Integrated hypertext and program understanding tools

by P. Brown

This paper describes some concepts and issues related to software tools integration. Questions regarding data integration and functional integration between tools are identified and discussed. Some techniques for handling large volumes of data are briefly described. A prototype tool is described in which hypertext links are automatically created between program analysis data and hypertext documentation. With this tool, end users can freely move between source code views and related documentation. A common annotation feature lets software developers and information developers share information and synchronize maintenance activities in a single tools environment.

Unless a program is adequately understood, effective maintenance of the program is impossible. Understanding a program involves building a mental model that represents a programmer's current comprehension of the program.[1] In the case of large, complex systems, this mental model is extremely difficult to construct without automated assistance. Program maintenance requires a fundamental capability to parse and analyze programs, to extract appropriate information, and to organize and present that information so that it is useful to humans. We call this capability *program understanding*, and it is a crucial subtask in achieving many programming deliverables, such as sizings, high-level design, low-level design, build plans, actual code, and fixes.[2]

Documentation is an integral part of the software development and maintenance process. An examination of software development guidelines reveals that about two-thirds of the software development tasks involve creating, modifying, or reviewing documentation. For example, the object of software reviews is usually a document.[3] Large, complex systems frequently have thousands of pages of associated documentation. When a failure occurs, locating relevant sections of the documentation to determine the appropriate corrective action can be time-consuming. The problem is compounded because of currency mismatches caused by frequent modifications of documentation and code to accommodate engineering changes, product updates, and newly identified problems and solutions. Victor Basili, of the University of Maryland, comments about the importance of documentation in the January 1990 issue of *IEEE Software*:

> Modification of complex software systems requires a deep understanding of the functional and nonfunctional requirements, the mapping of functions to systems components, and the interaction of components. Without good documentation of the requirements, design, and code with respect to function, maintenance becomes a difficult, expensive and error-prone task.[4]

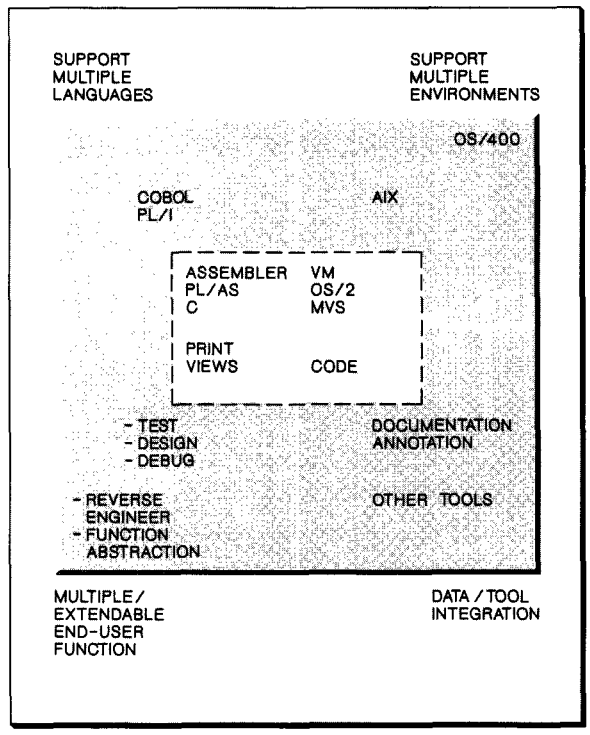
Faced with growing software bases, increased productivity demands, and mounting pressures to move into computer-aided software engineering (CASE) environments, software maintainers need a consistent mechanism to capture and organize information about their systems and to make it available to a variety of CASE-type tools. Although many of the CASE tools and methodologies are oriented either to application or system development, program understanding tools do not seem to have the same limitations. Because no specific methodology or process is enforced, the tool is not constrained by functionality or constructs contained in the system. An understanding tool should be applicable to the maintenance of both applications and systems (operating systems, database management systems, libraries, etc.) without substantial changes. User interfaces and navigation styles should be user-driven, allowing a user to follow an existing methodology or to use the tool in a discovery mode.

This paper describes a tool suite that integrates a hypertext tool and a program understanding tool. The first section introduces the topic and briefly describes the problems and concepts. Then Integrated Software Engineering Applications (ISEA), a cooperative processing tools platform, is presented. CodeNavigator, a program understanding tool, is discussed along with several design issues. Hypertext and related problems are summarized, and a hypertext tool, TRAILS (Text Retrieval And Information Linking System), is highlighted. A subsequent section discusses the integration of the tools and some of the issues involved in this process. The last section describes a scenario in which program understanding data and hypertext information are linked through integrated tools.

Problem summary

Software Engineering Tools is a tools development organization in IBM. Our "customers" are the IBM development laboratories that design large systems such as the System/370™ and the System/390™. Although the laboratories develop many different products, they are similar in that they maintain large bases of existing code and most of the products are intended to be used in conjunction with products developed in other laboratories. Most of the systems contain more than a million lines of code. Frequently, there are extensive functional and documentary linkages with

Figure 1 Maintenance tools requirements



other system products. Maintenance tools for the laboratories must be able to cope with large amounts of data and informational relationships between software products, possibly at different sites. Because of the large volumes of data, users need the capability to filter out unwanted or irrelevant information and maintain a sense of contextual continuity when exploring information from many sources.

A cooperative processing platform was developed to provide data and functional integration capabilities for the tools. Figure 1 gives an overview of our approach to the tools. It addresses four characteristics of our customers as described below.

Multiple environments—Our customers develop host operating systems and systems software. The Multiple Virtual Storage (MVS) and virtual machine (VM) operating systems have traditionally been the primary operating environments.

Multiple languages—Assembly language and PL/AS, an internal systems development language,

are the predominant programming languages. However, the base of C code is growing rapidly. Multiple language support must be present at the individual program level, since a program may be written in a higher-level language and contain inline assembler code.

Multiple data sources—Source libraries provide essential but incomplete information about systems. Valuable system knowledge is lost when documentation becomes outdated or when key personnel leave the project. Capabilities are required to capture system information contained in documentation, human knowledge, and information from other tools.

End-user function—Since program understanding and documentation are key parts of many other tasks, it was considered important to provide integration of function and data. End users also need to have a robust capability to manipulate the information, such as import or export functions, sophisticated print facilities, graph analysis tools, and automatic documentation capabilities.

The Integrated Software Engineering Applications platform

Integrated Software Engineering Applications (ISEA) is a tools platform developed to support CodeNavigator and other tools. It is a cooperative application with components on a host and a workstation. The host is either Multiple Virtual Storage/Extended Architecture (MVS/XA™) or Virtual Machine/Extended Architecture (VM/XA™). The workstation is a Personal System/2® (PS/2®) running Operating System/2® (OS/2®). The host is used as the primary data store, and the workstation is used for end-user interfaces. An overview of the ISEA platform is shown in Figure 2.

Although our expectation is that most tools will populate a central repository on the host and provide end-user function on the workstation, there is no requirement to do so. Tools developers are free to place function on either the host or the workstation in response to performance or other considerations.

The host component of CodeNavigator, for example, parses and analyzes the input data. After analysis, the resultant data are organized and loaded into the database. End-user presentation, certain data manipulations, and session services are provided on the workstation.

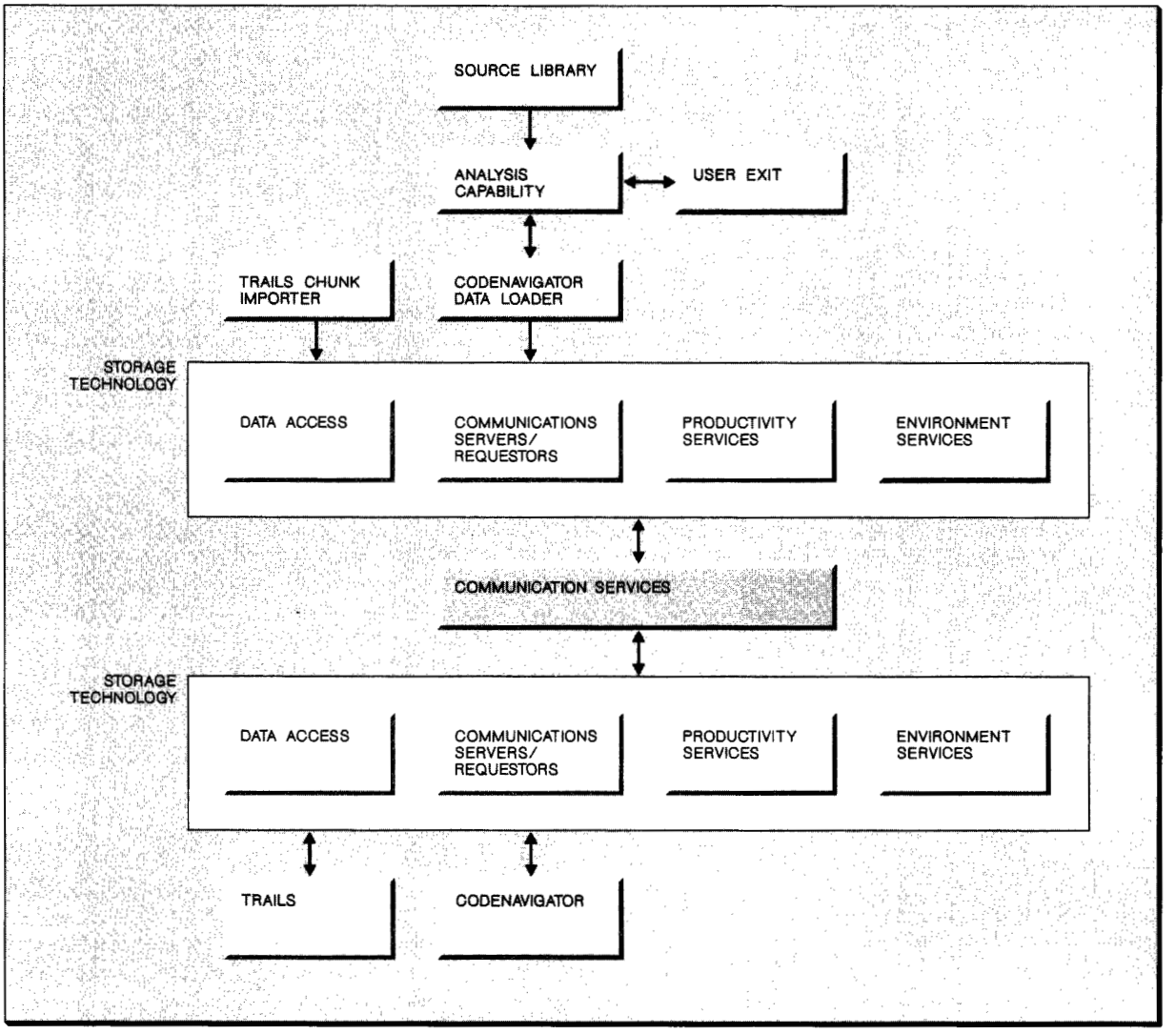
As we analyzed the requirements for several tools, it became apparent that many functions need to be provided for every tool. The requirements for dialog managers, graph services, memory management, and error handlers are similar from one tool to another. A set of building-block functions has been developed for use by tools and tool developers. The existence of services or building blocks enables the tool builder to focus on the specific business problem being implemented without worrying about universal functions which can, and should, become common utilities in an integrated tools suite. Inclusion of the several types of services shown in Figure 2 was driven by a desire to insulate the tools from environmental changes and to provide building blocks so new tools can be built. Approximately two-thirds of the CodeNavigator system is common ISEA code.

Tools developers are free to place function on either the host or the workstation.

Because of the evolving nature of our tools requirements, it was expected that the data content and structures would change. Our data access services, called the *object manager*, provide a layer between the application and the storage technology. Data are stored in the form of objects and relationships. By packaging related information into *objects*, we have been able to encapsulate the data and provide generic functions against the objects. Although inheritance is supported in the underlying data models, an object-oriented language is not currently provided for the tools. The object manager interface is very similar on the host and the workstation, providing the potential for functions to be moved freely between the host and the workstation.

The ISEA base supports both server-requester programming interface (SRPI) communications and advanced program-to-program communica-

Figure 2 ISEA system overview



tions (APPC) between the host and workstation. Each user has an option to choose either LU 2.0 or LU 6.2 communications at installation time. Thus, one set of application users might choose to use LU 2.0, whereas LU 6.2 might be chosen for another application. This choice provides some flexibility for applications to upgrade technology in a convenient time frame without being constrained by the tools platform.

