Stanford Artificial Intelligence Laboratory
Memo AIM-301

June 1977

Computer Science Department
Report No. STAN-CS-77-624

Recent Research in Computer Science

by

John McCarthy, Professor of Computer Science
Principal Investigator

Associate Investigators:
Thomas Binford, Research Associate in Computer Science
Cordell Green, Assistant Professor of Computer Science
David Luckham, Senior Research Associate in Computer Science
Zohar Manna, Research Associate in Computer Science
Terry Winograd, Assistant Professor of Computer Science

Edited by
Lester Earnest, Research Computer Scientist

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-301

June 1977

Computer Science Department
Report No. STAN-CS-77-624

Recent Research in Computer Science

by

John **McCarthy, Professor of Computer Science**
Principal Investigator

Associate Investigators:
Thomas **Binford,** Research Associate in Computer Science
Cordell Green, Assistant Professor of Computer Science
David **Luckham,** Senior Research Associate in Computer Science
Zohar Manna, Research Associate in Computer Science
Terry Winograd, Assistant Professor of Computer Science

Edited by
Lester Earnest, Research Computer Scientist

ABSTRACT

This report summarizes recent accomplishments in six related areas: (1) basic AI research and formal reasoning, (2) image understanding, (3) mathematical theory of computation, (4) program verification, (5) natural language understanding, and (6) knowledge based programming.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or any agency of the V. S. Government.

Reproduced in the U.S.A. Available from the National Technical information Service, Springfield, Virginia 22161.

TABLE OF CONTENTS

Section Page

1. Introduction

2. Basic Research in Artificial Intelligence and Formal Reasoning 2
   2.1 Formal reasoning related to AI questions 2
   2.2 Formal reasoning related to MTC questions 2
   2.3 A reas of work and specific accomplishments 3
   2.4 The FOL proof checker 4
   2.5 References 5

3. Image Understanding 7
   3.1 Achievements 7
      3.1.1 Vehicle Location 7
      3.1.2 Locating Buildings and Change Monitoring 9
      3.1.3 System Description 9
   3.2 References 16

4. Mathematical Theory of Computation 17
   4.1 Motivation 17
   4.2 The Goals 17
   4.3 Current Research 18
   4.4 References 22

5. Program Verification 24
   5.1 Overview 24
   5.2 The Stanford Interactive Verifier 24
   5.3 Summary of Recent Work 25
      5.3.1 Stanford Pascal Verifier 25
      5.3.2 Applications of Verifiers 27
      5.3.3 Operating Systems Verification 28
   5.4 Proposal 29
      5.4.1 Stanford Pascal Verifier 29
      5.42 Verification Experiments 30
      5.4.3 Design of a Concurrent Programming Language and Verifier 31
   5.5 References 32

Section Page

6. Natural Language Understanding 34
   6.1 Building the KRL-0 System 36
   6.2 Experiments using KRL-0 38
   6.3 The Problems 46
   6.4 Current Directions 54
   6.5 References 57

7. Knowledge Based Programming 59
   7.1 Introduction 59
   7.2 Research Objectives 60
   7.3 A Sample Session with PSI 61
   7.4 System Organization 64
   7.5 Conclusions 72
   7.6 Acknowledgements 72
   7.7 References 72

Appendices

A. Theses 74

B. Film Reports 78

C. External Publications 80

D. Abstracts of Recent Reports 88

1. Introduction

This report summarizes six related research projects, with both basic and applied research objectives.

- *Basic research in artificial intelligence and formal reasoning* addresses fundamental problems in the representation of knowledge and reasoning processes applied to this knowledge. Solution of these problems will make possible the development of analytical applications of computers with large and complex data bases, where current systems can handle only a very restricted set of data structures and queries.

- *Image understanding* is aimed at mechanizing visual perception of three-dimensional objects either from photographs *or* from passive imaging sensors. Advances in this field are expected to lead to much more efficient photointerpretation capabilities as well as automatic visual guidance systems.

- *Mathematical theory of computation* studies the properties of computer programs and digital logic. The goal is to provide a sound theoretical basis for proving correctness or equivalence of designs.

- *Program verification* is a closely related project whose goal is to improve the reliability of important classes of programs such as compilers, operating systems and realtime control systems, and to standardize techniques for program construction, documentation and maintenance.

- *Natural Language Understanding* research is developing a knowledge representation language (called KRL) that is expected to support sophisticated systems and theories of language understanding.

- *Knowledge based programming* is developing a new interactive approach to programming in which the computer assists the user in formulating the specification of his problem and in designing the procedures needed to solve it.

Readers who wish to dig deeper should see the references at the end of each section. Appendices list dissertations, films, and other recent reports as well as external publications by the staff.

2. Basic Research in Artificial Intelligence and Formal Reasoning

Personnel: John McCarthy,
Richard Weyhrauch, Martin Davis,
Student Research Assistants:
Juan Bulnes, Robert Filman,
Robert Moore, Andrew Robinson,
David Wilkins.

The long range goals of work in basic AI and formal reasoning are to make computers carry out the reasoning required to solve problems. We believe that our recent work has made it substantially clearer how the more formal approach to AI can be used not only in traditional AI areas but also applied to proving programs correct and hardware verification. This brings applications nearer and has changed the direction of some of our research.

The research we do is primarily technical in nature. When dealing with questions about the basic adequacy of systems of representations of data it is the technical details that are most important. The next two short sections describe the context in which we view formal reasoning to be applicable. We then will describe in detail our recent results.

2.1 Formal reasoning related to AI questions

We feel that for data bases to include many types of information that decision makers really need will require major advances in representation theory. In order for programs to use this information effectively will also require new modes of reasoning. Current data base technology at best allows simple relations to be represented - e.g. "Smith is the supervisor of Jones." Additions from current AI techniques would allow simple generalizations of relations ("Every employee has a supervisor except the director."), but this leaves a tremendous range of representation problems untreated:

1. Mental states - what a person believes, knows, wants, fears, etc.
2. Modalities - what may happen, what must happen, what ought to be done, what can be done, etc.
3. Counterfactual conditionals - if something were true what else would be the case.
4. Causality - how does one event follow because of another.
5. Actions and their modifiers.
6. Self reference - how can I be aware of myself and think about what I am thinking.

None of these concepts can be satisfactorily handled at present, and there are undoubtedly other phenomena which are yet to be discovered. What we are working on is an integrated system in which these kinds of notions can be represented.

2.2 Formal reasoning related to MTC questions

Here we are interested in how to verify to properties of computer programs. The problem, as above, is that there are many interesting questions about programs that existing verification schemes were not designed to answer. There are two main styles of program verification at present, the Hoare-Floyd type, and the the approach of Dana Scott, et al. Although both of these have advantages, neither will comfortably treat the range of problems below. Each example is followed by a typical question we would like to ask the verification system about the programs and specification language it admits.

1. Parsing - is p a well formed program; is s an acceptable specification?
2. Correctness - does a program, p, satisfy some specification, s?
3. Equivalence - do two programs do the same thing, i.e. meet the same specs?
4. Collections of programs - can we mention set of programs which only contain assignment statements.
5. Properties of such sets - can we state in the language that equivalence of any two of the above programs is decidable.

6. Lemmas - can the system specialize the above fact to specific programs?
7. Resources - how much storage does this program use?

We believe that it is possible to handle these questions in a unified system. Recent progress in our ability to represent the correctness of recursive programs in first order logic has been very encouraging.

2.3 Areas of work and specific accomplishments

The above remarks sets the context of our work. It briefly relates some of the questions we think are important. The sections below give some details of the work we have actually done together with some further remarks about questions above.

Representing general facts

The most developed logical system which deals with general facts is first order logic. Statements like "For all programs . .." are represented by using quantifiers. But even within first order logic, there are many possible ways of representing a particular kind of fact, and much further study is required. The FOL system has the ability to enter these general facts into its data base. A different kind of general statement is about facts themselves. For example, we want to be able to say, "Unbelievable statements cannot be true" or "The algorithm a, when applied to a number, generates a true sentence". The latter example is what is usually called an axiom schema. It is an example of a metamathematical sentence. R. Weyhrauch is interested in the problem of how to incorporate general statements into deductions and how to use metamathematics to reason about these facts rather than with them. His work has been primarily in designing and integrating the specific code described below.

Knowledge and belief

The notion X *thinks Y will soon know Z* is not unusually complex when adversaries try to outwit each other, but it presents problems for machine representation that haven't been conclusively solved but on which we have made recent progress. A good artificial intelligence program must be able to prove or conjecture it under appropriate circumstances and it must be able to draw correct conclusions from it - and not draw incorrect conclusions. The latter is the the more immediate problem. Let us use a simpler example. Suppose we have the sentences Pat *knows Mike's telephone number* and *Mike's telephone number is the same as Mary's,* A computerized deduction system that uses the rule that equals may be substituted for equals might conclude *Pat knows Mary's telephone number.* This is not a legitimate deduction, even though it would be legitimate to deduce that Pat dialed Mary's telephone number from the fact that he dialed Mike's number and the fact that the numbers are the same.

Recently McCarthy has discovered how to represent such facts in unmodified first order logic and the solution works no matter how many mental qualities must be treated. The work is described in (McCarthy 1977b) and will be further developed in the next year and a half.

Partial information

Robert Moore has found some new results on representing partial information about knowledge and belief. He has shown that some of the "multiple data base" approaches of previous AI work cannot represent partial knowledge - e.g. they cannot represent the assertion that the Russians know how many divisions the Chinese have, unless the program knows this also, so it can include the information in the data base representing the Russians' model of the world. Moore has shown how this and related difficulties can be avoided by talking not about beliefs themselves, but rather the possible worlds in which the beliefs are true or false. A very elegant theory has been developed based on this approach.

Minimal inference

It has long been recognized that standard logic does not represent the many kinds of reasoning that people use in forming conjectures. This reasoning requires the ability to conjecture that the known facts about a phenomenon are all the relevant facts.

J. McCarthy has recently found a partial solution to this problem. An axiom schema of first order logic called a *minimization schema* can be used to represent in a flexible way the conjecture that the entities that can be shown to exist on the basis of the information in a certain data base are all the relevant entities that exist. The flexibility comes from the fact that the set of information conjectured to be all the relevant information is readily changed. Martin Davis has helped in the mathematical formulation of this method.

Reasoning with observation

R. Filman has demonstrated that the chain of reasoning involved in a complex chess problem requires programs that observe a chess board as well as perform deductions if the solution is to be considered feasible. The point of his research was not to solve chess problems, but to explore how the ability to make direct observations of the world, in this case a chessboard, can be interspersed with deduction to better solve problems. A human player doesn't usually prove that his king is in check by reasoning from the rules. He simply looks at the board and sees that the rook can capture his king (or even more likely is that me hear's his opponent say check). The ability of a person to look at the real world is facilitated by what we have called the semantic attachment feature of FOL, which was designed by R. Weyhrauch. Filman's experience with observational reasoning shows that we still have only begun to understand it.

Facts about one's own knowledge

For a system to explain how it arrived at its conclusions it must be able to reason about its own program. This problem has two parts. One is how to reason about programs, which meshes with our interest in mathematical theory of computation. The aspect directly related to the formal reasoning project involves the question of how can you write a program that can reason about itself. Weyhrauch has designed a system that has some ability to reason about itself. It also can reason some about what it knows. This is a special but particularly tricky case of reasoning about knowledge mentioned above. This system requires several pieces of software the implement which are presently being coded.

Correctness of programs

One of the most important results is McCarthy's ideas for using axiom schemas to embed parts of Scott's style of doing program verification in first order logic. This work is a outgrowth of a thesis by Cartwright which puts in usable form some of the earlier ideas of Kleene. This work has made it possible for us to prove the correctness and termination of several programs and we hope to use these ideas to develop this new style of verification.

2.4 The FOL proof checker

Our main software tool for making a computer follow reasoning is a proof *checker.* Ours is called FOL (for First Order Logic) and checks proof in a system of first order logic that has been enhanced in many ways. We use this tool to formulate the facts involved in an intellectual problem and check that our representation is adequate to solve the problem. As stated above the facts we are studying are general facts about situations and events and actions and goals, the effects of actions that manipulate physical objects, and the facts about sources of information such as books, computer files, people and observation that are necessary in order for a program to obtain the information required to solve problems.

The building of FOL as a test ground for theoretical ideas is one way we keep from presenting ivory tower solutions to problems. We actually use FOL to implement our ideas about representation theory. We are interested in theories whose details can actually be realized as a computer program. Over the past year FOL has been improved in many ways.

It should be noted that three of the tasks described below: the semantic attachment code, the monadic predicate calculus decision procedure and the syntactic simplifier were each programming tasks comparable in scope to lisp interpreters, and this represents an enormous amount of work.

- A decision procedure for the monadic predicate calculus has been added to FOL to decide first-order statements about sorts.

- Semantic attachment has been completely rewritten and is now compatible with the full many sorted logic of FOL.

- A syntactic simplifier has been written. This program allows a user to do the symbolic evaluation various terms and well formed formulas of FOL.

- Several axiomatizations of set theory have been expressed in FOL in order to study their suitability for practical proof-checking. The work with Kelly set theory is a kind of benchmark for this work.

- The McCarthy-Painter compiler has been proved correct in FOL.

- FOL languages have been extended to include conditional terms and function parameters. Introduction and elimination rules corresponding to these notions have been added.

- Two new rules for manipulating quantifiers have been added to FOL.

- A new axiomatization of a theory of knowledge suitable for implementation in FOL has been developed.

2.5 References

[Kelley 1955] John Kelley, *General Topology,* D. van Nostrand Company, Inc., 1955.

[McCarthy 19591 John McCarthy, Programs with Common Sense, *Proc. Int. Conf. on Mechanisation of Thought Processes,* Teddington, England, National Physical Laboratory, 1959.

[McCarthy 19611 John McCarthy, A Basis for a Mathematical Theory of Computation, *Proc. of the Western joint Computer Conf.,* New York, Spartan Books Inc., 1961,

[McCarthy 1963a] John McCarthy, A Basis for a Mathematical Theory of Computation, in Braffort, P. and Herschberg, D. (eds.), *Computer Programming and Formal Systems,* North-Holland, Amsterdam, 1963.

[McCarthy 1963b] John McCarthy, Towards a Mathematical Science of Computation, in Popplewell, C.M. (ed.), *Information processing: Proceedings of IFIP Congress* 62, North Holland, Amsterdam, 1963.

[McCarthy 19641 John McCarthy, A Formal Description of a Subset of ALGOL, in Steel, T.B., Jr. (ed.), *Formal Language Description Languages for Computer Programming,* North Holland, Amsterdam, 1966.

[McCarthy 19651 John McCarthy, A Proof-Checker for the Predicate Calculus, Stanford AI Memo AIM-27, March 1965.

[McCarthy and Hayes 1969] John McCarthy and Patrick Hayes, Some Philosophical

Problem from the Standpoint of Artificial Intelligence, Stanford AI Memo AIM-73, November 1968; also in D. Michie (ed.), Machine *Intelligence,* American Elsevier, New York, 1969.

[McCarthy and Painter 1967] John McCarthy and James Painter, Correctness of a Compiler for Arithmetic Expressions, in Schwartz, J.T. (ed.), *Proc. of a Symposium in Applied Mathematics, Vol. 19 – Mathematical Aspects of Computer Science,* American Mathematical Society, Providence, Rhode Island, 1967.

[McCarthy 1977] First Order Theories of Individual Concepts and Propositions, forthcoming.

[McCarthy 1977] Minimal Inference - A way of jumping to conclusions, forthcoming.

[Prawitz 1965] Dag Prawitz, *Natural Deduction,* Almqvist & Wiksell, Stockholm, 1965.

[Weyhrauch 1977] Proofs using FOL, forthcoming.

3. Image Understanding

Personnel: Thomas Binford, *Student Research Assistants:* Reginald Arnold, Donald Gennery.

The objective of this research in Image Understanding is to build computer systems which locate and monitor buildings, airfields, aircraft and vehicles in aerial imagery. A scientific objective is to accomplish these tasks by building spatial structural models of observed scenes, and matching spatial models, as contrasted with image matching. This approach is taken in order to lead to systems which can use images taken from various viewpoints, sun angles, weather conditions, different sensors, and different seasons of the year.

3.1 Achievements

This research has demonstrated high potential for the use of passive ranging techniques for high resolution depth measurement. Aircraft flying at low altitude using terrain following radar are endangered if 'they use active ranging, which broadcasts their presence. Passive ranging has the advantage of covertness in hostile environments.

Sequences of images from a moving aircraft have been used to find the ground plane and separate objects from ground. The accuracy attained has been demonstrated to be 2" height error for 3" horizontal pixel size on the ground. The system should be effective with camouflaged surfaces. On a general purpose computer, the process requires about 8 seconds with no guidance information. That can likely be reduced at least a factor of 2. With accurate guidance information, the time required is estimated to be about 250 milliseconds (most missions probably fall in this class). The system is self-calibrating and highly reliable. Other groups in image understanding have begun using these algorithms and the code.

The system includes a promising solution to a problem in terminal guidance, guiding a vehicle to a target. This is solved by determining the Observer Model. Imagine an aircraft approaching a runway. As it moves, objects on both sides appear to move radially outward from a center, the fixed point. The center is the instantaneous direction of motion. The pilot knows that the point which does not appear to move is where he will touch down, unless he changes direction. The Observer Model contains the information necessary to calculate the distance of each point from the observer and from the vehicle path. The touchdown point can be calculated from the trajectory of instantaneous directions of motion. The system determines the transform from one view to another in a sequence of views from a moving observer.

3.1.1 Vehicle Location

The objective of this research is to locate cars in an aerial stereo pair of a suburban scene, using stereo. The goal was 80% recognition with 20 hours of processing.

Status: the system successfully separates vehicles from ground and has succeeded in describing the projection of a car as a rectangle of approximately the right size and orientation. The length and width of the car are accurate to about 5% in this example. The system is very near to labeling objects as cars. Cars have been isolated in both aerial and ground level images. Both feature-based and area-based depth mapping have achieved that level of performance. Feature-based stereo is based on edge fragments from the Hueckel operator [Hueckel]; a new technique of linking edge fragments in depth has been developed.

A sequence of steps ending in description of a car by the rectangular outline is shown in the figures which follow. It is not yet possible to estimate its recognition rate. The program finds a coarse depth map and finds the ground plane in about 8 seconds. Then it must make a denser depth map for describing the car.

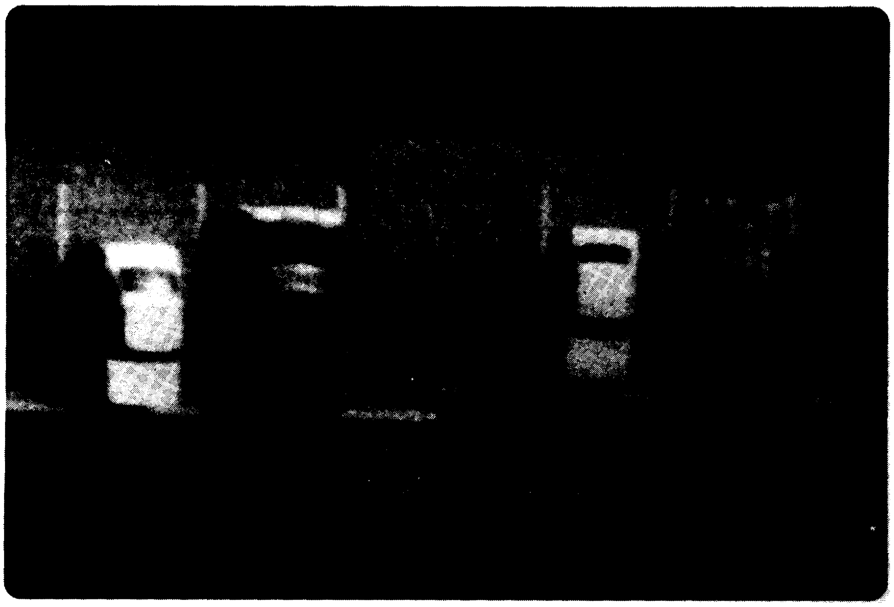


Figure 1. Segment of aerial photo

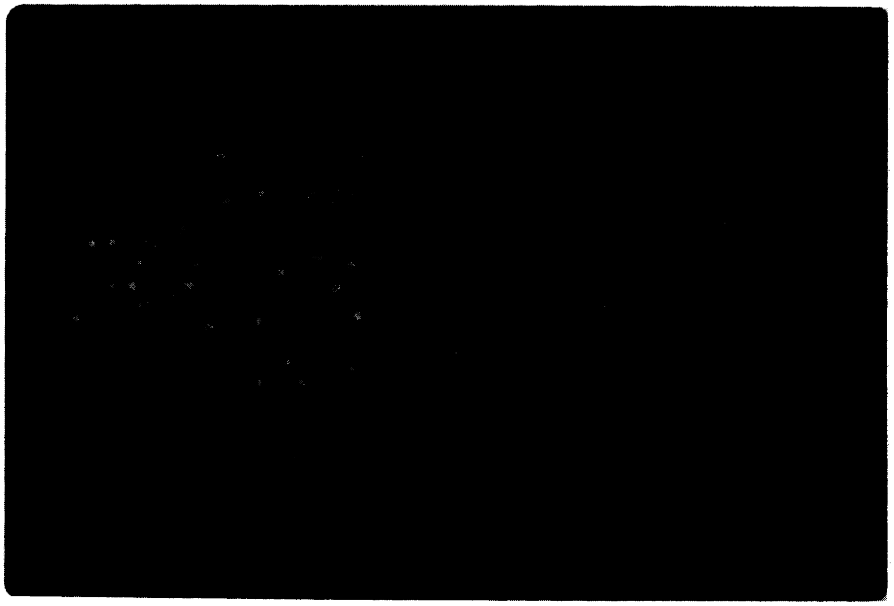


Figure 2. Features of interest

No evaluation has been made of the time required for making the denser depth map and for describing and matching the car, but a crude guess is that potentially about ten seconds is required.

3.1.2 Locating Buildings and Change Monitoring

The same stereo techniques are being applied to build models of buildings in aerial photos of suburban scenes. Because buildings are larger than cars and because they are more likely to have plane sides and box-like sections, the techniques are expected to work even better than for cars. The programs have been developed and preliminary results have been obtained which support this expectation.

3.1.3 System Description

Observer Model

The program first orients itself in the scene and finds an Observer Model, a model for the transform between the two views. This step takes 60% of the time required for finding the ground plane. If two views are an accurately calibrated stereo pair, this operation is not necessary. If accurate guidance information is available, this operation can be speeded up enormously. In any case, this process makes up for any inaccuracies in calibration, maintains a continuous self-calibration, and in the worst case, works even if no calibration or guidance information is available. The program finds a camera transform model by finding-a sample of features of interest in one image . and matching them with their corresponding view in the other image. It needs only to know pairs of corresponding points in the two views; it does not need to know where the points are.

The system automatically selects points of interest. Figure 1 shows a portion of a stereo pair of a parking lot. Figure 2 shows features of interest selected by the program. The Interest Operator requires about 75 milliseconds for a 256x256 frame. Interesting features are areas (typically 8x8) which can be localized in two dimensions without a camera transform. The operator chooses those areas with large variance along all grid directions. That is roughly equivalent to a large drop in autocorrelation along all grid directions, which means that the area can be localized closely. Points on a line will match anywhere along the line, which means that lines are not useful features at this stage, but corners are useful.

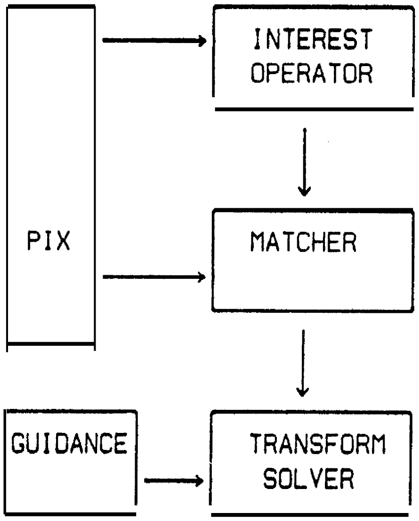


Figure 3
Observer Modeling System

The correlator matches the features in the other image by a coarse to fine strategy: It has versions of the picture at resolutions. of 256x256,128x 128,64x64, 32x32, and 16x 16.

Figure 4 shows the neighborhoods at each of those resolutions, centered around a pair of matching features in the stereo pair. It first matches by a small search on a very coarse version (16x16) of the image. It then performs a search in the next finer version of the image (32x32), in the neighborhood of the best match in the previous image. That step is repeated until the full resolution is reached. The matching process requires only 50 milliseconds for a match of a single feature no matter where it is in the image.

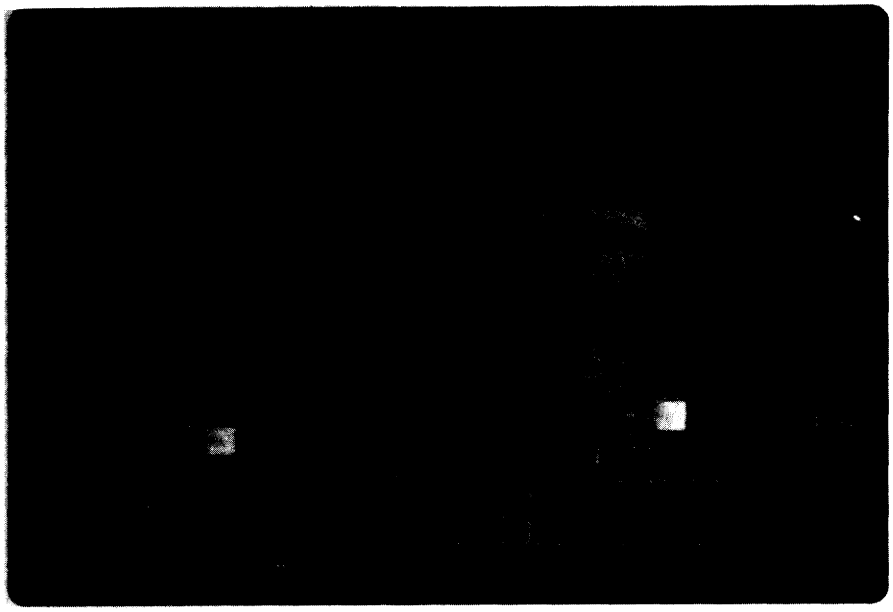


Figure 4. Neighborhoods at various resolutions

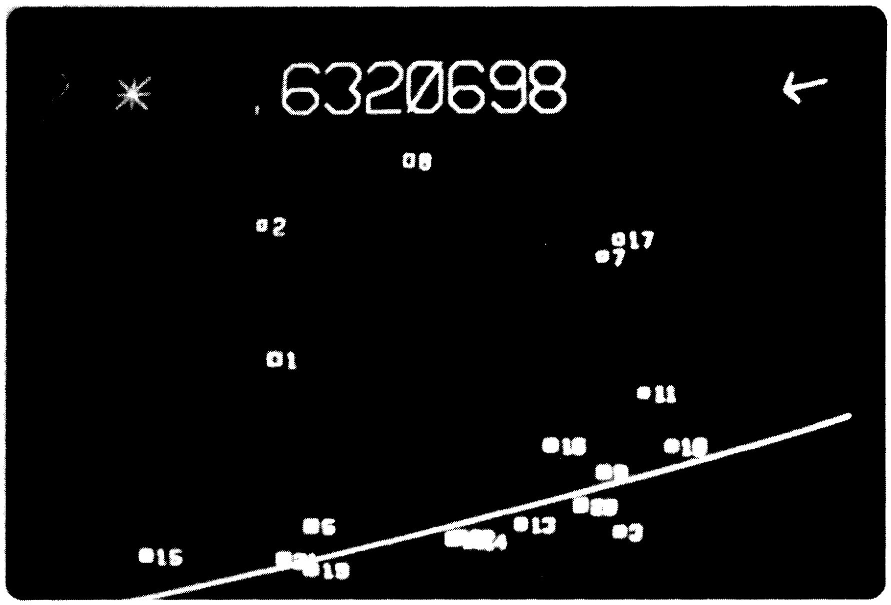


Figure 5. Ground plane estimates

Once the camera transform is known, search is necessary only along a line in the image. In this case, search is about a factor of seven faster. If the depth of neighboring points is used as a starting point for the search, the match is another factor of seven faster. It is planned to incorporate those speedups; neither is now used. The matcher makes about 10% errors. Both the camera transform solver and the ground surface solver reject the points with erroneous matches. It encounters fewer ambiguities than brute force matching, since not only must the feature match, but the surrounding context must also match. The procedure should not work for parts of scenes where the background of objects (context) changes drastically from one view to another. This is true only in parts of images at wide angles and close range. This is also a problem for matching images from very different sensors. The problem will be less important where guidance information is available, since matching breaks down at the coarse phase of coarse-to-fine matching.

Aerial views are mostly planar, so failure of matching should not be a problem, nor has it been in practice. The process requires about 50k of 36 bit words now. It is possible to implement the coarse-to-fine search strategy in a raster scan and keep only a portion of each image in core. This would cut memory size by a large amount, but it has not been done. A version is being designed using this strategy.

The program automatically determines the transform between the twb views. Given corresponding views of five points which are non-degenerate (i.e. no colinear and planar degeneracies) the relative transform of the two views can be found. It is not necessary to know the position of these points, only two views that correspond. The transform is determined except for a scale factor, the length of the stereo baseline. That does not affect subsequent matching of two views using the transform, and the scale factor can often be determined from known scene distances or guidance information.

If the scene is nearly flat, then certain parameters are ill-determined. However, that does not affect the accuracy of measuring heights using the transform. If the scene is nearly flat, then a special simplified form of the solver can be used. The special case version has been used on some images. It is much faster than the full transform solver. In the present form of the camera transform solver, it sometimes encounters stability problems in degenerate cases. It will be investigated to examine when it is limited by data, in which case more data are required, and when it is subject to difficulties in the numerical solution.

Part of the job of the transform solver is to deal with mistaken matches. The procedure calculates an error matrix for each point and iterates by throwing out wild points. It calculates an error matrix from which errors in depths of point pairs are calculated. The solver. uses typically 12 points and requires about 300 milliseconds per point. It requires about 20k of memory. About 60% of the time for finding the ground plane is spent in the camera solver.

With accurate guidance information, this operation would not be necessary. However, it can be used directly to find the instantaneous direction of the vehicle. As mentioned above, as the vehicle moves, points in the image appear to move radially away from the center which is the instantaneous direction of the vehicle. Three angles relate the coordinate system of one view with the other, and two angles specify the direction of the instantaneous direction of motion.

Ground Plane

The approach to discriminating objects is to find the local ground surface and then describe objects above the ground. Finding the ground requires a more dense depth sample. The camera transform model makes it economical to make a denser depth map. A point in one view corresponds to a ray in space which corresponds to a line in the other view. The search is limited to this line, and in addition, nearby points usually have about the same disparity as their neighbors. Thus, search is limited to a small interval on a line. A high resolution correlator has been developed which interpolates to the best match, and which calculates the precision of the match based on statistics of the area.

The system then finds a best ground plane or ground parabola to the depth points, in the least squares sense. Figure 5 shows the distribution of points and the best ground plane fit for the parking lot scene of figure 1. The ground surface finder gives no weight to points above the ground plane; it expects many of those. It includes points below the ground plane and near. the ground plane. Since points below the ground plane may be wild points, they are edited out in an iterative procedure. Of course, there may be holes. If they are small, there is no problem. If the hole is big, it becomes the ground plane. The ground plane finder requires 5 milliseconds per point.

Edge-Based Stereo

Feature-based stereo using edges is interesting, since it increases the accuracy with which boundaries of depth discontinuities can be found by about a factor of 25. That accuracy allows making accurate measurements of dimensions of objects, an important part of our approach. It also provides additional information about surface markings which are not available in stereo based on area correlation. Feature-based stereo is also potentially very fast, although now area-based techniques are considerably faster. Edge-based techniques have not been developed very far, and would benefit from "smart sensor" technology. Figure 6 shows the images from the parking lot scene transformed into the canonical stereo coordinate system, with stereo axis along the x axis. Figure 7 shows edges from the Hueckel operator [Hueckel] in the stereo coordinate system.

A new technique has been developed to use edge features in stereo. Edges are linked along smooth curves in 3d (in the image coordinates and in depth). The new technique is used in the object modeling and recognition modules of the system. Those edges out of the ground plane delimit bodies, if isolated. Figure 8 shows linked edges superimposed on one picture from the pair of images. The left image shows edges near the ground. The right image shows edges above ground level. A vertical rectangular parallelepiped fit to edges gives approximately the right size and direction for car examples. A first crude identification of cars is near. Figure 9 shows the rectangle superimposed on one of the images.

The photos were taken with a wide angle lens at an altitude of about 1500 feet and about 1500 feet apart. They subtend an angle of about sixty degrees. Several areas were digitized to resolutions of about 3" on the ground (courtesy the Image Processing Laboratory, USC). TV images from a parking lot were also used as examples of ground level images.

The limiting accuracy obtainable from a pair of images can be calculated accurately. Consider the vector connecting the two camera centers. Its length will be called b. The angle from this baseline to a point p will be called a. The angle shift of the two views on the unit sphere of the viewer is the same as the angle subtended by the baseline from the point p. That angle is:

theta = b*sin(a)/s.

Here s is the distance of p from the observer. Solve for s:

s = b*sin(a)/theta.

The distance of p from the instantaneous direction of motion is the target error, t.

t=s*sin(a).

The error in the range has two components. The first is a global scale error from errors in b. This affects all distances in the same way. The other is the error in angle measurements.

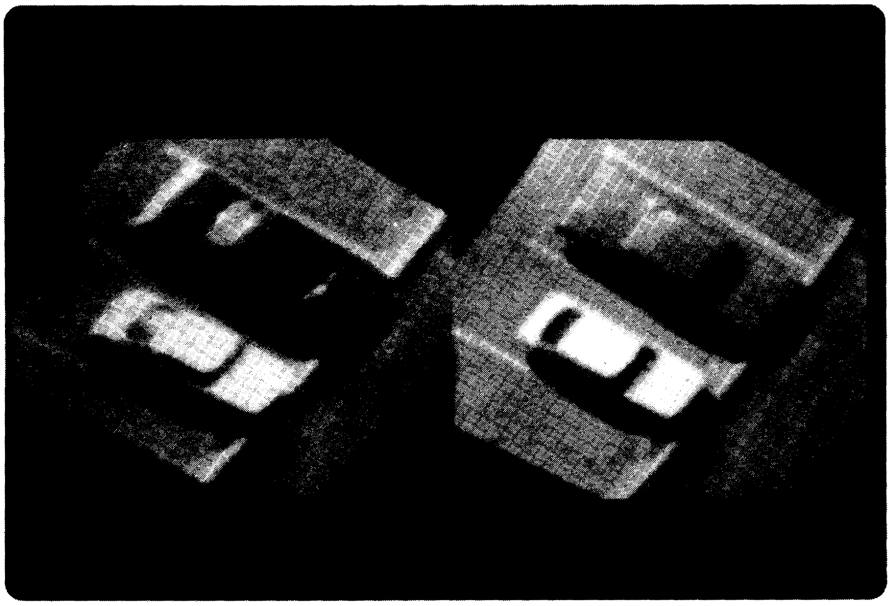


Figure 6. Stereo pair

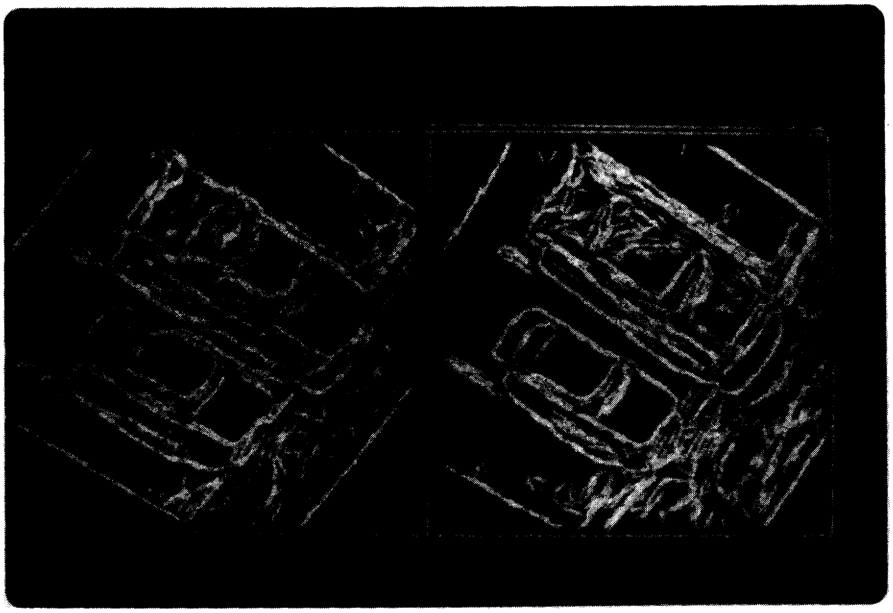


Figure 7. Edges from Hueckel operator

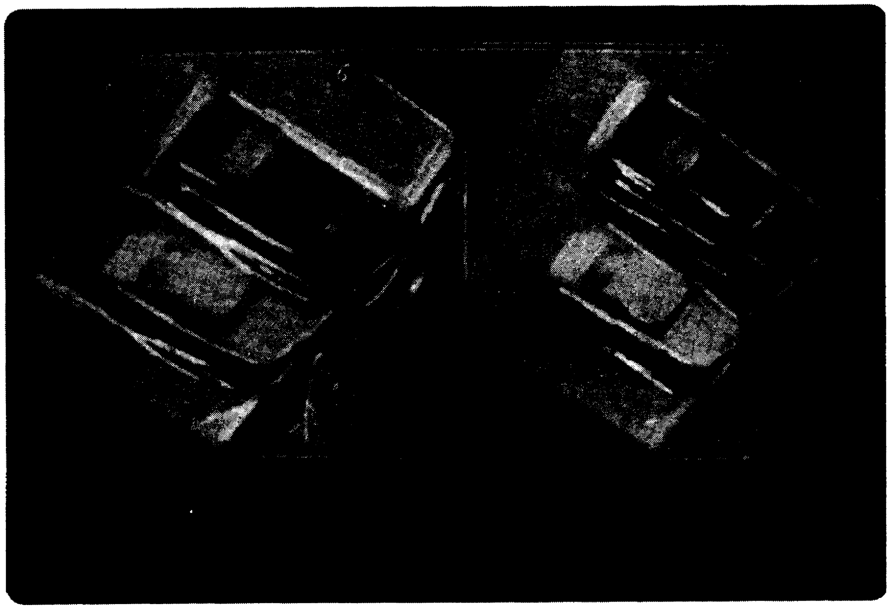


Figure 8. Linked edges near ground (left) and above ground (right).

