P.
Solowa, Elliot
Sorenson, Paul
Spaniol, Roland
Sprowl, James A.
Sridharan, N. S.

Standjev, Robert
Steels, Luc L.
Stefink, Mark
Steiner, Carl R.
Stodolsky, David
Sussman, Gerald
Sutherland, Duke

Teichroew, Daniel
Tempel, Michael
Tesler, Larry
Thomsen, Carl
Thompson, Claudia R.
Thurber, Kenneth J.
Tillinghast, James
Toellner, John
Toyama, Masaharu
Tripathi, S. K.
Tunis, James A.

Van Slyke, Richard
Van Tilborg, Andre M.
Veazie, Stephen M.
Vick, Charles R.

Walker, Donald E.
Wardle, Caroline
Waterman, D. A.
Watteeuw, Caroline
Weber, Larry
Wecksung, Mona
Wensley, John
Wetherbe, James C.
Whitten, Jeffrey L.
Wigington, Ronald
Wilk, Evelyn S.
Williams, A. C.
Williams, Martha
Winkler, Connie
Wittie, Larry D.
Wohl, Amy
Wong, Harry
Wood, David C.
Wu, Chialin

Yara, Ron
Young, Charles R.
Young, Robert F.

Ziegler, R. W., Jr.
Zikas, Algirdas J.
Zilles, Stephen

AUTHOR INDEX

- Alagar, Vangalur S., 443
Anderson, Roy E., 401
Avizienis, Algirdas, 27
- Baird, George N., 361
Balzer, Robert, 393
Barrett, Eamon, 453
Batchelor, William L., 389
Benson, Robert J., 593
Berg, Helmut K., 75
Bernstein, Philip A., 487
Bhargava, Bharat, 297, 543
Blanchard, Bernard, 443
Blankenship, P. E., 183
Blattner, Meera, 453
Boss, Richard W., 609
Briggs, Fayé A., 191
Brooks, Ruven, 453, 469
- Campbell, R., 209
Campbell, R. H., 231
Center, John W., 323
Chapin, Ned, 349
Cheng, W. Y., 209
Chow, Yuan-Chieh, 163
Chu, Wesley W., 137
Clemons, Eric K., 249
Cohen, Danny, 169
Constantinides, J., 203
Cook, Sandra, 689
Cordy, James R., 259
Costello, S. H., 217
Crickman, Robin, 613
Crocker, Stephen D., 19
Crowe, David R., 257
Crowley, Charles, 265
- Danielson, Ronald L., 375
Davidson, Scott, 81
Dayal, Umeshwar, 487
Decitre, Paul, 473
Dees, W. A., 11
De Figueiredo, A. C. D., 141
Doi, Norihisa, 407
Donato, Nola, 665
- Endicott, Lucian J., Jr., 389
- Feldman, J. A., 183
Fisher, Joseph A., 95
Fishman, P., 459
- Gallaher, L. E., 41
Gammill, Robert C., 415
Giloi, Wolfgang K., 49
Glaser, David, 443
Goldman, Neil, 393
- Goodell, Ross, 103
Goodman, Frederick L., 601
Goodman, Nathan, 487
Goyal, A., 11
Greene, Richard J., 481
Griggs, Ian H., 257
Gueth, Reinhold, 49
- Hafner, Carole D., 689
Hanna, W. L., 431
Hasegawa, Kiyoshi, 507
Heidorn, George E., 649
Heller, Andrew, 69
Hellerstein, Joseph, 137
Hendler, James, 643
Hewitt, Jay, 663
Hikita, Sadayuki, 507
Hirata, Masahiro, 407
Hirose, Ken, 407
Hobson, Richard F., 3
Hodson, Patrick, 677
Holt, Richard C., 257
Holthouse, M. A., 353
Howden, William E., 367
Hwang, Kai, 191
- Ingargiola, Giorgio, 383
- Jensen, Karen, 649
Johnson, David W., 235
Johnson, L. Arnold, 361
- Kambayashi, Yahiko, 555
Kartashev, Steven I., 111
Kartashev, Svetlana P., 111
Kearney, Joe, 317
Kehler, Thomas P., 643
Kerr, Edwin F., 597
Kini, Vittal, 19
Kolstad, R., 209
Koubias, S., 203
- Landers, Terry, 487
Landskov, David, 95
Lawson, Harold W., Jr., 57
Leavenworth, Burt, 537
Leventis, S., 203
Levinson, E., 241
Levy, L. S., 241
Lilien, Leszek, 543
Lin, Ken W. T., 487
Liu, J. W.-S., 209
Luhukay, J., 209
Lusk, E. L., 697
Lybrook, C. W., 353
Lyons, Michael J., 337
- Ma, Perng-Yi, 87
Ma, Y. W., 149
Maekawa, Mamoru, 515
Markel, J. D., 177
Matsushita, Yutaka, 507
McCarty, L. Thorne, 689
Meldman, Jeffrey A., 689
Meyer, Michael E., 225
Michaelis, Paul Roller, 643
Miller, Lance A., 649
Mishelevich, David J., 631
Miyamoto, Isao, 571
Mooney, James D., 145
Morse, Jane G., 565
- Nakatsu, Narao, 555
Nicolas, Georges S., 529
- O'Leary, G. C., 183
Oliver, Dennis M., 593
Osterweil, Leon J., 367
Othmer, E., 459
Othmer, S., 459
Overman, William T., 19
- Papadopoulos, G., 203
Papayanopoulos, L., 623
Parker, Alice C., 63
Parmar, K. M., 11
Patterson, Dave, 103
Pawlak, Zdzislaw, 453
Peterson, Mark, 689
Phillips, Brian, 643
Poe, Michael D., 103
Pramanik, Sakti, 521
Price, Camille C., 291
Protopapas, Dimitris A., 423
- Quatember, Bernhard, 125
- Rao, J. R., 431
Rathi, B. D., 11
Ray, S., 209
Relles, Nathan, 383
Reutter, John, III, 343
Richards, P. G., 231
Ricketts, S. R., 681
Rocchetti, Robert, 671
Roistacher, Richard C., 617
Rolland, Colette, 583
Ross, Kenneth M., 643
Rouse, Robert A., 593
- Saito, Nobuo, 407
Salisbury, J. B., 241
Scallan, P. Gerard, 249
Schneider, G. Michael, 317