Initial requirements stated that our tools operate on MVS and VM hosts. Additionally, we anticipated a requirement to provide an Advanced Interactive Executive™ (AIX®) version of our tools. Rather than build multiple versions of the tools, we developed a set of environmental services that isolate the tools from the operating environment. Currently both MVS/XA and VM/XA hosts are supported, and the workstation is using OS/2 Ex-

tended Edition 1.2. Migrations to new levels of OS/2 have been transparent to the applications, except where the tools chose to take advantage of new functions in the operating system.

A tool can be integrated with ISEA tools in several ways. The first way is to run entirely within ISEA, in which case both functional and data integration are a by-product of using the common services. An ISEA tool can be host-only, workstation-only, or cooperative, depending on the needs of the users. A second alternative is for a tool to exist outside of ISEA and call or be called by an ISEA tool. This alternative provides some degree of functional coupling and the possibility of sharing data between tools. A third alternative is for a tool to store or retrieve data using the ISEA object manager. This alternative provides an initial step toward data integration among independent tools.

Overview of CodeNavigator

Program understanding tools are intended to provide information about software systems. To be effective, they must be able to handle systems of significant size and be customized to meet the needs of a varied user set. Because program understanding is part of many development tasks (designing enhancements, problem-solving, testing, requirements analysis, etc.), the tools must provide a variety of levels of logical and physical views.

CodeNavigator is a program understanding tool designed to provide information about large-scale software systems. It is intended to improve productivity in:

- Educating programmers about software
- Analyzing system change requests
- Identifying software problems

The complete CodeNavigator system consists of a host and a PS/2. The host portion is used primarily for data gathering and data storage, whereas the PS/2 component provides the user interface that presents the derived information to the end user. The components of the CodeNavigator system are shown in the overview in Figure 3. Several components are of interest on the host:

The *batch controller* provides general control of the analysis processing. It controls access to the source libraries, invokes the appropriate language analysis, gives control to user exit routines when requested, and calls the functions to load the databases.

Analysis engines read the source code from a development library and create a physical model of the system. The analysis engine provides user exits either for manipulations of the source code before it is analyzed or to update the database following analysis, or both.

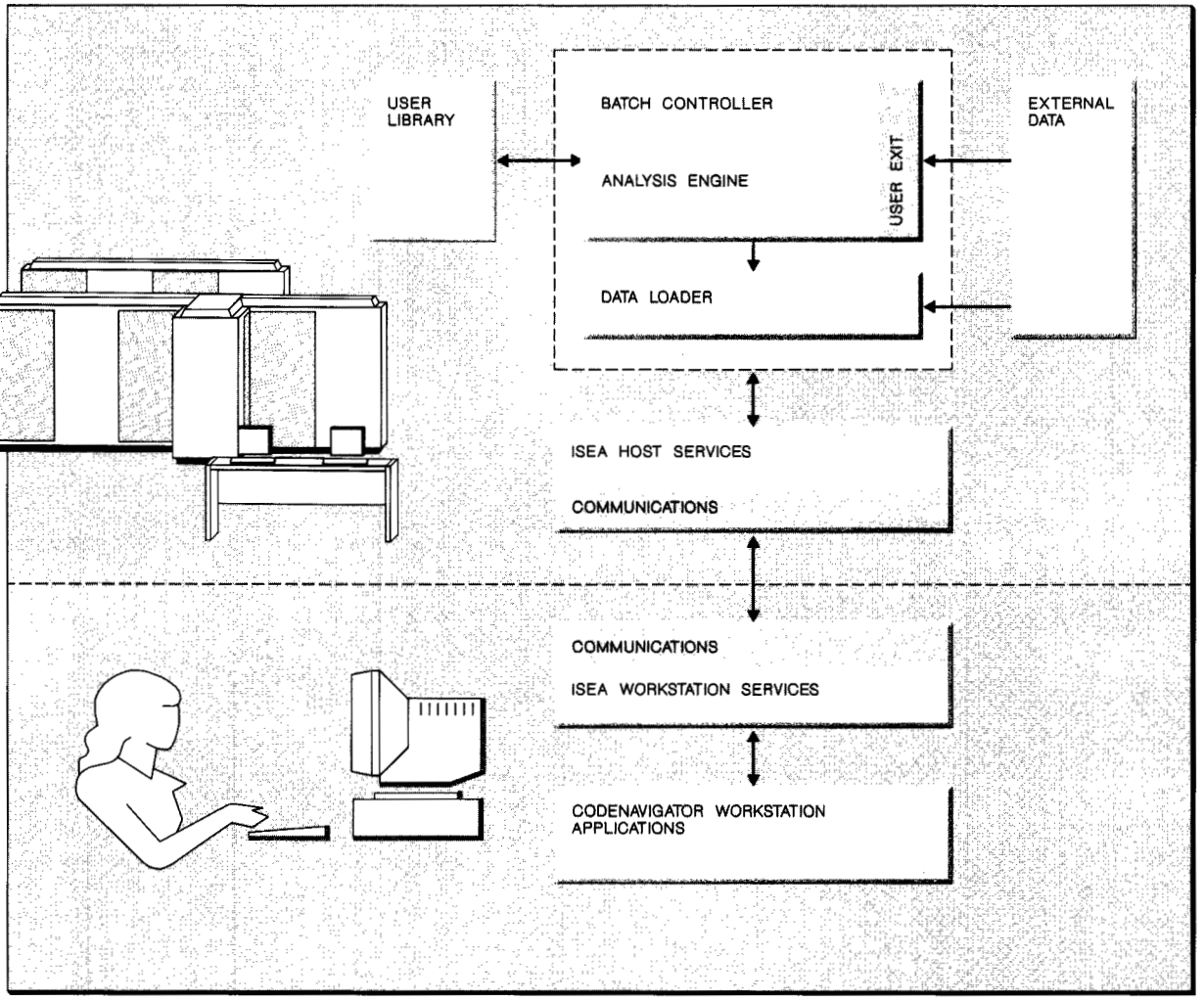
User exits enable each installation to customize the analysis of its code. Preprocessing exits can be used to enhance the analysis by making control flow or cross-reference data visible in situations where it might not normally be discovered. An example is providing information about routines that are commonly used but are not visible to the analysis engine, such as supervisor calls (SVCs) and system utilities or macros. Postprocessing provides information from outside sources or information that is derived from additional analysis of the database.

In the event that one or more modules are changed, only the changed modules need to be reanalyzed, and the database is updated with the changed information.

End-user interface. The end-user interface on the PS/2 provides a variety of logical views. A user may request a display of logical views of the individual source code modules or of the whole system. Each logical view is designed to assist the user's understanding of a particular aspect of the system being analyzed. The views can be summarized as follows:

- Lists are used to show traditional cross-reference information, such as *what-used, where-used, how-used* displays for symbols, macros, control blocks, subroutines, and modules.
- Directed graphs are used to show flow relationships. Logic flow and calling relationships at subroutine and module levels are displayed as directed graphs.
- Annotations may be created, browsed, updated, deleted, imported, or exported to either PS/2 or host files. The annotation is stored as an object in the database and may be associated with other objects in the database.
- Source code is displayed in a syntax-sensitive browser. The user can invoke other types of displays, such as cross-references, by pointing

Figure 3 CodeNavigator system overview



the cursor at tokens in the source code and selecting an action. Additionally, the source code display and the logic flow diagrams scroll synchronously, allowing you to easily follow the flow in both source and logic displays.

Program understanding design considerations. User requirements and early prototype experiences with CodeNavigator have identified several points that required special attention in our designs. The first category was the organization and internal representation of large amounts of data from a variety of sources. Representatives from several user sites and tools organizations worked together to develop a data model for the program understanding data.

The second broad class of problems was that of end-user presentations. CodeNavigator is used to assist education, design, testing, and other tasks. It is also intended to support users working in several languages and environments. The design of the user presentation had to accommodate these needs.

Host functions. The analysis functions and primary data store were placed on the host for the reasons of capacity and shared access. CodeNavi-

gator is intended to be used against large systems, and the direct-access storage device (DASD) requirements can be quite large. One of our performance benchmarks involves analyzing 500 000 lines of assembler code (KLOC). The resulting database exceeds the capacity of most workstation databases.

Most internal sites have their source libraries on the host. Given the large DASD requirements, existing host libraries, and the accessibility requirement, the decision to place analysis and data store functions on the host was straightforward.

Large quantities of data. Program analysis can create databases that may grow to many times the size of the original source library. Two key issues are deciding (1) what data to generate and save, and (2) how to most effectively store and present the data. The first issue is usually addressed by examining the user tasks and requirements. Although this approach seems straightforward, program comprehension and user interfaces are areas of continuing research. Therefore, user interface requirements and the data needed to support the requirements were not well understood at design time. We are approaching this problem by breaking the analysis into phases or stages and by using Prolog, a rules-oriented language.

Staged analysis allows us to incrementally extract or derive information from the source code and allows users to suppress creation of information that is not of interest to them. Staged analysis is used in providing the generic functions for control flow and data flow analysis shown in Figure 4. The rules-oriented approach offers the ability to add new types of analysis or to derive new data without major changes to existing parts of the analysis programs. Avoiding major changes was a consideration in the early prototypes when analysis changes were frequent.

The second issue is quite complex, oriented primarily around performance questions. The most obvious approach is to generate and store all required data as separate objects. However, the volume of data to be stored will probably create capacity issues in all but trivial cases. An alternative is to store only the essential raw data and to generate "derived" data when it is requested. This approach will reduce DASD usage but may introduce performance problems, since the query responses now involve the derivation of data.

Another alternative involves the "clustering" of related objects into a larger object. Clustering improves response time and reduces DASD usage, since fewer physical objects are stored and retrieved. However, it introduces some problems with cross-referencing of objects that are clustered. In our situation, symbol information accounted for approximately 60 percent of our captured data. Maintaining each symbol as a separate object required accessing and transferring hundreds of objects to satisfy a cross-reference request. Clustering them into a single file allowed us to satisfy the same request with one data access and one transfer.

We have explored several alternatives for optimizing query performance by caching data on the PS/2 or host database. The traditional approach is to request the data from the host database for each user query. An alternative is to store on the workstation all of the data that have been retrieved. The tool passes a request for information to the tools platform, and the workstation database is searched first. If the requested data are not found, a request is built and sent to the host database, and the appropriate data are returned. The tool never has any direct contact with the database and is not aware of whether the data come from the host or workstation. During session initialization, a check is made to see whether the workstation data have become obsolete, and the user is notified that the workstation database should be refreshed. A third approach is to attempt to anticipate which kind of data are likely to be requested next and to obtain the data in advance of the request. However, this approach presumes that the queries can be predicted.

CodeNavigator has implemented all three approaches. For most users, the first approach of going to the host for each request gives the most satisfactory performance. The second alternative, searching the workstation database before going to the host, is implemented as an installation option and is our intended direction. Performance is affected by overhead incurred when using the workstation database, and it makes this option undesirable to some users. Users who have small or heavily burdened host systems or who pay line charges for each data transfer find the second option very attractive. We have attempted a version of the third option, in which the system anticipates the next data request. Our experience was that we did not have sufficient

knowledge of usage patterns to anticipate effectively what data would be needed, and there was no appreciable benefit to the user who has some performance burden.

We have explored all of these performance alternatives to some degree. Our current approach is to tailor the solution to each circumstance. Because each of our sites has different host and workstation configurations, we have developed the ability to enable or disable caching as an installation parameter. Because of the variety of DASD situations at the various sites, we have developed user options to tailor the output of the analysis engines. We are using clustered objects to improve query response but maintain duplicate objects to support the indexing requirements.

CodeNavigator users can add data or perform additional code analysis through user exits in the analysis capability.

Individual needs of each site to add data. It is important to be able to extend the database with user-defined entities and to provide user exits for user-written functions before and after the standard analysis. Every organization we interviewed expressed a desire to extend the database with additional types of information. The data to be added were related to the system but not part of the general analysis. For example, operating system developers wanted to store information about the location of modules in the system nucleus, and support organizations wanted to keep problem report information. Also, a number of groups wanted to do additional analysis pertinent to their particular product but not suitable for general use.

CodeNavigator users have the ability to add data or perform additional code analysis through user exits in the analysis capability. Individual installations may add data or extract data that are important to them but not of general interest to other organizations.