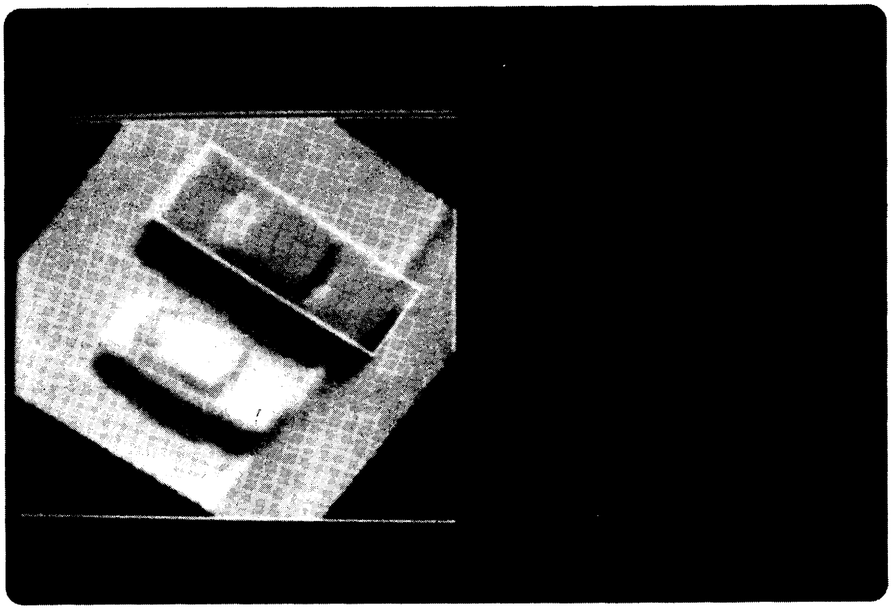


Figure 9. Rectangular parallelepiped fit

Relative distances can be determined to the accuracy with which angles can be measured. Relative distance errors are thus determined by dtheta. Differentiate the above equation for s and substitute for theta:

`ds = -(b*sin(a)/theta↑2)*dtheta`

`ds = -(s↑2/(b*sin(a))*dtheta.`

The range error thus increases as the square of s. Consider:

`ds/s= dtheta/theta.`

Relative ranging error is constant for constant theta. In the case of the above photos, this corresponds to errors of about 1/2 pixel in registration. Previous experience has been about a factor of 3 better than that.

Image Matching System

A system has been built which assists a user to program image matching tasks in about 20 minutes [Bolles]. The system is effective for tasks for which test images are nearly the same as training images; i.e. for which matching can be done in the image. It is more general than 2d since objects and features can move relative to one another. It has shape matching capability, but strictly 2d. It has no 3d models and only point features.

The system suggests features to the user, who chooses among them. Features are chosen by the Interest Operator, mentioned above. The system evaluates the expected cost and utility of each operator. Utility is defined as the contribution to the probability of a decision or as the contribution to positional accuracy. At training time, the system gathers statistics about the effectiveness of the operators. At planning time, it ranks operators according to their expected utility and estimates the total cost of the task by a simple best first strategy. At execution time, the system applies operators in a pre-ordered sequence, combines their results by least squares into confidence and precision, and stops when it reaches a desired confidence or positional accuracy or when it exceeds its cost limit.

Previously, an automated picture retrieval system was implemented. That system provided the user with a way to reference all pictures which covered a specified area. That picture retrieval system could be used with depth maps and other feature descriptors. This would make possible picture retrieval on con tent and location.

There has been progress in other areas. New results have been obtained in quantifying behavior of the Hueckel operator: an automatic determination has been obtained of sensor noise for sensors with square-root noise characteristic (USC digitizer); a sensitivity analysis has been performed for line features; theoretical estimates for errors in position and angle have been made which are in agreement with experimental error distributions; a degeneracy has been found in the edge solution. An improved program to link edges has been implemented. It is not yet operating adequately. However, a stereo version, described above, is operating. That version links edge fragments by both colinearity and depth continuity.

A previous achievement of the program was the formulation of the "generalized translational invariance" representation for complex volume shapes [Binford]. That representation and a laser triangulation system developed here were part of a research program which led to recognition of a doll, a toy horse, and objects of similar complexity [Agin], [Nevatia]. Now, the representation is being widely accepted. Marr has obtained an interesting result that the most reasonable simple assumptions about interpretation of 2d boundaries as 3d objects are equivalent to interpretation in terms of "generalized cones" [Marr].

3.2 References

[Agin] G.J.Agin and T.O.Binford; Representation and Description of Curved Objects; IEEE Transactions on Computers; Vol C-25, 440, April 1976;

[Binford] T.O.Binford; Visual Perception by Computers; Invited Paper IEEE Systems Science and Cybernetics; Miami Fla; Dec 1971.

[Bolles] R.C.Bolles; Verification Vision; submitted to 5th IJCAI, Boston, 1977;

[Gennery] D.B.Gennery; A Stereo Vision System for Autonomous Vehicles; submitted to 5th IJCAI, 1977;

[Hueckel] M.H. Hueckel, A Local Visual Operator which Recognizes Edges and Lines, J. ACM, October 1973.

[IU-1], T.O.Binford; Presentation to first Image Understanding Workshop; USC; April 19'76;

[IU-2], T.O.Binford; Presentation to second Image Understanding Workshop; Univ of Md; October 1976;

[Marr], D.Marr; Analysis of Occluding Contour; MIT AI Memo 372; October 1976;

[Nevatia] R.Nevatia and T.O.Binford; Structured Descriptions of Complex 0 b jects; Artificial Intelligence forthcoming; and Proc 3rd Int Joint Conf on AI (1973).

[Shortliffe], E.H.Shortliffe, R.Davis, S.G.Axline, B.G.Buchanan, C.C.Green, and S.N.Cohen; Computer-Based Consultations in Clinical Therapeutics: Explanation and Rule Acquisition Capabilities of the MYCIN system,; Computers and Biomedical Research, Volume 8, June 1975;

4. Mathematical Theory of Computation

Unified Framework for Program Verification

Personnel: Zohar Manna, John McCarthy, *Student Research Assistants:* Martin Brooks, Nachum Dershowitz, Chris Goad, Todd Wagner.

4.1 Motivation

In the past decade there has been intense research into the problem of proving the correctness of computer programs. As a result, a variety of different program verification techniques have appeared. These methods have different realms of application: some show partial correctness and others show termination, total correctness or equivalence; some apply to recursive and others to iterative programs; some apply to the program itself, but others require that the program be documented or altered.

Here are some of the principle program verification methods and their applications:

- *Invariant-assertion method:* partial correctness of iterative programs (Floyd[19673, Hoare[1969])
- *Well-founded* ordering *method*: termination of iterative programs (Floyd[19671)
- *Structural-induction method* total correctness and equivalence of recursive programs (Burstall[19691)
- *Subgoal-assertion method:* partial correctness of iterative and recursive programs (Manna[1971], Morris and Wegbreit[1977])
- *Counters method:* termination of iterative programs (Elspas et al.[1973], Katz and Manna[19751)
- *Intermittent-assertion method:* total correctness of iterative programs (Burstall[1974], Manna and Waldinger[19761).

"Most current verification systems employ only the invariant-assertion method for proving partial correctness and the counters method for proving termination, even though for some situations another of the techniques may be decidedly superior. For instance, these two techniques cannot be applied to verify certain classes of programs, such as recursive programs, or to prove certain properties, such as the equivalence of two programs. Furthermore, for some verification problems another of the techniques may yield significantly simpler proofs than those required by the two most commonly used methods.

Most verification methods require some form of documentation, in which the user expresses his intuition about how he expects the program to work. However, different techniques require very different sorts of documentation; that which is most natural and easy to provide varies in form from program to program.

4.2 The Coals

We are therefore conducting an investigation with the following goals.

(A) Comparing methods:

- We hope to discover which methods are equivalent in power and which methods are strictly stronger than others, and to determine situations in which a given method may yield simpler proofs than another.
- We want to compare the documentation requirements of the different methods in a rigorous way. Ideally, we wish the user of a verification system to be able to document his intuitions in whatever way he finds most convenient, and have the system select the technique that best matches the program and

documentation supplied and the property to be proved.

(B) Strengthening methods:

- We need to make the existing methods less dependent on the documentation supplied by the user. Thus, we will devise new ways of generating documentation automatically, and of altering and extending the documentation supplied by the user.

- We would like to extend the realm of application of some of the methods, e.g. find how to apply the intermittent-assertion method to show the equivalence of two iterative programs or the correctness of nondeterministic or parallel programs.

- We hope to develop more general techniques that will be able to draw on the advantages of all the existing techniques, and compensate for weaknesses in the existing methodology.

(C) Finding new applications:

We intend to apply methods devised for program verification to other problems such as:

- *Development:* constructing a program to meet given specifications.

- *Transformation:* altering a given program to compute the same output in a different way, generally in order to optimize the program.

- *Modification:* altering a given program to debug it, to adapt it to meet revised specifications, *or* to extend its capabilities.

A unified survey of verification techniques and various applications appears in Manna and Waldinger [July 1977].

4.3 Current Research

Unified Framework

In his Stanford PhD thesis, supported by this project, Robert Cartwright[1977] proposed a way of representing the functional equation of a Lisp program entirely within first order logic. Using this and some earlier results, McCarthy [1977] showed that Lisp and other recursive programs can be completely characterized within first order logic by the functional equation and a minimization axiom schema. It had been previously thought that such characterization required second order logic which is much more difficult to compute with. McCarthy further showed that the well known proof methods of invariant assertions and subgoal assertions were expressible as axiom schemata in first order logic. This unexpected result makes all first order methods of verification and proof-checking more valuable than expected, and it also permits postponing the expression of the full Scott fixed-point theory in first order logic.

John McCarthy plans to exploit this breakthrough by verifying more complex programs directly within first order logic. Because the breakthrough is very new (February 1977), it is not possible to say how far the new methods will go and whether it will still be necessary to develop the extensional form theory. Most likely, the extensional form theory will still be needed, but we will be able to distinguish programs of simple structure that don't require it for their verification.

Wolfgang Polak has an indication that the subgoal-assertion method is exactly McCarthy's minimization schema extended to relations. If this is true, the list of methods for inductively proving programs correct will be greatly shortened by showing that most of the existing methods are subcases of more general methods. This will make it much easier to write verifying programs, both the automatic kind and those that use human help.

McCarthy has investigated continuous functionals that don't arise from simple recursive programs. Some of them require parallel evaluation, and the work may lead to a treatment of program correctness that unifies parallel programs with the more usual sequential programs.

Program Annotation

Zohar Manna is investigating techniques by which an Algol-like program, given together with its input-output specifications, may be documented automatically, This documentation expresses invariant relationships that hold between program variables at intermediate points in the program, and explains the acutal workings of the program regardless of whether the program is correct. Thus this documentation can be used for proving the correctness of the program, or may serve as an aid in the debugging of an incorrect program.

He recently succeeded in unifying existing approaches to this problem, and improving most of the known methods. The techniques are expressed as rules which derive invariants from the assignment statements and from the control structure of the program, and as heuristics which propose relationships whose invariance must be verified. The implementation of this system is in progress.

Results along this line have beein reported in Katz and Manna [1976] and Dershowitz and Manna [1977].

Program Modification

Nachum Dershowitz (graduate student) is attempting to formulate techniques of program modification whereby a program that achieves one result can be transformed into a new program that uses the same principles to achieve a different goal. For example, a program that uses the binary search paradigm to calculate the square-root of a number may be modified to divide two numbers in a similar manner, or vice versa.

The essence of the approach lies in the ability to formulate an *analogy* between two sets of specifications, those of a program that has already been constructed and those of the program that we desire to construct. The analogy is then used as the basis for transforming the existing program to meet the new specifications.

Program debugging is considered as an important special case of program modification: the properties of an incorrect program are compared with the specifications, and a modification (correction) sought that transforms the incorrect program into a correct one.

This approach has been embedded in an experimental implementation and appears in Dcrshowitz and Manna [1976].

Program Synthesis

Manna and Waldinger are developing deductive techniqes for the automatic construction of recursive programs to meet given input-output specifications. These specifications express what conditions the output of the desired program is expected to satisfy. The deductive techniques involve transforming the specifications by a collection of rules, summoned by pattern-directed function invocation. Some of these transformation rules express the semantics of the subject domain; others represent more general programming techniques. The rules that introduce conditional expressions and recursive calls into the program are being investigated in detail.

The deductive techniques were embedded in a running system called SYNSYS. This system accepts specifications expressed in high-level descriptive language and attempts to transform them into a corresponding LISP program. The transformation rules are expressed in the QLISP programming language. The synthesis of several programs performed by the system are presented in Manna and Waldinger [Aug. 19771.

The Intermittent-Assertion Method

Zohar Manna explored a new technique for proving the correctness and termination of programs simultaneously. This approach, *which* he calls the *intermittent-assertion* method, involves documenting the program with assertions that must be true at some time when control passes through the corresponding point, but that need not be true every time. The method, introduced by Burstall [1974], promises to provide a valuable complement to the more conventional methods.

On all the examples attempted, the intermittent-assertion proofs turned out to be simpler than their conventional counterparts. On the other hand, he showed that a proof of correctness or termination by any of the conventional techniques can be rephrased directly as a proof using intermittent-assertions. The intermittent-assertin method can also be applied to prove the validity of program transformations and the correctness of continuously operating programs.

This work is described in a recent paper by Manna and Waldinger [1976]. Manna and Waldinger believe that the intermittent-assertion method will have practical impact because it often allows one to incorporate his intuitive understanding about the' way a program works directly into a proof of its correctness.

Hardware Verification

The research of Todd Wagner (graduate student) involves developing methods for detecting logical errors in hardware designs using symbolic manipulation techniques instead of simulation. A *very* simple register transfer language has been proposed which can be used to specify the desired behaviour of a digital system. The same language can also be used to describe the individual components used in the design. A series of logical transformations can then be used to prove that the set of interconnected components correctly satisfy the specifications for the overall system.

The process of hardware verification uses the basic boolean identities derived from switching theory along with some techniques for determining the possible effects of clock transitions as they move through a circuit. Methods for detecting timing anomalies such as races, hazards, and oscillations have also been studied.

A major goal of this research is to be able deal with fairly complex large scale integrated components without having to reduce their descriptions to the gate level. For example, if a designer requires several complex operations and uses a microprocessor in his circuit, it should only be necessary to demonstrate that the required operations are included in the microprocessor instruction set and that the timing and control logic are correct. Current methods would require a gate level model of the microprocessor and fairly exhaustive simulation. As components become more complex it will become increasingly advantageous to prove circuit correctness algebraically and at a fairly high level.

Preliminary results were presented at the Symposium on Design Automation and Microprocessors (Palo Alto, Ca., February 1977).

Automatic Debugging

Martin Brooks (graduate student) is currently developing methods for specifying and analyzing LISP programs, considering both the theoretical and practical aspects. One goal of this research is the design and implementation of an automatic LISP debugging system.

The automatic debugger is initially supplied with an undebugged program. The system then analyzes the program by symbolic evaluation and automatically generates a sufficient *finite* set of test inputs for which the user supplies the intended outputs. The system extracts the structure of the undebugged program and then use the input-output pairs to fill out the missing details, yielding a new debugged program.

The main advantage of this approach is that it does not require the user to give a formal specification, usually a difficult and error-prone task.

The emphasis of Brooks' research will be to develop a general theory for finding a finite set of input-output pairs which represent a complete specification of a program, depending on the structure of the program.

Generalizing--Proofs

Reasoning by example is a technique frequently used in human problem solving. For this reason,if one regards automatic theorem proving as a mechanical incarnation of human problem solving, the topic of the automatic generalization of proofs from special cases is important. Even if one has no faith in analogies between theorem provers and people, this topic has practical interest since generalizing proofs is likely to be a practically important tool in theorem proving. The effectiveness of a mechanical procedure for generalizing proofs depends on the degree to which it takes advantage of such systematic relationship as may exist between proofs of instances of theorems and proofs of the theorems themselves.

Chris Goad (graduate student) is currently working on a class of technical questions which are relevant to understanding this relationship. These technical questions arise from looking at pairs of formal systems, such that one system is a weak subsystem of the other. Specifically, let T and T' be two such systems, where T' is a weak subsystem of T, that is to say, every proposition provable in T' is also provable in T, but not the other way around. The idea here is that T' be a system within which proofs can be mechanically found with comparative ease, and T is a more general system whose theorems we wish to reduce by instantiation to theorems in T'. Then we may ask for (1) conditions under which the instances of a theorem in T are provable within T', and (2) a uniform method for obtaining proofs in T' of instances of a theorem in T from a proof of the theorem in T.

Goad has obtained solutions to the problems posed above for arithmetic and one of its weak subsystems, namely "proposition al" arithmetic. By "propositional" arithmetic is meant, roughly, the subtheory whose formulas contain no unbounded quantifiers (so each formula only concerns a finite collection of numbers), and where the proofs involve only "propositional" methods (e.g. induction is not allowed). The plan for future work includes the extension of the results on arithmetic, and the study of the same type of problem for other theories of inductively constructed objects, such as the theory of lists. An ultimate goal of this reasearch is the application of these results to program synthesis. That such applications exist is apparent, since mechanical methods are known for extracting programs from constructive proofs of the existence of the functions they compute. Thus any method for generalizing proofs extends immediately to a method for generalizing programs.

Fixedpoint Theory

The classical method for constructing the least fixedpoint of a recursive definition is to generate a sequence of functions whose initial element is the totally undefined function and which converges to the desired least fixedpoint. This method, due to Kleene, cannot be generalized to allow the construction of other fixedpoin ts.

Manna and Shamir [1997] have been investigating an alternate definition of convergence and a new "fixedpoint access" method of generating sequences of functions for a given recursive definition. The initial function of the sequence can be an arbitrary function, and the sequence will always converge to a fixedpoint that is "close" to the initial function. This defines a monotonic mapping from the set of partial functions onto the set of all fixedpoints of the given recursive definition.

They have also suggested a new approach which replaces the classical least fixedpoint with an "optimal" fixedpoint. An informal exposition of this approach appears in Manna and Shamir [1975] and a formal presentation in Manna and Shamir [1976].

4.4 References

1 Burstall, R. M. [Feb. 1969], *Proving properties of programs by structural induction,* Computing J., Vol. 12, No. 1, pp. *41-48.*

2 Burstall, R. M. [Aug. 1974], *Program proving as hand simulation with a little induction,* Information Processing 1974, North Holland, Amsterdam, pp. 308-312.

3 Cartwright, Robert [Jan. 1977], *A practical formal semantic definition system for typed* Lisp, Ph.D. Thesis, Stanford University, Stanford, Ca.

4 Dershowitz N. and Z. Manna [Dec. 19761, *The evolution of programs: automatic program modification,* IEEE Software Engineering (to appear).

5 Dekhowitz, N. and Z. Manna [July 1977], *Derivation rules for program annotation,* Acta Informatica (submitted).

6 Elspas, B., K. N. Levitt and R. J. Waldinger [Sept. 1973], *An interactive system for the verification of computer programs,* technical report, Stanford Research Institute, Menlo Park, Ca.

7 Floyd, R. W. [19671, *Assigning meaning to* programs, Proc. Symp. in Applied Mathematics, Vol. 19 (J.T. Schwartz, cd.), American Mathematical Society, Providence, R. I., pp. 19-32.

8 Katz, S. M. and Z. Manna [1975], *A closer look at termination,* Acta Informatica, Vol. 5, No. 4, pp. 333-352.

9 Katz, S. M. and Z. Manna [Apr. 19761, *Logical analysis of programs,* CACM, Vol. 19, No. 4, pp. 188-206.

10 Hoare, C. A. R. [Oct. 19691, *An axiomatic basis of computer programming,* CACM, Vol. 12, No. 10, pp. 576-580, 583.

11 Manna, Z. [June 1971], *Mathematical theory of partial correctness,* JCSS, Vol. 5, No. 3, pp. 239-253.

12 Manna, Z. and A. Shamir [Dec. 1975], *A new approach to recursive programs,* CACM (to appear).

13 Manna, Z. and A. Shamir [Sept. 19761 *The theoretical aspects of the optimal fixedpoint,* SIAM Journal of Computing, Vol. 5, No. 3, pp. 4 14-426.

14 Manna, Z. and A. Shamir [May 1977], *The convergence of functions to fixedpoints of recursive definitions,* Theoretical Computer Science (submitted).

15 Manna, Z. and R. Waldinger [Oct. 1976], *Is 'sometime' sometimes better than 'always'? intermittent assertions in proving program correctness,* CACM (to appear).

16 Manna, Z. and R. Waldinger [July 1977], *The logic of computer programming,* Computing Surveys (to appear).

17 Manna, Z. and R. Waldinger [Aug. 19771, *The automatic synthesis of recursive programs,* Fifth Intl. Joint Conf. on Artificial Intelligence, Cambridge, Ma.

18 McCarthy, J. [Feb. 19771, *Representation of recursive programs in first order logic,* draft of a technical report, Artificial Intelligence Lab., Stanford University.

19 Morris, J. H. and B. Wegbreit [1976], *Subgoal induction,* CACM (to appear).

5. Program Verification

Personnel: David C. Luckham,
Derek C. Oppen, ***Student Research Assistants:*** R.A. Karp, S. German,
W. Scherlis, R. Drysdale, C.G. Nelson,
W. Polak.

5.1 Overview

The work of the Verification Group is directed towards the development of new programming tools and techniques. Our goal is to improve the reliability of important classes of programs such as compilers, operating systems and realtime control systems, and to standardize techniques for program construction, documentation and maintenance.

Our major effort is in three research areas:

A. Design and implementation of on-line interactive program verifiers.

B. Applications of verifiers in the design, documentation, debugging and maintenance of programs.

C. Design of a high level specification and programming language for implementing and verifying multiprocessing and realtime systems.

Within each of these three main research areas we have pursued specific tasks as follows.

A. 1 Specification and implementation of an extended parser and verification condition generator for the verifier.
A.2 Design and implementation af fast, special purpose theorem provers to improve the capability of verifiers.
A.3 Design and implementation of a user-oriented interface to the verifier. This includes specification languages for programs and documentation, and a command language for the verifier.
A.4 Development of special-purpose verifiers for completely automatic detection of common runtime errors in some programs.
B.1 Standardization of techniques for documenting and verifying important classes of programs, based on our experience in verifying these classes of programs.
B.2 Extension of the use of verifiers to the design, documentation, debugging and maintenance of programs.
C.1 Specification and verification of components of the Solo operating system.

5.2 The Stanford Interactive Verifier

The Stanford Interactive Verifier is a verification system for proving properties of programs written in Pascal. The verifier accepts as input a documented Pascal program, and tries to prove either automatically or with interactive guidance that the program satisfies its documentation. The choice of Pascal is not crucial and the verifier can be changed to accept programs written in other "Algol-like" languages.

The verifier constructs its proofs within the Floyd-Hoare logic of programs. It requires as input a Pascal program together with documentation in the form of Entry and Exit assertions and inductive assertions at crucial points in the program. Fig. 1 shows what happens when the programmer gives this input to the verifier. The input goes first to a verification condition generator which gives as output a set of purely logical conditions called Verification Conditions (VC's). There is a VC for each path in the program. If all of the VC's can be proved, the program satisfies its specification. The next step is to try to prove the VC's using various simplification and proof methods. Those VC's that are not proved are displayed for analysis by the programmer. If the VC's are incorrect, this may reveal a bug in the program or insufficient documentation at some point. A modification is made to the input and the problem is rerun. If the unproven VC's are all correct this merely indicates that the proof procedures need more mathematical facts (called lemmas). The time for a complete cycle

(Fig. 1) in the AI Lab. interactive computing environment is on the order of a minute for a one page program.

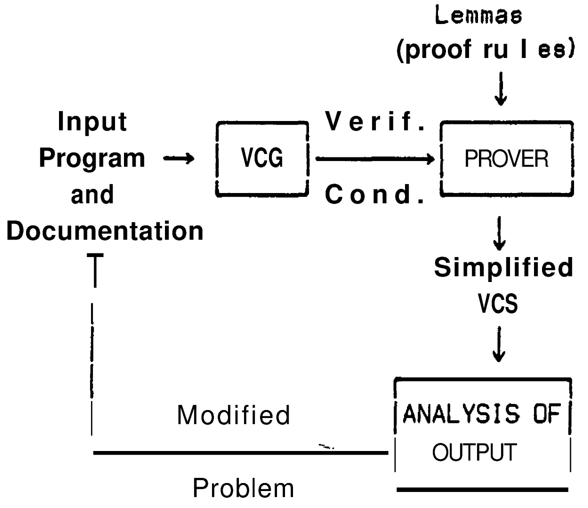


Figure 1.

The first Stanford verifier was written during the period 1972 - 1975. It was successfully used to verify about two hundred programs including about fifty programs involving pointer manipulation [7, 8, 11, 12, 131.

In the autumn of 1975, a comprehensive review of the verifier was made and it was decided to design and implement a more powerful version for general distribution. This was prompted by the successful initial experiments in verification and in the use of the verifier as a programming aid.

Version 2 was planned to include: (i) a flexible and robust interactive user interface, (ii) a more extensive assertion language for specifying properties of programs, (iii) a language for defining rules (lemmas), (iv) extensive syntactic and semantic error checking for both documented programs and rules, (v) more powerful theorem provers. It was planned to be extensible and modular so that it could be easily modified for other programming languages, including multiprocessing languages like Concurrent Pascal.

5.3 Summary of Recent Work

5.3.1 Stanford Pascal Verifier

Parser and Verification Condition Generator

The parser has been written and debugged. It parses both programs with inductive assertions and rulefiles containing lemmas necessary for verification. It gives error messages concerning syntax errors in Pascal code, ‘assertions, and rules. Its diagnostic capability for finding both syntactic and semantic errors in programs exceeds that of most of the Pascal compilers presently available. To accomplish this, it requires certain additional information (such as GLOBAL declarations) which improves the readability and reliability of programs in the language. It permits certain extensions of Pascal (e.g. dynamic arrays). It can be modified to accept other programming languages and other input character sets.

A new verification condition generator (VCC) has been written and is operational. It is a great deal more comprehensive and powerful than any other such generator we know of. The major effect is to substantially reduce the amount of documentation the programmer has to include in his program. Thus the VCG helps remove one of the major problems of using program verifiers.

Prover

The prover takes as input a verification condition and outputs either a proof of the verification condition or else whatever simplification of the verification condition is possible.

The design and construction of provers is the main research battleground in the construction of practical verifiers, simply because it is their deficiencies which have thus far been the main stumbling block to the general acceptance of program verifiers. We have made, substantial progress in this area, and believe that further major progress is feasible.

A completely new prover has been written. It consists of a top level driving routine which interacts with a series of fast, special purpose theorem provers over various useful types of data (integers, pointers, arrays, reference classes, records [13]). The top level driving routine and several special purpose provers have been implemented. Forthcoming reports on this work are [16,17].

Because we feel it essential that all our built-in provers be efficient and not waste the user's time with fruitless searches for proofs, we have not tried to make these special purpose provers too powerful or -general. Instead, we have programmed into them only the facts that are common to all programs. In this way, we are assured that they are always useful without being unnecessarily time-consuming. The problem with this approach is of course that many programs will involve concepts (such as orderedness in sorting programs, or fairness in operating systems) about which the prover has no built-in knowledge. We have solved this problem by designing the prover to be "extendible" in that the user may extend its power in a consistent fashion by interactively adding more useful knowledge.

To accomplish this, we have designed a new rule language for interactively adding information to the prover. The rule language effectively allows the user to write his own special purpose prover and to attempt to optimize its performance by specifying for each rule the amount of effort that should be expended in trying to apply it.

The top level driving routine of the prover stores these rules, and applies them when necessary to obtain a proof of a verification condition. A great deal of effort and empirical study has gone into designing the top level routine so that it applies these rules as efficiently as possible. The present top level routine appears to be very successful at minimizing search time for proofs, mainly by avoiding following long and fruitless paths in trying to find a proof. This part of the prover seems to be surprisingly good, and we feel it is substantially better than previous efforts.

How to write rules is an interesting study in itself, and we return to it later,

On-line User Interface

This has reached a semi-stable design. That is, we are willing to release a version with the present interface. It is intended to permit the user to alternate between fully automatic verification and user assisted verification. The latter is important in debugging and analysis of the verification BASIS.

The top level routine in the prover has been extensively redesigned to permit effective interaction with the user. The user may now interactively guide the proof by specifying which rule is to be applied next with what instantiation of variables, by cutting off lines of proof search that he knows will be fruitless, by interactively adding a rule to prove a lemma which cannot otherwise be proved and so on. The major effort in this has been to minimize the amount of unnecessary interaction required. The goal has been to have the prover ask the user for guidance only if it is trying to prove a particular atomic formula and is unable to find a proof quickly by itself. The user is thus spared having to give trivial commands such as "see if you have proved this lemma already" or provide trivial information such as "yes, $1 = 0$ is false", which is an unfortunate deficiency of many totally interactive provers. We feel that we have found a good compromise between no interaction and total interaction.

Runtime Error Checking

Experimentation has started with a special version of the verifier which is intended to check programs for common runtime errors (e.g. array indices out of bounds, dereferencing a NIL pointer, accessing an uninitialized variable, division by 0, numerical overflow). This is intended to require as little as possible documentation from the programmer. Theoretical principles of modifying verifiers to check automatically for runtime errors are given in [6]. About 20 programs have been checked so far including *Bubble Sort, Quicksort, Matrix Multiplication,* Wirth's version of the *Eight Queens,* and several programs operating on lists and queues by means of pointer manipulation. It was found, for example, that a Pascal version of the *Schorr-Waite* marking algorithm for garbage collection, the standard specifications of which had already been verified, could generate a runtime error by dereferencing a NIL pointer.

This is a highly experimental area and includes the automatic construction of inductive assertions for limited kinds of verification. It has a potentially high payoff in making compiled code more efficient by eliminating the need for code to check for runtime errors. The basic theory [6] has been implemented for these experiments. We make no claims about how "automatic" this kind of error checking can become. The results so far are interesting, and show some problems to be harder than was thought. Even "semi-automatic" checking for runtime errors may be really useful.

Distributable Versiott

The verifier is implemented in various versions of LISP (Mlisp, Lisp 1.6, Maclisp). Translators between these lisps have been written; and debugged by the group. Two thirds of the verifier has been compiled into Hear-n's Rlisp as an experiment. Rlisp is a standard Lisp available for a wide variety of machines (the Rlisp compiler is very good). Currently the Lisp 1.6 version of the complete verifier including workspace occupies about 80k PDP- 10 core. Documented source code listings are available. There is a plan to separate the parser-vcgen and the prover into segments for greater efficiency in timeshared environments.

We are preparing an Rlisp version of the verifier for distribution to selected users. We are also preparing a user manual and an implementation guide on the verifier to help other institutions to use our verifier, and to modify and extend it to meet their own needs.

Comparison with Previous Version

We have repeated all the correct previous experiments to make comparison of efficiency and power with the old verifier. In all cases the verification was faster and used fewer lemmas. The speed-up varies with the problem, and is usually better on the harder problems; some that used to take about 2 hours. of elapsed time at the console now take about 10 minutes. Only the parser is slower, the reason being that it is doing a lot more work.

As a result of user complaints and suggestions, the verifier has gone through over 40 versions since the system became operational in September 76. These reflect changes mainly to the Parser to handle new features of the Rule and assertion languages, and to the prover and user interface.

5.3.2 Applications of Verifiers

(i) MATHEMATICAL ALGORITHMS. This includes Sorting and various tricky programs. Verification experiments in this area have been directed towards standardization of techniques, and use of the verifier for debugging and documentation. New algorithms have been verified by beginning students (e.g. Knuth's In-Situ permutation [14]). The writing and verification of sorting programs on-line in realtime is now very close to being a routine task (a report is forthcoming [2]).

(ii) PROGRAMS WITH POINTERS. We are attempting to formulate standard methods for verifying programs that manipulate pointers based on extending our previous work [13]. Current experiments being attempted in this area include the balanced tree insertion and deletion program (a deep algorithm of about 3 pages of Pascal code), and cyclic list structure copying programs [3].

(iii) JUSTIFICATION OF VERIFICATION BASES. The drive towards standardizing the verification of classes of programs is based on producing the concepts that lie behind the programs, and are adequate for specifying them, and the necessary lemmas defining those concepts. We need justify the BASIS of lemmas once and for all. Most of the time the Bases are obviously correct. But this can get quite tricky when one starts dealing with pointer manipulations.

The Resolution Prover that was developed at Stanford some years ago has been run recently in an attempt to prove some verification Bases (e.g. the Basis for the In-Situ program which depends on some quite sophisticated mathematics about disjoint cycles within permutations). The results were surprisingly good although not overwhelming [14]; it shows that if justification becomes a problem, this line might be worth a little effort.

(iv) VERIFICATION ORIENTED PROGRAMMING. This concerns using the verifier as a programming aid. We have continued experiments reported in [11, 2] to develop methods of using the verifier to plan a program with some combination of specifications and code, and to test the plan at each step as more and more of its details are coded. The methods are based on the analysis of verification conditions to discover bugs and incomplete documentation. The verifier is now being used by one student to do some examination assignments in advanced programming courses at Stanford.

5.3.3 Operating Systems Verification

We have been studying the Solo operating system for the PDP-11. This is a working system, about 22 pages of source code. Solo is written in Concurrent Pascal and widely distributed as an experimental single user operating system. The claim has been made that such operating systems, written in a high level language which has the Monitor construct available to modularize (or package) the code, are easy to write, debug, and verify. The figures quoted for writing time is 3 man/months, and for debugging, 2 days. It is known that Solo is a slow system. But it is the first working system in such a language, and it is claimed that anyone can understand the whole system. We might view part of our work in this area as investigating these claims.

We are experimenting with the automation of the verification of Solo using the present verifier. We are redesigning Concurrent Pascal to improve runtime efficiency, and verifiability, and to extend the class of realtime systems that can be written in it.

RESULTS SO FAR: We have verified the correctness (including Fairness) of a queuing system for the implementation of Monitors [8]. We have verified two fundamental components of Solo-Fifo which controls queuing stategy of Solo, and Resource (a monitor with synchronization) which is used to protect other components of Solo [9]. The proof rules for Concurrent Pascal were applied by hand (very simple) and the resulting sequential Pascal problems (much longer) were given to the verifier. The verification of FIFO turned out to depend on a simple but unstated assumption about Solo, but was otherwise an easy verification. The proof that Resource enforces *mutual exclusion* (i.e. any process having access to a component that is protected by Resource has exclusive access) is interesting; it depends on a theory of how to state such properties of a continuously running multiprocessing environment as *mutual exclusion, fairness,* and *freedom from deadlock.* This theory depends on the use of virtual data structures.

Some other components of Solo, such as Terminal, are going to be difficult to verify in their present form. We have rewritten

Terminal so that it is more efficient and is verifiable.

We have also begun a study of lnaguage features for concurrent systems that facilitate verification [18]. Theoretical developments, proof rule design, and other initial research has been completed.

We draw some conclusions from this work.

(a) The verifier can be used to automate the checking of important properties of high level language operating systems.

(b) We can extend Concurrent Pascal with some simple and natural programming constructs which would make the checking of protection in such systems easy, eliminate bugs due to misuse of protector components, and speed up Solo.

(c) Techniques for writing specialized operating system components need to be refined, and versions of new language constructs already studied should be added for this purpose (this would eliminate problems with Terminal for example).

5.4 Proposal

In this section proposed research tasks are presented together with estimated dates for achieving planned milestones towards the completion of each task. Dates in parentheses refer to the expected duration of each task.

5.4.1 Stanford Pascal Verifier

Work in this area attempts to define standards for program documentation and certification, to introduce the use of verifiers as a programming aid, and to develop new methods of programming and program maintenance. Experience shows that ideas in this area must be subjected to testing by people other than the system implementors.

We propose to attempt this by very limited and cautious distribution of the verifier. The first distribution will begin in September 1977 to selected sophisticated users. Feedback will almost certainly require expansion of the user interface, and the theorem proving capabilities.

Our plans for improving the verifier are as follows:

(i) Typed Rule Language (7/77-9/78). Many errors in verification bases can be caught at parse time if the rule language contains the same type compatibility conventions as Pascal. Such type information can also be used by the theorem prover to select the correct special purpose prover and will speed the proof search on some complex data structure programs by a factor of 5. Furthermore programmers are already used to type declarations and will find this extension of the rule language natural; the type declarations of the program will simply be appended to the basis of lemmas.

(ii) Design Specification of Prover (7/77-12/77). Our concern here is to give a specification of the Prover which contains clear and uniform specifications for all sub-provers. This will enable a user to add his own special provers. A critical point is the specification of the interface between the controlling prover and any sub-prover; this must depend on limiting the interactions between the sub-provers. (We have theoretical studies showing that almost all such interactions are unnecessary). We are working on this now but the final specification will depend on introducing types into the rule language.

(iii) Special Purpose Provers (7/77-7/79). The success or failure of verifiers in the long run will depend largely on their proof capabilities. Much work needs to be done to find efficient provers for the sorts of data that appear in programs and in verification. We need to investigate the advantages of different special purpose provers. This involves experimental comparison of different methods as well as the design of new methods. We also need to design and implement special provers for new kinds of data structures (e.g. trees which appear in compilers).

(iv) **Analyser (7/77-7/79)**; first **version (7/78)**. This is a proposed new module of the verifier to aid in analysis of verification conditions as they relate to the program code. This is intended to aid debugging and documentation.

(v) Code Generator (12/77). We propose to add a code generator to the verifier to enable the user to alternate between verification and compilation. This will be for a subset of Pascal types, including Boolean, **Integer**, Scalar, Array, Record, but not Real. The amount of effort to do this with the current system is not large and should broaden the base of users. The compiler can be extended for the concurrent language (below).