Schwartz, Helen J., 605
Sedlmeyer, Robert L., 317
Segawa, Kiyoshi, 407
Shanthikumar, J. G., 311
Sheheen, Nancy, 677
Shriver, Bruce D., 49, 81, 95
Sibley, Edgar H., 249
Simmons, Dick B., 329
Smith, John Miles, 487
Smith, Mark K., 367
Smith, R. J., II, 11
Sondheimer, Norman K., 383
Sprowl, James A., 689
Sridharan, N. S., 689
Steely, Simon C., Jr., 103
Swager, James R., 501

Takata, Masayuki, 407
Taylor, Richard N., 367

Tennant, Harry R., 643
Thompson, Bozena Henisz, 657
Thompson, Fred, 657
Tierney, J., 183
Toy, W. N., 41
Tripp, Leonard L., 367
Tsui, R. Y., 11
Turba, T. N., 217

Van Dam, Andries, 69
Vannier, M. W., 459
Van Tilborg, Andre, 283

Wah, Benjamin W., 149, 191
Ward, Darrell L., 463
Waterman, D. A., 689
Whitney, V. Kevin, 565
Wilner, Wayne T., 63
Winfrey, W. R., 681

Winker, S. K., 697
Winslow, Leon E., 163
Wittie, Larry D., 283
Wong, Eugene, 487
Wortman, David B., 257
Wos, L., 697

Yajima, Shuzo, 555
Yamasaki, Toshiharu, 407
Yamazaki, Haruaki, 507
Yeh, Raymond T., 571
Young, Charles R., 273

Zolnowski, Jean Cochrane, 329

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES, INC. (AFIPS)

OFFICERS

President

J. Ralph Leatherman
Hughes Tool Company
Houston, TX

Vice President

Sylvia Charp
The School District of Philadelphia
Philadelphia, PA

Treasurer

M. Stuart Lynn
University of California
Berkeley, CA

Secretary

Arthur C. Lumb
Procter & Gamble Co.
Cincinnati, OH

Executive Director

Paul J. Raisig
AFIPS
Arlington, VA

BOARD OF DIRECTORS

Association for Educational Data Systems (AEDS)

Judith B. Edwards
Northwest Regional Educational Lab
Portland, OR

AFIPS Immediate Past President

Albert S. Hoagland
IBM Corporation
San Jose, CA

American Society for Information Science (ASIS)

James M. Cretsos
Merell National Labs
Cincinnati, OH

American Statistical Association (ASA)

George Minich
World Bank
Washington, DC

Association for Computational Linguistics (ACL)

Donald E. Walker
SRI International
Menlo Park, CA

Association for Computing Machinery (ACM)

Aaron Finerman
University of Michigan
Ann Arbor, MI

Raymond E. Miller
Georgia Institute of Technology
Atlanta, GA

Peter J. Denning
Purdue University
West Lafayette, IN

Data Processing Management Association (DPMA)

Robert Marrigan
Mail Communications, Inc.
Everett, MA

George Eggert
Chicago DCASR
Department of Defense
Chicago, IL

Robert A. Finke
Cummins Engine Co.
Columbus, IN

IEEE—Computer Society

Richard Merwin
The George Washington University
Washington, DC

Steven S. Yau
Northwestern University
Evanston, IL

Roland B. Arndt
Sperry Univac
St. Paul, MN

Instrument Society of America (ISA)

Chun H. Cho
Fisher Controls Co.
West Marshalltown, IA

Society for Computer Simulation (SCS)

Per Holst
The Foxboro Company
Foxboro, MA

Society for Industrial and Applied Mathematics (SIAM)

Donald L. Thomsen, Jr.
SIAM Institute for Mathematics and
Society
New Canaan, CT

Society for Information Display (SID)