Data to support activities. We interviewed maintenance programmers to get an understanding of what tasks they performed and what data they needed to perform those tasks. A consistent set of data views has been found to be common across activities such as education, analysis, design, and test. Constructing matrices of user tasks and the data needed to support the tasks has helped us understand how to design views that are usable across different programming activities. It was also of interest to note the combinations of data used by programmers to accomplish some particular task. Observing how programmers use CodeNavigator to perform the tasks is helping us refine the CodeNavigator displays to provide more complete information. Additionally, a comparison of tasks has identified opportunities to provide some user customization. Table 1 is an example of several tasks that are involved in learning about unfamiliar software. The subtasks listed in the table were identified as being typical tasks for programmers learning about unfamiliar code. The module listing is the primary source of information for most tasks. It is not assumed that every programmer performs all or any of these tasks on a regular basis; the tasks represent typical activities that a programmer might expect to perform as part of an assignment.

Although the programmers seem to have a relatively consistent set of tasks, they do not perform them consistently. This inconsistency suggests an opportunity to provide user-defined profiles that enable users to invoke a series of displays in some user-defined sequence. Thus, users could "program" the tool to perform specific tasks and to tailor the use of the tool to their particular preference or to local procedures.

Data models and multiple languages. The data model for CodeNavigator is a key element of almost every design decision. The content and structure of the data model drive the analysis and data extraction routines. End-user functions and displays are also constrained by the data model.

Our early prototype used a data model which, although quite simple, enabled us to provide a very useful set of displays. Modules are collected into logical groups by the user. Each module is made up of one or more subroutines, and each subroutine is made up of one or more blocks of code that have a single entry point. All lower-level data (data structures, macro, opcode, and

Table 1 Examples of educational program understanding tasks

Understanding Objectives	Steps in Understanding
Understand intended function	Read system documents to get description of function
Understand calling structure Develop a cross-reference of entry point calls	1. Find first module invoked in system documentation 2. From first module invoked, find flow to "my" modules 3. Build a cross-reference of entry point calls for my modules
Understand logic flow of module or entry point	1. Identify major functional pieces of code from listing 2. Develop flowchart from listing
Document understanding of modules Develop a notebook entry for each module	• Identify function of entry point • Define calling structure—who calls whom • Why this entry point is called (purpose) • List macros used • List control blocks used and updated • Write personal notes

symbol data) are related to the particular block or subroutine to which the data belong.

Although the initial data model was inadequate to support all of our requirements, it did support our initial set of functions. The limited support enabled us to try out many of the system features with a modest initial investment. By using the simple data model, we gained valuable experience in performance tuning and user interfaces that has influenced our design of future data models and functions.

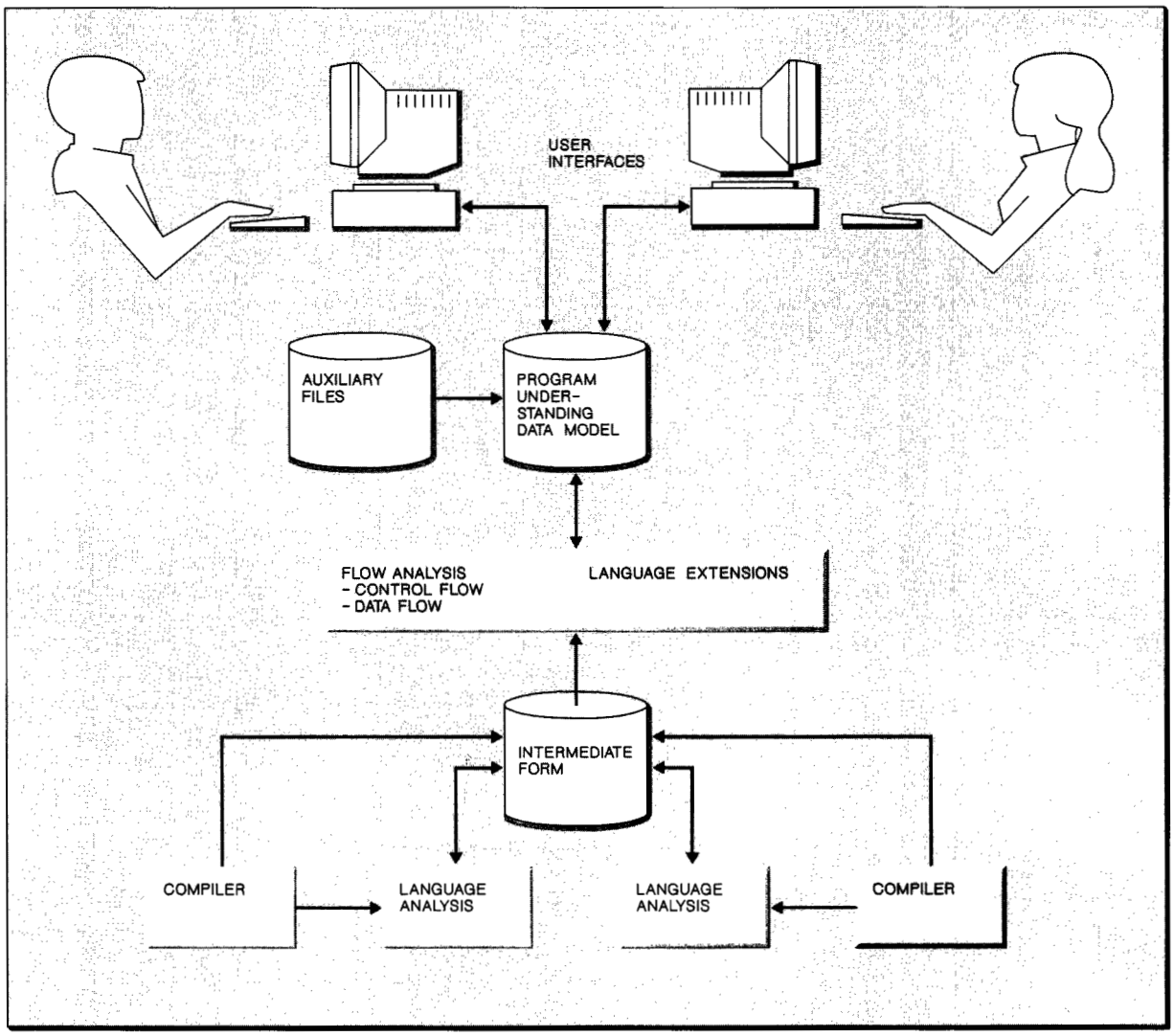
The new data model is a result of cooperation among tool developers and tool users from six sites. It supports the known requirements of the participating sites. Readers interested in the data model are encouraged to see the *IBM Systems Journal* article by Linore Cleveland, which refers to an earlier data model that influenced our model greatly.[5]

The model is intended to support multiple languages, particularly the languages commonly used in the laboratories: assembler, PL/AS, which is an IBM internally used language, and C. Some consideration was given to other languages, such as PL/I and COBOL, to ensure that no serious flaws were in the model. Currently CodeNavigator supports assembler in the laboratory sites, with PL/AS support to be provided in late 1991. Additionally, local versions of CodeNavigator analyze C and IBM Prolog to support the development of CodeNavigator itself.

Figure 4 illustrates our strategy for capturing information from multiple languages. A common language-independent representation has been defined for the output of the analysis, which we have called an intermediate form. This form has allowed us to build common routines to perform additional analysis and a single common database load capability. Aside from the obvious elimination of duplicate functions (a database loader for assembler and another loader for C, for example), it provides the capability to analyze mixed-language programs, which are quite common in our systems. When processing a mixed-language module, such as PL/AS and assembler, control is passed between the analyzers as the different languages are encountered. Since both analyzers will produce similar output, no gaps exist in the information generated for that module.

Several alternatives were examined before defining our own intermediate form. The new IBM Systems Application Architecture® (SAA™) compilers will have the capability to produce an output file for use by tools such as debuggers (methods to detect, diagnose, and eliminate program errors). Many compilers discard data element information during the optimization phase of compilation, making the output file unusable for our purposes. Design tool export files frequently are not granular enough to represent all of the implementation detail of a program, and many are restricted to structured constructs that make it difficult to represent an unstructured program.

Figure 4 Analysis of multiple languages



At this time, most analysis is done by custom-written parsers and analysis functions. It is desirable to have the analysis performed by compilers so as to eliminate the redundant effort of writing an analysis tool to perform parsing and analysis similar to that done by a compiler. It also eliminates the problem of keeping the custom-written analysis function up to date with language changes. One internally used compiler is producing output in our intermediate form.

Flexibility of end-user navigation. End users need the capability to survey, explore, and annotate the data they encounter. Each person does it differently, and no process or sequence can be imposed.[2] To a great extent, the techniques used depend on personal style; to some extent, on the task at hand. Our own experiences confirmed that programmers want the ability to jump from one view to another related view, since it is unusual for programmers to fully comprehend the system

they are trying to enhance or repair. Therefore, a key requirement was to allow the user to move from one view to any other contextually related view.

Our tool has a free-form navigation style that allows the user to control the scope and flow of investigation. Navigation in CodeNavigator is accomplished by selecting an object of interest within a bounded domain and invoking a specific query (e.g., "How is this symbol used in the following subroutines?" or "What is the logic flow of this module?"). The capability of simultaneously looking at several windows gives the user an ability to pursue several lines of investigation. The effect is similar to that of a programmer placing paper clips or bookmarks in a program listing to indicate places of interest or important discoveries.

User interfaces. Many of our systems contain modules written in several languages, and it was important that the end-user views remain consistent regardless of the source language being analyzed. This required developing a language-independent data model of our information and ensuring that our logical views were consistent across our target languages.

Frequently, users want to look at several logical views simultaneously. It is annoying for users to have to constantly reorient themselves in related views as they navigate through another view. Synchronous scrolling allows the control flow diagram and the source code browser to be viewed in tandem. When the user selects an object in one display, the other display is automatically reoriented, and the corresponding object is highlighted. Similarly, users can select an object from most list displays and request "Find in Source," which will display the source code and scroll to the first occurrence of the token selected.

The directed graphs that represent program logic flow or program calling relationships may become far larger than can be easily comprehended. This possibility has driven requirements for a number of usability features. Features such as a zoom capability and a focus window are needed to give users the ability to see a large graph and examine portions of interest in detail. Graph reduction is a method of decreasing the number of nodes and arcs while retaining the control structure of the graph. As the reduction takes place, each node in the reduced graph represents a subgraph of the original graph, and each arc in the reduced graph represents an arc in the original graph. In effect, we are collapsing structured subgraphs into a single node while retaining the overall control structure. For example, if a very complicated graph is reduced, the resulting graph will show the fundamental control structure, although much of the detail will have been eliminated.

We had originally expected that the workstation views would reduce user desire for documentation. However, we discovered that there is a very strong requirement for print capabilities. Two items of particular interest were the directed graphs representing calling relationships or logic flow and an ability to automatically produce a "programmer reference" document. A capture feature in CodeNavigator enables users to save a workstation display and pass the information to a host print facility. Some users are creating wall-sized posters of logic flow and calling structures as a hard-copy reference. A prototype automatic documentation facility allows the user to define the scope and level of detail to be included in the generated program reference document.