(vi) **Runcheck** Version (first version 12/78). As mentioned previously this includes automating the construction of inductive assertions for runtime error problems. Experiments towards the construction of such a version are already in progress. It is not clear which runtime errors in which kinds of program can be caught by fully automatic verification (no assertion required), and which errors will always require some user supplied information in order to be detected.

The runcheck version will be able to detect fide-effects in procedure and function calls. Some languages have disallowed this feature because it can lead to unexpected errors even though the feature is useful. We propose leaving in this feature, but locating bad side-effects with our **runcheck** verifier before compilation.

Milestones

- Limited distribution of the verifier with documentation (9/77).
- Redesign of the rule language (12177).
- Preliminary experimental version of Runcheck (12/77).
- Modification of Parser for the new rule language (3/78).
- Implementation and testing of new special provers (7/77-6/78).
- Completion of new proof system with extended rule language (9/78).

5.4.2 Verification Experiments

Proposed experiments in verifying programs are aimed at extending the classes of programs which can be verified and improving the verifier to achieve this. In particular, we will go beyond sorting programs, and develop standard techniques for operations on complex data structures by means of pointer manipulation. Programs with pointers have already been verified with this system (previous reports), but the methods are not yet standard. Here we mention tasks aimed at extending the kinds of programs that can be handled by verifiers. There are two categories of experiments.

Milestones

(i) Deep Properties (i.e. depending on mathematics) of Small Complex Programs:

- Balanced tree insertion and deletion (first results, 6/77; finished 12/77). (this program combines both sorting and pointer operations).
- Cyclic list structure copying algorithms (first results, 9/77; finished 6/78).
- Average running time estimates of mathematical algorithms (typical of properties depending on probabilistic analysis). (First results, 9/77, complete analysis of the problem, 9/78).

(ii) Shallow properties of large programs:

- Pascal compiler. This requires theory of segmentation into verifiable passes, and specification of each pass. We have started on the segmentation theory, and on verification of the lexical scanner pass. (First results, 6/77; finished 9/78).
- Operating Systems. We will investigate the verification of protection and deadlock problems for entire operating systems derived from Solo and other sources. See the next section.

5.4.3 Design of a Concurrent Programming Language and Verifier

We propose to design a concurrent programming language and implement a verifier for it. The reasons for this are:

1. A verifier for Concurrent Pascal will require simple changes to that language (below). So some redesigning has to be done anyway.

2. With the changes, the verification of some operating system specifications for the whole of Solo is not difficult. So the concurrent systems verifier should be useful. --

3. We wish to extend verification techniques to systems that cannot be written in Concurrent Pascal, e.g. realtime systems involving, for example, interrupt handling, memory allocation, and dynamic process invocation.

Implementation of the new language itself may be undertaken at a later time.

Some of the changes we propose making to Concurrent Pascal should make systems within the new language more efficient than Solo. This may be a step towards making high-level language operating systems more practical. So the extension of the present verifier to a concurrent systems verifier with a compile-and-run option is an interesting possibility.

Next we make some comments about changes to Concurrent Pascal.

Our study of Solo has led us to formulate extensions to Concurrent Pascal which express both the specifications of some Solo components and the programming discipline that has been used in Solo. Unless the specifications and discipline are expressed in the syntax of the language (in a form somewhat analogous to assertions and type declarations in sequential languages) the problem of verifying Solo is horrendous. This is not because the problems are hard, but because necessary information is never stated explicitly. Of course, some properties of single components (written as single Concurrent Pascal monitors) can be verified, but this is much different from verifying that the whole system possesses a certain simple but crucial property (e.g. all data structures that require protection are in fact protected).

For example, component classes and monitors in Solo perform specialized tasks such as scheduling, protection and buffering. None of these specialized tasks is declared, so there no explicit assertion to be checked about the function of the component. Also components that must be bound together (e.g. a scheduler and the component to be scheduled) are done so by means of a single general mechanism, binding of accessrights at initialization, which is not only uninformative for verification, but also leads to inefficient blocks of components. Scheduling is one of the bottlenecks in Solo, and protection works correctly only because the programmer bound the accessrights of Solo correctly in the initialization process (one slip there, and the system would never work). The monitor construct alone does not write correct operating systems, nor does it specify them.

Also note that some important parts of the Solo operating system (interrupt handling, memory allocation, process suspension and resumption) are handled by an invisible, unverifiable assembly language kernal. We propose to develop high level, verifiable language constructs, where possible, to make these aspects of operating systems visible. The work in [18] is a very small portion of this.

Examples of proposed extensions *are* to introduce:

(i) Declarations of the specialized function of certain components (from which we can tell what needs to be verified about them in the context of the whole system).

(ii) Composition of components which must always be invoked together in a particular order (this will make scheduling more efficient and eliminate some possible errors due to scheduling in a wrong order).

(iii) Statements of the protection requirements of a type at the type declaration, and declarations of protection guarantees (a protector component which is an accessright of a type, may be declared to guarantee protection for parameters of that type). These are to aid the verification of correct protection over the whole system.

Such extensions are natural in the sense that they permit the programmer to state *very* simple intentions; they are similar to type declarations in Pascal. They are crucial information for the verification of some standard operating system' specifications. Other natural extensions will become evident as the verification of Solo progresses.

Milestones

- Verification of more Solo components: Buffers and Processes (6/77).
- Verification of correct protection for whole of Solo (12/77).
- Preliminary verifier for a concurrent extension to Pascal (12/77).
- Implementation of components of operating systems within the extended language and their automatic verification using the preliminary version (9/77-6/78).
- Design of concurrent systems language (6/78).
- First version of concurrent systems verifier (6/78).
- Verification of a small realtime system (9/78).
- Pilot studies in implementation of the concurrent systems language (1/78-6/79).

5.5 References

[1] P. Brinch Hansen, The Solo Operating System, *Software-Practice and Experience* 6, 2 April-June 1976, 141-205.

[2] R.L. Drysdale, A Standard Basis for Automatic Verification of Sorting Algorithms, Stanford A.I. Lab. Memo, Stanford University, forthcoming.

[3] D.A. Fisher, Copying Cyclic List Structures in Linear Time Using Bounded Workspace, *CACM*, 18,5, May 1975, 251-252.

[4] S.M. German, B. Wegbreit, A Synthesizer of Inductive Assertions, *IEEE Trans. on Software Engineering,* Vol 1. pp.68-75., March, 1975.

[5] C.A.R. Hoare, and N. Wirth, An Axiomatic Definition of the Programming Language Pascal, *Acta Informatica,* Vol. 2, 1973, pp.335-355.

[6] G. Huet, D.C. Luckham, D.C. Oppen, Proving the Absence of Common Runtime Errors, Stanford A.I. Lab. Memo, Stanford University, forthcoming.

[7] S. Igarashi, R.L.London, D.C. Luckham, Automatic Program Verification I: Logical Basis and Its Implementation, *Acta Informatica,* Vol.4, 1975, pp. 145-182.

[8] R.A. Karp, D.C. Luckham, Verification of Fairness in an Implementation of Monitors, *Proc. Intnl. Conf. on Software Engineering,* San Francisco, pp. 40-46, Oct. 1976.

[9] R.A. Karp, D.C. Luckham, Automating the verification of operating systems: A case study - the Solo operating system, Preliminary internal report, Stanford Verification Group, Jan. 1977.

[10] D.C. Knuth. *The Art of Computer*

Programming, Vol. III - Sorting and Searching, Addison Wesley, Reading, Mass. 1973.

[11] F.W. v.Henke, DC. Luckham, A Methodology for Verifying Programs, *Proc. Intnl. Conf. on Reliable Software,* Los Angeles, California, pp. 156- 164, April 20-24, 1975.

[12] D.C. Luckham, N. Suzuki, Automatic Program Verificatiori IV: Proof of Termination within a Weak Logic of Programs, Stanford AI Memo AIM-269, Stanford University, 1975, to appear in *Acta Informatica.*

[13] D.C. Luckham, N. Suzuki, Verification Oriented Proof Rules for Arrays, Records, and Pointers, Stanford AI Memo AIM-278, Stanford University, April 1976.

[14] W. Polak, Verification of the In-Situ Permutation Program, Stanford A.I. Lab. Verification Project Report, forthcoming.

[15] R. S. Cartwright, D. C. Oppen, Generalized Procedure Calls in Hoare's Logic, forthcoming.

[16] C. C. Nelson and D. C. Oppen, An Efficient Decision Algorithm for the Extended Theory of Equality, forthcoming.

[17] C. G. Nelson, D. C. Oppen, A Theory of Independently Interacting Theorem Prdvers, forthcoming.

[18] R. A. Karp, The Scheduled Class as an Operating Systems Structuring Concept, submitted for publication, 1977.

6. Natural Language Understanding

A joint project with the Xerox Palo Alto Research Center.

EXPERJENCE WITH KRL-0
ONE CYCLE OF A KNOWLEDGE REPRESENTATION LANGUAGE

Daniel G. Bobrow, Terry Winograd,
and the KRL research group'

'The projects and implementation described in this paper were done at Xerox Palo Alto Research Center, Palo Alto, California by Dan Bobrow, Ron Kaplan, and Martin Kay from Xerox PARC; Jonathan King, David Levy, Paul Martin, Mitch Model, and Terry Winograd from Stanford; Wendy Lehnert from Yale; Donald A. Norman from U.C. San Diego; Brian Smith from M.I.T; and Henry Thompson from U.C Berkeley.

The goal of the KRL research group is to develop a knowledge representation language with which to build sophisticated systems and theories of language understanding. This is a difficult goal to reach, one that will require a number of years. We are using an iterative strategy with repeated cycles of design, implementation and testing. An initial design is described in an overview of KRL (Bobrow & Winograd, 1977). The system created in the first cycle is called KRL-0, and this paper describes its implementation, an analysis of what was learned from our experiments in using KRL-0, and a brief summary of plans for the second iteration of the cycle (the KRL-1 system). In writing this paper, we have emphasized our difficulties and disappointments more than our successes, because" the major lessons learned from the iterative cycle were in the form of problems. We mention only briefly in the summary of experiments those features of KRL-0 that we found most satisfactory and useful.

In order to put our experiments in some perspective, we summarize here the major intuitions we were testing in the design of KRL-0:

1. Knowledge should- be organized around conceptual entities with associated descriptions and procedures.
2. A description must be able to represent partial knowledge about an entity and accommodate multiple descriptors which can describe the associated entity from different viewpoints.
3. An important method of description is comparison with a known entity, with further specification of the described instance with respect to the prototype.
4. Reasoning is dominated by a process of recognition in which new objects and events are compared to stored sets of expected prototypes, and in which specialized reasoning strategies are keyed to these prototypes.
5. Intelligent programs will require multiple active processes with explicit user-provided scheduling and resource allocation heuristics.
6. Information should be clustered to reflect use in -processes whose results are affected by resource limitation and differences in information accessibility.
7. A knowledge representation language must provide a flexible set of underlying tools, rather than embody specific commitments about either processing strategies or the representation of specific areas of knowledge.

Some of these intuitions were explored in GUS (Bobrow, et al, 1977). a dialog system for making airline reservations. GUS used ideas of procedural attachment (Winograd, 1975), and context dependent description (Bobrow & Norman, 1975). Experience with GUS led to some changes to our ideas for KRL-0, although GUS and KRL-o were basically concurrent projects; we started programming GUS just prior to intensive design on KRL-o. The GUS system was primarily an attempt to explore the integration of already existing programming technology for a performance demonstration, while KRL-o was a first attempt at outlining a new basis for representation.

1. Building the KRL-o System

KRL-o was implemented in INTERLISP (Teitelman, 1975), along the lines described in Bobrow and Winograd (1977). The design was specified mostly during the summer of 1975. The initial KRL-o implementation was programmed primarily by Bobrow, Levy, Thompson, and Winograd during December and January, with parts of the development being done by the rest of the KRL group. It included basic structure manipulating facilities, a reader and printer for KRL structures, a simple agenda manager and scheduler, a procedure directory mechanism, and a matcher which handled only the most elementary cases. Many more pieces were built into this system by people working on the test projects over the following 6 months. The system was first implemented on the MAXC computer at Xerox PARC and later transferred to the SUMEX PDP-IO, (where one of the projects was done as an AIM Pilot project), and to the IMSSS PDP-10 at Stanford. When the test projects were complete, the system was retired from active duty.

As an experimental system, there was no commitment to continue support after the initial purposes were satisfied. Despite its avowed experimental nature, however, building KRL-0 was a major system programming effort; programming any “new AI language” for users is larger task than just trying out the new ideas. Having the many facilities of INTERLISP to build on eased our programming burden, but a number of new facilities were built for the project:

- Ron Kaplan developed a set of utilities, including special data structure manipulation and formatted printing routines, as a base for much of the implementation. The entire utility package (called USYS) was interfaced so smoothly that the user could think of it as simply an extended INTERLISP. This package will be used in the development of KRL-1.
- An on-line cross-reference and documentation system (called the NAMES system) was used to coordinate the efforts of the people doing interactive debugging of a shared set of programs. The facility was designed and built by Ron Kaplan and Martin Kay. It communicated with the editor and file package facilities in INTERLISP so that the programmer was prompted for a comment whenever programs or record

declarations were created or edited. The information available to the system (e.g. procedure name, variable names, etc.) was combined with user supplied comments in a standardized data base which could be interrogated on line. The programmer was automatically warned of potential naming conflicts with anything **anywhere** else in the system. It also provided facilities for entering comments associated with global variable names and file names. The file of names grew to contain over 1000 entries during the course of implementing **KRL-0.** For the **KRL-1** implementation we are extending the interface to work with Masterscope, the **INTERLISP** cross-reference and program analysis package written by Larry Masinter.

- A simulated match interface was built by Paul Martin, which enabled the programmer to intercept calls to the matcher and gather data on what kinds of problems came up before programming the necessary' extensions. The user returned an answer for the match, and on future identical matches the same answer was used.
- A tracing facility for the matcher was implemented by Jonathan King, to facilitate debugging of programs which were organized around matching

. As problems came up in using KRL-0, they were handled in several ways. Those which seemed general and could be handled within the existing framework were set up as tasks for the **KRL-0** programming effort. Usually design discussions were shared by everyone, and the implementation done by the person whose program faced the problem. Those problems which were either too specialized or obviously beyond the scope of our current design were programmed around by the problem-finder. Most of these cases led to changes in the KRL-1 design to **accomodate** solutions more naturally. Because KRL-0 was embedded in INTERLISP, "patching" was usually straightforward in that it was the same as what would have been involved in trying to write the program in a bare **INTERLISP** in the first place. Of course, sometimes these "patches" interacted with other parts of the KRL code in unpredicted and confusing ways. Those problems for which there was no acceptable way to escape were chalked up to experience, and the goals of the program reduced accordingly. Usually this was in cases where there had been an unresolved question as to how much the program should be expected to handle. Issues raised by these problems were a major driving force in the KRL-1 design.

A very rough draft of a manual was distributed, but became rapidly obsolete as the system evolved. It was highly incomplete (for example, the section on the matcher consisted of a single paragraph describing why the section was going to be difficult to write). It was never completed or re-edited, and those doing the programming had to rely on discussion with the implementers and on the source code of the interpreter for up to date information. It worked reasonably well, with some frustration, but not enough so that anyone ever felt moved to volunteer the time to do the writing needed to produce a real manual and keep it current. We were somewhere around the upper bound of the size of project (number of people, amount of programming) where so informal an approach was feasible.

2. Experiments using KRL-o

KRL-o notation and programs were tested in nine different small projects. Each of these projects was intended to test some aspect of the KRL-o language or system. They took from 3 to 15 person-weeks of effort each. In most cases, the goal was to produce an actual running program which could handle enough examples to convince us that it did what the original program was intended to. In no case was an effort made to do the kind of final debugging and polishing which would make the program robust or usable by anyone but the original author. We will describe three of these in detail: a cryptarithemetic problem solver; a story analysis program; and a medical diagnosis system. We list below the other projects that were done to give a flavor of the range of projects tried:

- LEGAL -- done by Jonathan King -- an implementation of a portion of a legal reasoning system sketched by Jeffery Meldman (1975) in his doctoral dissertation. This program forced consideration of matching in which both patterns and data could specify bindings that were needed.
- ARCHES -- done by Paul Martin -- a concept learning progam based on Patrick Winston's (1975) program for recognizing visual scenes. Matching sets of descriptions, and the use of instances as patterns were the interesting parts of this project
- COIL -- done by Wendy Lehnert -- a new program for drawing inferences about objects, based on methods related to those of conceptual dependency. This program used the contingent description mechanism to select knowledge to be used in a particular context, and the agenda to interweave syntactic and semantic processing of input English.
- FLOW -- done by Dan Bobrow and Don Norman -- a program sketch which simulated a person's access to long term memory while using a recently learned simple computer language. The indexing mechanism of KRL was used to simulate properties of human associative retrieval (including errors of various kinds).
- PHYSIOLOGY -- done by Brian Smith -- a program sketch which explored the problems of using KRL-o for a system which could reason about physiological processes. This project forced consideration of the gaps

in KRL-0 with respect to specifying temporal and causal structures, and the need for stronger structuring to factor information in units by viewpoints, e.g., information about the heart as viewed as a mechanism, versus information when viewing it from an anatomical perspective.

- **KINSHIP** - - done by Henry Thompson -- a theoretical paper, using the KRL-0 notation as a basis for comparing kinship terms in English and Sherpa. The attempt to communicate results of encoding to non-computer scientists led to a simplified notation which has contributed to the syntax for **KRL-1.**

Cryptarithmetic

The initial test program was a simple cryptarithmetic problem solver (see Newell and Simon, 1972 for a description of the domain) written by Terry Winograd and debugged and extended by Paul Martin. it exercised the basic data structures, agenda, and triggering facilities, and was successfully tested on several problems (including DONALD + GERALD = ROBERT with **D=5).** No attempt was made to provide complete coverage of the class of problems handled by humans. Interesting aspects of the design included:

- Use of triggers to combine goal directed and data directed processing

- Use of "patterns" to suggest strategies

- Use of levels on the agenda to control ordering of strategies

- Use of multiple descriptors to accumulate information about the value of a letter

- Use of contingencies to handle hypothetical assignments

- Use of the signal table to control work within hypothetical worlds

Much' of the processing was associated with procedures attached to the units for Column (a vertical column in the addition problem) and Letter. The Unit for Column is given below. **It** gives some idea of the use of procedural attachment to propagate information, search for patterns such as a column with **TwoBlanks** and trigger arithmetic processing (using the LISP function **ProcessColumn).**

```
[COLUMN UNIT Basic
<SELF {}>
<leftNeighbor (a Column)>
<rightNeighbor (a Column)>
<topLetter (a Letter)>
```

```
<bottomLetter (a Letter)
   (triggers (WhenKnown
              (DoWhenKnown (topLetter) Column
                      (TryToFurtherSpecify UNIT
                          '(TwoBlanks OneBlank TwinAddend)
                          'AddendType]>
<sumLetter (a Letter)
   (triggers (WhenKnown
              (DoWhenKnown (topLetter botomLetter) Column
                      (CheckSumEqualAddend UNIT]>
<topDigit (a Digit)
   (triggers (WhenKnown (Assign 'topLetter)(ProcessColumn)))>
< bot tomDigit (a Digit)
   (triggers (WhenKnown (Assign 'bottomLetter)(ProcessColumn)))>
<sumDigit (a Digit)
   (triggers (WhenKnown (Assign 'sumLetter)(ProcessColumn)))>
<sum { (an Integer)
       (which IsSumOf
           (AllItems (the carryIn) (the topDigit)(the bottomDigit)))}
    (triggers (WhenKnown (ProcessColumn)))>
<carryIn { (an Integer)
           (XOR 0 1)
           (the carryOut from Column(the rightNeighbor))(; CARRYOUT))
   (triggers (WhenKnown (GoFill 'CARRYOUT)(ProcessColumn)))>
<carryOut {(an Integer)
           (XOR 0 1)
           (the carryIn from Column (the leftNeighbor)) (; CARRYIN))
   (triggers (WhenKnown (GoFill 'CARRYIN) (ProcessColumn)))>]
```

There was a set of recognized patterns for columns (for example, a column with the sum letter identical to one of the addends) and a set of pattern driven strategies was associated with each. Each strategy was a LISP procedure which used the KRL-o structures only as a data base. Some of the strategies caused values to be computed. Whenever a new value was filled into a column, triggers caused data driven strategies to be suggested, such as trying to bound the possible value of other letters based on this information. Constraints on values were added in the form of new descriptions for the value of the letter, for example specifying that the value must be an even or odd integer. Each such description was added to the existing description of the value of that letter, so that at any point in the computation, some letters had a value described as a specific digit, while others had complex desciptions, such as "Greater than 3 and odd". Each time a new description was added, a trigger in the unit for Letter caused a procedure to be run which matched each still-unassigned digit against the accumulated description, and if only one matched, it was assigned.

When new strategies were suggested by a new value being filled in, or by the match of one of the patterns describing columns, all of the triggered strategies were put onto the agenda. They were assigned priority levels on the basis of a fixed scheme: Level 1 was immediate propagation of information (e.g. if the value of a letter is determined, then that value gets entered into all of the places where the letter appears). Level 2 was for straightforward arithmetic computations, Level 3 for the strategy being worked on currently, Level 4 for other simple strategies, Level 5 for more complex and less likely strategies, Level 6 for last-ditch strategies (brute force trial and error) and Level 7 contained a single entry which caused the problem to be abandoned.

This rather *ad hoc* use of agenda levels achieved a number of goals. The use of Level 1 for simple propagation served as a kind of data locking scheme to maintain consistency. As long as there were more results to be propagated, no other part of the program would run. This meant, for example, that if some letter were assigned to a digit, no other letter could be assigned to the same digit before the result had been properly recorded. The use of a separate level for the current strategy allowed it to trigger sub-strategies without getting put aside for work on a different strategy. This meant that each strategy could run to completion. The use of levels to distinguish how promising different strategies were allowed the system to focus its effort on whatever were the most likely things at the moment. Placing last-ditch strategies on lower levels when they were thought of made it easy for the program to fall back on them -- they automatically ran if nothing at any higher priority was scheduled. This provided a weak global structuring in what was inherently a data-driven process.

The mechanisms for multiple worlds and contingent descriptors made it possible to deal with hypothesized values while using the normal mechanisms. When all but two possible values had been eliminated for some letter, and no other strategies were pending, the program chose one of them, and created a hypothetical world, in which the letter had that value. Describing the letter as having that value hypothetically caused all of the same triggering as would noncontingent assignment of the value, leading to propagation of new information, computations, strategies, etc. However, by modifying the signal table, all derived information was asserted as contingent on that hypothetical world. This special signal table also affected the processing in two other ways: First, only simple strategies were allowed to be placed on the agenda. Second, if a contradiction occurred, the hypothesis was rejected instead of the problem being declared impossible. If a hypothesis was rejected, the contingent descriptors were not removed, but would not be accessed by programs looking for descriptions in other hypothetical worlds, or in the world of actually inferred facts.

Sam

David Levy implemented and tested a program which reproduced the simple text analysis and questioning aspects of the SAM program (Schank et. al, 1975) which uses scripts in analyzing short "stories" containing stylized sequences of events. It used Ron Kaplan's GSP parser(Kaplan, 1973), and a grammar written by Henry Thompson for the initial input of the stories. It processed two stories (Schank, p. 12), summarized them and answered a number of simple questions. It was a full fledged language-processor in that it took its input in English and generated English output. Questions were entered in an internal representation. Its main features were:

- Interfacing an existing parser (Kaplan's GSP) with a KRL-0 program which used the results of the parsing for further analysis
- Using slots to represent the basic elements (both events and participants) of scripts, and perspectives to represent instances of the scripts.
- Using the notion of "focus lists" as the basis for determining definite reference, including reference to objects not explicitly mentioned in the input text. It used the index mechanism to speed up search through the focus lists.
- Using the matcher in a complex way to compare story events to prototypical script events, with side effects such as identifiying objects for future reference
- Using units describing lexical items and English grammatical structures as the basis for analysis and generation, using signals and procedural attachment

SAM's basic processing loop consisted of parsing, construction of conceptual entities followed by script lookup:

Parsing. A sentence from the story was fed to GSP, which produced as output a surface syntactic parse identifying clauses, noun phrases, etc. as a KRL declarative structure. For example, for the sentence "John went to a restaurant" GSP produced the following rather shallow syntactic structure:

```
(a Declare with clause =
   (a Clause with
       surfaceForm = 'John went to a restaurant"
       verb = GO
       subject = (a NounPhrase with
                              head = JOHN)
       prepPt = (a PrepositionalPhrase with
                              preposition = TO
                              object = (a NounPhrase with
                                              head = RESTAURANT
                                              determiner = A))))
```

Construction of conceptual entities. The next step was to map this syntactic object into a set of conceptual objects with the help of declarative and procedural information stored in the prototypical syntactic units (Clause, **NounPhrase**, etc.) and in the lexical units. For example, the Clause unit specified that the filler of the verb slot would guide the mapping process for the entire clause, and the lexical representation of each verb included a case frame mapping from syntactic to conceptual structures. Following is a partial description of SAM's representation of the verb "go":

```
[GO UNIT Individual
  <self ((a Verb with
              root = "go"
              past = "went")
          (which IsAConstituentOf
            (a Clause with
              referent =
                (a Go with
                   goer = (the referent from NounPhrase
                                (the subject from Clause (a Clause)))
                   source = (the referent from NounPhrase
                                (the object from PrepositionalPhrase
                                   (a PrepositionalPhrase with
                                      preposition = FROM)))
                   destination = (the referent from NounPhrase
                                   (the object from PrepositionalPhrase
                                      (a PrepositionalPhrase with
                                         preposition = TO))))))} >]
```

As a description was created for each conceptual object (e.g. as it was determined that the appropriate filler for the goer slot in the above example was (a Person with name = "John")), this description was matched against a list of units in a *focus list* which contained the conceptual objects thus far created. If the description matched one of these objects, the slot was filled with a pointer to this object, and this object was moved to the front of the focus list. In order to make the search through the focus list faster, the index facility was used to find good potential matches from the list. If the description matched no object, a new object (a KRL unit) was created, the description was attached to it, and this object was pushed onto the front of the focus list. In this way referents were established and maintained.

This scheme handled pronominal as well as definite reference. From the word "she", for example, the conceptual description (a **FemalePerson**) was constructed, a description which would match the last mentioned reference (if any) to a female person (e.g. "the waitress").

Script lookup. Next the program tried to identify the conceptual event just created as a step in an active script. It did this by stepping' through the script from the last event identified, and *matching* the description of this prototypical event to the event just created from the input sentence. This process exercised the KRL matcher rather heavily. Once the step in the script (represented as a slot) was identified, this slot was filled with the new conceptual event. In addition, any previous steps not explicitly filled by story inputs were then filled by creating conceptual events from the prototypical descriptions contained in the script. These events too were added to the focus list. The program also dealt with *what-ifs* or predictable error conditions, but these will not be discussed here.

The result of this iterative process was therefore the construction of a representation for the story consisting of:

- a set of syntactic units representing the surface syntactic form of the input sentences
- a set of conceptual units representing story objects: people, events (including inferred events), physical objects
- a focus list containing these objects
- a (partially) instantiated script, whose event slots were filled with the conceptual events in the focus list

Having analyzed a story, SAM could then summarize, paraphrase, and answer questions.

The different stages of processing in the analysis of inputs were controlled through the use of special **signal** tables. These tables provided special responses to the addition of descriptions to units. For example, the search for a referent was keyed by a signal set off by the addition of a perspective, of type **NounPhrase.** The generation process used a different set of signal tables to direct the inverse process of building a surface syntactic construction from a conceptual object. SAM was an interesting exercise in system construction, useful mainly as a tool for understanding problems in representation and debugging KRL-0. When finished, it did not, and was not intended to, rival the power of the Yale group's original program.

Medical

Mitch Model implemented and tested a program for medical diagnosis based on a model for diagnosis which had not been directly implemented before (Rubin, 1975). In writing the program, it was necessary to fill in a number of details, and correct some minor inconsistencies in the original. The program successfully duplicated, with some minor exceptions, the performance described for Rubin's hypothesized system. Part of the reason for the exceptions was incomplete specifications in Rubin's thesis, but there was also a major problem in that the implementation LISP code and data base completely filled the storage available in the KRL system. (This program, SAM, and COIL were the most extensive tests, and all ran into space problems discussed below). Some of the major features of the implementation were:

- The use of the abstraction hierarchy to represent the set of disease types and finding types, with information and procedures attached at different levels of generality.
- The use of KRL-0 triggers to implement the conceptual “triggering’* of potential diagnoses on the basis of having relevant symptoms described
- The use of signals to provide run-time monitoring of what the system was doing as it generated new hypotheses and evaluated them
- A direct encoding of the declarative “slices” of Rubin's version into the declarative forms of KRL-0. This included extensive use of the “Using” descriptor (a declarative conditional) to explicitly represent the decision trees in the units for diagnosing different conditions

- There were four major kinds of representational objects in the system.
  - *“Elementary hypotheses” which corresponded to the “slices” of Rubin's thesis; these were named after the disease [e.g. *Glomerulitis* or *RenalInfarction*] the data structure was intended to represent. Elementary hypotheses had descriptions in slots to indicate such things as likely symptoms, links to other elementary hypotheses that might be related, and how to evaluate how well the patients symptoms would be accounted for by a diagnosis of this disease.
  - “Elementary hypothesis instances” were data structures created for each diagnosis the system decided might account for the presented symptoms; these contained pointers to the findings that suggested the diagnosis, and a pointer to the elementary hypothesis representing the disease of the diagnosis. It also contained values for how well the diagnosis accounted for the symptoms, obtained by applying the evaluation information represented in the elementary hypothesis to the specific details of the elementary hypothesis instance.
  - “Findings” were units for specific symptoms, facts, historical information, physical examination data, or lab data (e.g. *Fever, Hematuria,* or *Biopsy*); a finding was mostly a hook on which to hang procedural information about what to do when the patient exhibited something abnormal with respect to the

particular kind of finding.

- Finding instances were the input to the system, having a structure similar to that Rubin suggested in her thesis, having slots for such things as finding, duration, severity, and normality. There were also further specified finding instances such as symptom instance.

The system worked essentially as follows. A unit might be described by:

```
(a Symptominstance with
   mainConcept = Hematutia
   presence = "present"
   severity = "gross"
   time = (a TimePoint with .
                     direction = "past"
                     magnitude = (a Quantity with
                                          unit = "days"
                                          number   = 3)))
```

A *WhenKnown* trigger on the *presence* slot of the *SymptomInstance* prototype would be set off; examination of the specific description caused this entity to be described also as: (a **SymptomInstance** with normality = "abnormal") Further triggers and traps might result in the creation of new elementary hypothesis instances, according to the information found in the description. After all the information propagation activity, each of the currently active elementary hypothesis instances would be evaluated based on information found in the corresponding elementary hypotheses. Based on the evaluation, the status of the elementary hypothesis instances might be changed to reflect possible dispositions of the hypothesis such as acceptance, rejection, or alteration.

The indexing facility was used to facilitate operations such as obtaining a list of all the hypotheses activated by a finding. Functionals and *ToMatch* triggers on prototypes were defined to handle special time-related matches to enable the system to tell, for example, that "3 days ago" is more recent than "1 year ago" or that "48 hours" is the same as "2 days". Signal tables were used locally to govern the handling of error-like occurrences and globally to effect trace and printout; different degrees of detail were specified by use of several signal tables, and it was thus quite simple to change modes by pushing or popping a table. The agenda was used for organizing the flow of control in a manner similar to that described for the Cryptarithmatic program. The built-in triggering mechanisms provided the means for a very natural modeling of the kind of medical reasoning discussed in Rubin's thesis.

3. The problems

As we had hoped, these projects pointed out many ways in which KRL-o was deficient or awkward. People were able to complete the programs, but at times they were forced into ad *hoc* solutions to problems which the language should have dealt with. The problems can be grouped as:

- Basic representation problems -- ways in which it was difficult to express intuitions about the semantic and logical structure of the domain
- Difficulties in manipulating descriptions explicitly
- Shortcomings in the matcher
- The awkwardness of the LISP-KRL interface
- Facilities which should have been available as standardized packages
- Infelicitous syntax
- Cramped address space

Due to the embedding of KRL-o in INTERLISP, none of these problems were fatal. Even with the difficulties, we found it possible to write complex programs rapidly, and to experiment with interesting representation and processing 'strategies. This list also does not include the social and organizational problems which are bound to infect any effort of this nature. Everyone on the project exhibited heroism and stoicism, persisting in their programming without a manual and in a rapidly evolving language which kept slipping out from under the programs almost as fast as they could be modified.

Basic representation problems

KRL-o embodied a number of commitments as to how the 'world should be represented. Some of these seemed intuitively justifiable, but did not work out in practice. Others were too vague to implement in a way which seemed satisfactory.

The categorization of units: Each unit bad a category type (as described in Bobrow and Winograd (1977, pp 10-12)) of *Individual, Manifestation, Basic, Specialization,* or *Abstract Category.* This was based on a number of intuitions and experiments about human reasoning, and on the belief that it would facilitate mechanisms such as the quick rejection of a match if there was a basic category disagreement. In practice, these distinctions turned out to be too limiting. In many of the hierarchies for specialized domains (such as medicine) there was no obvious way to assign *Basic, Specialization,* and *Abstract.* In dealing with units describing events, the notion of *Manifestation* was not precise enough to be useful. It was generally felt that although the concepts involved were useful, they had been embedded at too low a level in the language.

Viewpoints: One of the major issues in developing KRL was the desire to have facilities for "chunking" knowledge into relevant units. This proved to work out well in most cases, but there was an additional dimension of organization which was lacking. For many purposes, it is useful to combine in a single unit information which will be used in several contexts, and to associate with each piece of the description some identifier of the context (or *viewpoint)* in which it will be used. In the natural language programs, it seemed natural to classify descriptions associated with words and phrases according to whether they related to the structure of syntactic phrases, or to meaning. In the physiology sketch, there were clear places where different viewpoints (e.g. looking at the form of an organ or looking at its function) called for using different information. There were two primitive mechanisms for doing this factoring in KRL-o -- attaching features to descriptors, and embedding information in contingencies. Both were used, but proved clumsy and felt *ad hoc.*

The relation between prototype and concept: KRL is built on the assumption that most of the information a system has about classes of objects is stored in the form of "prototypes" rather than in quantified formulas. In general, this proved to be a useful organizational principle. However, there were cases of complex interactions between instance and prototype. In the medical domain, for example, a disease such as **AcuteRenalFailure** could be thought of as an instance of the prototype for Disease but could also be thought of as a prototype for specific cases of this disease. There are a number of issues which arise in trying to represent these connections, and although KRL-o did not make obviously wrong choices, it also did not make obviously right ones. In general, we seem to have been hoping that too many consequences would just naturally fall out of the notation, when in fact they take more explicit mechanisms.

Further specification hierarchies: In simple network or frame systems (see, for example Goldstein and Roberts, 1977) there is a natural notion of hierarchy, in which each descendant inherits all of the slots (or cases) from its parent. Thus, if a Give is a further specified Act then it has a slot for actor as well as its own slots for object and recipient. In a system based on multiple description, the inheritance of slots is not as straightforward. This is especially true when there is an attempt to do Merlin-like reasoning and use perspectives to "view an x as a y". The basic inheritance mechanism in KRL-o does not include automatic inheritance of slots. This is vital for cases in which there are multiple descriptions using the same prototype units. However, it makes it awkward (though possible) to program the cases where the slots are to be inherited simply. Therefore, we included a mechanism for "further specification" which allowed a unit to inherit slots (along with their attached procedures) from a single parent. This was not fully implemented into the system, and was a dangling end in the implementation.

The factoring of context-dependent descriptions: One major design decision in KRL was the use of an object-factored data base, rather than a context-factored one. The unit for a particular object contained all of the different contingencies representing the facts about it in different worlds. This proved quite successful; however, when combined with the kind of descriptions provided by mappings, another issue arises. Using the example of the cryptarithmetic units given earlier, consider the problem of representing what is known about a column in the addition problem if worlds are used to represent hypothetical assignments. Imagine that we know that in the unmarked global world, Column1 is an instance of Column, with values for topletter, bottomletter, etc. If in a hypothetical **World1** (in which some value is assumed for a letter) we infer that its sum is 17, we want to add a contingent descriptor. This could be done in two ways:

```
[Column I UNIT Individual
   <self { (a Column with
                  topletter = A
                     ...)
            (during World1 then (a Column with sum =17))}>]
```

```
[Column I UNIT Individual
   <self { (a Column with
                  topLetter = A
                  sum = (during World1 then 17)
                     ...)}>]
```

These are equivalent at the semantic level, and the first was chosen in the initial implementation -- all factoring into contexts was done at the top level of slots. However this proved to be tremendously clumsy in practice, since it meant that much of the information was duplicated, especially in cases of recursive embedding. This was exacerbated by the fact that features (See Bobrow and Winograd, p. 14) demanded factoring as well, and were used for a variety of purposes, such as the viewpoints mentioned above. There was a reimplementation midway in the life of KRL-o in which the basic data structures were changed to make it possible to merge as much of the shared information as possible. There are a number of difficult tradeoffs between storage redundancy, running efficiency, and readability when debugging, and we never found a fully satisfactory solution within KRL-O.

Data structure manipulation

KKL-0 was not a fully declaratively recursive language in the sense that machine language and pure LISP are. It was not possible to write KRL-0 descriptions of the KRL-0 structures (e.g. units, slots, descriptions) themselves, and use the descriptive mechanisms to operate on them. Instead, there were a number of LISP primitives which accessed the data structures directly. People ran into a number of problems which could be solved by explicit surgery (i.e. using the LISP functions for accessing KRL data structures, and RPLACA and RPLACD) but which gave the programs a taint of *ad hocery* and overcomplexity. As an exercise in using KRL representational structures, Brian Smith tried to describe the KRL data structures themselves in KRL-0. A brief sketch was completed, and in doing it we were made much more aware of the ways in which the language was inconsistent and irregular. This initial sketch was the basis for much of the development in KRL-1.