Carlo P. Crocetti
Rome Air Development Center/XP
New York, NY

AFIPS EXECUTIVE COMMITTEE

J. Ralph Leatherman
Hughes Tool Company
Houston, TX

Sylvia Charp
The School District of Philadelphia
Philadelphia, PA

M. Stuart Lynn
University of California
Berkeley, CA

Arthur C. Lumb
Procter & Gamble Co.
Cincinnati, OH

Aaron Finerman
University of Michigan
Ann Arbor, MI

Steven S. Yau
Northwestern University
Evanston, IL

Per Holst
The Foxboro Company
Foxboro, MA

Robert Marrigan
Mail Communications, Inc.
Everett, MA

NATIONAL COMPUTER CONFERENCE BOARD MEMBERS

Chairman and AFIPS Representative

Sylvia Charp
The School District of Philadelphia
Philadelphia, PA

Vice Chairman and SCS Representative

Carl Malstrom
North Carolina State University
Raleigh, NC

Secretary and AFIPS Representative

George Minich
World Bank
Washington, DC

Treasurer and AFIPS Representative

M. Stuart Lynn
University of California
Berkeley, CA

AFIPS President

J. Ralph Leatherman
Hughes Tool Company
Houston, TX

*Chairman of the NCC Committee—
Ex Officio*

Irwin Sitkin
Aetna Life and Casualty
Hartford, CT

*IEEE—Computer Society President—
Ex Officio*

Richard Merwin
The George Washington University
Washington, DC

ACM President—Ex Officio

Peter J. Denning
Purdue University
West Lafayette, IN

*Chairman of the Industry Advisory
Panel—Ex Officio*

Dallas Talley
Qantel Corporation
Hayward, CA

DPMA Representative

George Eggert
DCASR—Chicago, U.S. Department
of Defense
Chicago, IL

*IEEE—Computer Society
Representative*

Dick B. Simmons
Texas A & M University
College Station, TX

ACM Representative

Seymour Wolfson
Wayne State University
Detroit, MI

SCS President—Ex Officio

Stewart I. Schlesinger
The Aerospace Corporation
Los Angeles, CA

DPMA President—Ex Officio

Roger Fenwick
New York Telephone Company
New York, NY

NATIONAL COMPUTER CONFERENCE COMMITTEE OF THE NCC BOARD

Chairman

Irwin J. Sitkin
Aetna Life & Casualty
Hartford, CT

Secretary

Floyd Harris
Life of Georgia
Atlanta, GA

NCC '81 Chairman

Al Hawkes
Sargent & Lundy Engineers
Chicago, IL

NCC '82 Chairman

Russell K. Brown
Moore Paper Co.
Houston, TX

Morton M. Astrahan
IBM Research Laboratory
San Jose, CA

Smith Dorsey
1018 N. Greenhaven Ave.
Fullerton, CA

Harvey L. Garner
Moore School of Electrical
Engineering
University of Pennsylvania
Philadelphia, PA

Jerry Koory
Rand Corporation
Santa Monica, CA

William Sitter
Tenneco, Inc.
Houston, TX

Arnold P. Smith
IBM Corporation
White Plains, NY

Robert C. Spieker
AT&T
New Brunswick, NJ

OAC '82 Chairman

Hans Pueshe
Fireman's Fund Insurance Co.
San Rafael, CA

NATIONAL COMPUTER CONFERENCE BOARD INDUSTRY ADVISORY PANEL

Chairman

Dallas Talley
Qantel Corp.
Hayward, CA

Members

Frederick M. Hoar
Apple Corporation
Cupertino, CA

S.A. (Sandy) Lanzarotta
Xerox Corporation
El Segundo, CA

William Lonergan
Xerox Development Corporation
Beverly Hills, CA

Richard Mau
Sperry Rand Corporation
New York, NY

Herbert Richman
Data General Corporation
Westboro, MA

Gordon Smith
Memorex Corporation
Santa Clara, CA

AFIPS HEADQUARTERS AND CONFERENCE SUPPORT STAFF

Executive Director
Paul J. Raisig

Executive Assistant
Jane Smith

Secretary/Receptionist
Terry DiMurro

Controller
Janis Miller

Accountant
Melinda Yost

Bookkeeper
William Hargrave

Administrative Coordinator
Ken Fields

*Public Information
Coordinator*
Nancy LeFebvre

Secretary
Cathy Chaney

AFIPS Press Manager
Christopher N. Hoelzel

Fulfillment Administrator
Olive Shilland

Secretary
Sharon Lee Conway

NCC Copy Editor/Proofreader
George Jansen, Jr.

Director, Washington Office
Alexander D. Roth

Research Associate
Ellen Law

Administrative Assistant
Lorraine Cummings

Secretary
Patricia Mayo

Conferences Director
James H. Kroell

Administrative Assistant
Betty Foley

Manager, Conference Operations
Sam Lippman

Operations Coordinator
Lisa Welke

Secretary, Conference Operations
Pantipa Dhanagom

Manager, Exhibit Operations & Sales
Larry Jennings

Exhibit Sales Coordinator
Luellen Hoffman

Secretary, Exhibit Sales
Jill Newman

Marketing Manager
Betty Lou Cooke

Marketing Coordinator
Diana Snow

Secretary, Marketing
Joyce Paige Davis