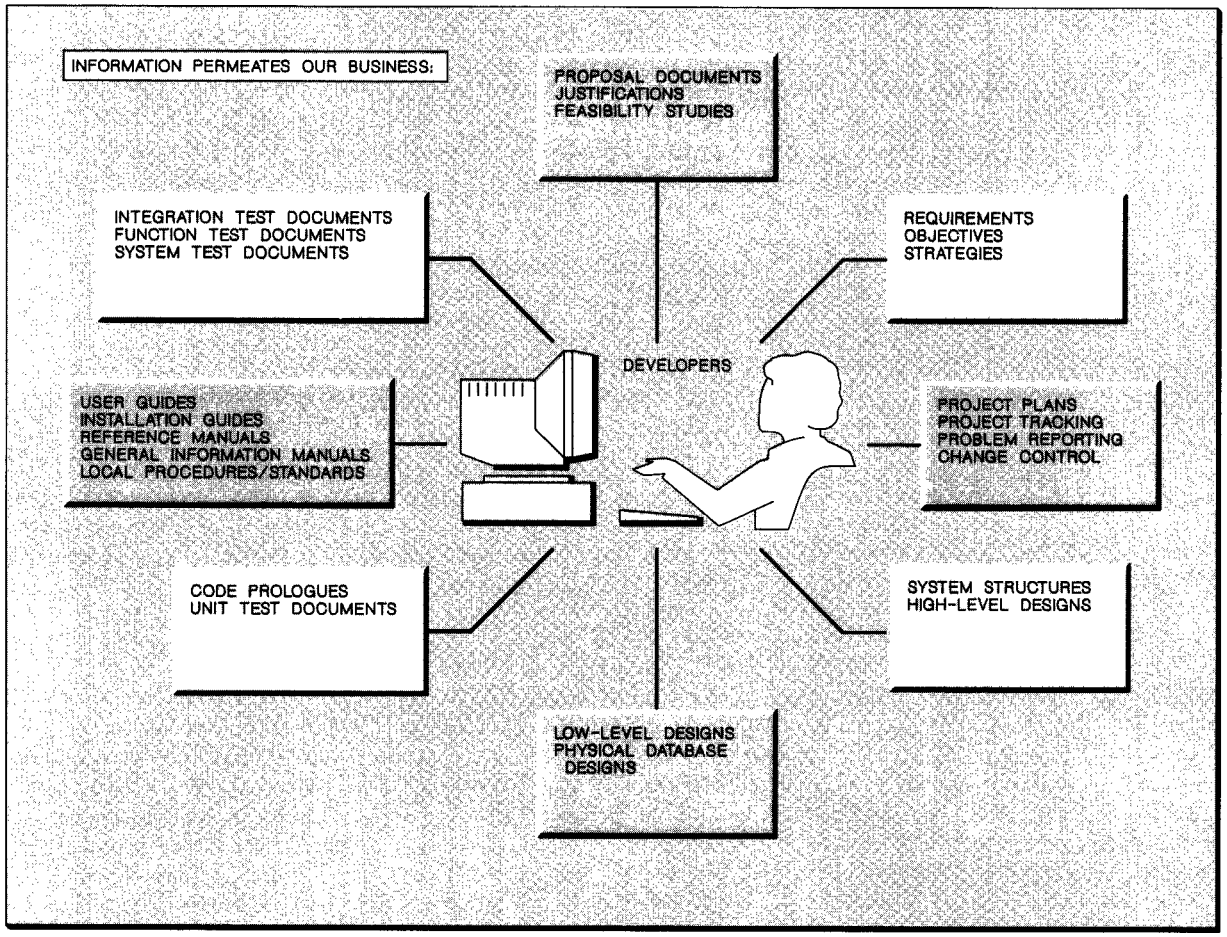
TRAILS overview

Hypertext. Hypertext is a way of displaying information on line and displaying "links" between different pieces of information. Links are used to show a reader that other information related to the current piece of information exists. Selecting among the links enables the reader to display the related information. Hypertext provides nonlinear access to data whereby the reader determines the path to be taken.

Hypertext tools are used by both authors and readers to create, manage, and understand relationships among documents. The tool must support multiple books or libraries of nontrivial size. There is a need to allow readers to integrate and link local documentation with system-supplied documents. For example, a reader may wish to link local coding standards to a programmer's reference manual or design document. This example requires authoring functions to identify and link the related pieces of information and reader functions to enable end users to find and follow the links.

Documentation is created and updated at every stage of the software development life cycle. Hypertext is a technology that enables authors and

Figure 5 Software life cycle documentation



readers to create, manage, and retrieve pertinent information throughout the development life cycle. The term *life cycle* implies that information is not static; it is constantly reviewed, updated, and recycled into other phases of the development cycle. Figure 5 provides an example of the documentation created in a software development cycle.

There are two general approaches to implementing hypertext:

- Book method—take existing books and put them on line and allow an author or a reader of the information to create or maintain links between chapters, sections, and subsections of the book and between books
- Node method—write the information into discrete "nodes" of information and interrelate the nodes with links

New documents may be written in a fashion to exploit hypertext tools and technologies. Existing documents must be rewritten or converted to be used in a hypertext tool. The TRAILS tool being described is a prototype hypertext tool used to explore issues in creating and managing hypertext documentation. It is a cooperative processing application with a host component (MVS or VM) and a workstation component on a PS/2. The host component provides the primary database and facilities for loading and managing the database. The workstation component provides user interfaces for retrieving, navigating, and manipulating hy-

pertext documents stored in the database. Figure 2 shows how TRAILS fits into the overall ISEA system.

The primary objective of the TRAILS prototype was to explore techniques and issues in developing hypertext documentation. In practice, it has also been useful for "hypertizing" existing softcopy documents, and it is currently being used to link and manage existing documentation in application maintenance organizations. Some features of the TRAILS prototype are:

- Text browser for displaying information with linkages and notes highlighted
- Graphic link map for displaying webs of related information
- Note facility to create, browse, update, delete, and maintain user-defined information
- Textual search within and across nodes
- Dynamic link creation, update, deletion, and traversal between words or phrases within or across nodes
- Author and reader mode of operation
- History display of actions during session that can be saved across sessions and re-executed
- Edit mode on nodes (for authors only) that allows text-editing while preserving links

Hypertext issues. Hypertext can be a valuable interactive medium for presenting information on line; it gives the reader the opportunity to navigate through related information according to interest, level of understanding, and other individually determined factors. Some important issues are evolving regarding the development and usage of hypertext information. Constructing a network of information nodes is not the same task as authoring an intelligible linear presentation of information. Some authoring issues are briefly described below.

Cognitive overhead (providing context). Marshall and Irish state the following:

> Although methods for maintaining coherence are more or less settled for conventional forms of writing, hypertext violates many of the assumptions underlying these methods. One important aspect of conventional forms absent from hypertext is the transitional text that helps the reader maintain a sense of materials coherence. The fragmentation characteristic of hypertext may also lead to a lack of interpretive context.[6]

The effect of following a hypertext link can be compared to reading a paragraph in one chapter and then suddenly reading another paragraph randomly chosen from the same book. Although the author who created the links knows the relationship between the paragraphs, the reader needs some additional clues to understand why and how they are related. The hypertext reader no longer has an understanding of sequence among paragraphs, and the explanatory text that introduces, concludes, and relates ideas to one another may not be visible.

Authors and hypertext system designers must be careful to provide contextual clues to assist the user. A variety of visual techniques are used in different systems, including persistent captions, information "beacons," graphical maps, and visual linkage across displays. These visual techniques provide the reader with a way of identifying a theme or consistent topic across many different displays. Another technique is to filter information so that only links that are relevant to the current topic are presented to the user.

TRAILS has developed several techniques to help a user understand what is being displayed and what other information is related to it. TRAILS users can limit an investigation to specific projects or domains. Requests for related information are bounded by the current project definition. Many times when a user is unsure of why a particular display is being shown, it is useful to recall the steps that preceded the display. A history function in TRAILS allows all or a portion of a session to be reviewed and re-executed. This history can be saved to provide an educational tutorial or "guided tour" for other users. Graphical maps position the user with regard to the current node and other related information nodes.

Lack of support for authoring activities. Authoring activities include idea processing, planning, organizing, and writing. When developing hypertext, an author is designing and developing an information web or network. The techniques and activities are often very different from those encountered in writing a book, which is usually written and read in a linear fashion. Hypertext authors may have to deal with issues of data models,

providing context for arbitrary queries, representation alternatives, and contextual filtering. Very few tools today provide support for these authoring activities.

Version 1 of the TRAILS prototype was developed to explore these types of issues. In the prototype, each node was stored as a bulk data file, and link information was maintained in the object manager. No authoring support was provided, so all organizing, filtering, and linking was done manually. The results of that study are guiding the development of a much more sophisticated data representation and user interface.

The organization and representation of hypertext is an area of active investigation.

Degree of integration among documents. With regard to using hypertext, Glushko states "The limited experimental literature on hypertext suggests that excessive integration through large numbers of links creates unusable spaghetti documents."[7]

There are really two aspects of this issue:

- What is the intended use of the hypertext information?
- How should the links be organized and represented in a reasonable fashion?

The intended use of the information will determine which documents should be linked and the types of linkages that are appropriate. Glushko offers this advice: "Select documents to include based on a user and task analysis. The extent to which the documents complement each other for the intended users and tasks will determine the extent to which it makes sense to combine them with hypertext links."[7] By developing task lists and identifying which documents are used and how they are used in support of the tasks, we can determine how much linkage and what kind of linkage is needed between documents.

The organization and representation of hypertext is an area of active investigation. A number of prototypes are using graph theoretic networks or data schema as an architecture for organizing the data. An underlying schema for the data simultaneously provides a great deal of power and the potential for severely limiting the use of the data. A data model has the advantage of enabling a tool to use established data management and query capabilities. A data model also provides opportunities to integrate other tools that have similar or overlapping data models. However, the data model must be carefully constructed to avoid constraining the use of the tool. In defining a data model, a fixed set of domains and contexts for the data is established. This fixed set may artificially constrain the use of hypertext data by imposing a contextual limitation that is not actually present in the document.

A subtlety of the task analysis mentioned earlier is that it may in fact artificially constrain the use of hypertext. Imagine that a task analysis is performed on testing activities and that a data model to represent the test activities is developed. There is a danger that documents linked using the test data model can only be referenced in a test context, even though they may be pertinent in design or other activities. Care must be taken to fully understand all possible or desired uses of data before establishing data models for hypertext.

Conversion of existing text into hypertext. Just as programmers are faced with the issue of understanding existing software, information developers are faced with enormous quantities of documentation about the existing software. Rewriting documentation for hypertext is often not economically feasible. Certainly a need exists for tools that can assist users in converting existing documents into hypertext. The question of what documents to convert or rewrite is one that must be answered on an individual basis. It seems likely that frequently used documents or documents describing critical asset software would be converted, assuming they were accurate. A task analysis that maps tasks against the documentation required for the tasks could be used to evaluate which documents are candidates for conversion or rewrites. Several articles on this topic may be found in the *Hypertext '87 Proceedings* published by the Association for Computing Machinery.[8]

TRAILS begins to address the problem by allowing documents to be "imported" with node delimiters imbedded in the formatted text. Linkages between nodes may be created manually or by running an automated analysis of the nodes. Several types of linking mechanisms will be discussed in a later section.

Readers of large hypermedia systems encounter major problems in navigation and contextual orientation. Some of the reader issues are now described.

Visualization and navigation of the underlying hypertext structures. The difficulty of navigation is described by Van Dyke Parunak: "One of the major problems confronting users of hypermedia systems is that of navigation: knowing where one is, where one wants to go, and how to get there from here."[9] This kind of problem may be encountered in several levels of detail. Examples of possible questions are: What is my overall frame of reference for this session?, What is the current subject and why am I looking at it?, and What specific document am I looking at?

The question of frame of reference deals with the intended use of the data and links. One approach is to use multiple models to represent the data in a hypertext database. Different models can be used to adjust the level of complexity, to provide alternative navigation schemes, and to present varying levels of detail. The second type of question may be caused by inadequate contextual clues or an unexpected shift in context. TRAILS provides a chronological, replayable view of the session through the history log. A link map gives the user a view of the current information node and all immediately connected nodes. The third kind of question is usually handled through use of a title or captions to identify the information being displayed.

User interface. There are many different types of user interfaces for hypertext systems. Different applications of hypertext seem to spawn varied types of user interfaces. Many systems have exploited windowing to show multiple hypertext nodes and to show representations of node relationships. Graphical structures are frequently used to illustrate connectivity between nodes, and users can often select elements from the graphs.

The TRAILS prototype interface was intended to provide some basic support to authors and readers of software documentation while carrying out a study to determine what interfaces are appropriate. Lists are used to display nodes when there is no obvious relationship among a collection of nodes. Directed graphs are used to display information nodes and the links that connect them. Each arc in the graph represents a link and is labeled with information provided by the author. A browser is used to display the text associated with an information node, and various types of highlighting are used to identify different types of links that are present in the node.

There are many different types of user interfaces for hypertext systems.

Query. All of the discussion to this point has assumed that a starting point or an initial object exists. In many cases, there is no initial object, and some form of query is needed. Keyword searches are notoriously inaccurate and often fail to locate appropriate nodes.[10] A variety of retrieval techniques have been described in the literature.[11] The query mechanism should be flexible to help filter information in a number of ways. Examples of filters might include keywords, search for existing relationships by name, lexical or contextual relationships, object type (if using an underlying data model), and level of detail desired.