Deletion of information: One of the consequences of seeing KRL-structures as descriptions, rather than uninterpreted relational structures was a bias against removing or replacing structures. Descriptions are by nature partial, and can be expanded, but the most natural style is to think of them as always applicable. Thus, for example, if a slot was to contain a list (say, the list of digits known to have been assigned in a cryptarithmetic problem), the descriptor used in an instance was the Items descriptor, which is interpreted as enumerating some (but not necessarily all) items in a set. If the description of some object changed over time, then it was most naturally expressed explicitly as being a time-dependent value, using the Contingency descriptor. There are some deep representational issues at stake, and the intuition of thinking of descriptions as additive was (and still is) important. However, it led to an implementation which made it impossible to delete descriptions (or remove items from lists) without dropping to the level of LISP manipulations on the descriptor forms. This caused problems both in cases where values changed over time, and in cases where the programmer wanted the program to delete unnecessary or redundant descriptors in order to gain efficiency. Although deletion and replacement were doable (and often done), they went outside of the KRL semantics in a rather unstructured way.

Explicit manipulation of descriptions: For some of the programs, it was useful to have parts of the code which dealt with the descriptions themselves as objects. For example, in the cryptarithmetic program, the set of descriptions being added to the value slot of an individual Digit could be thought of as a set of "constraints", and used in reasoning. One might ask "What unused digits match all of the descriptors accumulated for the value of the letter A". This is quite different from asking "Which unused digits match the description 'the value of letter A"'. Similarly, in the implementation of Winston's program, the descriptions themselves needed to be thought of and manipulated as relational networks. The ability to use descriptions in this style gave power in writing the programs, but it had to be done through LISP access of the descriptor forms, rather than through the standard match and seek mechanisms.

Problems with the matcher

Specifying the match strategies: The matcher in KRL-0 took a KRL-o description as a pattern, and matched" it against another description viewed as a datum. For each potential descriptor form in the pattern, there were a set of strategies for finding potentially matching descriptions in the datum. The ordering of these named strategies, and the interposition of special user-defined strategies was controlled by use of the signal mechanism. This was designed to give complete flexibility in how the match was carried out, and succeeded in doing so. Many specialized match proceses were designed for the different projects. However, the level at which they had to be constructed was too detailed, and made it difficult to write strategies which handled wide ranges of cases. The strategies were mostly reflections of the possible structures in the datum, and did not deal directly with the meaning of the descriptors. This led to-having to consider all of the possible combinations of forms, and to programs which did not function as expected when the descriptions contained different (even though semantically equivalent) forms from those anticipated.

Returning bindings: Since patterns for the matcher were simply KRL-o descriptors, and there was no coherent meta-description language to define procedural side effects, it was very difficult to extract the bindings from a match. This was handled in the legal example, which most needed it, by providing special signal tables for these matches, again leading to a feeling of *ad hocery* to get around a basic problem in matching.

Control of circularities: In using matching as a control structure for reasoning, it is often useful to expand the match by looking at the descriptions contained in the units being compared. Consider the units:

```
[Give UNIT Basic
    <self (A Receive with
                received+ (the given)
                receiver = (the . receiver)
                giver = (the giver))>
    <giver (A Person)>
    <receiver (A Person)>
    <given (A PhysicalObject)>]

[Receive UNIT Basic
    <self (A Give with
                given= (the received)
                receiver = (the receiver)
                giver = (the giver))>
    <giver (A Person)>
    <receiver (A Person)>
    <received (A PhysicalObject)>]

[Event 17 UNIT Individual
    <self (A Give with
                giver = Jane
                receiver = Joan
                given = (A Hammer))>
```

If asked whether the pattern (A Receive with received = (A Hammer)) matches Event17, the matcher needs to look in the unit for Give in order to see that every Give is indeed a Receive, and to match up the slots appropriately. However, this can lead to problems since descriptions in units could easily be self-referential, and mutually cross-referential. In a slightly more complex case, the matcher could try to match a Give by looking at its definition as a Receive, and then transform that to a Give, and so on. Some of the early match strategies we developed fell into this trap and looped. The simple solution that was adopted to limit such circular expansion was to adopt a depth first expansion policy, and to limit the total depth of expansion (recursion through definition). This obviously works both in this case, and to limit arbitrarily large non-circular searches. In the limited data bases we used, it never caused a match to be missed when the programmer expected it to be found. But it is a crude device which does not provide adequate control over search.

Inefficiencies due to generality: Since the matcher was designed to allow a wide range of strategies, a fairly large amount of processing was invoked for each call. Often, the programmer wanted to check for the direct presence of a certain descriptor, and to avoid the overhead, dived into LISP. Thus, instead of writing: .

```
(Match 'Event 17
       '(A Give with giver = Jane)
       'SimpleStructureMatchTable)
```

it was possible to write:

```
(EQ 'Jane
    (GetItem (GetFiller 'giver
                        (Get Perspective 'Give
                            (GetSlot 'self 'Event 17)))))
```

Given that the SimpleStructureMatchTable caused the matcher to look only at direct structural matches, the two forms were equivalent, and the second avoided much of the overhead. Many problems arose, however, in cases where later decisions caused the description form to be different (for example, embedded in a contingency) but to reflect equivalent information.

Problems in the interface between KRL and LISP

One of the major design decisions in KRL-0 was the use of LISP for writing procedures, rather than having a KRL programming language. This was viewed as a temporary measure, allowing us to quickly build the first version, and work out more of the declarative aspects before trying to formulate a complete procedural language in the following versions. A number of awkward constructs resulted from the need to interface LISP procedures and variables to the KRL environment.

Limited procedural attachment modes: Only the simplest forms of procedural attachment were implemented. Thus, for example, there was no direct way to state that a procedure should be invoked when some combination of slots was filled into an instance. Procedures had to be associated with a single condition on a single slot. It was possible to build more complex forms out of this by having a trigger establish further triggers and traps (there are examples of this in the unit for Column given above), but this led to some rather baroque programming.

Communication of context: When a trap or trigger was invoked, the code associated with it needed to make use of contextual information about what units were involved in the invocation and what state the interpreter was in (for example in the use of hypothetical worlds). This was done simply by adopting a set of LISP free variables which were accessible by any piece of code. and were set to appropriate values by the interpreter when procedures were invoked. This approach was adequate in power, but weak in structure, and a number of the detailed problems which arose in the projects grew out of insufficient documentation and stability of what the variables were, and what they were expected to contain when.

Unstructuredness of procedure directories: The notion of having a “signal table” containing procedural variables was a first step towards breaking out of the normal hierarchical definition scheme of LISP. The intention in developing a KRL procedural language- is to develop a set of structured control notions which make it unnecessary for the programmer to fill in the detailed responses to each possible invocation. In the absence of this, KRL-o signal tables had much the flavor of machine code. A clever programmer could do some striking things with them (as in their use in SAM for controlling language analysis and generation), but in general they were hard to manage and understand.

Underdeveloped Facilities

The KRL overall design (see Bobrow and Winograd, p. 3) involved a series of "layers" beginning with the primitive underlying system and working out towards more knowledge-specific domains. Part of the ability to implement and test the language so quickly came from deferring a number of problems to higher layers, and letting users build their own specialized versions of pieces of these layers as they needed them. In most cases this worked well, but there were some areas in which a certain amount of effort was wasted, and people felt hampered by not having more general facilities.

Sets and sequences: KRL-o provided only three primitive descriptors (Items, AllItems, and Sequence) for representing sets and sequences. Notions such as subset, position in sequence, member of set, etc. all had to be built by the user out of the primitives. Everyone needed some of them, and it became clear that a well thought out layer of standard units and procedures would have greatly simplified the use of the language.

Indexing schemes: The index mechanism built into KRL-o was based on simple collections of key words. It was assumed from the beginning that this was to be viewed not as a theory of memory access, but as a minimal primitive for building realistic access schemes. One of the projects (FLOW) attacked this directly, but the rest stuck to simple uses of indexing, and did not explore its potential in the way they might have if a more developed set of facilities had been provided initially.

Scheduler regimes: As with indexing, the scheduler mechanism of KRL-o was intended primarily as a primitive with which to build interesting control structures which explored uses of parallelism, asynchronous multi-processing, etc. The only structuring was provided by the use of a multi-layer queue Like the category types discussed above, it was an attempt to embed some much more specific representation decisions into a system which in most places tried for generality. It was not restrictive, since the system made it possible to ignore it totally, allowing for arbitrary manipulation of agenda items. However, because it (and no other scheme) was built in, it tended to be used for problems where other regimes would have been interesting to explore.

Notation

The KRL-0 notation was strongly LISP-based, using parenthesization as the primary means of marking structure. This made it easy to parse and manipulate, but led to forms which were at times cumbersome. This was especially true because of the use of different bracketing characters ("()", "{}","<>") for descriptors, descriptions and slots. At times a unit would end with a sequence such as "})})}>)". There was one simplification made during the course of the implementation, allowing the description brackets "{}" to be omitted around a description containing a single descriptor. The examples in this paper use this convention. In addition, better notations were needed for expressing sets and sequences, and were explored in the KINSHIP project.

Limited address space

One of the shortcomings which most strongly limited the projects was in the implementation, not the basic design. INTERLISP is a paged system, based on a virtual memory which uses the full 18 bits of the PDP-10 address space. The philosophy has always been that with some care to separate working sets, system facilities could grow to large sizes without placing extra overhead on the running of the program when they were not being used. This has led to the wealth of user aids and facilities which differentiate INTERLISP from other LISP systems.

As a result, more than half of the address space is used by the INTERLISP system itself. The KRL-0 system added another quarter to this, so only a quarter of the space was available for user programs (including program storage, data structure storage, and working space). Both of the extended systems (SAM and Medical) quickly reached this limit. This resulted in cutting back the goals (in terms of the number of stories and questions handled by SAM, and the amount of the sample diagnosis protocol handled by Medical), and also led the programmers to put a good deal of effort into squeezing out maximal use of their dwindling space. Some designs were sketched for providing a separate virtual memory space for KRL data structures, but their implementation was put off for later versions, since the lessons learned in using KRL-0 within the space limitation were quite sufficient to give us direction for KRL-1.

4. Current Directions

The projects described above were completed by the end of summer 1976. Since that time, we have been primarily engaged in the design of KRL-1, and as of this writing (June 1977) are in the midst of implementing it. The development has involved a substantial shift of emphasis towards semantic regularity in the language, and a formal understanding of the kinds of reasoning processes which were described at an intuitive level in the earlier paper. Much of this has been the result of collaboration with Brian Smith at M.I.T, who is developing a semantic theory (called KRS for Knowledge Representation Semantics) which grew out of attempts to systematize and understand the principles underlying systems like KRL.

The new aspects of KRL-1 include:

- A uniform notion of *meta-description* which uses the descriptive forms of KRL-1 to represent a number of things which were in different *ad hoc* forms in KRL-0. The old notions which are covered include features, traps and triggers, index terms, and a variety of other detailed mechanisms. The emphasis has been on providing a clear and systematic notion of how one description can describe another, and how its meaning can be used by the interpreter. A number of the problems related to the manipulation of description forms are solved by this approach.
- A more structured notion of the *access* and *inference* steps done by the interpreter. The interpreter is written in a style which involves operating on the *meaning* of the forms, rather than the details of the forms themselves. This makes possible a more uniform framework for describing matching and searching procedures, and the results they produce. It allows the language to be described in terms of a clear semantics (see Hayes, 1977 for a discussion of why this is important). We expect it to make the development of complex Match and Seek processes much easier.
- A notion of *data compaction* which makes it possible to use simple data record structures to stand for complex descriptor structures, according to a set of declarations about how they are to be interpreted. This enables the system to encode all of the internal structures (e.g. the structure which represents a unit) in a form which can be manipulated as though it were a full-fledged description.
- A compiler which converts simple Match, Seek, and Describe expressions into corresponding INTERLISP record structure manipulations, reducing the overhead on those instances of these processes in which only simple operations are to be done. This should make it possible to preserve efficiency while writing much more uniform code, with no need to use explicit LISP manipulations of the structures. Use of the notions of compiling and compaction allows the conceptually correct but notationally expensive use of uniform metadescription to be supported without excessive running cost in the common cases.

- A uniform notion of *system events* which allows more general kinds of procedural attachment, and includes traps, triggers, and signals. Also, by including much of the INTERLISP interface in description form, it has become more uniform and understandable as well.
- A simplified syntax, in which indentation is used to express bracketing, eliminating the need' for most parentheses. It also uses "footnotes" for attaching meta-descriptions, and has simple set and sequence notations.
- Simplified notions of categories, inheritance chains, and agendas, which avoid some of the specific commitments made in KRL-0.
- Expanded facilities for sets, sequences, scheduling, time-dependent values, category hierarchies, matching information and multiple-worlds. These are all built up out of the simpler, uniform facilities provided in the kernel, but they represent a substantial body of standardized facilities available to the user.

We are currently exploring a number of different solutions to the address space problem. Until LISP systems with a larger address space are available, some sort of swapping mechanism will be necessary, but we see this as a temporary rather than . long- term problem.

The cycle of testing on KRL-1 will be similar to the one described in this paper, but with an emphasis on a smaller number of larger systems, instead of the multiple mini-projects described above. We feel that with KRL-0 we explored a number of important representation issues, but were unable to deal with the emergent problems of large systems. Issues such as associative indexing, viewpoints, concurrent processing, and large-scale factoring of knowledge can only be explored in systems large enough to frustrate simplistic solutions. Several programs will be written in KRL-I, on the order of magnitude of a doctoral dissertation project. Current possibilities include: a system for comprehension of narratives; a system which reasons about the dynamic state of a complex multi-process program, and interacts with the user about that state; and a travel arrangement system related to GUS (Bobrow et. al., 1977). Current plans include much more extensive description and documentation of the system than was the case with KRL-0.

We do not view KRL-1 as the final step, or even the next-to-last step in our project. In Bobrow and Winograd, 1977 (pp. 34-36) we discussed the importance of being able to describe procedures in KRL structures. Our plan at that time was to design a comprehensive programming formalism as part of KRL-I. In light of the shift of emphasis towards better understanding the aspects which we had already implemented, we have postponed this effort for later versions, still considering it one of the major foundations needed for a full KRL system. There remains the large and only vaguely understood task of dealing in a coherent descriptive way with programs and processes. It is likely that to develop this aspect will take at least two more cycles of experience, and as we learned so well with KRL-0, there is always much much more to be done.

References

Bobrow, D.G., Kaplan, R.M., Kay, M., Norman, D.A., Thompson, H., and Winograd, T., GUS, a frame driven dialog system. *Artificial Intelligence,* 1977 V 8. No. 2.

Bobrow, D.G. and Norman D.A., Some principles of memory schemata, in D.G. Bobrow and A.M. Collins (Eds.), *Representation and Understanding,* New York: Academic Press, 1975, 131-150.

Bobrow, D.G. and Winograd, T., An overview of KRL-0, a knowledge representation language. *Cognitive Science,* V. 1, No. 1, 1977

Hayes, P., In defense of logic, (Draft paper) 1977

Kaplan, R. A general syntactic processor. In R. Rustin (Ed.), *Natural language processing. New* York: Algorithmics Press, 1973.

Lehnert, W., Question Answering in a story understanding system, Yale University Computer Science Research Report #57, 1975.

Meldman, J.A., A preliminary study in computer-aided legal analysis, MIT project MAC TR 157, 1975.

Minsky, M., A framework for representing knowledge, In Winston, P. (Ed.), *The psychology of computer vision,* McGraw-Hill, 1975.

Newell, A., and Simon, H.A., *Human Problem Solving,* Prentice Hall, 1972.

Norman, D.A., & Bobrow, D.G. On data-limited and resource-limited processes. *Cognitive Psychology,* 1975, 7, 44-64.

Rubin, A.D., Hypothesis formation and evaluation in medical diagnosis (MIT-AI Technical Report 316). Cambridge: Massachusetts Institute of Technology, 1975.

Schank, R.C. (Ed.), *Conceptual information processing,* Amsterdam: North-Holland, 1975.

Schank, R., and the Yale AI Project, SAM -- A story understander, Yale University Computer Science Research Report #43, August, 1975.

Teitelman, W., INTERLISP reference manual. Xerox Palo Alto Research Center, December, 1975.

Thompson, H., “A Frame Semantics approach to Kinship”, ms. Univ. of California, Berkeley. 1976

Winograd, T., Frame representations and the declarative procedural controversy. In Bobrow, D.G. and Collins, A. (Eds.), *Representation and Understanding,* New York: Academic Press. 1975.

Winston, P., Learning structural descriptions from examples. in P. Winston (Ed.), *The psychology of computer vision* New York: McGraw-Hill, 1975.

THE DESIGN OF
THE PSI PROGRAM SYNTHESIS SYSTEM

Cordell Green

Artificial intelligence Laboratory,Computer Science Department,Stanford University

Keywords: automatic programming, program understanding, knowledge-based systems, natural language, program Inference, algorithm analysis, automatic coding, program synthesis, program modelling

Abstract

This paper presents an overview of the current state of the PSI automatic program synthesis system and dicusses the design considerations. The PSI system allows a user to specify a desired program in a dialogue using natural language and traces. PSI then synthesizes a program meeting these specifications. The target programs are simple symbolic computation programs in LISP.

PSI may be described as a knowledge-based program understanding system. It is organized as a collection of closely interacting modules, or experts in the areas of natural language, discourse, traces, application domain, high-level program modelling, coding, and efficiency. An implementation effort is underway and several modules are now working.

1. Introduction

This paper describes the research goals and system design of a knowledge-based automatic programming system, PSI (sometimes referred to as Ψ). The PSI program allows a user to interactively describe in natural language a desired program. PSI then synthesizes a program meeting the specifications. The PSI system is a group project being done at the Stanford University Artificial intelligence Laboratory. The personnel include David Barstow, Jerrold Ginsparg, Cordell Green, Elaine Kant, Brian McCune, Jorge Phillips, Louis Steinberg, and Ronny van den Wuevel. Former members include Avra Cohn and Bruce Nelson.

PSI deals with the world of symbolic computation (as opposed to numeric computation) and produces programs in LISP. -Examples of symbolic computation include list processing, searching, sorting, set operations, data storage and retrieval, and pattern matching.

There is a variety of programming applications that can be made up out of these kinds of techniques, including algebraic simplification, symbolic learning programs, simple data management systems, etc. We expect the class of programs PSI can write to grow as the project proceeds We are planning for PSI to be able to write a series of application programs that use increasing amounts of both low-level programming knowledge and of higher-level or application-specific knowledge. We present later in this paper an example of a specific learning program to be written by PSI.

The user specifies or describes the desired program through an interactive dialogue between the system and the user, where each is able to take the initiative in leading the discussion and asking questions as the situation requires. The intent is that the dialogue be as natural as possible to the user for the particular class of programs being produced. We allow two specification methods in the initial implementation. The principal method is the use of natural language, in particular a reasonably useful subset of English. A second specification method available to the user of PSI is *traces,* a Sequence of "snapshots" of the state of execution of the desired program. Effectively the user shows the system in a step-by-step manner how the desired program should work. and then the system writes the actual program.

PSI is organized as a knowledge-based system containing a great deal of information about the process of writing a program and about discussing a program with a user. The system is organized as a set of closely Interacting modules or experts. There are programmed modules for the following areas:

Parser-Interpreter
Discourse (including User Model)
Application Domain
Trace Understanding
Model Building (constructs a model from fragments)
Coding (pure programming knowledge)
Efficiency (Including algorithm analysis)

There is currently no explanation or natural language generation system, although such an expert is clearly needed.

The operation of the system may be said to fall into two (not entirely distinct) phases. The first is the model acquisition phase in which a high-level model of the desired program is built up through conversation with the user. In the second phase, an efficient program that meets these specifications is produced.

As of November 1976 an initial implementation of the PSI system has been completed and has successfully synthesized several programs from natural language dialogues. In particular, the two synthesis group modules, the coder and the efficiency expert, have synthesized many test programs. The parser-interpreter works properly on several of the target dialogues, including the one presented in this paper. The model builder can currently construct program models from a few dialogues. There are partial implementations for the other modules in the acquisition group, but they are not yet in a state to be interfaced or tested. In general, our progress is encouraging, and we hope to be able to report in later papers the details of the implementations and the successes and limitations of our design.

How does this system compare with others? At the time of its design there were no really comparable systems in existence, although a few are similar. The closest was Heidorn's natural language programming system [Heidorn, 1974], which was a specialized design for writing simulation programs. There are three other automatic programming research projects which rely on natural language, at MIT, ISI, and IBM Yorktown Heights. All of these projects have the same global goal of making programming easier. Since all of these projects, including ours, are continually undergoing redefinition, it is somewhat speculative to compare them, but we can try. Our system is distinguished from the others perhaps by its scope-- ranging from the use of traces as inputs to reliance on discourse expertise, domain expertise and some automatic analysis of algorithms for efficiency. In addition to using all of these parts, we are concentrating on finding solutions to the various problems of system integration. The other projects may be distinguished from ours as follows: At MIT the OWL project [Hax and Martin, 1973). emphasizes natural language. A second MIT project (PROTOSYSTEM I) deals with a system specially devised for the problems of data management and inventory control. Currently, these two efforts are not integrated into one system, At ISI [Balzer, 1974], there has been greater emphasis on acquisition of domain knowledge from English rather than having this knowledge built in, as in our system. That ISI effort focused on the program model acquisition phase rather than producing efficient code from the model. IBM [Mikelsons, 1975] has shifted major emphasis from synthesis of new programs to understanding of existing programs, utilizing Heidorn's system for the natural language processing. In summary, these efforts are complementary, each emphasizing somewhat different aspects of automatic programming. There is, additionally, much research on relevant independent subparts of the entire problem, ranging from natural language to analysis of algorithms. The research in related areas is too voluminous to discuss here, but two recent papers survey the field of automatic programming [Heidorn, 1976],[Biermann, 1976).

2. Research Objectives

Our major reason for attempting a synthesis of the many aspects of automatic programming was the belief that an attempt to integrate a total system would serve to focus research efforts. There has been much research on particular related sub-problems, e.g. program Inference from traces or examples, and program synthesis from formal specifications, but we felt that an overall framework was missing. Without knowing where and how the pieces would fit in, it was difficult to decide on optimal research strategies for the parts. For this reason we have chosen relatively long-term total system goals. We are evolving a framework to meet these goals.

In our efforts, we hope to establish the feasibility of this approach to automatic programming, and to answer some key questions, such as: How should a total system be organized? How much programming knowledge is needed to accomplish a particular task, and can this knowledge be codified in machine form? How critical are questions of efficiency, and can algorithm analysis techniques be automated in any useful way? Do there exist alternative natural ways of expressing programs that are better than current programming languages? We also hope to shed light on other research issues, although In a sense they are less pressing, due to the amount of effort already being expended on them by others. These areas include work on parsers for limited natural language and the use of inference and problem-solving techniques for program synthesis.

2.1 Desiderata

In order to accomplish our research objectives, it was necessary to carefully draw up a set of design constraints that was ambitious enough to force us to look at the important issues and yet limited enough to make the project feasible. The design constraints decided upon are as follows:

Program Specification: The specification process may be interactive. The range of specificity allowed the user is large, varying from one extreme in which the user gives great detail, to the other extreme in which the system uses its domain knowledge to suggest a complete program. More than one natural specification method will be available to the user (forcing the system to internally integrate different forms of program description). A program can be specified relative to an existing program (this is similar to a program modification capability, but in our system the modification occurs at the program description level).

Languages for Program Specification: These should include examples, traces, and a reasonable subset of English.

User Interaction: Both the system and the user will be able to explain the program, ask necessary questions, and provide answers. A mixed-initiative dialogue will be possible.

Documentation: The system shall be able to explain its operation and its finished program by answering "why" and "how" questions at each significant level. These reasons should form the basis for a later system that would produce readable documentation. This would require additional information on how to convert reasoning chains into readable prose. We are not currently working on this problem.

User: The user must have in mind a general idea of the desired program, should be a programmer, and should be able to stay within a limited subset of English. If the user's general view of the program is at odds with the domain knowledge of the system, then the user will have to be very specific, but still will need no detailed knowledge of the target language.

System Model of the User: The system will have a reasonable model of the user and of the discourse. The model will Include topics under discussion, degree of user initiative, discourse history, etc.

Class of Target Programs and Knowledge: They are, generally, symbolic computation programs. At the *lower* levels, the type of programming knowledge necessary to write them includes such subjects as list processing. set operations, sorting, pattern matching, etc. The system shall have domain knowledge about higher-level application or problem domains, e.g. data management programs, symbolic classification programs, simple learning programs, etc. Initially, the system shall know about one high-level domain-- that of concept formation programs. This domain is rather broad and presumes some simpler application domains, such as classification and symbolic (as opposed, say, to statistical) pattern recognition.

Variability in Target Programs: The system will allow a variety of algorithm and data structures in the programs being constructed. For low-level algorithms (such as table look-up or set intersection) there should be several varieties of algorithm and data structures that the system can synthesize for each problem. Data structures should include lists, arrays, records and references. Control structures should include iteration and recursion. At higher levels, the system should be able to produce many significantly different types of programs (although we don't quite know how to characterize this desired variety except to list typical target programs).

Efficiency of Target Programs: The programs produced should be efficient. i.e. comparable to those produced by an average programmer. This will require some algorithm analysis capability beyond conventional optimization techniques.

2.2 Assumptions about the Future World of Programming

Underlying this set of constraints and choices about what abilities are important to include and to omit in planning an automatic programming system for the future, there are a few assumptions about what the programming environment of the future will *be* like.

Perhaps the most significant assumption is that much higher-level languages will come into use in our interactions with computers. Limited English is a very high-level language suitable to certain applications. (This does not imply that an algorithmic language such as ALGOL has no place, but merely that it can often be successfully replaced.) When we are specifying a program in English, there is little need for the user to go through the target language program and make changes at that level. If the user wishes to modify a program, the desired modification would be expressed at the English level, not at the target language level. Reflection upon this point implies that our system need not necessarily modify object or target programs in the conventional sense of program modification. Instead, a high-level internal model or description of the desired program would be modified, and a new and efficient program could be produced from this model. Whether or not any of the old program would be used or modified would be a question of synthesis efficiency to be decided by the code-producing part of the automatic programming system. Compilers provide a good analogy. Users need to look at target code only for special purposes; most interaction is carried out at the source code level.

A second assumption is that some part of the art of computer programming can be made into a science or a theory of computer programming. In addition, we think that this theory can be a detailed, machine-usable theory of the process of programming. We are assuming that such a theory can be put into some form of rules, embodied in a computer program. Evidence that this is possible is provided by the machine-usable theories of sorting [Green and Barstow, 1976].[Green and Barstow, 1975] and theories of hash tables [Rich and Shrobo, 1974). The important questions are the amount of knowledge that will be necessary to carry out synthesis of a particular class of programs and the difficulty of embedding this knowledge in useful programs.

Although there is an optimistic feeling that conventional synthesis knowledge can be encoded in a machine-usable theory. we imagine that it will be more difficult to be able to automatically synthesize novel and very clever algorithms. That is, we do not expect to automatically devise really good sort algorithms from first principles. Instead, our system will be able to synthesize a reasonable sort routine or even a variation that is suitable for the task at hand, but it will know the principles of sorting and programming necessary for that derivation from the outset.

3. “A Sample Session with PSI

3.1 The Specification Dialogue

To show how our system will work we present here a dialogue In which the user describes a desired concept-formation program. This dialogue has been used for hand simulations to guide the design of PSI and should be representative of its capabilities. We have prepared 20 dialogues for 20 different target concept formation programs, to characterize the desired performance; the program discussed here is approximately mid-range in the difficulty level of the target dialogues. Other dialogues we prepared specified a symbolic pattern classification program and a simple data storage and retrieval program.

We would like to warn the reader that the dialogue is illustrative of our intent but should not be construed too literally. In particular, this dialogue endows PSI with an impressive English synthesis ability. While this makes for easier reading, our near-term implementation will use simpler standard responses, since synthesis of natural language is not the focus of our research. We also anticipate the necessity of more interaction to disambiguate the user sentences.

To understand this dialogue, we digress briefly to discuss in general what a “concept formation program” is. It is a type of simple learning program that takes as Input several objects that are instances of some concept and also several objects that are not Instances of the concept. As output, it produces an abstracted description of the concept so that given a new instance, it can classify that instance as representing that concept or not. A typical concept formation program, given the instances

part of the concept	*not part of the concept*
blue circle	**blue** square
blue oval	**red** ellipse

might conclude that the concept includes “blue, curved shapes” as features. Then, given a new instance "yellow circle," the program would say “not an instance of the concept.” In our dialogue the input instances are called **"scenes".** For further explanation of concept formation programs, the reader may peruse section 4.3. Concept formation programs can also be quite complex; for example Winston’s doctoral thesis [Winston, 1975) was based on such a concept formation program. Our current example is one that lies somewhere between these two extremes. Concept formation is in fact quite a good domain, since the programs involved call upon an interesting body of programming ideas and these programs can vary considerably in difficulty. They exercise our system well. Our target programs are typically several pages of moderately complex LISP code.

The dialogue is as follows:

[1] USER: Please write me a theory formation program.

[2] PSI: What would you like to call the theory formation program?

[3] USER: "TF".

[4] PSI: Tell me more about TF.

[5] USER: Where should I start?

[6] PSI: First, let's make sure we're talking about the same thing. I will assume that by "theory formation" you mean what Is frequently referred to as concept formation. To me, a concept formation program is a program which accepts scenes from a user end forms theories to explain or model the scenes. Is that close to what you mean?

[7] USER: Yes.

[8] PSI: There are now several things to talk obout: the input scenes, the structure of the theories, and the theory formation algorithm. Is that OK with you?

[9] USER: Yes.

[10]PSI: What is the structure of the input scenes?

[11]USER: An input scene is a collection of relations, each of which is a property name followed by the names of the objects over which the property holds.

[12]PSI: OK. I think we're done with scenes for now, Is it all right if we go on now to talk about the theories?

[13]USER: Yes.

[14]PSI: Should the program maintain several different theories at the same time, or only one theory?

[15]USER: I call theories "concepts". There should be only one of them. It should be a set of relations, each marked either as "possible" or "necessary".

[16]PSI: OK. I think we're done with concepts for now. Is it OK if we go on now to talk about how a scene is used to form and modify the theories?

[17]USER: Yes.

[18]PSI: Please describe the top level of TF.

[19]USER: It inputs a scene, tests whether it fits the concept, verifies the result of this test with the user, and updates the concept. Then it repeats the process.

[20]PSI: I take it you mean that it repeats until the user provides an exit signal.

[21]USER: Yes, stop when the user types quit instead of an input scene.

[22]PSI: OK. Now we need to talk about inputting a scene, testing whether the scene fits the concept, Verifying the result of the test with the user, and updating the concept. Would you describe the test that a *scene* fits the concept?

[23]USER: OK. The input scene fits the concept if all Of the relations in the concept that are marked "necessary" are part of the scene.

[24]PSI: What is the concept initially?

[25]USER: It has no relations stored initially.

[26]PSI: OK. Can we talk about how to verify the result of this test with the user now?

[27]USER: Yes. Output the result of the test, ask the user if this is correct, and input the user's response. The response should be either *correct* or wrong.

[28]PSI: What should be done if the user's response is neither?

[29]USER: Output "type correct or wrong" and input a new response.

[30]PSI: In what output format should the result of the test be output given?

[31]USER: output *fit* if the scene fit and *didn't fit* if it didn't.

[32]PSI: OK. I think we are done talking about verifying the result of the test. Can we talk about how to update the concept now?

[33]USER: OK. There are four cases, depending on whether the scene fit or didn't and whether the user said the program's guess was correct or wrong. If the scene fit and the user said the guess was correct, then every relation that is part of the scene but not part of the concept should be added to the concept and marked "maybe".

[34]PSI: [interrupts] Above you said that relations in the concept were to be marked "possible" or "necessary". Do you want to add "maybe" as another marking?

[35]USER: No, "maybe" means the same as "possible".

[36]PSI: Go on.

[37]USER: Where am I?

[38]PSI: You were just talking about four different cases for updating the concept. You had just talked about the case where the scene fit and the user said the guess was correct.

[39]USER: OK. The next case is If the scene fit, and the user said “wrong”: then find a relation in the concept which is marked “possible” and which is not in the scene. Change its marking to “necessary”. If the scene did not fit, and the user said correct, then do nothing. If the scene did not fit, and the user said wrong, then for each relation in the concept marked “necessary” that is not in the scene, re-mark it “possible”.

[40]PSI: In the case where the scene fit and the user said wrong, what if there is no description in the concept marked possible that is not in the scene?

[41]USER: Do nothing.

[42]PSI: Can we talk about input formats now?

[43]USER: OK. Input a scene as a list of descriptions, where each description is a list of the property name followed by its objects.

[44]PSI: OK. How should TF prompt the user for an input scene?

[45]USER: By typing ready. --

[46]PSI: That seems to complete the description of TF. Anything else?

[47]USER: No.

[48]PSI: Then I will begin writing the program now and let you know if further questions arise.

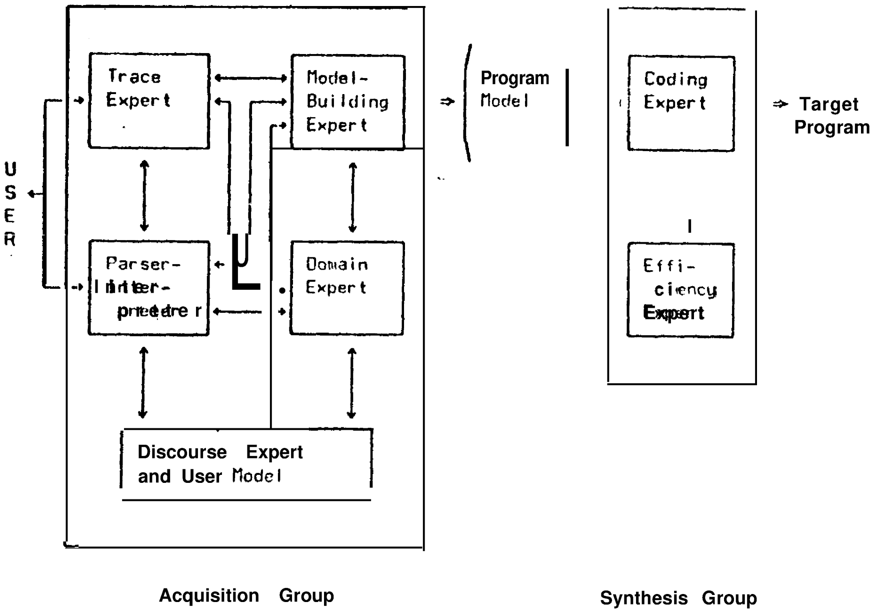
3.2 The Final Program

The resulting program produced consists of nine LISP functions in the hand simulation of PSI. We exhibit it here to give the reader a rough idea of the complexity of the output. See section 4.6 for a high-level algorithmic description of this program, our program model which serves as the principal intermediate representation.

```
(PROGRAM-X     .
  [LAMBDA NIL
    (PROG (NECESSARY POSSIBLE SCENE GUESS USER)
          (INITIALIZE)
       START
        . (TERPRI)
          (PRIN1 "Ready")
       STARTX
          (SETQ SCENE (READ))
          [COND
            ((ATOM SCENE)
              (COND
                ((EQP SCENE (QUOTE QUIT))
                  (RETURN NI L))
                (T (TERPRI )
                   (PRIN1 "Type 'quit' or a scene")
                   (GO STARTX]
          (SETQ GUESS (TEST: SUBSET (CDR NECE:SSARY)
                                    SCENE))
          (SETQ USER (VERIFY GUESS))
          (UPDATE SCENE GUESS USER)
          (GO START] )

(INITIALIZE
  [LAMBDA N I L
    (SETQ NECESSARY (LIST (QUOTE NECESSARY: )))
    (SETQ POSSIBLE (LIST (QUOTE POSSIBLE: ]))
```

```
(TEST: SUBSET
  [LAMBDA (X Y)
    (PROG (STATE)
          (SETQ STATE (CDR X))
      RPT (COND
            ((NULL STATE)
              (RETURN T))
            ((MEMBER (CAR STATE) Y)
              (SETQ STATE (CDR STATE) )
              (GO RPT))
            (T          --
              (RETURN NIL])

O'ER I FY
  [LAMBDA (TRY)
    (PROG (RESPONSE)
          (TERPRI)
          (PRIN1 (COND
                   (TRY "fit")
                   (T "didn't fit")))
          (TERPRI)
          (PRIN1 "correct or wrong? ")
      IN  (SETQ RESPONSE (READ))
          (COND
            ((EQP RESPONSE (QUOTE CORRECT))
              (RETURN T))
            ((EQP RESPONSE (QUOTE WRONG))
              (RETURN NIL))
            (T (PRIN1 "Type 'correct' or 'wrong'")
              (GO IN])

(UPDATE
  [LAMBDA (INPUT GUESS RESPONSE)
    (COND
      [GUESS (COND
               (RESPONSE (UPDATE: FIT: CORRECT INPUT))
               (T (UPDATE: FIT: WRONG INPUT]
      ((NOT RESPONSE)
        (UPDATE: DI DNT: WRONG INPUT) )
      (T T])

(UPDATE: FIT: CORRECT
  [LAMBDA (INPUT)
    (MAPC INPUT (FUNCTION ADD: POSSIBLE?] )

(ADD: POSS I BLE?
  [LAMBDA (X)
    (COND . .
      ((OR (MEMBER X (CDR NECESSARY)
           (MEMBER X (CDR POSS 1 BLE))))
      (T (RPLACD POSSIBLE
                 (CONS X (CDR POSSI BLE])

(UPDATE: FIT: WRONG
  [LAMBDA (INPUT)
    (PROG (PRED)
          (SETQ PRED POSSIBLE)
      RPT (COND
            ((NULL (CDR PRED))
              (RETURN T))
            ((NOT (MEMBER (CADR PRED)
                          INPUT))
              (RPLACD NECESSARY (CONS (CADR PRED)
                                      (CDR NECESSARY) ))
              (RPLACD PRED (CDDR PRED) )
              (RETURN T))
            (T (SETQ PRED (CDR PRED))
              (GO RPT])

(UPDATE: DIDNT: WRONG
  [LAMBDA (INPUT)
    (PROG (PRED)
          (SETQ PRED NECESSARY)
     RPT1 (COND
            ((NULL (CDR PRED))
              (RETURN T))
            ((NOT (MEMBER (CADR PRED)
                          INPUT))
              (RPLACD POSSIBLE (CONS (CADR PRED)
                                     (CDR POSSIBLE)))
              (RPLACD PRED (CDDR PRED))
              (GO RPT1))
            (T (SETQ PRED (CDR PRED))
              (GO RPT1])
```

For the reader concerned about a more "structured" program. the coder does indeed produce a cleaner intermediate program: an example is given in [Green and Barstow, 1976].



4. System Organization

We will first describe the overall design and then discuss each module in detail.

4.1 The System Design

The structure of the system is shown in the block diagram here. Arrows indicate paths of communication, and the experts appear in the frames.

The experts may be conveniently divided into two groups, acquisition and synthesis. The former group of experts acquires a model of the desired program from the user. The latter group synthesizes an efficient target program that satisfies the model. The majority of the interactions within the system are within each group, and the program model is the major communication between the two groups. (It turns out that this simplified view is not quite accurate, since, for example, certain questions from the efficiency expert can reach the user, but the division is convenient and mostly true.)

The program model can be seen as a very high-level language program, or alternatively, as a description of a program. It allows high-level procedures and information structures, as well as assertions about these items. We have developed an exact language for specifying this model. It is concrete enough that it can be interpreted, albeit slowly compared to the synthesized program. The interpreter for the model language has been implemented by Bruce Nelson [Nelson, 1976).

The specification phase consists of interaction between the user and the system. The flow of the dialogue is governed by the discourse expert, based on its measurements and estimations about the user's state of mind (the user model) and on the current topic under discussion. The discourse expert adjusts to changes in initiative (whether the user or the system seems to be leading the discussion at a given time), changes in topics (different users may want to discuss different aspects of their programs in different orders), and the degree of user control (different users may want the system to make different choices automatically). Since the dialogue is to be conducted in a subset of English, there is a parser-interpreter which parses the sentences and then partially interprets them into a relational structure. The ability to complete the interpretation of the incoming utterances (and to make reasonable responses or requests) requires the help of two other modules: the domain expert and the model-building expert. The domain expert interprets terms with domain-specific meanings, providing disambiguation information to the natural language expert. It provides guidance to the discourse expert regarding the user's apparent knowledge of the domain, and provides help to both the user and the model-building expert regarding possible algorithms and information structures to be used. In addition to the English dialogue input, input in the form of traces and examples is allowed. The trace expert interprets such inputs, receiving help in the process from the domain expert. Its output to the model builder is a partial description of the program. The model-building expert constructs a complete and consistent high-level program model by assembling fragmentary program descriptions from the user, the domain expert, and the trace expert, and by asking questions when necessary. It also acts as a source of information for the user and all of the other experts regarding the program model being constructed. Its knowledge includes very general high-level program description knowledge.

In the synthesis phase PSI takes the program model and derives a LISP program from it. The process is based on a body of systematized, codified programming knowledge to which the coding expert will have access, When faced with a choice, the coding expert is guided by the efficiency export, which evaluates the alternatives in terms of relative space and time efficiency. The efficiency expert uses heuristic analysis of algorithms techniques.

A comment on the modularity of our system may be appropriate at this point. In this presentation, and in our design, we have drawn somewhat over-simplified boundaries around the various modules or experts, Two purposes are served by this simplification: firstly, the system is made easier to understand, and secondly, both responsibility and credit for the implementation of particular system capabilities becomes easier to assign, in practice, such a clean and simple separation is not always possible or efficient. A closer examination will reveal that the required degree of communication between certain modules can be quite large, and that certain tasks may require cooperation between two or more experts. For this reason, the implementation is more complex than this design indicates; some knowledge is duplicated for the sake of efficiency, and some code "belonging" to one expert is physically located within another expert. For a further discussion of some of these issues, see [Barstow and Kant, 1976]. Now that the reader has been warned that the design presented-here is over-simplified and in some cases untested, we proceed with the exposition,

4.2 Parser-interpreter

The function of the parser-interpreter is to parse and partially interpret sentences into less linguistic, and more program-oriented terms. it consists of two parts, READER and SPAN. both described in [Ginsparg, 1976]. READER is a nondeterministic parser. The SPAN program converts the output of the parser into a more usable form called INT. INT is a relational structure, suited for describing programs. in INT some pronoun and noun references have been found. Verbs and nouns have taken on subject-specific senses. For example, "of" can be interpreted to be set membership, "is" can be interpreted to be data-structure definition, etc.

We now show a simple example of what the natural language expert does. Given the two sentences,

> "It should be a set of relations, each marked either as possible or necessary"
>
> "The scene fits the concept if all of the relations in the- concept that are marked "necessary" are part of the scene"

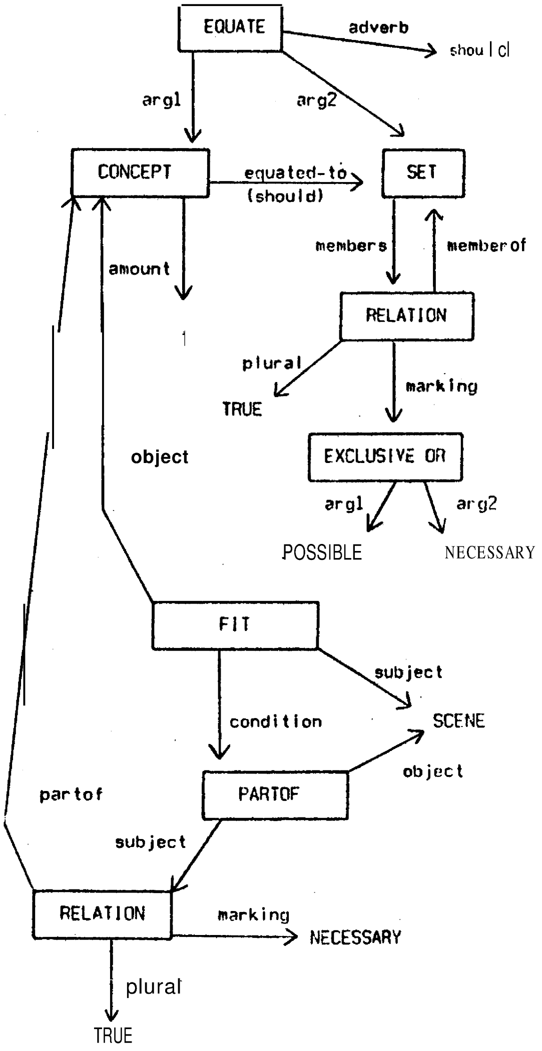
it produces the following parses:

```
(BE (NN SHOULD)
    (SUBJECT 1 T)
    (OBJECT
       (SET A
           (OF (RELATIONS
                   (DESCRIPTION EACH
                    (MARK PN
                          (OBJ ECT NOUNDUM2)
                          (MARKI SG (OR (EITHER)
                                         POSSIBLE
                                         NECESSARY
                                       )))))))))}
```

```
(FIT NN
     (SUBJECT (SCENE THE) )
     (OBJECT KOSCEPT THE) )
     (IF (BE NN
             (SUBJECT (ALL
                        (OF (RELATIONS THE
                                (IN KOSCEPT THE) )
                                (MARK NN
                                      (OBJECT THAT)
                                      (MARKING NECESSARY)
                               )))))
             (OBJECT (PART (OF (SCENE THE) )))))}
```

where NN and PN are tense markers.

From this parsed form, it produces the following INT form.



The natural language programs interact with the dialogue expert, domain expert, and model builder, All of these experts play a role in the disambiguation of the input. Take, for example, the sentence

> *"It* inputs a scene, tests whether *it* fits the concept, verifies the result with the user, and updates the concept. Then *it* repeats *the process."*

The references for the italicized parts may be found as follows: the first *if* is solved by the parser-interpreter and dialogue expert by knowing which topic was just asked about: the second *it* is solved by the domain expert using knowledge of what Can fit a concept; the last *it* and *the process* use knowledge from the domain expert and model builder about what a concept-formation program can do, and what constitutes a process.

4.3 Trace Expert

The trace expert allows the use of an alternative input language to specify a program. A trace is a series of snapshots of a program's execution. Example input/output pairs are treated as subcases of traces. The trace expert is strongly oriented toward inductive inference; i.e. it does more ‘guessing than natural language input requires, which in turn is more than conventional programming language input requires.

Let us illustrate how one would input a trace of the ‘desired program to the system. The user gives a sequence of inputs and internal states. In the example below, these states and inputs correspond to the formation of the concept "tower". The input consists of two parts, a scene and its *category*. The internal concept consists of two parts, necessary and *possible*.

The trace shown here is shortened and specifies a simpler program than the TF program specified in the dialogue. (For example, TF Includes the ability to classify a new input and ask the user if that is correct.)

Note that in this case, a trace is a convenient way for a human user to communicate the general idea about how a program works. (It is a good way for the program to reveal to the user how it works.) In this case, and in certain others we have studied, traces are more natural than limited English or programming languages, although this is not universally true. For discussion see [Green et al, 1974].

Since there is no graphic input available, the actual form of the trace input will look more like the following excerpt from a dialogue specifying the TF program.

The concept is initially null. The scene

(Block a)
(Block b)
(Supports b a)

is input to the TF program. Match the scene and the concept. The scene fits the concept. Output "fits" to the user. Prompt the user. The user replies "correct", and the concept becomes

((Block a) possible)
((Block b) possible)
((Supports a b) possible)

Assume now that the input *scene* is

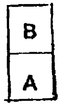
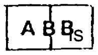
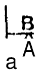
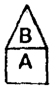
(Block a)
(Block b)

Match the scene and the concept. The scene fits the concept, so output "fits" to the user and prompt her for a reply. The user replies "wrong" so the concept becomes

((Supports a b) necessary)
((Block a) possible)
((Block b) possible)

The trace expert is being designed and implemented by Jorge Phillips. Its operation is described in more detail in [Phillips, 1976]. It is the most recent addition to the PSI system. The trace expert uses the natural language parser and interpreter on the English sentences. From these and special knowledge of input formats, a sequence of state-characterizing schemata is produced. From these, a set of program model fragments is inferred and sent to the model builder.

The trace expert has several new and significant characteristics. It allows the use of high-level traces expressed in natural language. It also incorporates into the trace the use of example input-output pairs. Another feature is that it draws upon a large knowledge base, incorporating knowledge of Input formats and knowledge about inference of algorithms and data structures. Finally, it utilizes knowledge from the domain expert, which we feel is an important source of constraint in a heavily Inductive inference oriented system.

	INPUT		*CONCEPT*	
	SCENE	CATEGORY	NECESSARY	POSSIBLE
(1)	none	none	empty set	empty set
(2)		**is a tower**	empty set	(BLOCK A) (BLOCK B) (SUPPORTS A B)
(3)		is *not* a tower	(SUPPORTS A B)	(BLOCK A) (BLOCK B)
(4)		is not a tower	(SUPPORTS A B) (BLOCK A)	(BLOCK B)
(5)		is a tower	(SUPPORTS A B) (BLOCK A)	(BLOCK B) (PYRAMID B)

4.4 The Discourse Expert (including the User Model)

The discourse expert (or dialogue expert) allows for a very flexible interface with the user. This flexibility is one of the core issues of automatic programming; how well can a computer system bend itself to the current task and user, rather than vice-versa? . The discourse expert is being designed and implemented by Louis Steinberg [Steinberg, 1976].

The function of the discourse expert is to model the user, the dialogue, and the state of the system. Using these models the discourse expert selects appropriate questions or statements to present to the user. It determines whether user or expert has the initiative and at what level and on what subject. Additionally, it helps to disambiguate the natural language input by keeping track of the dialogue context.

The parts of the discourse expert include a semantic structure and a discourse tree. The semantic structure encodes the semantic relations among the possible topics of discussion. These relations, such as super and subalgorithm, and use of data structure by algorithm define the set of potential paths the dialogue may take, in that any relation is a potential path for changing topics. The Dialogue Tree encodes the path actually taken by the dialogue. Finally, there is a user model that includes what topic(s) the user is talking about, how much initiative the user has taken, how confused the user is, etc.

We can illustrate the discourse expert's operation by example. In the dialogue given earlier, we have a reasonably unconfused user, but one who asks for some guidance in the discussion. In this dialogue the discourse expert generally suppresses questions, since the user's statements seem to fit internal models of the domain. The first significant question "Tell me more about TF" allows the user to take the initiative and guide the discussion. At first the user is unsure and asks where to start, so the discourse expert delivers a topic outline provided by the domain expert and suggests beginning with the first topic, the form of the input. A heuristic is that users and programmers often like to begin with the format of the input. This indeed works and the discussion is underway.

After digesting the input, the system suggests the next topic. Typically, with complex programs, programmers get lost (lose context) and some assistance is helpful. This occurs in more dramatic form in line 37 where, in the middle of a complex algorithm the user asks, "Where am I?".

Interruptions occur in line 20 and 40, where the system perceives a fairly serious omission. In the first case, a missing exit test implies an infinite input loop, and the system suggests that the user provide a 'stop' command. In the second instance, the user is listing what to do in each case and forgets to say what to do in the last case. The discourse expert decided that an interruption in the form of a question would be less disruptive at this time than in a later context.

At any moment, there are many questions being asked internally by the various experts. The discourse expert selects from among these the ones that will be presented to the user. Some heuristics include the fact that a supertopic subsumes a subtopic, and that usually, the domain expert's questions subsume any related general programming questions. Thus, the questions tend to be phrased in terms the user finds meaningful, i.e., terms related to the problem domain, rather than the more programming-oriented terms used by the model builder.

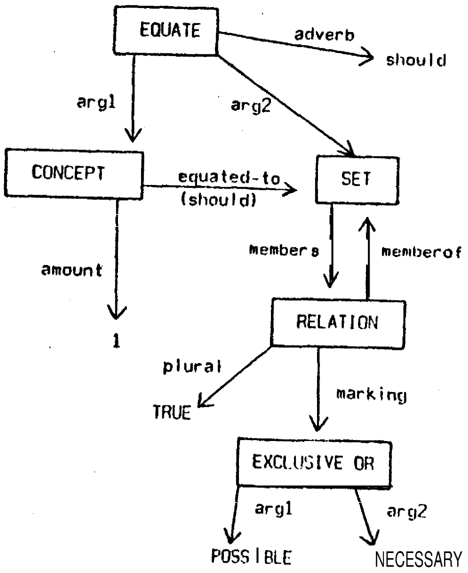
4.5 The Domain Expert

The domain expert functions in several ways in the PSI system. Its principal purpose is to take partially interpreted sentences as input from the natural language expert, and produce as output more specific fragments of program description, which are given to the model builder to assemble. In this process, the information is converted from a linguistically oriented form to a more program-usable form, and much of the ambiguity is removed. The domain expert also makes domain-specific inferences to fill in missing information. The output from the domain expert is in a language called FRAGMENTS. FRAGMENTS is a program description-oriented language, but is not specific enough to be interpretable as a program in any ordinary sense. The domain expert is described in more detail in [Steinberg, 1976]. Avra Cohn, Louis Steinberg, Jorge Phillips, and Brian McCune all helped in defining and formulating the domain expert. The current design and implementation is being done by Ronny van den Heuvel.

As an example of the domain expert's performance, suppose it received the INT form of the sentences

> "It should be a set of relations, each marked either as possible or necessary."
>
> "The scene fits the concept if all of the relations in the concept that are marked "necessary" are part of the scene."

For the sentence referring to concepts, "It should be a set of relations, each marked either 'possible' or 'necessary,'" the input to the domain expert in INT form is



The program model fragments produced as an output provide the following information:

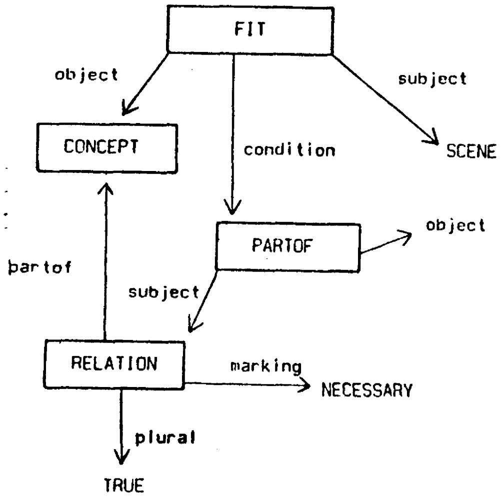
> A Concept is an information structure. It is a set. There is only one concept. It changes during the execution of the program. Elements of the set are *marked features.*
>
> *A marked feature* is a pair (plex) consisting of a *mark* and a *feature.* A mark is a primitive. There are two types of markings, may-marking and must-marking.
>
> A *feature* is a pair (plex) consisting of a relation name and its argument list.
>
> .
>
> .
>
> .

The above information is coded in the special internal fragment language and really looks more like

```
(NAME CONCEPT
CLASS INFORMATION STRUCTURE
TYPE SET
SIZE (MINIMUM 0)
DYNAMIC YES
ELEMENT MARKED-FEATURE)
```

and so on. Observe that the domain expert has produced a much more precise description of the data structure information "concept", adding information of its own that was not previously known about concepts and features. Often, as in this case, much of the detail omitted from the input dialogue can be supplied by the domain expert. Also note that fragments do not correspond in a one-to-one fashion to sentences, but can be the result of several sentences *or* of part of a sentence.

As a second example, the INT form for the sentence "The *scene* fits **the concept if** all the relations that are marked 'necessary' **are part of** the scene," is



The output produced is roughly,

fit (input scene, concept) is defined to be

$$(\forall\, rel \in concept)\,Marking(rel)=\text{"necessary"} \Rightarrow part\text{-}of(rel, input\ scene)$$

Again, it takes a slightly different internal form,

The domain expert serves other purposes as well. It helps to disambiguate input sentences by answering questions for the natural language expert during the course of parsing and interpreting the sentence. Some disambiguation is done interactively with the natural language expert and further disambiguation occurs after the natural language expert is through and INT has been passed to the domain expert. Similarly, the domain expert answers questions posed by the trace expert, the model builder and even the efficiency expert. To whatever extent possible, the domain expert prevents the nonessential questions from propagating to the user. The domain expert is a very important information source, and without one, an automatic programming system has significantly less capability and utility.

In addition, the domain expert should enable the system to carry on a relatively coherent dialogue by phrasing questions and comments which are in domain-specific terms and which are directly relevant to the user's particular idea of the domain. Finally, the domain expert should be able to inform the discourse expert about how well the user's apparent idea of a program matches that of the domain expert, thereby helping to judge how well the system is following the user (and how well the user is following the system).

Let us now look at the role of the domain expert in the sample dialogue. We can see the use of general knowledge about, concept formation when input becomes the topic of conversation:

[8] PSI: There are now several things to talk about: the input scene, what the theories are like, and how the scene is used to form and modify the theories. Is that OK with you?

Since the concept formation expert seems to be following the user fairly well, the discourse expert assumes that responses suggested by the concept formation expert are the most apt. As part of its information about concept formation, the expert knows about the nature and purpose of inputs to typical concept formation programs, and in particular that inputs are information structures. The form of information structures is of interest, since it helps specify what sort of concept formation program the user wants. Therefore, a question is posed to the user:

[10] PSI: What is the structure of the input scene?

Further information about inputs enables the concept formation expert (and the system) to follow the rest of the discussion about inputs.

It is our intent to separate the domain knowledge as much as possible from the rest of the system. This modularity should help to maintain the generality of the rest of the system, so that a new domain can eventually be added. Accordingly, this expert is organized with most of the knowledge encoded in an easy-to-modify relational data structure.

4.6 The Model-Building Expert

The model-building expert (or simply "model builder") contains high-level, general programming knowledge and rules for assembling fragments of program description into a complete algorithm and information structure model. These fragments come from the domain expert or trace expert. In the event that the domain expert or trace expert cannot help in interpreting INT, the model builder can attempt to perform the same interpretation and disambiguation functions as the domain expert, although the task is more difficult without domain knowledge. The model builder is being designed and implemented by Brian McCune [McCune, 1976).

The program model (also called the algorithm model) itself may be thought of as a very high-level program, allowing annotation, such as comments about the information structures, the constraints on the program, which parts of the program use what data, likely sizes of data sets, and sometimes estimates about the probable outcome of tests used for branching. The high-level data structures are' called Abstract Information Units (AIUs) and include correspondences (mappings), collections (sets, lists, ordered sets, and multisets), plexes (record structures), booleans, and strings. The high-level procedural units, Abstract Control Units (ACUs), Include partially ordered sets of actions, tests, cases, loops, and many high-level operations on the AIUs (e.g., take the inverse of a correspondence). The model builder's programming knowledge occurs at two levels: (1) the syntax and semantics of the primitives of the modelling language and (2) high-level knowledge built upon this base. This second type of knowledge includes ways of expressing common constructs such as names of objects, relations, data which are input, input-and-process loops, and correspondences.

To clarify the notion of a program model, Brian McCune has prepared a simplified program model for the TF program, which Is shown here. This example represents about half of the program model; all information structure descriptions, set sizes, branching probabilities, cross-references. and other annotations have been omitted. Also, we do not have sufficient space to define the terms used. Still, we feel that from a perusal of the version given here the reader may gain a better understanding of the procedural portion of the model.

THE PROCEDURAL PART OF THE PROGRAM MODEL FOR TF

```
Input. and_ process:
    Initialize:
        parbegin
        Initialize. labels:
            necessary ← new_ primitive(necessary prototype);
            possible ← new- primitive(possible. prototype);
        Initialize. concept:
            concept ← new_correspondence(concept_ prototype,∅);
        parend;
    Input._and_process._body:
        loop until exit;
        Input .and. test .for. exit:
            Input:
                input. data ← input(input. data. prototype,"Ready","Type 'Quit' or a scene.");
            Test-for-exit:
                if input_data = "Quit" then exit else distinguish(input data,relations);
        Process:
            Classify:
                Test._fit:
                    test_fit_ result ← concept⁻¹(necessary) ⊆ relations;
                Verify:
                    Output. test fit_ result:
                        inform user(if test. fit. result then "Fit" else "Didn't fit");
                    Input-verification
                        user .response ← input(user response prototype,
                            "Is this correct or wrong?","Type 'Correct' or 'Wrong'.");
            Undate:
                case (test_fit_ result,user. response) of
                    begin
                    if test. fit result A user_ response = "Correct" then
                        V rel ∈ relations | rel ∉ domain(concept)
                        do establish .correspondence(concept,rel → possible);
                    if test. fit. result ∧ user. response = "Wrong" then
                        if 3 rel ∈ concept⁻¹(possible) | rel ∉ relation s
                        then establish correspondence(concept,rel → necessary);
                    if ¬test fit result A user response = "Correct" then nil;
                    if ¬test fit result A user response = "Wrong" then
                        V rel ∈ concept⁻¹(necessary) | rel ∉ relations
                        do establish. correspondence(concept,rel → possible);
                    end;
        repeat;
    Exit:
```

As an example of the operation of the model builder, suppose that it received the model fragments discussed in the domain expert section. Recall that one set of fragments was the definition of a concept as a collection of marked features. The model builder uses this as the basis for constructing an Abstract Information Unit. In this case the concept is represented as a correspondence between relations and their labels, "necessary" and "possible". The correspondence representing the concept is marked as many-to-one, with the set of relations in the concept as the domain, and the set of two labels as the range. Correspondences can later be implemented by the coder in many ways, e.g., using. tables, association lists, functions, etc. The Internal representation for this AIU is

```
(NAME CONCEPT
   CLASS AIU
   TYPE CORRESPONDENCE
   SUPER-AIUS NO
   INSTANCES (CONCEPT-INSTANCE)
   DOMAIN-AIU RELATIONS-IN-CONCEPT
   DOMAIN SIZE (MINIMUM 0 MAXIMUM NIL
                MEAN NIL VARIANCE NIL)
   RANGE-AIU LABELS
   RANGE SIZE (MINIMUM 0 MAXIMUM 2
               MEAN NIL VARIANCE NIL)
   WHAT-TO-ONE MANY
```

Another type of high-level structure used by the model builder is the Abstract Control Unit, which provides control of procedure flow. As an example of the use of ACUs, consider our other sentence stating that the scene fits the concept if all relations in the concept that are marked "necessary" *are* part of the scene. The input representing this sentence is the model fragment discussed in the previous section,

fit (input *scene,* concept) is defined to be

($\forall$ rel $\in$ concept) Marking (rel)="necessary"
$\Rightarrow$ Part-of(rel,input scene)

This fragment (along with knowledge from other fragments, including the information structure selected for the correspondence) yields the high-level algorithm structure which we may describe as

test-fit-result ← [Concept^{-1} (necessary) $\subseteq$ relations]

This statement then forms the test-fit section of the procedural part of the model shown earlier. This means that the result of the "fit" test is true if the inverse mapping of the range element "necessary" under the correspondence "concept" is a subset of the relations of the input scene. Observe that the test has been mapped into suitable high-level operations (inverse and subset) on the high-level information structure, and that the result will be remembered in the Boolean test-fit-result for later use.

Once this program model has been completed to the satisfaction of the model-building expert, control is turned over to the coding expert for synthesis of a LISP program which satisfies the descriptions in the model. Note, however, that this process is not necessarily sequential: the coding expert or the efficiency expert may have questions for the model-building expert about possible further assumptions which, if considered before coding occurs, would lead to a better program.

The model-building expert also acts as a source of information for the user and the other experts in the system. An example of its providing help to the dialogue expert is the disambiguation of referents in the dialogue. it does this by keeping track of the most recently discussed portion of the model and determining which possible referents are plausible in terms of the semantics of the model. An example of help to the efficiency expert might consist of providing a list of all the accesses of an information structure so that an efficient data representation can be selected.

The model-building expert must check that the final program model produced has no obvious inconsistencies, either internally or as it relates to the desires of the user. Consistency checking at each step will thus help assure a program model which is both legal and correct. This capability is useful for program modification, where small local changes can cause interactions with other parts of the model and have tremendous implications for later implementation choices.

4.7 The Coding Expert

The purpose of the coding expert or "coder" is to take as input the program model and produce as output an efficient target language program that satisfies the program model. The coder interacts closely with the efficiency expert in this task. The knowledge base for the coder consists of relatively "pure," domain-independent knowledge about the process of programming. The Coding Expert is being designed and implemented by David Barstow [Barstow and Kant, 1976].

To show by example how it works, consider the high-level information structure, a correspondence, used to represent a concept. The coder selects a proper data structure to implement the correspondence. Recall that the correspondence maps the set of necessary relations into the term "necessary" and the set of possible relations into the term "possible." By analyzing all uses of the correspondence, as discussed in the next section on the efficiency expert, an appropriate data structure is chosen. The set of "necessary" relations is represented by a linked list, and that list is referenced by being the value of the atom "N". In order to facilitate insertions and deletions, a special element is kept as the first element of the list. Thus (CDR N) is a pointer to the list of elements. The "possible" set is represented similarly.

Consider next the test to see if the input fits the concept. Recall that the condition for successful fit was that the set of "necessary" relations must be a subset of the input scene relations. So a subset test must be constructed. The input scene is represented as a linked list and is the value of a variable, "I". The completed program for the test is

```
(PROG (STATE)
      (SETQ STATE (CDR N))
  RFT (COND
         ((NULL STATE)
            (RETURN T))
         ((MEMBER (CAR STATE) I)
            (SETQ STATE (CDR STATE))
            (GO RFT))
         (T
            (RETURN NIL))))
```

The coding expert uses a large knowledge base of programming techniques to generate, in successively finer detail, alternative algorithms and data structures that satisfy its goals. These techniques, along with simple knowledge bases, are discussed in [Green and Barstow, 1976],[Green and Barstow, 1976]. The knowledge and methodology, as expanded in these papers and in the implementation of the coder, constitute our first attempt at a theory of the process of programming,

To exemplify the coding process, consider the subset operation above. One method of testing whether set A is a subset of set B, is to test whether all elements in A are members of B. This may be done by enumerating over A, testing each element for membership in B. In order to enumerate A, an enumeration order is selected, a method for saving the state of the enumeration is selected, and then a loop is written, having appropriate body, initializer, incrementer, and termination test. Each part is selected intelligently, e.g., if the set is a linked list, the enumeration order is front-to-back, and the state saving scheme is a pointer that moves along the list, and if the target language is LISP, then the termination test is a test that the pointer has the value "NIL."

The coder proceeds through these steps of generating alternative data structures and algorithms. At choice points, the alternatives are passed to the efficiency expert for recommendations about which path to choose. Since an exact analysis could not be completed until all code is finished, this efficiency analysis is heuristic in nature, using estimates on the efficiency of the process on subparts not yet written.

The knowledge base for the coder is in the form of a set of rules. Some examples of such rules, given here in informal English, are:

> "One technique for representing a correspondence is to use a collection where the elements are plexes of size 2, with one part being the domain element and the other part being the range element."
>
> "In order to write an enumeration for an explicitly represented collection, first determine the order for generating the elements, then select an appropriate scheme for saving the state of the enumeration between the production of the elements, then write the body, initializer, Incrementor, and termination test."
>
> "In LISP, the function CAR applied to a pointer to a list returns the first element in the list."

A more detailed discussion of the nature of these rules can be found in [Barstow and Kant, 1976]. For this overview a short summary will suffice. it appears that approximately a thousand rules will be necessary for the task we have chosen, and the level of competence at which we aim. The rules span many levels, from high-level concepts such as correspondences, to low-level LISP-specific concepts, such as knowledge about CONS-cells. The rules seem to fail into two broad classes: those dealing with general programming techniques and those dealing with LISP-specific details. Currently we estimate that about two-thirds of the rules are general and the rest are specific to the LISP language. However, we expect the set of LISP-specific rules to stay fixed, while the data base of general rules grows.

4.8 Efficiency Expert

The function of the efficiency expert is to select efficient algorithms and data structures, from the alternatives offered by the coding expert. The tools available are analysis of algorithm techniques, heuristics, and simulation. The efficiency expert uses primarily the first two methods. Based upon sizes of data sets, probabilities of the outcomes of tests, algorithm and data structures, and costs of operations, it is able to calculate symbolic space-time cost functions of competing alternatives. The efficiency expert is being designed and implemented by Elaine Kant [Barstow and Kant, 1976].

As an example of the operation of the efficiency expert, consider the choice of the data structures to represent a concept. The concept is a correspondence between the necessary relations and the label "necessary", and between the possible relations and the label "possible". There are many ways to represent a correspondence, including various forms of tables, bit maps, or functions. Let us examine the choice between only two alternatives. We shall call the two choices "set of pairs" and "one set per range element". To illustrate, let the relations be simple propositions such as blue, triangle, curved, tilted, red, etc. Then we can illustrate the two cases as follows:

(a) one set per range element

(blue, curved, tilted) ↔ Necessary
(square, triangular, red) ↔ Possible

(b) set of pairs

((blue necessary) (curved necessary) (square possible)
(triangular possible) (tilted necessary) (red possible))

Note that (b) corresponds to a form of association list or property list (with parentheses added for clarity). Note that in (a) we do not specify how the sets are associated with their labels; the sets may be the value of a variable corresponding to the label, or the label could be a hash link from the list, or the set could be on the property list of the label, or the set and label could be a plex, etc.

Next, consider how the efficiency expert chooses between these two data structures. Since the data structure is accessed in many places, the calculation actually made by the efficiency expert is rather complex. For our example, we will simplify matters by considering only one access to the data, the "too many necessaries" update.

The "too many necessaries" case occurs in the learning portion of the TF program, in which the concept being learned is incorrect and must be modified. Because too many relations were marked necessary, a *scene* that was a correct instance was rejected as not being an instance of the concept. Accordingly, some of the "necessaries" must be made into "possibles". In particular, any relations that are not in the scene are not necessary, so any such relations in the concept must be changed to possible.

In the model, this update action is represented roughly as

```
(∀ rel ∈ concept$^{-1}$(necessary) such that rel ∉ input
    do establish-correspondence (concept, rel, possible)
```

For each possible data structure implementation, the coder produces the appropriate algorithms in a special intermediate language used for analysis. These analysis language programs are passed to the efficiency expert to make the cost calculation. The two programs passed to the efficiency expert look something like:

(a) one set per range element

```
(Forall x in Nec. such that x ∉ Input
    do move x from Nec. to Poss.)
```

(b) set of pairs

```
(Forall pairs <x,y> in Concept
    such that [y = Nec. and x ∉ input]
      do re-pair(x,possible))
```

Thus in case (a) we must loop through the "necessary" set, test for membership in the input-and then move elements from one set to another. In case (b) we loop through the "concept" set (of pairs) and if an element is paired with necessary and also satisfies the test for membership in the input, then pair the element with possible instead of necessary (i.e., re-label x).

To do the analysis, the efficiency expert needs to know the size of the sets that are enumerated, the cost of enumeration, the probability of each test being true, and the cost of each operation. Thus in (a) we need the size of the set "necessary", the cost of the test "x ∉ Input" (which might require knowing the size of input), the likelihood of that test being true, and the cost of the move operation. In case (b) we need to know the size of the set "concept", the cost of enumerating it, the probability of truth of the conjunction of the two tests, "x paired with necessary" and "x ∉ input", and the cost of changing the label. From this information, the cost of the update operation is estimated for each alternative data structure choice.

This provides the efficiency expert the basic information with which to calculate the cost of each method, since we can now predict the number of times each operation is executed and the cost per operation. But the calculation is somewhat more complex than is apparent from this example. Remember that this is only one use of the data structure; it is accessed in at least four other places in the program. The cost of each access must also be part of the calculation. However, the exact costs of each sub-operation are often not known exactly, since further synthesis by the coder may be required to know how sub-operations are implemented. So these costs must be estimated. Also, the cost includes the space as well as time costs, so the calculation must take into account the (sometimes changing) sizes of the data structures extant during the algorithm. The knowledge for probability estimates comes from the data supplied by the domain expert or by the user. However, these probabilities may not be in the right form, since transformations made by the coder can change the program structure. In this case, the probabilities must also be transformed. If no probability information is available, the probabilities may be treated as variables In which case it may be possible to compare symbolic cost functions.

For the example considered above the probabilities were estimated by the domain expert, and the space-time cost function turned out to be a polynomial function of the sizes of the sets of Concept, Necessary, and Input. For reasonable assumptions about the size of these sets, the program selects choice (a), one set per range element.

5. Conclusions

In conclusion, we have specified some of the desired capabilities of an automatic programming system. We have created an overall rough system design to meet these specifications. An implementation effort is underway and several key parts of the system are working. It is too early to make an evaluation of our goals, designs, and implementation at this time.

6. Acknowledgements

I would like to thank Louis Steinberg, Jorge Phillips, Bruce Nelson. Brian McCune, Elaine Kant, Jerrold Ginsparg, Avra Cohn, and David Barstow for their contributions to the design of PSI and their assistance in the preparation of this paper.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206. Computer time was provided by the Information Sciences Institute of the University of Southern California and the Artificial Intelligence Center of Stanford Research Institute.

7. References

[Balzer, 1974)
Balzer, Robert, Greenfeld, Norton, Kay, Martin, Mann, William, Ryder, Walter, Wilczynski, David, and Zobrist, Albert, "Domain-Independent Automatic Programming", in Rosenfeld, Jack L., editor, Software, *Information Processing 74: Proceedings of IFIP Congress* 74, Volume 2, American Elsevier Publishing Company, Inc., New York, New York, 1974, pages 326-330.

[Barstow and Kant, 1976]
Barstow, David R. and Kant, Elaine, "Observations on the Interaction between Coding and Efficiency Knowledge in the PSI Program Synthesis System", Second International Conference on Software Engineering, San Francisco, California, October 1976.

[Biermann, 1975)
Biermann, Alan W., "Approaches to Automatic Programming", in *Advances in Computers,* Volume 15, Academic Press, 1975.

[Ginsparg, 1976]
Ginsparg, Jerrold, "A Parser and interpreter for the PSI System", in preparation.

[Green and Barstow, 1976)
Green, Cordell, and Barstow, David, A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System, Memo AIM-258, Artificial Intelligence Laboratory, Report STAN-CS-75-476, Computer Science Department, Stanford University, Stanford, California, January 1975; to appear in Elcock, E. W., and Michie, D., editors, *Machine Representations of Knowledge,* D. Reidel Publishing Company, Dordrecht, The Netherlands, 1976.

[Green and Barstow, 1975)
Green, Cordell, and Barstow, David, "Some Rules for the Automatic Synthesis of Programs", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Volume I, Artificial intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 232-239.

[Green et al, 1974]
Green, C. Cordell, Waldinger, Richard J., Barstow, David R., Elschlager, Robert, Lenat, Douglas B., McCune, Brian P., Shaw, David E., and Steinberg, Louis I., Progress Report on Program Understanding Systems, Memo AIM-240, Artificial Intelligence Laboratory, Report STAN-CS-74-444, Computer Science Department, Stanford University, Stanford, California, August 1974.

[Hax and Martin, 1973)
Hax, Arnoldo C., and Martin, William A., "Automatic Generation of Customized, Model-Based Information Systems for Operations Management", in Morgan, Howard Lee, editor, Proceedings of the Wharton Conference on Research on Computers in Organizations, Data Base, Volume 5, Numbers 2-4, Winter 1973, pages 140-144.

[Heidorn, 1974]
Heidorn, George E., "English as a Very High-Level Language for Simulation Programming", Proceedings of a Symposium on Very High-Level Languages, *SIGPLAN Notices,* Volume 9, Number 4, April 1974, pages 91-100.

[Heidorn, 1976]
Heidorn, George E., "Automatic Programming through Natural Language Dialogue: A Survey", *IBM Journal of Research and Development 20, 4* (July 1976), pages 302 - 313.

[McCune, 1976]
McCune, Brian P., "The PSI Algorithm Model Builder: Synthzsis of Very High-Level Algorithms", in preparation.

[Mikelsons, 1975]
Mikelsons, "Computer-Assisted Application Definition", *Second ACM Symposium on the Principles of Programming Languages,* January 1975.

[Nelson, 1976]
Nelson, Bruce, "The PSI Interpreter", unpublished Master's Project, Computer Science Department, Stanford University, June 1976.

[Phillips, 1976)
Phillips, Jorge V., "The PSI Trace Expert: Understanding Traces Using Multiple Knowledge Sources", in preparation.

[Rich and Shrobu, 1974]
Rich, Charles, and Shrobe, Howard E. "Understanding LISP Programs: Towards a Programmer's Apprentice", Working Paper 82, AI Laboratory, Massachusetts Institute of Technology, December, 1974.

[Steinberg, 1976]
Steinberg, Louis I., "The PSI Program Synthesis System: Discourse and Domain Expertise", in preparation.