Integration of the tools

This section describes our approach to solving several problems in integrating the tools.

A brief look at a software technical manual will usually reveal quite a number of specific references to programs or parts of programs. Depending on the document being reviewed, references may be found to macros, control blocks (global data structures), symbols, entry points, subroutines, modules, or subsystems. A number of ques-

tions arise about relating documentation to program data:

- What documentation should be related?
- What program data should be linked?
- What techniques can be used to automatically generate links?
- How are appropriate links identified?
- Are navigation styles consistent across tools?

What documents to link. It is assumed that the target documentation will be formatted and available for on-line hypertext displays. The choice of documents eligible for linking to program data will depend on the user tasks and user set to be supported. Maintenance, support, and user organizations will have different tasks and will probably choose to link different sets of documentation to the program information.

Documentation generated from code deserves special attention. Many software development organizations have tools that generate one or more forms of documentation by analyzing source code. The control block descriptions and logic diagrams in some IBM technical documents are generated in this fashion. Generated documentation has two characteristics that make it desirable to link to program data. First, it has a high degree of correlation with the programs. Second, the fact that a software organization expended the effort to create a document generator is a good indication of a strong need for that specific documentation. In our scenario, the generated design document in TRAILS was used with the control flow diagram from CodeNavigator.

Program data to be linked. Two issues in linking a hypertext node to a knowledge base are the recall (the proportion of appropriate references found) and the precision (the proportion of references that are relevant) of the linkages.[10] Once again an examination of the intended uses of the data is necessary to develop a good linking process. It seems likely that externally visible data, such as module names and module entry points, will provide useful linkages. In our scenario, it will become evident that error codes may also be a good candidate for linking documents and program data. Our recall will be improved by adding additional links based on data that programmers typically use in problem analysis. Additionally, we will find that our precision can be improved by introducing levels of detail in the linkages.

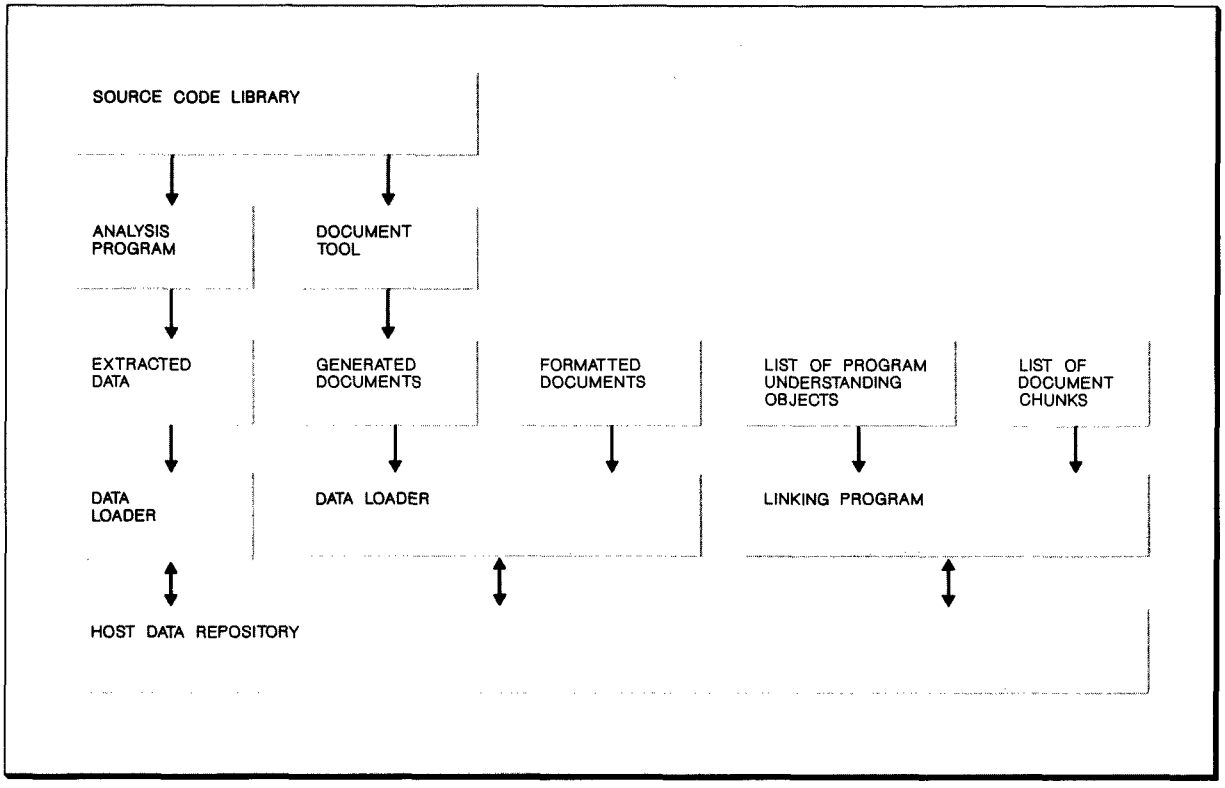
Our initial approach has been to link module and subroutine data to documents and add other data over time. As we gain information about the utility of the various linkages and gain experience in managing the links, we will increase the types and volumes of links.

Generated documentation has two characteristics that make it desirable to link to program data.

Linking methods. Links between the program understanding objects and the hypertext document objects are automatically generated after both databases are loaded. We will discuss several possible approaches. The first and simplest approach uses a character string scan. A list of objects is created from the program understanding database. The names of those objects are the targets of a character string search in the hypertext nodes. Similarly, a list of candidate hypertext nodes is created. Whenever the name of an object is found in a hypertext document, a link is automatically created between the program understanding object and the hypertext node. Although this approach is crude, it is also fairly effective for detailed documents, such as installation guides and logic manuals. Variations of the approach can be achieved by using keywords as search targets also. Keywords provide some opportunity for functional descriptions to influence the link creation. Our scenario might have been improved by using error codes issued by each program as keywords.

This approach has been implemented in the CodeNavigator/TRAILS prototype. An overview of the analysis is shown in Figure 6. Source code is analyzed by the CodeNavigator analysis program, and the results are stored in the database. A document tool is used to create the hierarchical-input-process-output (HIPO) style diagrams shown in Figure 7. These diagrams are imported into the database by the TRAILS data loader. Existing documentation is also loaded using the TRAILS data loader. Linkages between program understanding data and information nodes are

Figure 6 Overview of establishing links between CodeNavigator and TRAILS

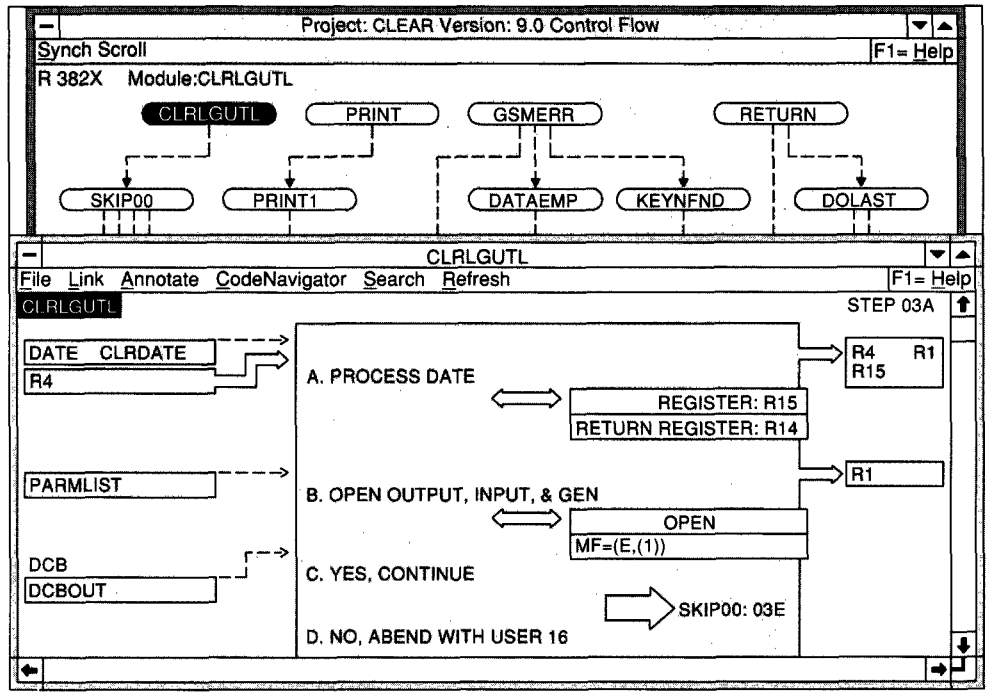


created by the linking program. Links are established when a program or subroutine name contained in the program understanding database is found in an information node created by TRAILS.

A second approach is to perform a lexical affinity analysis on the documentation and also on the prologues and block comments in the modules. A lexical affinity analysis identifies the most important word pairs (or triples) in a document. Words such as "is," "a," "the," "and," and "for" are ignored. Simply stated, this analysis identifies the frequency of occurrence of word groups across the target nodes, the number of nodes containing each word group, and the frequency of occurrence of word groups in a node. Word groups with a high frequency of occurrence are considered to have little information value, whereas seldom-occurring word groups are considered to have high information value. Linkages are established among nodes that have high occurrences of infrequently used word groups. The technique is heuristic and must be tailored to provide usable results. This approach is more appropriate for linking general information documents and high-level control modules or external interface modules. The approach also assumes a consistently rigorous commentary policy in the code.

A third approach involves use of a data model for hypertext nodes and attributes of objects in the data model. Garg introduces the concept of a "well attributed hypertext": "Attributes of objects are properties (of objects) which can be used to identify the objects from different perspectives."[12] The information content of an information node is represented by the attributes of the node. Levels of generality and possible contextual references may also be encoded as attributes. He further describes the use of operators to retrieve information nodes based on attributes and attribute values.[12]

Figure 7 Detail information about module CLRLGUTL



The need for this capability is illustrated in the scenario. Our linking technique made no distinction between overview and detailed documentation related to a module. Some context is provided in the names of the information nodes and in the names of the links between the program data and information nodes. Future work may use the level of detail of the CodeNavigator display to filter the TRAILS links. For example, if the user is looking at overview displays, such as subfunctions, overview types of information nodes would be presented to the user before detailed data. Similarly, from a low-level view, such as logic flow, the user should be presented with detailed information nodes.

How to identify appropriate links. A user moving from one application to another will have established not only a domain boundary but a working context. Both of these contexts must be passed to the invoked application. CodeNavigator and TRAILS collect information into projects and versions. These project-versions provide a boundary for investigations. The grouping of objects into project-versions is arbitrary and may be based on departmental organization, function relationships in the software, or team assignments. The decision of how to define project scope will include factors about the local definition of a project as well as careful consideration of the size of the potential search domains. TRAILS and CodeNavigator users can organize data into similar project-versions. The CLEAR project might contain both the program data and documentation data for the CLEAR product.

Additional context can be provided by using underlying data models. Bigelow and Riley describe a software engineering environment that uses a data model to support the integration of source code and hypertext documentation. Attributes on the nodes and links enable a user to retrieve and follow specific types of information linkages.[13] Garg has defined filtering that can be performed using attributes on nodes and links.[12] Garg and Scacchi have gone further by encoding information about the development process and user roles into their data model. Individual users define a profile indicating the information they want to see. These filters provide a form of dynamic

context by taking into consideration the attributes of nodes that a user has recently visited.[14]

Consistency of navigation across tools. The contrast of navigation styles between CodeNavigator and TRAILS mirrors the contrast in their analysis capabilities. Analysis of program source code is specific and rigorous. Natural language analysis is considerably less rigorous and is more heuristic.

Programming languages have closely defined grammars and finite numbers of operators or keywords. The number of possible relationships between programs is relatively small and well-defined. Navigation in CodeNavigator is accomplished by selecting an object of interest within a bounded domain and invoking a specific query (e.g., "How is this symbol used in the following subroutines?" or "What is the logic flow of this module?"). The granularity of the queries is offset by the variety of presentations (lists, text, directed graphs).

The TRAILS prototype, in contrast, has a very broad, nonspecific navigation style. This nonspecific style of navigation occurs because of the richness of natural language and the wide variety of types of documents. A user may be presented with many types of links across a fairly large range of nodes. The types of nodes to be returned must be filtered to some extent through the use of the context information. A graphic map is used to display nodes, links between nodes, and link names to assist the user in determining which links may be relevant.

Human interfaces. It has become evident that personal preference and task orientation drive the need for a variety of representations and navigation styles. We have found that people seem to have preferences for styles of graphic presentation. An example is the use of trees and directed graphs to show flow relationships. Many people have a strong preference for one or the other even though both contain the same information.

The networks that represent either program understanding data or relationships between hypertext documents may become far larger than can be comprehended easily. Because of the large data volumes, users need a capability to navigate at an abstract level as well as the ability to navigate through instance data. We call the high-level traversals "surface navigation" because the user is traversing the entities and relationships of several data models. We are developing a browser that allows a user to examine and traverse the entities and relationships of the various data models. Navigating at the data model level will enable a user to see the structural relationships between models. At any point a user can ask to see the instance data for an entity or to invoke a tool against the instance data.

"Submerged navigation" occurs when the user follows relationships between instances of the data. Traversing the instance data allows the user to understand and follow relationships in specific contexts. In this case, the user is operating beneath the surface of the data model.

Personal preference and task orientation drive the need for a variety of representations and navigation styles.

Multiple views of the database will enable the user to select the appropriate complexity, navigation style, and level of detail.

Scenario of linking hypertext and program understanding

This section of the paper is intended to provide an understanding of how CodeNavigator and TRAILS interact with each other. The scenario presented here illustrates a potential usage of integrated hypertext and program understanding tools. The scenario is a hypothetical example developed to demonstrate the capabilities of the tools. The code and documentation examples are taken from an internal library system.

The scenario has been built using information gathered when we interviewed programmers on how they learned to understand unfamiliar software. Although there was no consistent sequence of activities, several activities were consistent. For example, all of the programmers traced the

calling relationships among the relevant programs, all of them developed some sort of control flow analysis, most of them referred to existing documentation for additional information, and most of them created and maintained a private notebook which contained their personal discoveries and notes. Although this list of activities is not exhaustive, it will be the basis of our scenario.

Multiple views of the database will enable the user to select the appropriate complexity, navigation style, and level of detail.

CodeNavigator provides user displays that show calling relationships and program control flow. An annotation capability enables users to create and maintain private or public annotation files that are stored in the CodeNavigator database. TRAILS and CodeNavigator can be invoked from each other, which enables users to transfer from one tool and database to another tool and database in the same user session. In our scenario, a programmer's reference manual and a messages and codes manual will be automatically linked to program information. Additionally, a detailed design document has been included to illustrate the use of generated documentation and to highlight some of the issues described earlier.

Linking these kinds of documents involves several of the issues discussed previously in this paper. CodeNavigator is primarily an understanding and analysis tool. Therefore, the linkages to the documentation are expected to be oriented around understanding tasks and data. Similarly, our choice of tasks—that of learning about unfamiliar programs, suggests a type of document and level of granularity for linkages to program information. As a starting point, we made a decision to restrict linkages to occurrences of program names. Thus, all documents will be scanned for the character strings that represented programs in the target system. When matches are found, a link will be automatically created between the document and the program understanding database. The decision to restrict ourselves to program names is driven by a fear of creating an overwhelming number of linkages. For example, we could easily create a linkage for each symbol name in the program understanding database and its occurrence in the documents. However, such linkage would involve thousands of links for even a small number of programs and would provide little more information than can be obtained through a system cross-reference listing. Our approach is to begin with a minimal number of links and add additional ones as we discover those that have value.

Our scenario assumes that both CodeNavigator and TRAILS are being used. Hypertext linkages between program information and hypertext documents are automatically built when the databases are created. The links between documents and program information have been created simply by searching for the names of programs in the documents. All links are bi-directional so that they can be traversed in either direction.

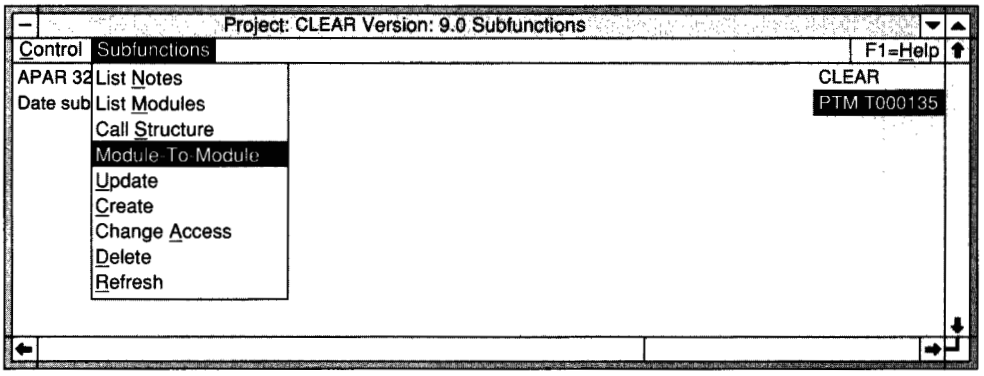
We will assume that the program understanding and hypertext document databases have been established and have been used by programmers for some time. In our scenario a programmer has been asked to review some code and documentation that is new to the programmer. Review of the code has three objectives for the programmer:

- Become educated about the reporting function in the bulk storage subsystem
- Identify changes that need to be made to the existing documents
- Create a programmer's notebook for the maintenance team

The programmer uses the tools to identify changes to existing documents and simultaneously create information to be included in a programmer's reference manual.

Step 1a—Identify a subset of the system to study. The first step is to identify the modules that are to be investigated. Since the task is to learn about a subsystem, a mechanism is needed to identify the modules of interest and filter out all other unwanted information. In this case, the programmer is specifically interested in the modules that perform reporting functions for the bulk storage sub-

Figure 8 Subfunction T000135



system. CodeNavigator provides a capability to group modules into logically related groups, called subfunctions. Subfunctions can be any arbitrary grouping of modules. Examples might be subsystems, modules supported by a programmer, or modules involved in a design change or a Program Trouble Memorandum (PTM). The groupings can be overlapping and may be changed as necessary. The intent is to provide a scope or boundary for investigation.

The investigation starts with PTM T000135, which involved changes to some of the BULKRPT reports. In Figure 8 subfunction PTM T000135 is selected from the list of subfunctions authorized to be seen. Subfunction PTM T000135 contains a list of the modules affected by a recent enhancement to the system. The pull-down menu lists the operations that can be performed on a subfunction. Among the several choices are:

- List notes associated with the subfunction
- List modules in the subfunction
- Display calling relationships between subroutines (detailed view)
- Display calling relationships between modules
- Change access authorization
- Update the contents of the subfunction

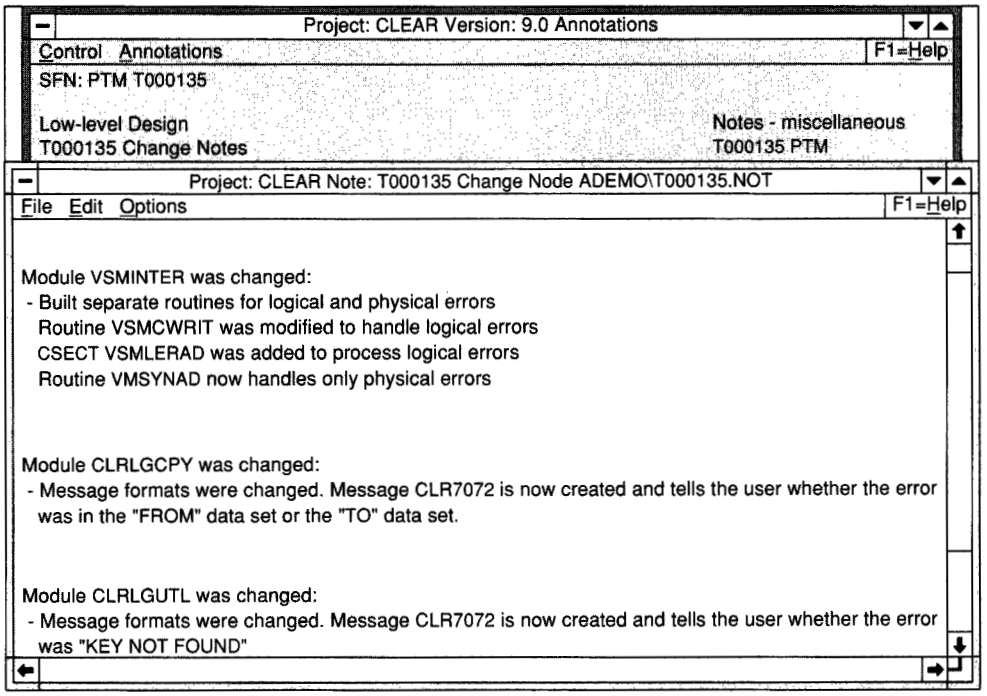
Step 1b—Review programmer annotations. The first source of information will be the notes and remarks made by programmers who worked on the code beforehand. One of the current objectives is to create a programmer's notebook containing information about the modules being studied. The purpose of this step is to review any notes other programmers felt were important enough to record in the database. These notes will be captured and included as part of the programmer's notebook. Additionally, the comments and remarks may provide information that will be useful in directing the investigation.

Two points of interest are in this step. The first is that the annotation feature provides a mechanism for people to add information to an automatically created database. History, ambiguities, hidden intentions, side effects, design considerations, test information, and inspection results are examples of the kind of information that may not be derived from code analysis and might be placed into annotation files. A second point is that the annotations are associated with objects in the database. This association provides the flexibility to make a general notation about a group of modules or a specific comment at the individual module level.

From the menu, List Notes is selected to display the programmer annotations associated with the subfunction. From the list of annotations, the T000135 changes are selected, and then the information recorded about the change is looked at. Figure 9 shows the annotation information. Notice that three modules were changed: VSMINTER, CLRLGCPY, and CLRLGUTL.

Each display has a pull-down menu that provides the ability to capture the contents of the display as either a workstation or host metafile. The cap-

Figure 9 Annotation on subfunction T000135



tured metafile can then be converted into a formatted file for inclusion in a programmer's notebook. The several displays shown in this scenario are captured, and they are converted into BookMaster™ (an IBM document markup language for text processing) files. These BookMaster files will be included in the programmer's notebook.

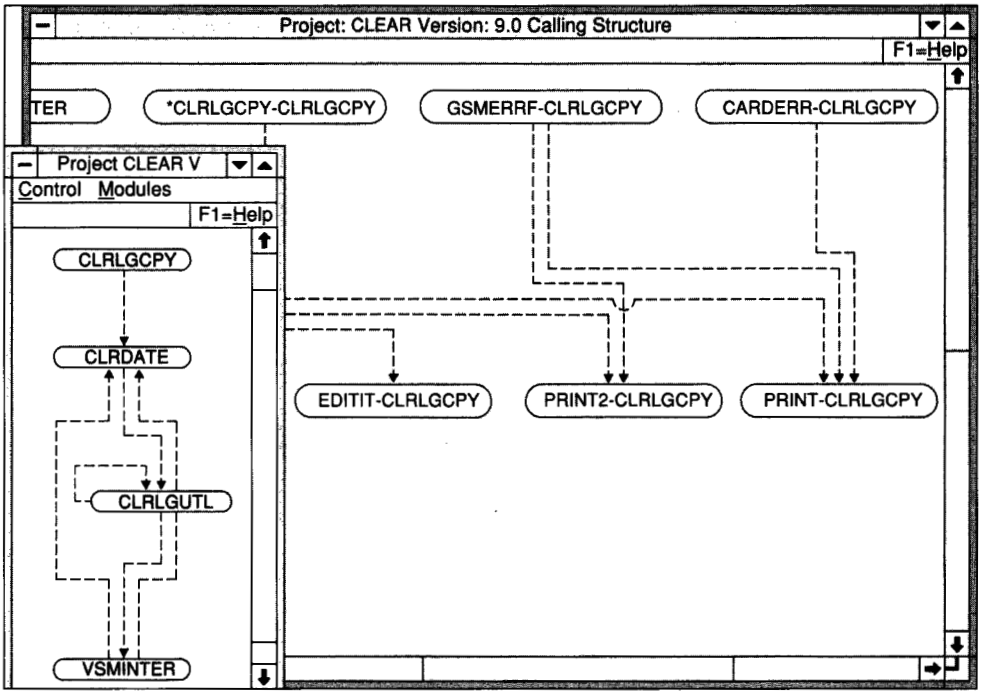
Step 2a—Investigate calling relationships. Most of the programmers we interviewed in IBM laboratories indicated that they begin learning about unfamiliar code by examining the calling relationships between modules. For each module of interest, they collect information about those modules or subroutines that call or are called by the module. This piece of information completes the context definition in the sense that the programmer has established a boundary for the investigation and also knows which other modules are immediately related to the modules of interest. This method is similar to the hypertext user's question of "What is the current node and what are the related nodes?"