[Winston, 1975]
Winston, Patrick Henry, "Learning Structural Descriptions from Examples", in Winston, Patrick Henry, editor, *The Psychology of Computer Vision,* McGraw-Will Book Company, Inc., New York, New York, 1975, pages 157-209.

Appendix A
Theses

Theses that have been published by this laboratory are listed here. Several earned degrees at institutions other than Stanford, as noted. This list is kept in diskfile THESES [BIB,DOC] @SU-AI.

D. Raj. Reddy, AIM-43
An Approach to Computer Speech Recognition by Direct Analysis of the Speech Wave,
Ph.D. in Computer Science,
September 1966.

S. Persson, AIM-46
Some Sequence Extrapolating Programs: a Study of Representation and Modeling in Inquiring Systems,
Ph.D. in Computer Science, University of California, Berkeley,
September 1966.

Bruce Buchanan, AIM-47
Logics of Scientific Discovery,
Ph.D. in Philosophy, University of California, Berkeley,
December 1966.

James Painter, AIM-44
Semantic Correctness of a Compiler for an Algol-like Language,
Ph.D. in Computer Science,
March 1967.

William Wichman, AIM-56
Use of Optical Feedback in the Computer Control of an Arm,
Eng. in Electrical Engineering,
August 1967.

Monte Callero, AIM-58
An Adaptive Command and Control System Utilizing Heuristic Learning Processes,
Ph.D. in Operations Research,
December 1967.

Donald Kaplan, AIM-60
The Formal Theoretic Analysis of Strong Equivalence for Elemental Properties,
Ph.D. in Computer Science,
July 1968.

Barbara Huberman, AIM-65
A Program to Play Chess End Gatnes,
Ph.D. in Computer Science,
August 1968.

Donald Pieper, AIM-72
The Kinematics of Manipulators under Computer Control,
Ph.D. in Mechanical Engineering,
October 1968.

Donald Waterman, AIM-74
Machine Learning of Heuristics,
Ph.D. in Computer Science,
December 1968.

Roger Schank, AIM-83
A Conceptual Dependency Representation for a Computer Oriented Semantics,
Ph.D. in Linguistics, University of Texas,
March 1969.

Pierre Vicens, AIM-85
Aspects of Speech Recognition by Computer,
Ph.D. in Computer Science,
March 1969.

Victor D. Scheinman, AIM-92
Design of Computer Controlled Manipulator,
Eng. in Mechanical Engineering,
June 1969.

Claude Cordell Green, AIM-96
The Application of Theorem Proving to Question-answering Systems,
Ph.D. in Electrical Engineering,
August 1969.

James J. Horning, AIM-98
A Study of Grammatical Inference,
Ph.D. in Computer Science,
August 1969.

Michael E. Kahn, AIM-106
The Near-minimum-time Control of Open-loop Articulated Kinematic Chains,
Ph.D. in Mechanical Engineering,
December 1969.

Joseph Becker, AIM-119
An Information-processing Model of Intermediate-Level Cognition,
Ph.D. in Computer Science,
May 1972.

Irwin Sobel, AIM-121
Camera Models and Machine Perception,
Ph.D. in Electrical Engineering,
May 1970.

Michael D. Kelly, AIM-130
Visual Identification of People by computer,
Ph.D. in Computer *Science,*
July 1970.

Gilbert Falk, AIM-132
Computer Interpretation of Imperfect Line Data as a Three-dimensional Scene,
Ph.D. in Electrical Engineering,
August 1970.

Jay Martin Tenenbaum, AIM-134
Accommodation in Computer vision,
Ph.D. in Electrical Engineering,
September 1970.

Lynn H. Quam, AIM-144
Computer Comparison of Pictures,
Ph.D. in Computer Science,
May 1971.

Robert E. Kling, AIM-147
Reasoning by Analogy with Applications to Heuristic Problem Solving: a Case Study,
Ph.D. in Computer Science,
August 1971.

Rodney Albert Schmidt Jr., AIM-149
A Study of the Real-time Control of a Computer-driven Vehicle,
Ph.D. in Electrical Engineering,
August 1971.

Jonathan Leonard Ryder, AIM-155
Heuristic Analysis of Large Trees as Generated in the Game of Go,
Ph.D. in Computer Science,
December 1971.

Jean M. Cadiou, AIM-163
Recursive Definitions of Partial Functions and their Computations,
Ph.D. in Computer Science,
April 1972.

Gerald Jacob Agin, AIM-173
Representation and Description of Curved Objects,
Ph.D. in Computer Science,
October 1972.

Francis Lockwood Morris, AIM-174
Correctness of Translations of Programming Languages – an Algebraic Approach,
Ph.D. in Computer Science,
August 1972.

Richard Paul, AIM-177
Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm,
Ph.D. in *Computer Science,*
November 1972.

Aharon Gill, AIM-178
Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination,
Ph.D. in Electrical Engineering,
October 1972.

Ruzena Bajcsy, AIM-180
Computer Identification of Textured Visiual Scenes,
Ph.D. in Computer Science,
October 1972.

Ashok Chandra, AIM-188
On the Properties and Applications of Programming Schemas,
Ph.D. in Computer Science,
March 1973.

Gunnar Rutger Grape, AIM-201
Model Based (Intermediate Level) Computer Vision,
Ph.D. in Computer Science,
May 1973.

Yoram Yakimovsky, AIM-209
Scene Analysis Using a Semantic Base for Region Growing,
Ph.D. in Computer Science,
July 1973.

Jean E. Vuillemin, AIM-218
Proof Techniques for Recursive Programs,
Ph.D. in Computer Science,
October 1973.

Daniel C. Swinehart, AIM-230
COPILOT: A Multiple Process Approach to Interactive Programming Systems,
Ph.D. in Computer Science,
May 1974.

James Gips, AIM-231
Shape Grammars and their Uses
Ph.D. in Computer Science,
May 1974.

Charles J. Rieger III, AIM-233
Conceptual Memory: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances,
Ph.D. in Computer Science,
June 1974.

Christopher K. Riesbeck, AIM-238
Computational Understanding: Analysis of Sentences and Context,
Ph.D. in Computer Science,
June 1974.

Marsha Jo Hannah, AIM-239
Computer Matching of Areas in Stereo Images,
Ph.D. in Computer Science,
July 1974.

James R. Low, AIM-242
Automatic Coding: Choice of Data Structures,
Ph.D. in Computer Science,
August 1974.

Jack Buchanan, AIM-245
A Study in Automatic Programming
Ph.D. in Computer Science,
May 1974.

Neil Goldman, AIM-247
Computer Generation of Natural Language From a Deep Conceptual Base
Ph.D. in Computer Science,
January 1974.

Bruce Baumgart, AIM-249
Geometric Modeling for Computer Vision
Ph.D. in Computer Science,
October 1974.

Ramakant Nevatia, AIM-250
Structured Descriptions of Complex Curved Objects for Recognition and Visual Memory
Ph.D. in Electrical Engineering,
October 1974.

Edward H. Shortliffe, AIM-251
MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection
Ph.D. in Medical Information Sciences,
October 1974.

Malcolm C. Newey, AIM-257
Formal Semantics of LISP With Applications to Program Correctness
Ph.D. in Computer Science,
January 1975.

Hanan Samet, AIM-259
Automatically Proving the Correctness of Translations Involving Optimized Coded
PhD in Computer Science,
May 1975.

David Canfield Smith, AIM-260
PYGMALION: A Creative Programming Environment
PhD in Computer Science,
June 1975.

Sundaram Ganapathy, AIM-272
Reconstruction of Scenes Containing Polyhedra From Stereo Pair of Views
Ph.D. in Computer Science,
December 1975.

Linda Gail Hemphill, AIM-273
A Conceptual Approach to Automated Language Understanding and Belief Structures: with Disambiguation of the Word 'For'
Ph.D. in Linguistics,
May 1975.

Norihsa Suzuki, AIM-279
Automatic Verification of Programs with Complex Data Structures
Ph.D. in Computer Science,
February 1976.

Russell Taylor, AIM-282
Synthesis of Manipulator Control Programs From Task-Level Specifications
PhD in Computer Science,
July 1976.

Randall Davis, AIM-283
Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases
Ph.D. in Computer Science,
July 1976.

Rafael Finkel, AIM-284
Constructing and Debugging Manipulator Programs
Ph.D. in Computer Science,
August 1976.

Douglas Lenat, AIM-286
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search
Ph.D. in Computer Science,
July 1976.

Michael Roderick, AIM-287
Discrete Control of a Robot Arm
Engineer in Electrical Engineering,
August 1976.

Robert C. Bolles, AIM-295
Verification Vision Within a Programmable Assembly System
Ph.D. in Computer *Science,*
December 1976.

Robert Cartwright, AIM-296
Practical Formal Semantic Definition and Verification Systems
Ph.D. in Computer Science,
December 1976.

Appendix B
Film Reports

Prints of the following films are available for distribution. This list is kept in diskfile FILMS [BIB,DOC] &U-AI.

1. Art Eisenson and Gary Feldman, Ellis D. Kroptechev and Zeus, his Marvelous Time-sharing System, 16mm B&W with sound, 15 minutes, March 1967.

The advantages of time-sharing over standard batch processing are revealed through the good offices of the Zeus time-sharing system on a PDP-1 computer. Our hero, Ellis, is saved from a fate worse than death. Recommended for mature audiences only.

2. Gary Feldman, Butterfinger, 16mm color with sound, 8 minutes, March 1968.

Describes the state of the hand-eye system at the Artificial Intelligence Project in the fall of 1967. The PDP-6 computer getting visual information from a television camera and controlling an electrical-mechanical arm solves simple tasks involving stacking blocks. The techniques of recognizing the blocks and their positions as well as controlling the arm are briefly presented. Rated "G".

3. Raj Reddy, Dave Espar and Art Eisenson, Hear Here, 16mm color with sound, 15 minutes, March 1969.

Describes the state of the speech recognition project- as of Spring, 1969. A discussion of the prdblems of speech recognition is followed by two real time demonstrations of the current system. The first shows the computer learning to recognize phrases and second shows how the hand-eye system may be controlled by voice commands. Commands as complicated as 'Pick up the small block in the lower lefthand corner', are recognized and the tasks are carried out by the computer controlled arm.

4. Gary Feldman and Donald Peiper, Avoid, 16mm color, silent, 5 minutes, March 1969.

An illustration of Peiper's Ph.D. thesis. The problem is to move the computer controlled mechanical arm through a space filled with one or more known obstacles. The film shows the arm as it moving through various cluttered environments with fairly good success.

5. Richard Paul and Karl Pingle, Instant Insanity, 16mm color, silent, 6 minutes, August, 1971.

Shows the hand/eye system solving the puzzle *Instant Insanity*. Sequences include finding and recognizing cubes, color recognition and object manipulation. [Made to accompany a paper presented at the 1971 IJCAI. May be hard to understand without a narrator.]

6. Suzanne Kandra, Motion and Vision, 16mm color, sound, 22 minutes, November 1972.

A technical presentation of three research projects completed in 1972: advanced arm control by R. P. Paul [AIM-177], visual feedback control by A. Gill [AIM-178], and representation and description of curved objects by G. Agin [AIM-173]. Drags a bit.

7. Larry Ward, Computer Interactive Picture Processing, (MARS Project), 16mm color, sound, 8 min., Fall 1972.

This film describes an automated picture differencing technique for analyzing the variable surface features on Mars using data returned by the Mariner 9 spacecraft. The system uses a time-shared, terminal oriented PDP- 10 computer. The film proceeds at a breathless pace. Don't blink, or you will miss an entire scene.

8. D.I. Okhotsimsky, et al, Display Simulations of **6-Legged** Walking, Institute of Applied Mathematics – USSR Academy of Science, (titles translated by Stanford AI Lab and edited by Suzanne Kandra), 16mm black and white, silent, 10 minutes, 1972.

A display simulation of a 6-legged ant-like walker getting over various obstacles. The research is aimed at a planetary *rover* that would get around by walking. This cartoon favorite beats Mickey Mouse hands down. *Or* rather, feet down.

9. Richard Paul, Karl Pingle, and Bob Bolles, Automated Pump Assembly, 16mm color, silent (runs at sound speed!), 7 minutes, A pril, 19'73.

Shows the hand-eye system assembling a simple pump, using vision to locate the pump body and to check for errors. The parts are assembled and screws inserted, using some special tools designed for the arm. Some titles are included to help explain the film.

10. Terry Winograd, Dialog with a robot, MIT A. I. Lab., 16mm black and white, silent, 20 minutes, 1971.

Presents a natural language dialog with a simulated robot block-manipulation system. The dialog is substantially the same as that in *V nderstanding Natural Language* (T. Winograd, Academic Press, 1972). No explanatory or narrative material is on **the** film.

11. Karl Pingle, Lou Paul, and Bob **Bolles, Programmable** Assembly, Three Short Examples, 16mm color, sound, 8 minutes, October 1974.

The first segment demonstrates the arm's ability to dynamically adjust for position and orientation changes. The task is to mount a bearing and seal on a crankshaft. Next, the arm is shown changing tools and recovering from a run-time error, Finally, a cinematic first: *two* arms cooperating to assemble a hinge.

12. Brian Harvey, Display **Terminals** at Stanford, 16mm B&W, sound, 13 minutes, May 1975.

Although there are many effective programs to use display terminals for special graphics applications, very few general purpose timesharing systems provide good support for using display terminals in normal text display applications This film shows a session using the display system at the Stanford AI Lab, explaining how the display support features in the Stanford monitor enhance the user's control over his job and facilitate the writing of display-effective user programs.

Appendix C
External Publications

Articles and books by project members that have appeared since July 1973 are listed here alphabetically by lead author. Earlier publications are given in our ten-year report [Memo AIM-2281 and in diskfile PUBS.OLD [BIB,DOC]@SU-AI. The list below is kept in PUBS[BIB,DOC]@SU-AI.

1. Agin, Gerald J., Thomas 0. Binford, Computer Description of Curved Objects, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

2. Aiello, Mario, Richard Weyhrauch, Checking Proofs in the Metamathetnatics of First Order Logic, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence*, Vpl. 1, pp. 1-8, September 1975.

3. Ashcroft, Edward, Zohar Manna, Amir Pnueli, Decidable Properties of Monodic Functional Schemas, *J. ACM*, July 1973.

4. Ashcroft, Edward, Zohar Manna, Translating Program Schemas to While-schemas, *SIAM Journal on Computing*, Vol. 4, No. 2, pp. 125-146, June 1975.

5. Bajcsy, Ruzena, Computer Description of Textured Scenes, *Proc. Third Int. Joint Conf. on Artificial Intelligence*, Stanford U., 1973.

6. Barstow, David, Elaine Kant, Observations on the Ineraction between Coding and Efficiency Knowledge in the PSI Program Synthesis System, *Proc. 2nd lnt. Conf. on Software Engineering*, IEEE Computer Society, Long Beach, California, October 1976.

7. Barstow, David, A Knowledge-Based System for Automatic Program Construction, *Proc. Int. Joint Con. on A.I.*, August 1977.

8. Biermann, A. W., R.I. Baum, F.E. Petry, Speeding Up the Synthesis of Programs from Traces, *IEEE Trans. Computers*, February 1975.

9. Bobrow, Daniel, Terry Winograd, An Overview of KRL, a Knowledge Representation Language, *J. Cognitive Science*, Vol. 1, No. 1, 1977.

10. Bobrow, Dan, Terry Winograd, & KRL Research Group, Experience with KRL-0: One Cycle of a Knowledge Representation Language, *Proc. ht. Joint Con. on A.I.*, August 1977.

11. Bolles, Robert C. Verification Vision for Programmable Assembly, *Proc. Int. Joint Con/. on A.I.*, August 1977.

12. Chandra, Ashol, Zohar Manna, On the Power of Programming Features, *Computer Languages*, Vol. 1, No. 3, pp. 219-232, September 1975.

13. Chowning, John M., The Synthesis of Complex Audio Spectra by means of Frequency Modulation, *J. Audio Engineering Society*, September 1973.

14. Clark, Douglas, and Green, C. Cordell, An Empirical Study of List Structure in LISP, *Communications of the ACM*, Volume 19, Number 11, November 1976.

15. Colby, Kenneth M., *Artificial Paranoia; A Computer Simulation of the Paranoid Mode*, Pergamon Press, N.Y., 1974.

16. Colby, K.M. and Parkison, R.C. Pattern-matching rules for the Recognition of Natural Language Dialogue Expressions, *American Journal of Computational Linguistics*, 1, September 1974.

17. Dershowitz, Nachum, Zohar Manna, On Automating Structural Programming, *Collogues IRIA on Proving and Improving Programs,* Arc-et-Senans, France, pp. 167-193, July 1975.

18. Dershowitz, Nachum, Zohar Manna, The Evolution of Programs: a System for Automatic Program Modification, *Proc. 4th Symp. on Principles of Programming Languages,* Los A ngeles, pp. 144- 154, January 1977.

19. Dobrotin, Boris M., Victor D. Scheinman, Design of a Computer Controlled Manipulator for Robot Research, *Proc. Third Int. Joint Conf. on Artificial Intelligence,* Stanford U., 1973.

20. Enea, Horace, Kenneth Mark Colby, Idiolectic Language-Analysis for Understanding Doctor-Patient Dialogues, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

2 1. Faught, William S., Affect as Motivation for Cognitive and Conative Processes, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence,* Vol. 2, pp. 893-899, September 1975.

22. Feldman, Jerome A., James R. Low, Comment on Brent's Scatter Storage Algorithm, *Comm. ACM,* November 1973.

23. Feldman, Jerome A., Yoram Yakimovsky, Decision Theory and Artificial Intelligence: I A Semantics-based Region Analyzer, *Artificial Intelligence J.,* Vol. 5, No: 4, Winter 1974.

24. Finkel, Raphael, Russell Taylor, Robert Bolles, Richard. Paul, Jerome Feldman, An Overview of AL, a Programming System for Automation, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence,* Vol. 2, pp. 758-765, September 1975.

25. Fuller, Samuel H., Forest Baskett, An Analysis of Drum Storage Units, *J. ACM,* Vol. 22, No. 1, January 1975,

26. Funt, Brian, WI-IISPER: A Problem-solving System utilizing Diagrams and a Parallel Processing Retina, *Proc. Int. Joint Con. on A.I.,* August 1977.

27. Gennery, Don A Stereo Vision System for an Autonomous Vehicle, *Proc. ht. Joint Conf. on A.I.,* August 1977.

28. Goldman, Neil M., Sentence Paraphrasing from a Conceptual Base, *Comm. ACM,* February 1975.

29. Goldman, Ron, Recent Work with the AL System, *Proc. ht. Joint Con. on A.I.,* August 1977.

30. Green, Cordell, David Barstow, Some Rules for the Automatic Synthesis of Programs, *Adv. Papers of 4th Int. joint Conference on Artificial Intelligence,* Vol. 1, pp. 232-239, September 1975.

31. Green, Cordell, and Barstow, David, Some Rules for the Automatic Synthesis of Programs, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Volume 1, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 232-239.

32. Green, Cordell, The Design of the PSI Program Synthesis System, *Proc. 2nd ht. Conf. on Software Engineering, IEEE* Computer Society, Long Beach, California, October 1976.

33. Green, Cordell, The PSI Program Synthesis System, 1976, *ACM '76: Proceedings of the Annual Conference,* Association for Computing Machinery, New York, New York, October 1976, pages 74-75.

34. Green, C. C., and Barstow, D. R., .A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System, in Elcock, E. W., and Michie, D., editors, Machine *Intelligence S: Machine Representations of Knowledge,* Ellis Horwood, Ltd., and John Wiley and Sons, Inc., New York, New York, 1976.

35. Green, C. C., A Summary of the PSI Program Synthesis System, *Proc. ht. Joint Conf. on A.I.,* August 1977.

36. Harvey, Brian, Increasing Programmer Power at Stanford with Display Terminals, *Minutes of the DECsystem-10 Spring-75 DECVS Meeting,* Digital Equipment Computer Users Society, Maynard, Mass., 1975.

37. Hieronymus, J. L., N. J. Miller, A. L. Samuel, The Amanuensis Speech Recognition System, *Proc. IEEE . Symposium on Speech Recognition,* April 1974.

38. Hieronymus, J. L., Pitch Synchronous Acoustic Segmentation, *Proc. IEEE Symposium on Speech Recognition,* April 1974.

39. Hilf, Franklin, Use of Computer Assistance in Enhancing Dialog Based Social Welfare, Public Health, and Educational Services in Developing Countries, *Proc. 2nd Jerusalem Conf. on Info. Technology,* July 1974.

40. Hilf, Franklin, Dynamic Content Analysis, *Archives of General Psychiatry,* January 1975.

41. Hueckel, Manfred H., A Local Visual Operator which Recognizes Edges and Lines, *J.* ACM, October 1973.

42. Igarashi, S., R. L. London, D. C. Luckham, Automatic Program Verification I: Logical Basis and its Implementation, *Acta Informatica,,* Vol. 4, pp. 145-182, March 1975.

43. Ishida, Tatsuzo, Force Control in Coordination of Two Arms, *Proc.* Int. *Joint Conf. on A.I.,* August 1977.

44. Kant, Elaine, The Selection of Efficient Implementations for a High-level Language, *Proc. SIGART-SIGPLAN Symp.* on *A.I. & Prog. Lang.,* August 1977.

45. Karp, Richard A., David C Luckham, Verification of Fairness in an Implementation of Monitors, *Proc. 2nd Intnl. Conf. on Software Engineering, PP.* 40-46, October 1976.

46. Katz, Shmuel, Zohar Manna, A Heuristic Approach to Program Verification, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

47. Katz, Shmuel, Zohar Manna, Towards Automatic Debugging of Programs, *Proc. Int. Con. on Reliable Software, Los Angeles, April 1975.*

48. *Katz, Shmuel, Zohar* Manna, Logical Analysis of Programs, *Comm. ACM,* April 1976.

49. Katz, Shmuel, Zohar Manna, A Closer Look at Termination, *Acta Informatica,* Vol. 5, pp. 333-352, April 1977.

50. Lenat, Douglas B., BEINGS: Knowledge as Interacting Experts, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence,* Vol. 1, pp. 126-133, September 1975.

51. Luckham, David C., Automatic Problem Solving, *Proceedings of the Third International Joint Conference on Artificial*

Intelligence, Stanford University, August 1973.

52. Luckham, David C., Jack R. Buchanan, Automatic Generation of Programs Containing Conditional Statements, *Proc. AISB Summer Conference,* U. Sussex, July 1974.

53. Luckham, David C., Program Verification and Verification-oriented Programming, *Proc. I.F.I.P.*, August 1977.

54. Manna, Zohar, Program Schemas, in *Currents in the Theory of Computing* (A. V. Aho, Ed.), Prentice-Hall, Englewood Cliffs, N. J., 1973.

55. Manna, Zohar, Stephen Ness, Jean Vuillemin, Inductive Methods for Proving Properties of Programs, *Comm. ACM,* August 1973.

56. Manna, Zohar, Automatic Programming, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

57. Manna, Zohar, *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974.

58. Manna, Zohar, Amir Pneuli, Axiomatic Approach to Total Correctness, *Acta Informatica,* Vol. 3, pp. 243-263, 1974.

59. Manna, Zohar, Richard Waldinger, Knowledge and Reasoning in Program Synthesis, *Artificial Intelligence,* Vol. 6, No. 2, pp. 175-208, 1975.

60. Manna, Zohar, Adi Shamir, The Theoretical Aspects of the Optimal Fixpoint, *SIAM Journal of Computing,* Vol. 5, No. 3, pp.414-426, September 1976.

61. Manna, Zohar, Richard Waldinger, Is 'Sometime' sometimes better than 'Always'? Intermittant Assertions in Proving Program Correctness, *Proc. 2nd Int. Con. on Software Engineering,* IEEE Computer Society, San Francisco, California, October 1976.

62. Manna, Zohar, Richard Waldinger, The Automatic Synthesis of Recursive Programs, *Proc. SIGART-SIGPLAN Symp. on A.I. & Prog. Lang.,* August 1977.

63. Manna, Zohar, Richard Waldinger, The Automatic Synthesis of Systems of Recursive Programs, *Proc. ht. Joint Conf. on A.I.*, August 1977.

64. McCarthy, John, Mechanical Servants for Mankind, *Britannica Yearbook of Science and the Future,* 1973.

65. McCarthy, John, Book Review: Artificial Intelligence: A General Survey by Sir James Lighthill, *Artificial Intelligence,* Vol. 5, No. 3, Fall 1974.

66. McCarthy, John, Modeling Our Minds Science Year *1975,* The World Book Science Annual, Field Enterprises Educational Corporation, Chicago, 1974.

67. McCarthy, John, Proposed Criterion for a Cipher to be Probable-word-proof, *Comm. ACM,* February 1975.

68. McCarthy, John, An Unreasonable Book, a review *of Computer Power and Human Reason* by Joseph Weizenbaum (W.H. Freeman and Co., San Francisco, 1976), *SIGART Newsletter* #58, June 1976.

69. McCarthy, John, Review: *Computer Power and Human Reason,* by Joseph Weizenbaum (W.H. Freeman and Co., San Francisco, 1976) in *Physics Today,* 1977.

70. McCarthy, John, Another SAMEFRINGE, *SIGART Newsletter No.* 61, February 1977.

71. McCarthy, John, The Home Information Terminal, *The Grolier Encyclopedia*, 1977.

72. McCarthy, John, M. Sato, S. Igarashi, T. Hayashi, On The Model Theory of Knowledge, *Proc. 5th international joint Conference on Artificial Intelligence*, M.I.T, Cambridge, 1977 (to appear).

73. McCarthy, John, M. Sato, T. Hayashi, S. Igarashi, On the Model Theory of Knowledge, *Proc. ht. Joint Con. on A.I.*, August 1977.

74. McCarthy, John, Epistemological Problems of Artificial Intelligence, *Proc. Int. Joint Con.. on A.I.*, August 1977.

75. McCune, Brian, The PSI Program Model Builder: Synthesis of Very High-level Programs, *Proc. SIGART-SIGPLAN Symp. on A.I. & Prog. Lang.*, August 1977.

76. Miller, N. J., Pitch Detection by Data . Reduction, *Proc. IEEE Symposium on Speech Recognition*, April 1974.

77. Moore, Robert C., Reasoning about Knowledge and 'Action, *Proc. Int. Joint Conf. on A.I.*, August 1977.

78. Moorer, James A., The Optimum Comb Method of Pitch Period Analysis of Continuous Speech, *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, No. 5, October 1974.

79. Moorer, James A., On the Transcription of Musical Sound by Computer, *USA-JAPAN Computer Conference*, August 1975.

80. Morales, Jorge J., Interactive Theorem Proving, *Proc. ACM National Conference*, August 1973.

81. Moravec, Hans, Towards Automatic Visual Obstacle Avoidance, *Proc. ht. Joint Conf. on A.I.*, August 1977.

82. Nevatia, Ramakant, Thomas 0. Binford, Structured Descriptions of Complex Objects, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.

83. Newell, A., Cooper, F. S., Forgie, J. W., Green, C. C., Klatt, D. H., Medress, M. F., Neuburg, E. P., O'Malley, M. H., Reddy, D. R., Ritea, B., Shoup, J. E., Walker, D. E., and Woods, W. A., *Considerations for a Follow-On ARPA Research Program for Speech Understanding Systems*, Information Processing Techniques Office, Advanced Research Projects Agency, Department of Defense, Arlington, Virginia, August 1975.

84. Oppen, Derek, S.A. Cook, Proving Assertions about Programs that Manipulate Data Structures, *Acta Informatica*, Vol. 4, No. 2, pp. 127-144, 1975.

85. Phillips, Jorge, T. H. Bredt, Design and Verification of Real-time Systems, *Proc. 2nd Int. Con- on Software Engineering*, IEEE Computer Society, Long Beach, California, October 1976.

86. Phillips, Jorge, Program Inference from Traces using Multiple Knowledge Sources, *Proc. Int. Joint Conf. on A.I.*, August 1977.

87. Quam, Lynn, Robert Tucker, Botond Eross, J. Veverka and Carl Sagan, Mariner 9 Picture Differencing at Stanford, *Sky* and *Telescope*, August 1973.

88. Rubin, Jeff, Computer Communication via the Dial-up Network, *Minutes of the DECsystem-10 Spring-75 D ECU S Meeting*, Digital Equipment Computer Users Society, Maynard, Mass., 1975.

89. Sagan, Carl, J. Veverka, P. Fox, R. Dubisch, R. French, P. Gierasch, L. Quam, J. Lederberg, E. Levinthal, R. Tucker, B. Eross, J. Pollack, Variable Features on Mars II: Mariner 9 Global Results, *J. Geophys. Res., 78, 4 163-4* 196, 1973.

90. Schank, Roger C., Neil Goldman, Charles J. Rieger III, Chris Riesbeck, MARGIE: Memory, Analysis, Response Generation and Inference on English, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

91. Schank, Roger C., Kenneth Colby (eds), *Computer Models of Thought and Language,* W. H. Freeman, San Francisco, 1973.

92. Schank, Roger, The Conceptual Analysis of Natural Language, in R. Rustin (ed.), *Natural Language Processing,* Algorithmics Press, New York, 1973.

93. Schank, Roger, Charles J. Rieger III, Inference and Computer Understanding of Natural Language, *Artificial Intelligence J.*, Vol.5, No. 4, Winter 1974.

94. Schank, Roger C., Neil M. Goldman, Charles J. Rieger III, Christopher K. Riesbeck, Interface and Paraphrase by Computer, *J. ACM,* Vol 22, No. 3, July 1975.

95. Shaw, David E., William R. Swartout, C. Cordell Green, Inferring LISP Programs from Examples, *Adv. Papers of 4th Int. Joint Conference on Artificial Intelligence,* Vol. 1, pp. 260-267, September 1975.

96. Shortliffe, Edward H., Davis, Randall, Axline, Stanton G., Buchanan, Bruce G., Green, C. Cordell, and Cohen, Stanley N., Computer-Based Consultations in Clinical Therapeutics: Explanation and Rule Acquisition Capabilities of the MYCIN System, *Computers and Biomedical Research,* Volume 8, Number 3, June 1975, pages 303-320.

97. Smith, David Canfield, Horace J. Enea, Backtracking in MLISP2, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

98. Smith, Leland, Editing and Printing Music by Computer, *J. Music Theory,* Fall 1973.

99. Sobel, Irwin, On Calibrating Computer Controlled Cameras for Perceiving 3-D *Scenes, Proc. Third Int. Joint Conf. on Artificial Intelligence,* Stanford U., 1973; also in Artificial Intelligence J., Vol. 5, No. 2, Summer 1974.

100. Suzuki, N., Verifying Programs by Algebraic and Logical Reduction, *Proc. Int. Conf. on Reliable Software,* Los Angeles, Calif., April 1975, in *ACM SIGPLAN Notices,* Vol. 10, No. 6, pp. 473-481, June 1975.

101. Tesler, Lawrence G., Horace J. Enea, David C. Smith, The LISP30 Pattern Matching System, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

102. Thomas, Arthur J., Puccetti on Machine Pattern Recognition, *Brit.J.Philosophy of Science,* 26:227-232, 1975.

103. Veverka, J., Carl Sagan, Lynn Quam, R. Tucker, B. Eross, Variable Features on Mars III: Comparison of Mariner 1969 and Mariner 1971 Photography, *Icarus,* 21, 317-368, 1974.

104. von Henke, F. W., D.C. Luckham, A Methodology for Verifying Programs, *Proc. Int. Conf. on Reliable Software,* Los Angeles, Calif., April 1975, in ACM

SIGPLAN Notices, Vol. 10, No. 6, pp. 156-164, June 1975.

105. Wilks, Yorick, The Stanford Machine Translation and Understanding Project, in R. Rustin (ed.), *Natural Language Processing,* Algorithmics Press, New York, 1973.

106. Wilks, Yorick, Understanding Without Proofs, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

107. Wilks, Yorick, Annette Herskovits, An Intelligent Analyser and Generator of Natural Language, *Proc. Int. Conf. on Computational Linguistics,* Pisa, Italy, *Proceedings of the Third Internation Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

108. Wilks, Yorick, The Computer Analysis . of Philosophical Arguments, *CIRPHO,* Vol. 1, No. 1, September 1973

109. Wilks, Yorick, An Artificial Intelligence Approach to Machine Translation, in Schank and Colby (eds.), *Computer Models of Thought and Language,* W. H. Freeman, San Francisco, 1973.

110. Wilks, Yorick, One Small Head – Models and Theories in Linguistics, *Foundations of Language,* Vol. 10, No. 1, January 1974.

111. Wilks, Yorick, Preference Semantics, E. Keenan (ed.), *Proc. 1973 Colloquium on Formal Semantics of Natural Language,* Cambridge, U.K., 1974.

112. Wilks, Yorick, The XGP Computer-driven Printer at Stanford, *Bulletin of Assoc. for Literary and Linguistic Computing,,* Vol. 2, No. 2, Summer 1974.

113. Wilks, Y., Semantic Procedures and Information, in *Studies in the Foundations of Communication,* R. Posner (ed.), Springer, Berlin, forthcoming.

114. Wilks, Yorick, A Preferential, Pattern-Seeking Semantics for Natural Language Inference, *Artificial Intelligence J.,* Vol. 6, No. 1, Spring 1975.

115. Wilks, Y., An Intelligent Analyser and Understander of English, *Comm. ACM,* May 1975.

116. Winograd, Terry, A Process Model of Language Understanding, in Schank and Colby (eds.), *Computer Models of Thought and Language,* W. H. Freeman, San Francisco, 1973.

117. Winograd, Terry, The Processes of Language Understanding in Benthall, (ed.), *The Limits of Human Nature,* Allen Lane, London, 1973.

118. Winograd, Terry, Language and the Nature of Intelligence, in G.J. Dalenoort (ed.), *Process Models for Psychology,* Rotterdam Univ. Press, 1973

119. Winograd, Terry, Breaking the Complexity Barrier (again), *Proc. SIGPLAN-SIGIR interface Meeting,* 1975; *ACM SIGPLAN Notices,* 10:1, pp. 13-30, January 1975.

120. Winograd, Terry, Artificial Intelligence – When Will Computers Understand People?, *Psychology Today,* May 1974.

121. Winograd, Terry, Frame Representations and the Procedural - Declarative Controversy, in D. Bobrow and A. Collins, eds., *Representation and Understanding: Studies in Cognitive Science,* Academic Press, 1975.

122. Winograd, Terry, Reactive Systems, *Coevolution Quarterly,* September 1975

123. Winograd, Terry, Parsing Natural Language via Recursive Transition Net, in Raymond Yeh (ed.) *Applied Computation Theory,* Prentice-Hall, 1976.

124. Winograd, Terry, Computer Memories - a Metaphor for Human Memory, in Charles Cofer (ed.), *Models of Human Memory,* Freeman, 1976.

125. Yakimovsky, Yoram, Jerome A. Feldman, A Semantics-Based Decision Theoretic Region Analyzer, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford University, August 1973.

126. Yolks, Warwick, There's Always Room at the Top, or How Frames gave my Life Meaning, *SIGART Newsletter, No. 53,* August 1975.

Appendix D
Abstracts of Recent Reports

Abstracts are given here for Artificial Intelligence Memos published since July 1973. For earlier years, see our ten-year report [Memo AIM-2281 or diskfile AIMS.OLD [BIB,DOC] &U-AI. The abstracts below are kept in diskfile AIMS [BIB,DOC] @SU-AI and the titles of both earlier and more recent A. I. Memos are in AIMLST[BIB,DOC] @SU-AI.

In the listing below, there are up to three numbers given for each report: an "AIM" number on the left, a "CS" (Computer Science) number in the middle, and a NTIS stock number (often beginning 'AD...") on the right. Special symbols preceding the 'AIM" number indicate availability at this writing, as follows:

- + hard copy or microfiche,
- ⊗ microfiche only,
- * out-of-stock.

If there is no special symbol, then it is available in hard copy only, Reports that are in stock may be requested from:

Documentation Services
Artificial Intelligence Laboratory
Stanford University
Stanford, California 94305

Rising costs and restriction, on the use of research funds for printing reports have made it necessary to charge for reports at their replacement cost. By doing so, we will be able to reprint popular reports rather than simply declaring them "out of print".

Alternate Sources

Alternatively, reports may be ordered (for a nominal fee) in either hard copy or microfiche from:

National Technical Information Service
P. 0. Box 1553
Springfield, Virginia 22161

If there is no NTIS number given, then they may or may not have the report. In requesting copies in this case, give them both the 'AIM-' and "CS-nnn" numbers, with the latter enlarged into the form "STAN-CS-yy-nnn", where "yy" is the last two digits of the year of publication.

Memos that are also Ph.D. theses are so marked below and may be ordered from:

University Microfilm
P. 0. Box 1346
Ann Arbor, Michigan 48106

For people with access to the ARPA Network, the texts of some A. I. Memos are stored online in the Stanford A. I. Laboratory disk file. These are designated below by "Diskfile: <file name>" appearing in the header.

* AIM-211 cs-383 AD769673
Yorick Wilks,
Natural Language Inference,
24 pages, September 1973.

The paper describes the way in which a Preference Semantics system for natural language analysis and generation tackles a difficult class of anaphoric inference problems (finding th correct referent for an English pronoun in context): those requiring either analytic (conceptual) knowledge of a complex sort, or requiring weak inductive knowledge of the course of events in the real world. The method employed converts all available knowledge to a canonical template form and endeavors to create chains of non-deductive inferences from the unknowns to the possible referents. Its method of selecting among possible chains of inferences is consistent with the overall principle of 'semantic preference' used to set up the original meaning representation, of which these anaphoric inference procedures are a manipulation.

* AIM-212 CS-384 AD769379
Annette Herskovits,
The Generation of French from a Semantic Representation,
20 pages, September 1973.

The report contains first a brief description of Preference Semantics, a system of representation and analysis of the meaning structure of natural language. The analysis algorithm which transforms phrases into semantic items called templates has been considered in detail elsewhere, so this report concentrates on the second phase of analysis, which binds templates together into a higher level semantic block corresponding to an English paragraph, and which, in operation, interlocks with the French generation procedure. During this phase, the semantic relations between templates are extracted, pronouns are referred and those word disambiguations are done that require the context of a whole paragraph. These tasks require items called *paraplates* which are attached to keywords such as prepositions, subjunctions and relative pronouns. The system chooses the representation which maximizes a carefully defined 'semantic density'.