After the annotations are reviewed it is desirable to gain an overall understanding of the structure of the subsystem. Our interest is in how the modules in the subfunction are related, so Module to Module is chosen to display a directed graph of the calling relationships among the modules. Calling Structure is also selected to show the interrelationships at the subroutine level. Figure 10 provides information about the module relationships and subroutine, or process, relationships. Notice that there is a choice of levels of detail—intermodule and intramodule level diagrams. The view chosen depends on the particular question being investigated and the size of the problem. In our case, since the objective is to acquire a general understanding of the system, both views are selected.

Information about modules and their relationships is extracted during program analysis and stored in the object manager. CodeNavigator users can invoke several displays that show different representations of those relationships. An investigation can be continued by selecting a module or subroutine from one of the displays and requesting additional information.

Step 2b—Review a list of documents about the module. At this point an understanding of the call-

Figure 10 Module-to-module calls and subroutine calling structure



ing structure of the subsystem has been achieved. The program analysis data are limited to information about the physical implementation of the system. Logical or conceptual information about the system is found in the system documentation, which provides information about the purpose of the subsystem and how it is related to other parts of the system.

Let us suppose there is a request to see documentation about CLRLGUTL, one of the modules in the BULKRPT subsystem. The pull-down menu in Figure 11 shows TRAILS being invoked from a CodeNavigator display with a request to show documentation that is linked to the highlighted module, CLRLGUTL.

Since several documents are linked, and there is no apparent relationship among them, a list of "nodes" or document fragments that contain related information is presented as in Figure 12. The CLRLGUTL Description tells how the program can be invoked from the command line in the library system panels. BULKRPT–Verb Reference Manual is an overview of how the BULKRPT verb fits into the bulk storage subsystem, and the CLRLGUTL HIPO Diagram contains detailed design information.

The list of documentation is a little confusing because it contains three different levels of detail: a general description of the program, a subsystem overview, and a detailed design description. Such confusion indicates that our linking scheme could be enhanced to differentiate among the levels of detail. As a result we need to find ways to enable authors to create linkages that represent different levels of detail and also to enable readers to understand and select the level of detail that is desired.

An approach that could be used is to separate the documentation into levels of detail and create linkages or relationships that represent those levels. Users would see either one or another set of linkages depending on the type of display that generates the request for documentation. In our case, we would categorize the subfunctions display as being at the logical level and the intramodule displays as physical-level objects. In the same way, the hypertext nodes will be catego-

Figure 11 Invoking TRAILS from CodeNavigator

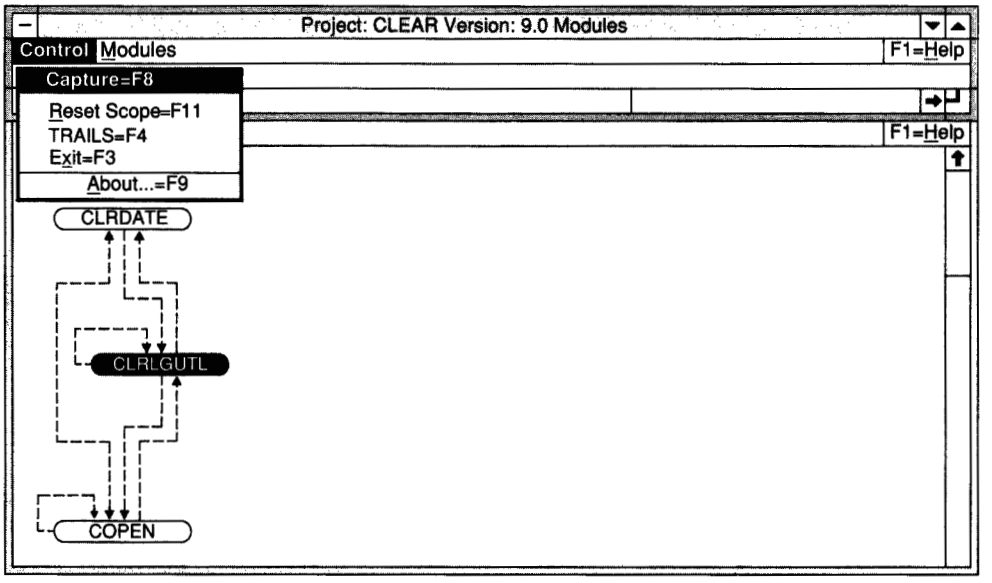
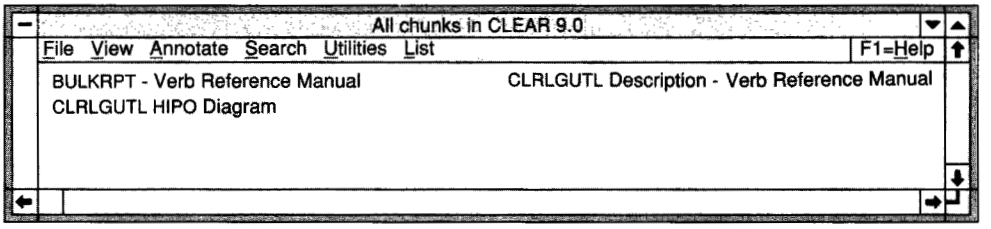


Figure 12 Documentation nodes linked to module CLRLGUTL

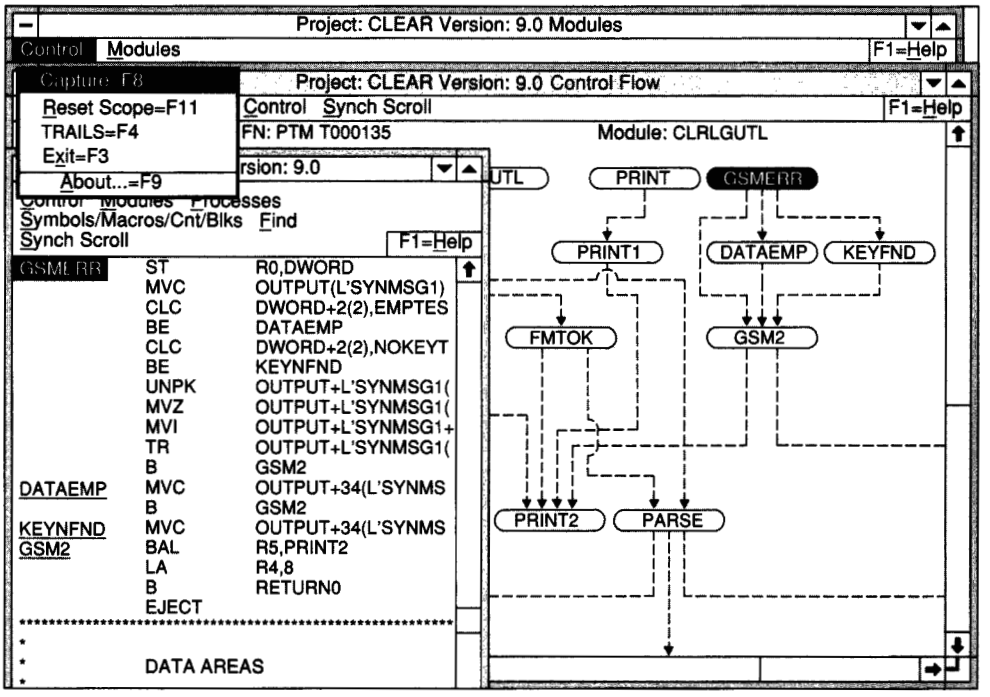


rized as either logical or physical levels of detail so that the BULKRPT node, which describes how CLRLGUTL fits into the system, would be a logical-level node and the others would be physical-level nodes. Linkages representing different levels of detail will be created in separate runs of the automatic linking process. The first run will contain only the logical-level document nodes. When program names are found, a relationship will be created between the document node and module that represents a logical level of detail. In the second run, only physical-level document nodes will be processed, and the automatically generated relationships will represent physical levels of detail.

Therefore, if the subfunctions menu is being displayed and a request is made to see the documents related to module CLRLGUTL, only the BULKRPT node, which describes how the BULKRPT subsystem is related to other subsystems, will be seen. In an intramodule display, a request to see related documentation will present the module invocation description and the detailed design information. This scheme allows use of the implicit granularity of one system to filter information being requested from a related database.

Step 3a—Examine logic flow within a module. Our interviews revealed that most programmers sketched the logic flow of the individual programs, although there was no consistency about when they did it. Some did it early in the investigation, and others left it until the end of their

Figure 13 Logic flow and source in CodeNavigator



learning process. This sketch gives them an overview of the module showing the number of entry points and subroutines and some information about the complexity of the module. Although the logic flow is an important piece of information, it is not generally useful by itself. Logic flow is an abstraction of a program and is mostly used to provide a detailed visual image of the program as the programmer reads the source code.

A strong need exists to maintain the same position in multiple displays as the programmer scrolls through the source and shifts his or her attention from the source code to the logic diagram and back to the source code. The source code display shows the source text in the order in which the programmer wrote the code. The logic diagram shows a graphic representation of the code in the sequence in which the instructions might be executed. There is no guarantee that sequential instructions in the source display will be represented by contiguous nodes in the logic diagram. In fact, branching instructions will frequently have targets that are not contained in the visible part of the logic diagram.

In CodeNavigator, the source code display and the graph are synchronized so that when an object in either display is selected, the corresponding object in the other display will be highlighted and scrolled into the viewing area. Notice that the label GSMERR is highlighted in both the source code and control flow display.

After an examination of the system structure, it is desirable to look at a summary of the logic flow of the modules. All of the nodes in the calling relationship graphs are selectable, meaning that a node with the cursor can be selected and an action invoked against the object represented by the node. The module CLRLGUTL is selected, and a source code display is requested (see Figure 13). With the source code display, a request is made for a control flow diagram, which shows the logic paths through the module. Source code is displayed in a syntax-sensitive browser. Tokens such as subroutine names, symbol names, macros, and control blocks can be selected in the source code, and an action initiated against them.

Step 3b—Display detailed design information. The CLRLGUTL HIPO Diagram is documentation that

shows a diagram of the internal logic of the module. It was generated by another tool and is known to TRAILS. Figure 7 shows an example of detailed design information contained in TRAILS along with the control flow diagram generated by CodeNavigator. The bottom screen shows a section of a HIPO style diagram that represents the CLRLGUTL node in the top diagram. The logic flow of the entire module is in the top window, and the detailed design of selected parts of the module is in the bottom window.

This example is another in which the two displays need to have a synchronous scrolling capability. Whenever two displays show different representations of the same object at the same level of detail, the user should be given an option to scroll them synchronously.

Step 4—Follow links between modules and documents. The links between documents and source code are bi-directional, so the links can be traversed from either tool. The CodeNavigator entry on the action bar enables a link from TRAILS to be followed back to CodeNavigator (Figure 7).

Notice that CLRLGUTL, the target of the hypertext link, is highlighted in the document display. Highlighting is used to indicate the existence of a hypertext link and enable other related information nodes to be viewed.

Several considerations are to be made in building connections between tools and related databases. One of them is maintaining a consistent context. This context includes the specific object of interest as well as the level of detail and synchronization of displays, as we have seen. Queries that are passed from one tool to another must include this type of information or allow the user to specify it if none is currently present. Another consideration is the similarity of user interfaces between the tools. For example, both of our tools have a point-and-click user interface, which enables users to select nodes in directed graphs, select items from lists, and select tokens from source information displayed in browsers. However, if one tool were to use a query-language interface and the other tool used a point-and-click style interface, the transition between tools might be disruptive to end users.

The granularity and organization of data should be similar across tools that are to be integrated. In our scenario, both databases contain networks of information about software systems. CodeNavigator and TRAILS are designed to operate on webs or networks of data. If one tool operated on hierarchically organized data and the other operated on relational data, the transition from one tool to the other could become awkward or confusing. If the organization of data is significantly different between tools, it may be desirable to provide a mechanism to help orientation in each database. An example of this might be to display a stylized image of the data model for each tool and an indicator to highlight one's current position. Thus, referring to the overview window could be used for reorientation in terms of what types of entities are related to the current object and what navigation paths are potentially available.