A system for the generation of French sentences is described, based on the generation of French sentences is described, based on the recursive evaluation of procedural generation patterns called *stereotypes*. The stereotypes are semantically context sensitive, are attached *to* each sense of English words and keywords and are carried into the representation by the analysis procedure. The representation of the meaning of words, and the versatility of the stereotype format, allow for fine meaning distinctions to appear in the French, and for the construction of French differing radically from the English origin.

AIM-213 CS-385
Ravindra B. Thosar,
Recognition of Continuous Speech: Segmentation and Classification using Signature Table Adaptation,
37 pages, September 1973. Cost: $2.75

This report explores the possibility of using a set of features for segmentation and recognition of continuous speech. The features are not necessarily *distinctive* or minimal, in the sense that they do not divide the phonemes into mutually exclusive subsets, and can have high redundancy. This concept of feature can thus avoid apriori binding between the phoneme categories to be recognized and the set of features defined in a particular system.

An adaptive technique is used to find the probability of the presence of a feature. Each feature is treated independently of other features, An unknown utterance is thus represented by a feature graph with associated probabilities. It is hoped that such a representation would be valuable for a hypothesize-test paradigm as opposed to a one which operates on a linear symbolic input.

AIM-214 CS-386 AD767332
Walter A. Perkins, Thomas 0. Binford,
A Corner Finder for Visual Feedback,
59 pages, September 1973. Cost: $3.35

In visual-feedback work often a model of an object and its approximate location are known and it is only necessary to determine its location and orientation more accurately. The purpose of the program described herein is to provide such information for the case in which the model is an edge or corner. Given , a model of a line or a corner with two or three edges, the program searches a TV window of arbitrary size looking for one or all corners which match the model. A model-driven program directs the search. It calls on another program to find all lines inside the window. Then it looks at these lines and eliminates lines which cannot match any of the model lines. It next calls on a program to form vertices and then checks for a matching vertex. If this simple procedure fails, the model-driver has two backup procedures. First it works with the lines that it has and tries to form a matching vertex (corner). If this fails, it matches parts of the model with vertices and lines that are present and then takes a careful look in a small region in which it expects to find a missing line. The program often finds weak contrast edges in this manner. Lines are found by a global method after the entire window has been scanned with the H ueckel edge operator.

∗ AIM-215 CS-387 A D769380
Bruce G. Buchanan, N. S. Sridharan,
Analysis of Behavior of Chemical Molecules: Rule Formation on Non-homogeneous Classes of Objects,
15 pages, September 1973.

An information processing model of some important aspects of inductive reasoning is presented within the context of one scientific discipline. Given a collection of experimental (mass spectrometry) data from several chemical molecules the computer program described here separates the molecules into *well-behaved* subclasses and selects from the space of all explanatory processes the *characteristic* processes for each subclass. The definitions of *well-behaved* and *characteristic* embody several heuristics which are discussed. Some results of the program are discussed which have been useful to chemists and which lend credibility to this approach.

∗ AIM-216 CS-389 AD771299
Larry Masinter, N.S. Sridharan, J. Lederberg, S. H. Smith,
Applications of Artificial Intelligence for Chemical Inference: XII. Exhaustive Generation of Cyclic and Acyclic Isomers,
60 pages, September 1973.

A systematic method of identification of all possible graph isomers consistent with a given empirical formula is described. The method, embodied in a computer program, generates a complete list of isomers. Duplicate structures are avoided prospectively.

∗ AIM-217 cs-39 1 AD770610
N. S. Sridharan,
Search Strategies for the Task of Organic Chemical Synthesis,
32 pages, August 1973.

A computer program has been written that successfully discovers syntheses for complex organic chemical molecules. The definition of the search space and strategies for heuristic search are described in this paper.

∗ AIM-218 CS-393 AD772063/4WC
Jean Etienne Vuillemin,
Proof Techniques for Recursive Programs,
Thesis: Ph.D. in Computer Science,
97 pages, October 1973.

The concept of least fixed-point of a continuous function can be considered as the unifying thread of this dissertation. The connections between fixed-points and recursive programs are detailed in Chapter 2, providing some insights on practical implementations of recursion. There are two usual characterizations of the least fixed-point of a continuous function. To the first characterization, due to Knaster and Tarski, corresponds a class of proof techniques for programs, as described in Chapter 3. The other characterization of least fixed points, better known as Kleene's first recursion theorem, is discussed in Chapter IV. It has the advantage of being effective and it leads to a wider class of prrof techniques.

∗ AIM-219 cs-394 AD769674
C. A. R. Hoare,
Parallel Programming: an Axiomatic Approach,
33 pages, October 1973.

This paper develops some ideas expounded in [1]. It distinguishes a number of ways of using parallelism, including disjoint processes, competition, cooperation, communication and "colluding". In each case an axiomatic proof rule is given. Some light is thrown on traps or ON conditions. Warning: the program structuring methods described here are not suitable for the construction of operating systems.

AIM-220 CS-396 AD772064/2WC
Robert Bolles, Richard Paul,
The use of Sensory Feedback in a Programmable Assembly Systems,
26 pages, October 1973. Cost: $2.45

This article describes an experimental, automated assembly system which uses sensory feedback to control an electro-mechanical arm and TV camera. Visual, tactile, and force feedback are used to improve positional information, guide manipulations, and perform inspections. The system has two phases: a *planning* phase in which the computer is programmed to assemble some object, and a *working* phase in which the computer controls the arm and TV camera in actually performing the assembly. The working phase is designed to be run on a mini-computer.

The system has been used to assemble a water pump, consisting of a base, gasket, top, and six screws. This example is used to explain how the sensory data is incorporated into the control system. A movie showing the pump assembly is available from the Stanford Artificial Intelligence Laboratory.

⊗ AIM-221 CS-447 AD787631/1WC
Luigia Aiello, Mario Aiello, Richard Weyhrauch,
The Semantics of PASCAL in LCF,
78 pages, October 1974.

We define a semantics for the arithmetic part of PASCAL by giving it an interpretation in LCF, a language based on the typed λ-calculus.. Programs are represented in terms of their abstract syntax. We show sample proofs, using LCF, of some general properties of PASCAL and the correctness of some particular programs. A program implementing the McCarthy Airline reservation system is proved correct.

+ AIM-222 cs-467
Mario Aiello, Richard Weyhrauch,
Checking Proofs in the Metamathematics of First Order Logic,
55 pages, August 1974. Cost: $3.25

This is a report on some of the first experiments of any size carried out using the new first order proof checker FOL. We present two different first order axiomatizations of the metamathematics of the logic which FOL itself checks and show several proofs using each one. The difference between the axiomatizations is that one defines the metamathematics in a many sorted logic the other does not.

⊗ AIM-223 cs-400 AD772509
C. A. R. Hoare,
Recursive Data Structures,
32 pages, December 1973.

The power and convenience of a programming language may be enhanced for certain applications by permitting data structures to be defined by recursion. This paper suggests a pleasing notation by which such structures can be declared and processed; it gives the axioms which specify their properties, and suggests an efficient implementation method. It shows how a recursive data structure may be used to represent another data type, for example, a set. It then discusses two ways in which significant gains in efficiency can be made by selective updating of structures, and gives the relevant proof rules and hints for implementation. It is shown by examples that a certain range of applications can be efficiently programmed, ithout introducing the low-level concept of a reference into a high-level programming language.

⊗ AIM-224 cs-403 AD773391
C. A, R. Hoare,
Hints on Programming Language Design,
29 pages, December 1973.

This paper (based on a keynote address

presented at the *SIGACT/SIGPLAN Symposium on Principles of Programming Languages,* Boston, October 1-3, 1973) presents the view that a programming language is a tool which should assist the programmer in the most difficult aspects of his art, namely program design, documentation, and debugging. It discusses the objective criteria for evaluating a language design, and illustrates them by application to language features of both high level languages and machine code programming. It concludes with an annotated reading list, recommended for all intending language designers.

⊗ AIM-225 CS-406 AD775645/5WC
W. A. Perkins,
Memory Model For a Robot,
118 pages, January 1974.

A memory model for a robot has been designed and tested in a simple toy-block world for which it has shown clarity, efficiency, and generality. In a constrained psüedo-English one can ask the program to manipulate objects and query it about the present, past, and possible future states of its world. The program has a good understanding of its world and gives intelligent answers in reasonably good English. Past and hypothetical states of the world are handled by changing the state the world in an imaginary context. Procedures interrogate and modify two globabl databases, one which contains the present representation of the world and another which contains the past history of events, conversations, etc. The program has the ability to create, destroy, and even resurrect objects in its world.

+ AIM-226 CS-407 AD778310/3WC
F.H.C. Wright II, R. E. Gorin,
FAIL,
6 1 pages, April 1974. Cost: $3.40

This is a reference manual for FAIL, a fast, one-pass assembler for PDP-10 and PDP-6 machine language. FAIL statements, pseudo-operations, macros, and conditional assembly features are described. Although FAIL uses substantially more main memory than MACRO- 10, it assembles typical programs about five times faster. FAIL assembles the entire Stanford time-sharing operating system (two million characters) in less than four minutes of CPU time on a KA-10 processor. FAIL permits an ALGOL-style block structure which provides a way of localizing the usage of some symbols to certain parts of the program, such that the same symbol name can be used to mean different things in different blocks.

+ AIM-227 CS-408 ADA003483
A. J. Thomas, T. 0. Binford,
Information Processing Analysis of Visual Perception: A Review,
50 pages, June 1974. Cost: $3.10

We suggest that recent advances in the construction of artificial vision systems provide the beginnings of a framework for an information processing analysis of human visual perception. We review some pertinent investigations which have appeared in the psychological literature, and discuss what we think t be some of the salient and potentially useful theoretical Concepts which have resulted from the attempts to build computer vision systems. Finally we try to integrate these two sources of ideas to suggest some desireable structural and behavioural concepts which apply to both the natural and artificial systems.

⊗ AIM-228 CS-409 AD776233/9WC
Lester Earnest (ed.),
FINAL REPORT: The First Ten Years of Artificial Intelligence Research at Stanford,
118 pages, July 1973.

The first ten years of research in artificial intelligence and related fields at Stanford University have yielded significant results in computer vision and control of manipulators, speech recognition, heuristic programming, representation theory, mathematical theory of computation, and modeling of organic chemical processes. This report summarizes the accomplishments and provides bibliographies in each research area.

⊗ AIM-229 cs-4 11
D.B. Anderson, T.O. **Binford,** A.J. **Thomas,**
R.W. Weyhrauch, Y.A. Wilks,
AFTER LEIBNIZ . . . : Discussions **on**
Philosophy and Artificial Intelligence,
43 pages, April 1974.

This is an **edited** transcript of informal conversations which we have had over recent months, in which we looked at some of the issues which seem to arise when artificial intelligence and philosophy meet. Our aim was to see what might be some of the fundamental principles of attempts to **build** intelligent machines. The major topics covered are the relationship of AI and philosophy and what help they might be to each other; the **machanisms** of natural inference and deduction; the question of what kind of theory of meaning would be involved in a successful natural language understanding program, and the nature of models in AI research.

⊗ AIM-230 CS-412 **AD786721/1WC**
Daniel C. Swinehart,
COPILOT: A Multiple Process Approach to
Interactive Programming Systems,
Thesis: Ph.D. in Computer Science,
2 13 pages, August 1974.

The addition of multiple processing facilities to a language used in an interactive **computing** environment requires new techniques. This dissertation presents one approach, emphasizing the characteristics of the interface between the user and the system.

We have designed an experimental interactive programming system, COPILOT, as the concrete vehicle for testing and describing our methods. COPILOT allows the user **to** create, modify, investigate, and control programs written in an Algol-like language, which has been augmented with facilities for multiple processing. Although COPILOT is **compiler**-based, many of our solutions could also be applied to an interpretive system.

Central to the design is the use of CRT displays to present programs, program data, and system status. This continuous display of information in context allows the user to retain comprehension of complex program environments, and to indicate the environments to be affected by his commands.

COPILOT uses the multiple processing facilities to its advantage to achieve a kind of interactive control which we have termed ***non-preemptive.*** The user's terminal is continuously available for commands of any kind: program editing, variable inquiry, program control, etc., independent of the execution state of the processes he is **controlling**. No process may unilaterally gain possession of the user's input; the user retains control at all times.

Commands in COPILOT are expressed as statements in the programming language. This single language policy adds consistency to the system, and permits the user to construct procedures for the execution of repetitive or complex command sequences. An abbreviation facility is provided' for the most common terminal operations, for convenience and speed.

We have attempted in this thesis to extend the facilities of interactive programming systems in response to developments in language design and information display technology. The resultant system provides an interface which, we think, is better matched to the interactive needs of its user than **are** its predecessors.

⊗ AIM-231 cs-413 ADA001814
James Gips,
Shape Grammars and their Uses,
Thesis: Ph.D. in Computer Science,
243 pages, August 1974.

Shape grammars are defined and their uses are investigated. Shape grammars provide a means for the recursive specification of shapes. A shape grammar is presented that generates a new class of reversible figures. Shape grammars are given for some well known mathematical curves. A simple method for constructing shape grammars that simulate Turing machines is presented. A program has been developed that uses a shape grammar to solve a perceptual task involving the analysis and comparison of line drawings that portray three-dimensional objects of a restricted type. A formalism that uses shape grammas to generate paintings is defined, its implementation on the computer is described, and examples of generated paintings are shown. The use of shape

⊗ AIM-232 CS-4 14 **AD780452/9WC**
Bruce G. Baumgart,
GEOMED - A Geometric Editor,
45 pages, May 1974.

GEOMED is a system for doing 3-D geometric modeling; used from a keyboard, it is an interactive drawing program; used as a package of SAIL or LISP accessible subroutines, it is a graphics language. With GEOMED, arbitrary polyhedra can be constructed; moved about and viewed in perspective with hidden lines eliminated. In addition to polyhedra; camera and image models are provided so that simulators relevant to computer vision, problem solving, and animation may be constructed.

⊗ AIM-233 CS-419 **ADA000086/9WC**
Charles J. Rieger, III,
Conceptual Memory: A Theory and Computer Program for Processing the Meaning Content of Natural Language Utterances,
Thesis: Ph.D. in Computer Science,
393 pages, June 1974.

Humans perform vast quantities of spontaneous, subconscious computation in order to understand even the simplest natural language utterances. The computation is principally meaning-based, with syntax and traditional semantics playing insignificant roles. This thesis supports this conjecture by synthesis of a theory and computer program which account for many aspects of language behavior in humans. It is a theory of language and memory.

Since the theory and program deal with language in the domain of conceptual meaning, they are independent of language form and of any specific language. Input to the memory has the form of analyzed conceptual dependency graphs which represent the underlying meaning of language utterances. Output from the memory is also in the form of meaning graphs which have been produced by the active (inferential) memory processes which dissect, transform, extend and recombine the input graphs in ways which are dependent upon the meaning context in which they were perceived.

A memory formalism for the computer model is first developed as a basis for examining the inferential processes by which comprehension occurs. Then, the notion of inference space is presented, and sixteen classes of conceptual inference and their implementation in the computer model are examined, emphasizing the contribution of each class to the total problem of understanding. Among the sixteen inference classes are: causative/resultative inferences (those which explain and predict cause and effect relationships relative to the memory's model of the world), motivational inferences (those which infer the probable intentions of actors), enabling inferences (those which predictively fill out the circumstances which were likely to have obtained at the time of an action), action prediction inferences (those which make guesses about what a person might be expected to do in some situation), knowledge propagation inferences (those which predict what knowledge is available to a person, based on what the memory already knows or can infer he knows), normative inferences (those which assess the "normality" of a given piece of information), and state duration inferences (those which predict the probable duration of specific states in the world). All inferences are probabilistic, and "backup" is deemphasited as a programming tool.

The idea of points of contact of information structures in inference space is explored. A point of contact occurs when an inferred unit of meaning from one starting point within one utterance's meaning graph either confirms (matches) or contradicts an inferred unit of meaning from another point within the graph, or from within the graph of another utterance. The quantity and quality of points of contact serve as the primary definition of understanding, since such points provide an effective measure of the memory's ability to relate and fill in information,

Interactions between the inference processes and (1) word sense *promotion* (how meaning context influences the language analyzer's choice of lexical senses of words during the parse), and (2) the processes of reference (how memory pointers to tokens of real world entities are established) are examined. In particular, an important inference-reference relaxation cycle is identified and solved.

The theory forms a basis for a computationally effective and comprehensive theory of language understanding by conceptual inference. Numerous computer examples are included to illustrate key points. Most issues are approached from both psychological and computational points of view, and the thesis is intended to be comprehensible to people with a limited background in computers and symbolic computation.

⊗ AIM-234 cs431 not at NTIS
Kenneth Mark Colby, Roger C. Parkison, Bill Faught,
Pattern-Matching Rules for the Recognition of Natural Language Dialogue Expressions,
23 pages, June 1974.

Man-machine dialogues using everyday conversational English present difficult problems for computer processing of natural language. Grammar-based parsers which perform a word-by-word, parts-of-speech analysis *are* too fragile to operate satisfactorily in real time interviews allowing unrestricted English. In constructing a simulation of paranoid thought processes, we designed an algorithm capable of handling the linguistic expressions used by interviewers in teletyped diagnostic psychiatric interviews. The algorithm uses pattern-matching rules which attempt to characterize the input expressions by progressively transforming them into patterns which match, completely or fuzzily, abstract stored patterns. The power of this approach lies in its ability to ignore recognized and unrecognized words and still grasp the meaning of the message. The methods utilized are general and could serve any "host" system which takes natural language input.

+ AIM-235 CS-432 ADA006898/1WC
Richard W. Weyhrauch, Arthur J. Thomas,
FOL: A Proof Checker for First-order Logic,
57 pages, September 1974. Cost: $3.30

This manual describes a machine implementation of an extended version of the system of natural deduction described by Prawitz. This language, called FOL, extends Prawitz's formulation to a many-sorted logic allowing a partial order *over* sorts. FOL also allows deductions to be made in some intuitionistic, modal and strict-implication logics. It is intended to be a vehicle for the investigation of the metamathamatics of first-order systems, of problems in the theory of computation and of issues in representation theory.

⊗ AIM-236 CS-433 AD784513/4WC
Jack R. Buchanan and David C. Luckham,
On Automating the Construction of Programs,
65 pages, May 1974.

An experimental system for automatically generating certain simple kinds of programs is described. The programs constructed are expressed in a subset of ALGOL containing assignments, function calls, conditional statements, while loops, and non-recursive procedure calls. The input is an environment of primitive programs and programming methods specified in a language currently used to define the semantics of the output programming language. The system has been used to generate programs for symbolic manipulation, robot control, everyday planning, and computing arithmetical functions.

+ AIM-237 CS-436
Yorick Wilks,
Natural Language Understanding Systems Within the AI Paradigm - A Survey and Some Comparisons,
40 pages, December 1974. Cost: $2.85

The paper surveys the major projects on the understanding of natural language that fall within what may now be called the artificial intelligence paradigm for natural language systems. Some space is devoted to arguing that the paradigm is now a reality and different in significant respects from the generative paradigm of present day linguistics. The comparisons between systems center around questions of the relative perspicuity of procedural and static representations; the aduantages and disadvantages of developing systems over a period to test their limits; and the degree of agreement that now exists on what are the sorts of information that must be available to a system that is to understand everyday language.

⊗ AIM-238 CS-437 A DA 005040
Christopher K. Riesbeck,
Computational Understanding: Analysis of Sentences and Context,
Thesis: Ph.D. in Computer Science,
245 pages, May 1974.

The goal of this thesis was to develop a system for the computer analysis of written natural language texts that could also serve a a theory of human comprehension of natural language. Therefore the construction of this system was guided by four basic assumptions about natural language comprehension. First, the primary goal of comprehension is always to find meanings as soon as possible. Other tasks, such as discovering syntactic relationships, are performed only when essential to decisions about meaning. Second, an attempt is made to understand each word as soon as it is read, to decide what it means and how it relates to the rest of the text. Third, comprehension means not only understanding what has been seen but also predicting what is likely to be seen next. Fourth, the words of a text provide the cues for finding the information necessary for comprehending that text.

⊗ AIM-239 CS-438 AD786720/3WC
Marsha Jo Hannah,
Computer Matching of Areas in Stereo Images,
Thesis: Ph.D. in Computer Science,
99 pages, July 1974.

This dissertation describes techniques for efficiently matching corresponding areas of a stereo pair of images. Measures of match which are suitable for this purpose are discussed, as are methods for pruning the search for a match. The mathematics necessary to convert a set of matchings into a workable camera model are given, along with calculations which use this model and a pair of image points to locate the corresponding scene point. Methods are included to detect some types of unmatchable target areas in the original data and for detecting when a supposed match is invalid, Region growing techniques are discussed for extend matching areas into regions of constant parallax and for delimiting uniform regions in an image. Also, two algorithms are presented to show some of the ways in which these techniques can be combined to perform useful tasks in the processing of stereo images.

⊗ AIM-240 cs-444 AD787035
C. Cordell Green, Richard J. Waldinger,
David R. Barstow, Robert Elschlager, Douglas
B. Lenat, Brian P. McCune, David E. Shaw,
and Louis I. Steinberg,
Progress Report on Program-understanding
Systems,
47 pages, August 1974.

This progress report covers the first year and one half of work by our automatic programming research group at the Stanford Artificial Intelligence Laboratory. Major emphasis has been placed on methods of program specification, codification of programming knowledge, and implementation of pilot systems for program writing and understanding. List processing has been used as the general problem domain for this work.

+ AIM-241 CS-446 AD786723/7WC
Luigia Aiello, Richard W. Weyhrauch,
LCFsmall: an Implementation of LCF,
45. pages, August 1974. Cost: $2.95

This is a report on a computer program implementing a simplified version of LCF. It is written (with minor exceptions) entirely in pure LISP and has none of the user oriented features of the implementation described by Milner. We attempt to represent directly in code the metamathematical notions necessary to describe LCF. We hope that the code is simple enough and the metamathematics is clear enough so that properties of this particular program (e.g. its correctness) can eventually be proved. The program is reproduced in full.

⊗ AIM-242 CS-452 ADA000500/9WC
James R. Low,
Automatic Coding: Choice of Data
Structures,
Thesis: Ph.D. in Computer Science,
110 pages, August 1974.

A system is described which automatically chooses representations for high-level information structures, such as sets, sequences, and relations for a given computer program. Representations are picked from a fixed library of low-level data structures including linked-lists, binary trees and hash tables. The representations are chosen by attempting to minimize the predicted space+time integral of the user's program execution. Predictions are based upon statistics of information structure use provided directly by the user and collected by monitoring executions of the user program using default representations for the high-level structures. A demonstration system has been constructed. Results using that system are presented.

⊗ AIM-243 CS-456 ADA003815
Raphael Finkel, Russel Taylor, Robert Bolles,
Richard Paul, Jerome Feldman,
AL, A Programming System for
Automation,
130 pages, November 1974.

AL is a high-level programming system for specification of manipulatory tasks such as assembly of an object from parts. AL includes an ALGOL-like souce language, a translator for converting programs in runnable code, and a runtime system for controlling manipulators and other devices. The system includes advanced features for describing individual motions of manipulators, for using sensory information, and for describing assembly algorithms in terms of common domain-specific primitives. This document describes the design of AL, which is currently being implemented as a successor to the Stanford WAVE system.

+ AIM-244 cs-457 not at NTIS
Kenneth Mark Colby,
Ten Criticisms of PARRY,
7 pages, September 1974. Cost: $1.90

Some major criticims of a computer simulation of paranoid processes (PARRY) are reviewed and discussed.

⊗ AIM-245 CS-458 AD784816/1WC
Jack Buchanan,
A Study in Automatic Programming,
Thesis: Ph.D. in Computer Science,
148 pages, May 1974.

A description of methods and an implementation of a system for automatic generation of programs is given. The problems of writing programs for numerical computation, symbol manipulation, robot control and everyday planning have been studied and some programs generated. A particular formalism, i.e. a FRAME, has been developed to define the programming environment and permit the statement of a problem. A frame, F, is formulated within the Logic of Programs [Hoare 1969, Hoare and Wirth 1972] and includes--primitive functions and procedures, axioms definitions and rules of program composition. Given a frame, F, a problem for program construction may be stated as a pair <I,G>, where I is an input assertion and G is an output assertion. The program generation task is to construct a program A such that $I\{A\}I'$, where $I' \supset G$. This process may be viewed as a search in the Logic of Programs for a proof that the generated program satisfies the given input-output assertions. Correctness of programs generated using the formal algorithm is discussed.

⊗ AIM-246 CS459 ADA000085/1WC
Terry Winograd,
Five Lectures on Artificial Intelligence,
93 pages, September 1974.

This publication is a slightly edited transcription of five lectures delivered at the Electrotechnical Laboratory in Tokyo, Japan from March 18 to March 23, 1974. They were intended as an introduction to current research problems in Artificial Intelligence, particularly in the area of natural language understanding. They are exploratory in nature, concentrating on open problems and directions for future work. The five lectures include: A survey of past work in natural language understanding; A description of the SHRDLU system; A comparison of representations used in AI programs; A rough sketch of some ideas for a new representation which combines features of the previous ones; A discussion of the applications of these ideas to programming systems.

⊗ AIM-247 CS-461 ADA005041/9WC
Neil Goldman,
Computer Generation of Natural Language
From a Deep Conceptual Base,
Thesis: Ph.D. in Computer Science,
318 pages, January 1974.

For many tasks involving communication between humans and computers it is necessary for the machine to produce as well as understand natural language. We describe an implemented system which generates English sentences from Conceptual Dependency networks, which are unambiguous, language-free representations of meaning. The system is designed to be task independent and thus capable of providing the language generation mechanism for such diverse problem areas as question answering, machine translation, and interviewing.

+ AIM-248 CS-462
Karl Pingle, Arthur Thomas,
A Fast, Feature-Driven Stereo Depth
Program,
15 pages, May 1975. Cost: $2.15

In this paper we describe a fast, feature-driven program for extracting depth information from stereoscopic sets of digitized TV images. This is achieved by two means: in the simplest case, by statistically correlating variable-sized windows on the basis of visual texture, and in the more complex case by pre-processing the images to extract significant visual features such as corners, and then using these features to control the correlation process.

The program runs on the PDP-10 but uses a PDP-I 1/45 and an PSP-41 Signal Processing

Computer as subsidiary processors. The us of the two small, fast machines for the performance of simple but often-repeated computations effects an increase in speed sufficient to allow us to think of using this program as a fast 3-dimensional segmentation method, preparatory to more complex image processing. It is also intended for use in visual feedback tasks involved in hand-eye coordination and automated assembly. The current program is able to calculate the three-dimensional positions of 20 points in an image to within 5 millimeters in less than 5 seconds of computation.

+ AIM-249 CS-463 A DA 00226 1
Bruce Baumgart,
Geometric Modeling for Computer Vision,
Thesis: Ph.D. in Computer-Science,
14 I pages, October 1974. Cost: $5.65

A 3-D geometric modeling system for application to computer vision is described. In computer vision geometric models provide a goal for descriptive image analysis, an origin for verification image synthesis, and a context for spatial problem solving. Some of the design ideas presented have been implemented in two programs named CEOMED and CRE; the programs are demonstrated in situations involving camera motion relative to a static world.

⊗ AIM-250 CS-464 A DA 003486
Ramakant Nevatia,
Structured Descriptions of Complex Curved Objects for Recognition and Visual Memory,
Thesis:. Ph.D. in Electrical Engineering,
126 pages, October 1974.

Description and recognition of three-dimensional objects from range data obtained by a laser triangulation technique are described. A complex object is described by decomposition into sub-parts and relations of these sub-parts. The individual parts are described by generalized cones, which are defined by a space curve known as the axis, and arbitrary shaped normal cross-sections along this axis.

Techniques for segmenting an object into sub-parts and generating structured, symbolic, graph like descriptions are described. These symbolic descriptions are matched with stored descriptions and the best match is picked for recognition. A limited amount of indexing capability exists to efficiently retrieve a sub-class of similar objects from the models stored in the memory. Indexing is a necessity if a large number of visual models is to be used.

Results of working programs for the stated tasks on many actual scenes are presented. The scenes consist of single as well as multiple models is to be used.

⊗ AIM-251 CS-465 ADA001373
Edward H. Shortliffe,
MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection,
Thesis: Ph.D. in Medical lnformation Sciences,
409 pages, October 1974.

This thesis describes a rule-based problem-solving system, termed MYCIN, which is designed to assist physicians with the selection of appropriate therapy for patients with bacterial infections. After a brief survey of medical computing, with an emphasis on computer-based medical decision making, the report describes the clinical problem and th design considerations necessary for a consultation program to gain acceptance by the physicians for whom it is intended. The three system components are then described in detail: 1) a Consultation System which interacts with the physician and gives therapeutic advice, 2) an Explanation System which seeks to justify the program's adivce, and 3) a Rule-Acquisition System which accepts rules from experts and codes them for use during future consultation sessions. MYCIN's quantitative model of inexact reasoning in medicine is also described in detail, and the results of an evaluation study comparing MYCIN's advice to that of experts are presented. The report closes with speculations regarding future extensions and applications of a system such as MYCIN and with a discussion of the program's contributions to medical decision making and artificial intelligence.

+ AIM-252 CS-466 A DA 002246
Lester Earnest (ed.),
Recent Research in Artificial Intelligence, Heuristic Programming, and Network Protocols,
74 pages, July 1974. Cost: $3.80

This is a progress report for ARPA-sponsored research projects in computer science for the period July 1973 to July 1974. Accomplishments are reported in artificial intelligence (especially heuristic programming, robotics, theorem proving, automatic programming, and natural language understanding), mathematical theory of computation, and protocol development for communication networks. References to recent publications are provided for each topic.

+ AIM-253 cs-471 ADA003487
Bill Faught, Kenneth Colby, Roger Parkison,
The interaction of Inferences, Affects, and Intentions in a Model of Paranoia,
38 pages, December 1974. Cost: $2.75

The analysis of natural language input into its underlying semantic content is but one of the tasks necessary for a system (human or non-human) to us natural language. Responding to natural language input requires performing a number of tasks: 1) deriving facts about the input and the situation in which it was spoken; 2) attending to the system's needs, desires, and interests; 3) choosing intentions to fulfill these interests; 4) deriving and executing actions from these intentions. We describe a series of processes in a model of paranoia which performs these tasks. We also describe the modifications made by the paranoid processes to the normal processes. A computer program has been constructed to test this theory.

+ AIM-254 CS-472 ADA005407/2WC
Lynn Quam, Marsha Jo Hannah,
Stanford Automatic Photogrammetry Research,
15 pages, November 1974. Cost: $2.15

This report documents the feasibility study done at Stanford University's Artificial Intellignece Laboratory on the problem of computer automated aerial/orbital photogrammetry. The techniques investigated were based on correlation matching of small areas in digitized pairs of stereo images taken from high altitude or planetary orbit, with the objective of deriving a S-dimensional model for the surface of a planet.

t AIM-255 CS-473 ADA005412/2WC
Norihisa Suzuki,
Automatic Program Verification II: Verifying Programs by Algebraic 256 Logical Reduction,
29 pages, December 1974. Cost: $2.50

Methods for verifying programs written in a higher level programming language are devised and implemented. The system can verify programs written in a subset of PASCAL, which may have data structures and control structures such as WHILE, REPEAT, FOR, PROCEDURE, FUNCTION and COROUTINE. The process of creation of verification conditions is an extension of the work done by Igarashi, London and Luckham which is based on the deductive theory by Hoare. Verification conditions are proved using specialized simplification and proof techniques, which consist of an arithmetic simplifier, equality replacement rules, fast algorithm for simplifying formulas using propositional truth value evaluation, and a depth first proof search process. The basis of deduction mechanism used in this prover is Gentzen-type formal system. Several sorting programs including Floyd's TREESORTS and Hoare's FIND are verified. It is shown that the resulting array is not only well-ordered but also a permutation of the input array.

● AIM-256 CS-474 ADA007563/0WC
Friedrich W. V.Henke, David C. Luckham,
Automatic Program Verification III: A Methodology for Verifying Programs,

45 pages, December 1974.

The paper investigates methods for applying an on-line interactive verification system designed to prove properties of PASCAL programs. The methodology is intended to provide techniques for developing a debugged and verified version starting from a program, that - is possibly unfinished in some respects, - may not satisfy the required specifications,i.e., may contain bugs, - may be incompletely documented in the sense that the assertions provided by the programmer are not sufficient for proving correctness. It deals with programs that may be written in non-standard ways, e.g., permits user defined data structures.

The methodology involves - techniques for describing data structures, type constraints, and properties of programs and rubprograms (i.e. lower level procedures); - the use of (abstract) data types in structuring programs and proofs. - interactive application of a verification condition generator, an algebraic simplifier and a theorem-prover;

Within each unit (i.e. segment of a problem), the interactive use is aimed at reducing verification conditions to manageable proportions so that the non-trivial factors may be analysed. Analysis of verification conditions attempts to localize errors in the program logic, to extend assertions inside the program, to spotlight additional assumptions on program subfunctions beyond those already specified by the programmer, and to generate appropriate lemmas and assumptions that allow a verification to be completed. Methods for structuring correctness proofs are discussed.

A detailed case study of a pattern matching algorithm illustrating the various aspects of the methodology (including the role played by the user) is given.

● AIM-257 CS-475 ADA005407/2WC
Malcolm C. Newey,
Formal Semantics of LISP With Applications to Program Correctness,
Thesis: Ph.D. in Computer Science,
184 pages, January 1975.

Described are some experiments in the formaiisation of the LISP programming language using LCF (Logic for Computable Functions). The bulk of each experiment was concerned with applying the formalisation to proofs of correctnes of some interesting LISP functions using Milner's mechanised version of LCF.

A definition of Pure LISP is given in an environment which includes an axiomatisation of LISP S-expressions. A primitive theory (a body of theorems in LCF) of Pure LISP is derived and is applied to proving the correctness of some simple LISP functions using the LCF proof checking system. A proof of correctness of McCarthy's interpreter is described and a machine checked proof of the partial correctness is outlined,

A more substantial subset of LISP and a subset of LAP (a LISP-oriented assembly language for the PDP-10 computer) were formalised and simple theories for the two languages were developed with computer assistance. This was done with a view to proving the correctness of a compiler, written the LISP subset, which translates LISP functions to LAP subroutines. The coarse structure of such a compiler correctness proof is displayed.

Particular attention is paid, in describing the experiments, to deficiencies revealed in the expressive power of LCF as a logical language and to limitations on the deductive power of the machine implementation of the logic.

⊗ ATM-258 CS-476 ADA006294/3WC
Cordell Green, David **Barstow,**
A Hypothetical Dialogue Exhibiting **a Knowledge** Base for a **Program-**
Understanding System,
38 pages, January 1975.

A hypothetical dialogue with a fictitious program-understanding system is presented. In the interactive dialogue the computer carries out a detailed synthesis of a simple insertion sort program for linked lists. The content, length and complexity of the dialogue reflect the underlying programming knowledge which would be required for a system to accomplish this task. The nature of the knowledge is discussed and the codification of such programming knowledge is suggested as a major research area in the development of program-understanding systems.

+ AIM-259 CS-498
Hanan Samet,
Automatically **Proving** the Correctness of Translations **Involving** Optimized Code,
Thesis: PAD in Computer Science,
2 14 pages, May 1975. Cost: $7.70

A formalism is described for proving that programs written in a higher level language are correctly translated to assembly language. In order to demonstrate the validity of the formalism a system has been designed and implemented for proving that programs written in a subset of LISP 1.6 as the high level language are correctly translated to LAP (an assembly language for the PDP-10) as the low level language. This work involves the identification of critical semantic properties of the language and their interrelationship to the instruction repertoire of the computer executing these programs. A primary use of the system is as a postoptimization step in code generation as well as a compiler debugger.

The assembly language programs need not have been generated by a compiler and in fact may be handcoded, The primary restrictions on the assembly language programs relate to calling sequences and well-formedness. The assembly language programs are processed by a program understanding system which -simulates their effect and returns as its result a representation of the program in the form of a tree.

The proof procedure is independent of the intermediary mechanism which translates the high level language into the low level language. A proof consists of applying valid transformations to show the equivalence of the forms corresponding to the assembly language program and the original higher level language program, for which there also exists a tree-like intermediate form.

Some interesting results include the ability to handle programs where recursion is implemented by bypassing the start of the program, the detection and pinpointing of a wide class of errors in the assembly language programs, and a deeper understanding of the question of how to deal automatically with translations between high and extremely low level languages.

⊗ AIM-260 CS-499 ADA01681 1/2WC
David Canfield Smith,
PYGMALION: A Creative Program **ming Environment,**
*Thesis: **PhD** in Computer Science,*
193 pages, June 1975.

PYGMALION is a two-dimensional, visual programming system implemented on an interactive computer with graphics display. Communication between human being and computer is by means of visual entities called "icons", subsuming the notions of "variable", "reference", "data structure", "function" and "picture". The heart of the system is an interactive "remembering" editor for icons, which executes and (optionally) saves operations for later re-execution. The display screen is viewed as a document to be edited. Programming consists of creating a sequence of display frames, the last of which contains the desired information. Display frames are modified by editing operations. PYGMALION employs a powerful paradigm that can be incorporated in virtually any other programming language:

Every operation has both visual (aesthetic) semantics and internal (mechanical) semantics.

In fact, every operation in PYGMALION has three responsibilities:

(a) for accomplishing a given internal machine task – the machine "semantics" of the operation;

(b) in display mode, for generating a representative visual action;

(c) in remember mode, for adding onto a code list the operation(s) necessary to reproduce itself.

Thus the system includes an incremental "iconic compiler". Since each operation has visual semantics, the display becomes a visual metaphor for computing. The programmer need deal with operations only on the display level; the corresponding machine semantics are managed automatically. The mechanical aspects of programming languages has been and is continuing to be well studied. The focus in this paper is on developing and interacting with an articulate visual presentation.

PYGMALION is a computational extension of the brain's short term memory. It is designed to relieve the load on the short term memory by providing alternative storage for mental images during thought. The display screen is seen as a "dynamic blackboard", on which ideas can be projected and animated. Instead of abstract symbols, the programer uses explicit display images. Considerable flexibility is provided for designing icons; the programmer may give them any shape that can be generated by a routine. This helps to reduce the translation distance between representations used in the mind in thinking about a problem and representations used in programming the problem.

The main innovations of PYGMALION are:

(1) a dynamic representation for programs – an emphasis on doing rather than telling;

(2) an iconic representation for parameters and data structures requiring less translation from mental representations;

(3) a "remembering" editor for icons;

(4) descriptions in terms of the concrete, which PYGMALION turns into the abstract.

The responsive, visual characteristics of PYGMALION permit it to play an active role in human problem solving. The principal application has been in assisting the design and simulation of algorithms.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

+ AIM-26 1 CS-501 ADA016808/8WC
Odd Pettersen,
Procedural Events as Software Interrupts,
8 pages, June 1975. Cost: $1.95

The paper deals with procedural events, providing a basis for synchronization and scheduling, particularly applied on real-time program systems of multiple parallel activities ("multi-task").

There is a great need for convenient scheduling mechanisms for minicomputer systems as used in process control, but so far mechanisms somewhat similar to those proposed here are found only in PL/I among the generally known high-level languages. PL/I, however, is not very common on computers of this size. Also, the mechanisms in PL/I seem more restricted, as compared to those proposed here.

A new type of boolean program variable, the EVENTMARK, is proposed. Eventmarks represent events of any kind that may occur within a computational process and are believed to give very efficient and convenient activation and scheduling of program modules in a real-time system. An evenfmark is declared similar to a procedure, and the proposed feature could easily be amended as an extension to existing languages, as well as incorporated in future language designs.

+ AIM-262 CS-502 **ADA016810/4WC**
Odd Pettersen,
Synchronization of Concurrent Processes,
14 pages, July 1975. Cost: $2.10

This paper gives an overview of commonly used synchronization primitives and literature, and presents a new form of primitive expressing conditional critical regions.

A new solution is presented to the problem of "readers and writers", utilizing the proposed synchronization primitive. The solution is simpler and shorter than other known algorithms. The first sections of the paper give a tutorial introduction into established methods, in order to provide a suitable background for the remaining parts.

+ AIM-263 cs-503
Odd Pettersen,
The Macro Processing System **STAGE2:**
Transfer of **Comments** to the **Generated Text**,
20 pages, July 1975. Cost: $2.25

This paper is a short description of a small extension of STAGE2, providing possibilities to copy comments etc. from the source text to the generated text. The description presupposes familiarity with the STAGE2 system: its purpose, use and descriptions, like [1] to [9]. Only section 3 of this paper requires knowledge of the internal structures and working of the system, and that section is unnecessary for the plain use of the described feature.

The extension, if not used, is completely invisible to the user: No rules, as described in the original litterature, are changed. A user, unaware of the extension, will see no difference from the original version.

+ AIM-264 CS-506
Michael Gordon,
Operational Reasoning and Denotational
Semantics,
33 pages, August 1975. Cost: $2.65

"Obviously true" properties of programs can be hard to prove when meanings are specified with a denotational semantics. One cause of this is that such a semantics usually abstracts away from the running process - thus properties which are obvious when one thinks about this lose the basis of their obviousness in the absence of it. To enable process-based intuitions to be used in constructing proofs one can associate with the semantics an abstract interpreter so that reasoning about the semantic can be done by reasoning about computations on the interpreter. This technique is used to prove several facts about a semantics of pure LISP. First a denotational semantics and an abstract interpreter are described. Then it is shown that the denotation of any LISP form is correctly computed by the interpreter. This is used to justify an inference rule - called "LISP-induction" which formalises induction on the size of computations on the interpreter. Finally LISP-induction is used to prove a number of results. In particular it is shown that the function eval is correct relative to the semantics - i.e. that it denotes a mapping which maps forms (coded asd S-expressions) on to their correct values.

+ AIM-265 cs-507
Michael Gordon,
Towards a Semantic Theory of Dynamic Binding,
28 pages, August 1975. Cost: $2.50

The results in this paper contribute to the formulation of a semantic theory of dynamic binding (fluid variables). The axioms and theorems are language independent in that they don't talk about programs - i.e. syntactic objects - but just about elements in certain domains. Firstly the equivalence (in the circumstances where it's true) of "tying a knot" through the environment (elaborated in the paper) and taking a least fixed point is shown. This is central in proving the correctness of LISP "eval" type interpreters. Secondly the relation which must hold between two environments if a program is to have the same meaning in both is established. It is shown how the theory can be applied to LISP to yield previously known facts.

+ AIM-266 cs-517 ADA019641
Randall Davis, Bruce Buchanan, Edward Shortliffe,
Production Rules as a Representation for a Knowledge-Based Consultation Program,
37 pages, October 1975. Cost: $2.75

The MYCIN system has begun to exhibit a high level of performance as a consultant on the difficult task of selecting antibiotic therapy for bactercmia. This report discusses issues of representation and design for the system. We describe the basic task and document the constraints involved in the use of a program as a consultant. The control structure and knowledge representation of the system are examined in this light, and special attention is given to the impact of production rules as a representation. The extent of the domain independence of the methodology is also examined.

+ AIM-267 CS-520 ADA019664/2WC
Friedrich W. von Henke,
On the Representation of Data Structures in LCF with Applications to Program Generation,
41 pages, September 1975. Cost: $2.85

In this paper we discuss techniques of exploiting the obvious relationship between program structure and data structure for program generation. We develop methods of program specification that are derived from a representation of recursive data structures in the Logic for Computable Functions (LCF). As a step towards a formal problem specification language we define definitional extensions of LCF. These include a calculus for (computable) homogeneous sets and restricted quantification. Concepts that are obtained by interpreting daa types as algebras are used to derive function definition schemes from an LCF term representing a data structure; they also lead to techniques for the simplification of expressions inthe extended language. The specification methods are illustrated with a detailed example.

+ AIM-268 CS-521 ADA019663/4WC
Clark Thompson,
Depth Perception In Stereo Computer Vision,
16 pages, October 1975. Cost: $2.15

This report describes a stereo vision approch to depth perception; the author has build upon a set of programs that decompose the problem in the following way: 1) Production of a camera model: the position and orientation of the cameras in 3-space. 2) Generation of matching point-pairs: loci of corresponding features in the two pictures. 3) Computation of the point in S-space for each point-pair. 4) Presentation of the resultant depth information.

+ AIM-269 CS-522 ADA019569/3WC
David C. Luckham, Norhisa Susuzki,
Automatic Program Verification IV: Proof of Termination Within a Weak Logic of Programs,
29 pages, October 1975. Cost: $2.50

A weak logic of programs is a formal system in which statements that mean "the program halts" cannot be expressed. In order to prove termination, we would usually have to use a stronger logical system. In this paper we show how we can prove termination of both iterative and recursive programs within a weak logic by adding pieces of code and placing restrictions on loop invariants and entry conditions. Thus, most of the existing verifiers which are based on a weak logic of programs can be used to--prove termination of programs without any modifidation. We give examples of proofs of termination and of accurate bounds on computation time that were obtained using the Stanford Pascal program verifier.

+ AIM-270 CS-523 ADA019467
John F. Reiser,
BAIL – A debugger for SAIL,
26 pages, October 1975. Cost: $2.45

BAIL is a debugging aid for SAIL programs, where SAIL is an extended dialect of ALGOL60 which runs on the PDP-10 computer. BAIL consists of a breakpoint package and an expression interpreter which allow the user to stop his program at selected points, examine and change the values of variables, and evaluate general SAIL expressions. In addition, BAIL can display text from the source file corresponding to the current location in the program. In may respects BAIL is like DDT or RAID, except that BAIL is oriented towards SAIL and knows about SAIL data types, primitive operations, and procedure implementation.

t AIM-271 CS-524 ADA019702/0WC
Randall David, Jonathan King,
An Overview of Production Systems,
40 pages, October 1975. Cost: $2.85

Since production systems were first proposed in 1943 as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. This paper is an attempt to provide an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other, and to other methodologies.

Accordingly, we use the term 'production system' in a broad sense, and attempt to show how most systems which have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally-based techniques, but with reference also to some of the current developments in programming and the organization of data and knowledge bases.

This is a slightly revised version of a paper to appear in *Machine Representations of Knowledge,* Dordrecht, D. Reidel Publishing Company (1976).

+ AIM-272 CS-525
Sundaram Ganapathy,
Reconstruction of Scenes Containing Polyhedra From Stereo Pair of Views,
Thesis: Ph.D. in Computer Science,
204 pages, December 1975. Cost: $7.40

The problem of constructing a 3-D description of a scene given two views of it, taken from widely different angles, is attacked in this thesis. The program accepts line drawing data as input and uses a large number of rules to match the corresponding features in the two views. These rules are derived from observation and are based on the fact that the scene consists of polyhedral bodies only.

There are several possible approaches to the problem. We have taken one approach which involves matching up the corresponding vertices and building up from that. The problem of matching up the corresponding vertices is combinatorial in nature and is somewhat analogous to graph matching. However the structure of objects provides helpful clues and constrains the search considerably. In fact quite often no search is' necessary, if the rules *are* combined properly.

Towards this end, the individual rules have been studied in great detail and their power analysed both from a theoretical and experimental standpoint. We have developed a theoretical framework, in which the power and. the probability of application of these rules can be studied. A scheme has been designed which combines these rules in such a way so as to make the best match (based on scoring functions) the right match with a very high probability. But in as much as no algorithm exists for doing vision, the best match is not always the right match. However a wrong match would result in an interpretation that is inconsistent. Ideas have been developed which will verify the consistency of a match. A procedure has been designed which will backtrack and correct the errors in an incorrect match.

All these ideas have been implemented as a computer program which works extremely well on idealised drawings. The problems encountered in applying these ideas to real data have been analysed. Finally suitable modifications have been made to make these same ideas work well on real-data as well.

- AIM-273 cs-534

Linda Gail Hemphill,
A Conceptual Approach to Automated Language Understanding and Belief Structures: with Disambiguatiora of the Word 'For',
Thesis: Ph.D. in Linguistics,
254 pages, May 1975.

This thesis deals with the problem of human language understanding, that is, what kinds of information does a person use in order to be able to understand what is said to him. The word "for" is examined because it can have more than twenty different meanings, and yet, a person rarely misinterprets an instance of "for" or finds it ambiguous.

The problem is approached from the standpoint of a computer understanding model, that is, what kinds of information must a computer understanding model have to interpret sequences of language, in particular, those in which "for" occurs, as an American English-speaking person might. This model would of necessity be idiosyncratic, since a person's idiosyncratic background determines the way in which he interprets certain utterances.

It is shown that a conceptual approach to analysis must be used in order for an understanding system to perform the tasks that a human being does in understanding. In order for an understanding model to assign a meaning representation to utterances, it must manipulate conceptual information. For example, the model must make inferences from the sentences under analysis; it must analyze two syntactically different sentences which are paraphrases of each other into the same meaning representation; and it must interact with a memory structure; each of these tasks requires that a conceptual approach to language analysis be used. Conceptual Dependency Theory is the approach used here.

The memory structure required by an

understanding model must have certain types of conceptual information. The memory information that must interact in the disambiguation of "for" is varied. Certain conceptual features that apply to objects must be specified, as well as conceptual features that apply to actions; the concept of different Scales, and terms that designate evaluations on those scales, interact in the understanding of "for". Often the understanding model must interact with *Expectancy **Rules**.* The Expectancy Rules deal with the different types of non-linguistic information that interact in the understanding process. The Expectancy Rule is of the form:** IF situation A occurs, THEN EXPECT B. This type of rule . interacts constantly in language understanding, and examples are given where this type of rule must interact in a-. model of language generation. This type of rule (or one might say "belief") is so basic that people do not feel the need to state it explicitly in language, and thus it must be in the memory of an understanding system.

The context of an utterance, both linguistic and non-linguistic, determines the way in which that utterance is interpreted. Therefore, the understanding model must store information as a "conversation" proceeds, because context ultimately determines the meaning of "for", not the sentence which contains "for". Specific procedures for the disambiguation of each meaning of "for" are given, which are based on elements of the "for" sentence itself; however, context can set up a completely different-interpretation for a "for" by providing the *conceptual format* underlying a particular meaning of "for", in which case the model would choose the contextual interpretation.

The theory of an understanding model that would correctly interpret all instances of "for" points out many of the problems that any natural language understanding model must handle, and the types of information needed for an understanding system to correctly interpret "for" were shown to interact in other instances of language understanding and generation as well.

+ AIM-274 CS-536 ADA020942/9WC
David Grossman, Russell Taylor,
Interactive Generation of Object Models with a Manipulator,
32 pages, December 1975. Cost: $2.60

Manipulator programs in a high level language consist of manipulation procedures and object model declarations. As higher level languages are developed, the procedures will shrink while the declarations will grow. This trend makes it desirable to develop means for automating the generation of these declarations. A system is proposed which would permit users to specify certain object models interactively' using the manipulator itself as a measuring tool in three dimensions. A preliminary version of the system has been tested.

+ AIM-275 CS-537 ADA020943/7WC
Robert C. Bolles,
Verification Vision Within a Programmable Assembly System: An Introductory Discussion,
82 pages, December 1975. Cost: $4.00

This paper defines a class of visual feedback tasks called ***Verification** Vision* which includes a significant portion of the feedback tasks required within a programmable assembly system. It characterizes a set of general-purpose capabilities which, if implemented, would provide a user with a system in which to write programs to perform such tasks. Example tasks and protocols are used to motivate these semantic capabilities. Of particular importance are the tools required to extract as much information as possible from planning and/or training sessions. Four different levels of verification systems are discussed. They range from a straightforward interactive system which could handie a subset of the verification vision tasks, to a completely automatic system which could plan its own strategies and handle the total range of verification tasks. Several unsolved problems in the area are discussed.

+ AIM-276 CS-539 **ADA021055/9WC**
Zohar Manna, **Adi** Shamir,
A New Approach to Recursive Programs,
25 pages, December 1975. Cost: $2.40

In this paper we critically evaluate the classical least-fixed point approach towards recursive programs. We suggest a new approach which extracts the maximal amount of valuable information embedded in the programs. The presentation is informal, with emphasis on examples.

⊛ AIM-277 CS-542 ADA027454
Zohar Manna, **Adi** Shamir,
The Theoretical Aspects of-the Optimal **Fixedpoint**,
24 pages, March 1976.

In this paper we define a new type of fixedpoint of recursive definitions and investigate some of its properties. This optimal fixedpoint (which always uniquely exists) contains, in some sense, the maximal amount of "interesting" information which can be extracted from the recursive definition, and it may be strictly more defined than the program's least fixedpoint. This fixedpoint can be the basis for assigning a new semantics to recursive programs.

+ **A-IM-278** cs-549 ADA027455
David **Luckham,** Norihisa Suzuki,
Automatic Program Vcrificatiorr Vı
Verification-Oriented Proof Rules for
Arrays, Records and **Pointers,**
48 pages; March 1976. Cost: $3.05

A practical method is presented for automating in a uniform way the verification of Pascal programs that operate on the standard Pascal data structures ARRAY, RECORD, and POINTER. New assertion language primitives are introduced for describing computational effects of operations on these data structures. Axioms defining the semantics of the new primitives are given. Proof rules for standard Pascal operations on pointer variables are then defined in terms of the extended assertion language. Similar rules **for** records and arrays are special cases. An extensible axiomatic rule for the Pascal memory allocation operation, NEW, is also given.

These rules have been implemented in the Stanford Pascal program verifier. Examples illustrating the verification of programs which operate on list structures implemented with pointers and records are discussed. These include programs with side-effects.

⊛ AIM-279 CS-552
Norihsa Suzuki,
Autotnatic Verification of Programs with
Complex Data Structures,
*Thesis: **Ph.D.** in Computer Science,*
194 pages, February **1976.**

The problem of checking whether programs work correctly or not has been troubling programmers since the earliest days of computing. Studies have been conducted to formally define semantics of programming languages and derive proof rules for correctness of programs.

Some experimental systems have been built to mechanically verify programs based on these proof ruler, However, these systems are yet far from attacking *real* programs in a real environment. Many problems covering the ranges from theory to artificial intelligence and programming languages must be solved in order to make program verification a practical tool. First, we must be able to verify a complete practical programming language. One of the important features of real programming languages which is not treated in early experimental systems is complex data structures. Next, we have to study specification methods. In order to verify programs we have to express what we intend to do by the programs. In many cases **we** are not sure what we want to verify and how we should express them. These specification methods are not independent of the proof rules. Third, we have to construct an efficient prover so that we can interact with the verification process. It is expected that repeated verification attempts will be necessary because programs and specifications may have errors at first try. So the time to complete one verification attempt is *very* important in real environment.

We have chosen Pascal as the target language. The semantics and proof rules are studied by Howe & Wirth and Igarashi, London & Luckham. However, they have not treated complex data structures obtained from arrays, records, and pointers. In order to express the state of the data structures concisely and express the effects of statements we introduced special assertion language primitives and new proof rules. We defined new methods of introducing functions and predicates to write assertions so that we can express simplification rules and proof search strategies. We introduced a special language to document properties of these functions and predicates. These methods enable users to express assertions in natural ways so that verification becomes easier. The theorem prover is constructed so that it will be efficient for proving a type of formulas which appear very often as verification conditions.

We have successfully verified many programs. Using our new proof rules and specification methods we have proved properties of sorting programs such as permutation and stability which have been thought to be hard to prove. We see no theoretical as well as practical problems in verifying sorting programs. We have also verified programs which manipulate pointers. These programs change their data structures so that usually verification conditions tend to be complex and hard to read. Some study about the complexity problem seems necessary.

The verifier has been used extensively by various users, and probably the most widely used verifier implemented so far. There is yet a great deal of research necessary in order to fill the gap between the current verifier and the standard programming tools like editors and compilers.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

+ AIM-280 cs-555
David D. Grossman,
Monte Carlo Simulation of Tolerancing in Discrete Parts Manufacturing and Assembly,
25 pages, May 1976. Cost: $2.40

The assembly of discrete parts is strongly affected by imprecise components, imperfect fixtures and tools, and inexact measuremets. It is often necessary to design higher precision into the manufacturing and assembly process than is functionally needed in the final product. Production engineers must trade off between alternative ways of selecting individual tolerances in order to achieve minimum cost, while preserving product integrity. This paper describes a comprehensive Monte Carlo method for systematically analysing the stochastic implications of tolerancing and related forms of imprecision. The method is illustrated by four examples, one of which is chosen from the field of assembly by computer controlled manipulators.

+ AIM-28 1.1 CS-558 AD-A031 406/2WC
Zohar Manna, Richard Waldinger,
Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness,
41 pages, June 1976, revised March 1977.
Cost: $2.85

This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, which we call the [intermittentl-[assertion method], involves documenting the program with assertions that must be true at some time when control is passing through the corresponding point, but that need not be true every time. The method, introduced by Knuth and further developed by Burstall, promises to provide a valuable complement to the more conventional methods.

We first introduce and illustrate the technique with a number of examples. We then show that a correctness proof using the invariant assertion method or the subgoal induction method can always be expressed using' intermittent assertions instead, but that the reverse is not always the case. The method can also be used just to prove termination, and any proof of termination using the conventional well-founded sets approach can be rephrased as a proof using intermittent assertions. Finally, we show how the method can be applied to prove the validity of program transformations and the correctness of continuously operating programs.

This is a revised and simplified version of a pevious paper with the same title (AIM-281, June *1976).*

+AIM-282 *CS-560*
Russell Taylor,
Synthesis of Manipulator Control Programs from Task-level Specifications,
Thesis: Ph.D. in Computer Science,
229 pages, July 1976. Cost: $8.10

This research is directed towards automatic generation of manipulator control programs from task-level specifications. The central assumption is that much manipulator-level coding is a process of adapting known program constructs to particular tasks, in which coding decisions are made by well-defined computations based on *planning information.* For manipulator programming, the principal elements of planning information are: (1) descriptive information about the objects being manipulated; (2) situational information describing the execution-time environment; and (3) action information defining the task and the semantics of the execution-time environment.

A standard subtask in mechanical assembly, insertion of a pin into a hole, is used to focus the technical issues of automating manipulator coding decisions. This task is first analyzed from the point of view of a human programmer writing in the target language, AL, to identify the specific coding decisions required and the planning information required to make them. Then, techniques for representing this information in a computationally useful form are developed. Objects are described by attribute graphs, in which the nodes contain shape information. the links contain structural information, and properties of the links contain location information. Techniques are developed for representing object locations by parameterized mathematical expressions in which free scalar variables correspond to degrees of freedom and for deriving such descriptions from symbolic relations between object features. Constraints linking the remaining degrees of freedom are derived and used to predict maximum variations. Differential approximations are used to predict errors in location values. Finally, procedures are developed which use this planning information to generate AL code automatically.

The AL system itself performs a number of coding functions not normally found in algebraic compilers. These functions and the planning information required to support them are also discussed.

⊗ AIM-283 *CS-552*
Randall Davis,
Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Know ledge Bases,
Thesis: Ph.D. in Computer Science,
304 pages, July 1976.

The creation and management of large knowledge bases has become a central problem of artificial intelligence research as a result of two recent trends: an emphasis on the use of large stores of domain specific knowledge as a base for high performance programs, and a concentration on problems taken from real world settings. Both of these mean an emphasis on the accumulation and management of large collections of knowledge, and in many systems embodying these trends much time has been spent on building and maintaining such knowledge bases. Yet there has been little discussion or analysis of the concomitant problems. This thesis attempts to define some of the issues involved, and explores steps taken toward solving a number of the problems encountered. It describes the organization, implementation, and operation of a program called TEIRESIAS, designed to make possible the interactive transfer of expertise from a human expert to the knowledge base of a high performance program, in a dialog conducted in a restricted subset of natural language.

The two major goals set were (*i*) to make it possible for an expert in the domain of application to "educate" the performance program directly, and (ii) to ease the task of assembling and maintaining large amounts of knowledge.

The central theme of this work is the exploration and use of what we have labelled *meta level knowledge*. This takes several different forms as its use is explored, but can be summed up generally as "knowing what you know". It makes possible a system which has both the capacity to use its knowledge directly, and the ability to examine it, abstract it, and direct its application.

We report here on the full extent of the capabilities it makes possible, and document cases where its lack has resulted in significant difficulties. Chapter 3 describes efforts to enable a program to explain its actions, by giving it a model of its control structure and an understanding of its representations. Chapter 5 documents the use of abstracted models of knowledge (rule *models*) as a guide to acquisition. Chapter 6 demonstrates the utility of describing to a program the structure of its representations (using data *structure schemata*). Chapter 7 describes the use of strategies in the form of *meta* rules, which contain knowledge about the use of knowledge.

⊗ AIM-284 cs-567
Rafael Finkel,
Constructing and Debugging Manipulator Programs,
Thesis: Ph.D. in Computer Science,
171 pages pages, August 1576.

This thesis presents results of work done at the Stanford Artificial Intelligence Laboratory in the field of robotics. The goal of the work is to program mechanical manipulators to accomplish a range of tasks, especially those found in the context of automated assembly. The thesis has three chapters describing significant work in this domain. The first chapter is a textbook that lays a theoretical framework for the principal issues involved in computer control of manipulators, including types of manipulators, specification of destinations, trajectory specification and planning, methods of interpolation, force feedback, force application, adaptive control, collision avoidance, and simultaneous control of several manipulators. The second chapter is an implementation manual for the AL manipulator programming language. The goals of the language are discussed, the language is defined, the compiler described, and the execution environment detailed. The language has special facilities for condition monitoring, data types that represent coordinate systems, and affixment structures that allow coordinate systems to be linked together. Programmable side effects play a large role in the implementation of these features. This chapter closes with a detailed programming example that displays how the constructs of the language assist in formulating and encoding the manipulation task. The third chapter discusses the problems involved in programming in the AL language, including program preparation, compilation, and especially debugging. A debugger, ALAID, is designed to make use of the complex environment of AL. Provision is made to take advantage of the multiple-processor, multiple-process, real-time, interactive nature of the problem. The principal conclusion is that the debugger can fruitfully act as a uniform supervisor for the entire process of program preparation and as the means of communication between cooperating processors.

⊗ AIM-285 CS-568 PB-259 130/3WC
T. O. Einford, D. D. Grossman, C. R. Lui, R. C. Bolles, R. A. Finkel, M. S. Mujtaba, M. D. Roderick, B. E. Shimano, R. H. Taylor, R. H. Goldman, J. P. Jarvis, V. D. Scheinman, T. A. Gafford,
Exploratory Study of Computer Integrated Assembly Systems, Progress Report 3,
336 pages, August 1976.

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices, including computer controlled manipulators and vision systems. A complete hardware system has been implemented that includes manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals. An advanced software system call AL has been developed that can be used to program: assembly applications. Research currently underway includes refinement of AL, development of improved languages and interactive programming techniques for assembly and vision, extension of computer vision to areas which are currently infeasible, geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

⊗ AIM-286 cs-570
Douglas Lenat,
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search,
Thesis: Ph.D. in Computer Science,
350 pages, July 1976.

A program, called "AM", is described which models one aspect of elementary mathematics research: developing new concepts under the guidance of a large body of heuristic rules. "Mathematics" is considered as a type of intelligent behavior, not as a finished product.

+ AIM-287 cs-571
Michael Roderick,
Discrete Control of a Robot Arm,
Thesis: Engineer in Electrical Engineering,
98 pages, August 1976. Cost: $4.45

The primary goal of this thesis was to determine the feasibility of operating the Stanford robot arm and reduce sample rates. A secondary goal was to reduce the effects of variations in inertia and sampling rates on the control system's stability.

A discrete arm model was initially developed to illustrate the effects of inertia and sampling rate variations on the present control system. Modifications were then suggested for reducing these effects. Finally, a method was demonstrated for reducing the arm sampling rate from its present value of 60 hertz to approximately 45 hertz without significantly effecting the arms performance.

+ AIM-288 CS-572
Robert Filman, Richard Weyhrauch,
An FOL Primer,
36 pages, September 1976, Cost: $2.70

This primer is an introduction to FOL, an interactive proof checker for first order logic. Its examples can be used to learn the FOL system, or read independently for a flavor of our style of interactive proof checking. Several example proofs are presented, successively increasing in the complexity of the FOL commands employed.

FOL runs on the computer at the Stanford Artificial Intelligence Laboratory. It can be used over the ARPA net after arrangements have been made with Richard Weyhrauch (network address RWW@SU-AI).

+ AIM-2S9 cs-574
John Reiser (ed.),
SAIL,
178 pages, August 1976. Cost: $6.70

SAIL is a high-level programming language for the PDP-10 computer. It includes an . extended ALGOL 60 compiler and a companion set of execution-time routines. In addition to ALGOL, the language features: (1) flexible linking to hand-coded machine language algorithms, (2) complete access to the PDP- 10 I/O facilities, (3) a complete system of compile-time arithmetic and logic as well as a flexible macro system, (4) a high-level debugger, (5) records and references, (6) sets and lists, (7) an associative data structure, (8) independent processes, (9) procedure variables, (IO) user modifiable error handling, (11) backtracking, and (12) interrupt facilities.

This manual describes the SAIL language and the execution-time routines for the typical SAIL user: a non-novice programmer with some knowledge of ALGOL. It lies somewhere between being a tutorial and a reference manual.

+ AIM-290 cs-575
Nancy W. Smith,
SAIL Tutorial,
54 pages, November 1976. Cost: $3.20

This TUTORIAL is designed for a beginning user of Sail, an ALGOL-like language for the PDP 10. The first part covers the basic statements and expressions of the language; remaining topics include macros, records, conditional compilation, and input/output. Detailed examples of Sail programming are included throughout, and only a minimum of programming background is assumed.

⊗ AIM-291 cs-577
Bruce Buchanan, Joshua Lederberg, John McCarthy,
Three Reviews of J. Weizenbaum's Computer Power and Human Reason,
28 pages, November 1976.

Three reviews of Joseph Weizenbaum's Computer Power and Human Reason (W.H. Freeman and Co., San Francisco, 1976) are reprinted from other sources. A reply by Weizenbaum to McCarthy's review is also reprinted.

+ AIM-292 cs-580
Terry Winograd,
Towards a Procedural Understanding of Semantics,
30 pages, October 1976. Cost: $2.55

The term "procedural semantics" has been used in ‘a variety of ways, not all compatible, and not. all comprehensible. In this paper, I have chosen to apply the term to a broad paradigm for studying semantics (and in fact, all of linguistics). This paradigm has developed in a context of writing computer programs which use natural language, but it is not a theory of computer programs or programming techniques. It is "procedural" because it looks at the underlying structure of language as fundamentally shaped by the nature of processes for language production and comprehension. It is based on the belief that there is a level of explanation at which there are significant similarities between the psychological processes of human language use and the computational processes in computer programs we can construct and study. Its goal is to develop a body of theory at this level. This approach necessitates abandoning or modifying several currently accepted doctrines, including the way in which distinctions have been drawn between "semantics" and "pragmatics" and between "performance" and "competence".

The paper has three major sections. It first lays out the paradigm assumptions which guide the enterprise, and elaborates a model of cognitive processing and language use. It then illustrates how some specific semantic problems might be approached from a procedural perspective, and contrasts the procedural approach with formal structural and truth conditional approaches. Finally, it discusses the goals of linguistic theory and the nature of the linguistic explanation.

Much of waht is presented here is a speculation about the nature of a pradigm yet to be developed. This paper is an attempt to be evocative rather than definitive; to convey intuitions rather than to formulate crucial arguments which justify this approach over others. It will be successful if it suggests some ways of looking at language which lead to further understanding.

@ AIM-293 cs-5s 1
Daniel Bobrow, Terry Winograd,
An Overview of KRL,
40 pages, November 1976.

This paper describes KRL, a Knowledge Representation Language designed for use in understander systems. It outlines both the general concepts which underlie our research and the details of KRL-0, an experimental implementation of some of these concepts. KRL is an attempt to integrate procedural knowledge with a broad base of declarative forms. These forms provide a variety of ways to express the logical structure of the knowledge, in order to give flexibility in associating procedures (for memory and reasoning) with specific pieces of knowledge, and to control the relative accessibility of different facts and descriptions. The formalism for declarative knowledge is based on structured *conceptual objects* with associated *descriptions*. These objects form a network of *memory units* with several different sorts of linkages, each having well-specified implications for the retrieval process. Procedures can be associated directly with the internal structure of a conceptual object. This *procedural attachment* allows the steps for a particular operation to be determined by characteristics of the specific entities involved.

The control structure of KRL is based on the belief that the next generation of intelligent programs will integrate data-directed and goal-directed processing by using multi-processing. It provides for a priority-ordered multi-process agenda with explicit (user-provided) strategies for scheduling and resource allocation. It provides *procedure* directories which operate along with process *frameworks* to allow procedural parameterization of the fundamental system processes for building, comparing, and retrieving memory structures. Future development of KRL will include integrating; procedure definition with the descriptive formalism.

+ AIM-294 cs-586
Nachum Dershowitz, Zohar Manna,
The Evolution of Programs: A System for Automatic Program Modification,
45 pages, December 1976. Cost: 82.95

An attempt is made to formulate techniques of program modification, whereby a program that achieves one result can be transformed into a new program that uses the same principles to achieve a different goal. For example, a program that uses the binary search paradigm to calculate the square-root of a number may be modified to divide two numbers in a similar manner, or vice versa.

Program debugging is considered as a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results. the application of abstract program schemata to concrete problems is also viewed from the perspective of modification techniques.

We have embedded this approach in a running implementation; our methods are illustrated with several examples that have been performed by it.

+ AIM-295 cs-59 1
Robert C. Bolles,
Verification Vision Within a Programmable Assembly System,
Thesis: Ph.D. in Computer Science,
245 pages, December 1976. Cost: $8.55

The long-range goal of this research is to simplify visual information processing by computer. The research reported in this thesis concentrates on a subclass of visual information processing referred to as *verification* vision (abbreviated VV). VV includes a significant portion of the visual feed back tasks required within programmable assembly. There are several types of information available in VV tasks that can facilitate the solution of such tasks. The main question addressed in this thesis is how to use all of this information to perform the task efficiently. Two steps are involved in answering this question: (1) formalize the types of tasks, available information, and quantities of interest and (2) formulate combination rules that use the available information to estimate the quantities of interest.

The combination rules that estimate confidences are based upon Bayes' theorem. They are general enough to handle operators that are not completely reliable, i.e., operators that may find any one of several features or a surprise. The combination rules that estimate precisions are based upon a least-squares technique. They use the expected precisions of the operators to check the structural consistency of a set of matches and to estimate the resulting precisions about the points of interest. An interactive VV system based upon: these ideas has been implemented. It makes it possible for a person who is not an expert in vision research to program visual feedback tasks. This system helps the programmer select potentially useful operator/feature pairs, provides a training session to gather statistics on the behavior of the operators, automatically ranks the operator/feature pairs according to their expected contributions, and performs the desired task. The VV system has also been interfaced to the AL control system for the mechanical arms and has been tested on tasks that involve a combination of touch, force, and visual feedback.

+ AIM-296 CS-592
Robert Cartwright,
Practical Formal Semantic Definition and Verification Systems,
Thesis: Ph.D. in Computer Science,
158 pages, December 1976. Cost: $6.15

Despite the fact that computer scientists have developed, a variety of formal methods for proving computer programs correct, the formal verification of a non-trivial program is still a formidable task. Moreover, the notion of proof is so imprecise in most existing verification systems, that the validity of the proofs generated is open to question. With an aim toward rectifying these problems, the research discussed in this dissertation attempts to accomplish the following objectives:

1. To develop a programming language which is sufficiently powerful to express many interesting algorithms clearly and succintly, yet simple enough to have a tractable formal semantic definition.

2. To completely specify both proof theoretic and model theoretic formal semantics for this language using the simplest possible abstractions.

3. To develop an interactive program verification system for the language which automatically performs as many of the straightforward steps in a verification as possible. →[continued next page].univ .next page

The first part of the dissertation decribes the motivation for creating TYPED LISP, a variant of PURE LISP including a flexible data type definition facility allowing the programmer to create arbitrary recursive types. It is argued that a powerful data type definition facility not only simplifies the task of writing programs, but reduces the complexity of the complementary task of verifying those programs.

The second part of the thesis formally defines the semantics of TYPED LISP. Every function symbol defined in a program P is identified with a function symbol in a first order predicate calculus language Lp. Both a standard model Mp and a natural deduction system Np are defined for the language Lp. In the standard model, each function symbol is interpreted by the least call-by-value fixed-point of its defining equation. An informal meta-mathematical proof of the consistency of the model Mp and the deductive system Np is given.

The final part of the dissertation describes an interactive verification system implementing the natural deduction system Np.

The verification system includes:

1. A subgoaler which applies rules specified by the user to reduce the proof of the current goal (or theorem) to the proof of one or more subgoals.

'2. A powerful simplifier which automatically proves many non-trivial goals by utilizing user-supplied lemmas as well as the rules of Np.

With a modest amount of user guidance, the verification system has proved a number of interesting, non-trivial theorems including the total correctness of an algorithm which sorts by successive merging, the total correctness of the McCarthy-Painter compiler for expressions, the termination of a unification algorithm and the equivalence of an iterative algorithm and a recursive algorithm for counting the leafs of a tree. Several of these proofs *are* included in an appendix.

+ AIM-297 CS-610
Terry Winograd,
A Framework for Understanding Discourse,
24 pages, April 1977. Cost: $2.40

There is a great deal of excitement in linguistics, cognitive psychology, and artificial intelligence today about the potential of understanding discourse. Researchers are studying a group of problems in natural language which have been largely ignored or finessed in the mainstream of language research *over* the past fifteen years. They are looking into a wide variety of phenomena, and although results and observations are scattered, it is apparent that there are many interrelationships. While the field s not yet at a stage where it is possible to lay out a precise unifying theory, this paper attempts to provide a beginning framework for studying discourse. Its main goal is to establish a general context and give a feeling for the problems through examples and references. Its four sections attempt to:

Delimit the range of problems covered by the term "discourse."

Characterize the basic structure of natural language based on a notion of communication.

Propose a general approach to formalisms for describing the phenomena and building theories about them

Lay out an outline of the different schemas involved in generating and comprehending language

+ AIM-298 CS-6 11
Zohar Manna, Richard Waldinger,
The Logic of Computer Programming,
90 pages, June 1977. Cost: $4.25

Techniques derived from mathematical logic promise to provide an alternative to the conventional methodology for constructing, debugging, and optimizing computer programs. Ultimately, these techniques are intended to lead to the automation of many of the facets of the programming process.

In this paper, we provide a unified tutorial exposition of the logical techniques, illustrating each with examples. We assess the strengths and limitations of each technique as a practical programming aid and report on attempts to implement these methods in experimental systems.

+ AIM-299 CS-614
Zohar Manna, Adi Shamir,
The Convergence of Functions to
Fixedpoints of Recursive Definitions,
45 pages, May 1977. Cost: $2.95

The classical method for-constructing the least fixeclpoint of a recursive definition is to generate a sequence of functions whose initial element is the totally undefined function and which converges to the desired least fixedpoint. This method, due to Kleene, cannot be generalized to allow the construction of other fixedpoints.

In this paper we present an alternate definition of convergence and a new [fixedpoint access] method of generating sequences of functions for a given recursive definition. The initial function of the sequence can be an arbitrary function, and the sequence will always converge to a fixedpoint that is "close" to the initial function. This defines a monotonic mapping from the set of partial functions onto the set of all fixedpoints of the given recursive definition.

+ AIM-300 CS-617
Terry Winograd,
On some contested suppositions of
generative linguistics about the scientific
study of language,
25 pages, May 1977. Cost: $2.40

This paper is a response to a recently published paper which asserts that current work in artificial intelligence is not relevant to the development of theories of language. The authors of that paper declare that workers in AI have misconstrued what the goals of an explanatory theory of language should be, and that there is no reason to believe that the development of programs which could understand language in some domain could contribute to the development of such theories. This paper concentrates on the assumptions underlying their view of science and language. It draws on the notion of "scientific paradigms" as elaborated by Thomas Kuhn, pointing out the ways in which views of what a science should be are shaped by unprovable assumptions. It contrasts the procedural paradigm (within which artificial intelligence research is based) to the currently dominant paradigm typified by the work of Chomsky. It describes the ways in which research in artificial intelligence will increase our understanding of human language, and through an analogy with biology, raises some questions about the plausibility of the Chomskian view of language and the science of linguistics.