Earlier in the scenario it became apparent that there is a need for help in identifying and selecting appropriate levels of detail when shifting between tools. Overview documents were inappropriately linked to low-level program understanding displays. Our solution was to examine the intended use of the documents and displays and to develop links that are consistent with that usage.

Another area to be considered is that of scope. In CodeNavigator the limits of the subfunction provide constraints to prevent one from becoming lost in potentially enormous volumes of information. The TRAILS prototype has no similar boundary. It is possible to follow some hypertext links in TRAILS and then return to CodeNavigator to find that one has wandered beyond the limits of the subfunction. In the event this happens, a list of subfunctions pertinent to the TRAILS context is presented. Selecting one of the candidate subfunctions will establish the boundaries that CodeNavigator needs to limit the information it presents.

A familiar problem in hypertext discussions is the "lost in hyperspace" syndrome in which the users have lost track of what they are looking at or how they got there. The likelihood of this happening is increased as users traverse between databases while using several tools. TRAILS has a history log function which displays a sequential list of all windows invoked during the TRAILS session. Any part or all of the session can be replayed by selecting the windows to be redisplayed. This func-

tion helps to determine how the current window was arrived at. Also, steps can be retraced or a previous window state repeated with this function.

Using different tools and looking at data in several databases will present even more opportunities for becoming lost. A capability similar to the log function is needed to operate across tools so that it is possible to be reoriented or to return to an earlier window state to pursue an alternate line of investigation.

A familiar problem in hypertext discussions is the "lost in hyperspace" syndrome.

Finally, the navigation style should be compatible between the tools. TRAILS and CodeNavigator share several similar navigation techniques, so the transition from one tool to another is not an abrupt change. In our tool the user selects an object of interest from a window and then invokes action in a free-form fashion. Integrating tools that use differing navigation styles such as sequential processing or a refinement process in a design methodology may cause confusion because of the unexpected shifts in the way the investigation is continued. The integration of tools whose navigation styles differ will have to include some mechanism for providing an alert that a navigation change is occurring and some mechanism to assist in using the new style.

Step 4a—Find discrepancies between program data and documents. In this step, a discrepancy between the program data and the documentation is identified. Change information contained in the program, module prologues, and the annotation files is not reflected in the system documentation.

Recall that reviewing program change information was the initial step (Figure 9). The changes concerned error messages produced by the programs. Hypertext links are now followed to look at the error message documentation.

After a return to CodeNavigator, the same documentation review is performed for module VSMINTER. Among the linked documents is a section of the messages and code manual that describes message CLR7072. Following the link to TRAILS, the messages and codes node is browsed.

The module name VSMINTER is highlighted in messages and codes, as shown in Figure 14. The node describes error message CLR7072 and references modules VSMINTER and CLRGCPY, but not module CLRLGUTL. Referring back to the notes created for PTM T000135 (Figure 9), note that module CLRLGUTL should be referenced in the messages and codes document.

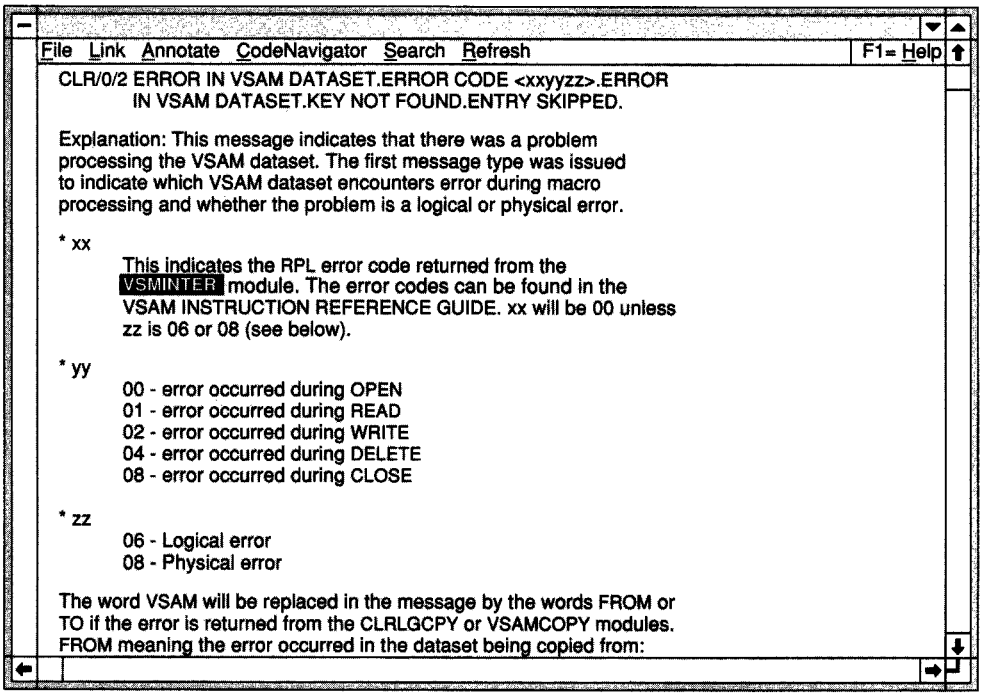
This example highlights two points for us. The first is that we have identified an error in the documentation through comparison of the source code and documentation, which was one of the original objectives. The example also emphasizes the need to understand the ways in which people may use the tool. The cross-reference of the messages and codes document with the program understanding data brings to mind the fact that this information is frequently used in problem analysis. The messages and codes manual is often the initial information source when trying to find the cause of a problem. Up until this point our task analysis and linking schemes have all been oriented to education activities.

A task analysis to support problem-solving might well suggest additional types of linkages. For example, based on this scenario, one might create a list containing module names and the message numbers produced in each module. An enhancement to the linking process would look for the message number in the documents in addition to the module name. Linkages would be automatically created between document nodes that reference a message number and the module that produces it. A filter would be needed to avoid creating redundant or duplicate links.

In the same way, other tasks such as design analysis, inspections, and testing can be considered to see if there are any key pieces of documentation that are used in the task. Those key pieces of documentation become candidates for linking to the program understanding data.

Step 4b—Create an annotation. From the display of the message and codes node depicted in Figure

Figure 14 Node messages and codes with hypertext links



14, an annotation file is opened and a note created indicating that the documentation must be updated. Since the note facility is available to both TRAILS and CodeNavigator users, it provides a means of sharing information between programmers and information developers.

This example is intended to illustrate the idea of integrating hypertext and program understanding tools. Our scenario deliberately focused on existing code and existing documentation, since most users are faced with that problem. Other examples of usage are for design and testing documentation. Linking specifications to program data will enhance our ability to ensure that designs are completely and correctly implemented. Test coverage analysis can be improved by relating test case documentation to program information. Documentation created throughout the entire development cycle can be linked with other pertinent documents as well as with the programs being developed or maintained. Inspections will be improved, since code and documentation can be linked and annotated.

Concluding remarks

Source code and documentation are the raw materials used to understand a software system. Because information is not captured and maintained as part of the system, analysis is repeated for each release of the system. We have described an integrated tool that provides program understanding and hypertext documentation capabilities for large software systems. These capabilities are applicable to many types of tasks in maintenance, so the tools need to be easily extendable and integrated with other tools and capabilities.

Data integration and functional integration are fundamental to providing tools integration capabilities. Data issues include the intended use of the data, granularity, underlying data models, and linking schemes to relate the different databases. A task analysis that identifies the intended tasks and the data used in the tasks will help ensure that appropriate links are created. Functional integration includes the mechanics of communicating between tools as well as maintaining contextual continuity for the user. Navigation mechanisms,

shared contextual information, orientation and "replay" capabilities, and automated filtering of information are all elements in the functional integration of tools.

Integration and distribution may well be among the key concerns of tools developers in the next several years. Distribution of function and data across the tool platform will provide effective use of resources and will separate individual tool components from common services or platform functions. Integration is becoming important because organizations cannot afford multiple tools development efforts to solve similar problems. The concept of a stand-alone tool is fading, just as stand-alone systems faded away in the 1970s.

Distributed data models seem to offer some opportunity for enhancing performance and exploiting the different advantages of host and workstation. An example in program understanding might be to place systems view data models on the host with relevant detail stored in bulk files. As data are downloaded to the workstation, the bulk files can be exploded to populate a workstation data model that supports intramodule views. This approach complements our staged analysis so that lower-level analysis can be done selectively, rather than for the whole system.

Portability of tools is also an emerging concern. Users are asking that the same tool function be provided in multiple environments. Individual tool components will be developed to be operating-system-independent, and the tools platforms will provide common functions to isolate the tools from operating systems.

Acknowledgments

The work described in this paper is being done in the Software Engineering Tools organization at IBM Sterling Forest. Each member of the development team contributed to the products. I would like to thank Frank Galdun for his vision of a day in the life of a programmer and George Rapalje for his unfailing support and positive attitude. Key technology transfers were provided by Linore Cleveland and Ashok Maholtra of the IBM Thomas J. Watson Research Center facility at Hawthorne, New York. Significant contributions to the architecture and requirements for analysis functions were developed in work sessions that included Sam Bailey, Linore Cleveland, Andrew Coleman, Joe Faga, Tom A. Gambino, Emmett G. Hayes, Roberta R. Hirth, Kurt T. Kresge, James L. Liu, Jeff Urs, and the author. Several people provided important leadership: Jim Caffrey and Yoshiro Akiyama for their early prototype and initiative, Joe Faga for refinement of the data model, Gary Laskoski and Enis Olgac for early analysis functions, Tim Montgomery for TRAILS development, and Tom Gargiulo for performance enhancements.

Cited references

1. M. T. Harandi and J. Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software* **7**, No. 1, 74 (January 1990).
2. T. A. Corbi, "Program Understanding Challenge for the 1990s," *IBM Systems Journal* **28**, No. 2, 294–306 (1989).
3. P. Brown, "Managing Software Development," *Datamation* **31**, No. 8, 133–136 (April 15, 1985).
4. V. R. Basili, "Viewing Maintenance as Reuse-Oriented Software Development," *IEEE Software* **7**, No. 1, 19–25 (January 1990).
5. L. Cleveland, "PUNS—A Program Understanding Tool," *IBM Systems Journal* **28**, No. 2, 324–344 (1989).
6. C. C. Marshall and P. M. Irish, "Guided Tours and On-Line Presentations: How Authors Make Existing Hypertext Intelligible for Readers," *Hypertext '89 Proceedings* (November 1989), pp. 15–26.
7. R. J. Glushko, "Design Issues for Multi-Document Hypertexts," *Hypertext '89 Proceedings* (November 1989), pp. 51–60.
8. *Hypertext '87 Proceedings*, Chapel Hill, NC (November 1987), pp. 143–188 (published by ACM).
9. H. Van Dyke Parunak, "Hypermedia Topologies and User Navigation," *Hypertext '89 Proceedings* (November 1989), pp. 43–50.
10. P. Hayes, "Towards an Integrated Maintenance Advisor," *Hypertext '87 Proceedings* (November 1987), pp. 119–128.
11. B. W. Croft and H. Turtle, "A Retrieval Model for Incorporating Hypertext Links," *Hypertext '89 Proceedings* (November 1989), pp. 213–224.
12. P. K. Garg, "Abstraction Mechanisms in Hypertext," *Hypertext '87 Proceedings* (November 1987), pp. 375–396.
13. J. Bigelow and V. Riley, "Manipulating Source Code in Dynamic Design," *Hypertext '87 Proceedings* (November 1987), pp. 397–408.
14. P. K. Garg and W. Scacchi, "On Designing Intelligent Hypertext Systems for Information Management in Software Engineering," *Hypertext '87 Proceedings* (November 1987), pp. 409–432.

Patrick Brown *IBM Enterprise Systems Division, Sterling Forest, P.O. Box 700, Suffern, New York 10901.* Mr. Brown works in the architecture department of the Software Engineering Tools organization. He joined Software Engineering Tools to manage a program understanding project in 1986. Prior to joining the organization, he was a programmer and manager of application development in the former Office Products Division. Mr. Brown received a B.S. in computer science from Syracuse University in 1976.

Reprint Order No. G321-5441.