TOWARD A SOFTWARE INFORMATION SYSTEM

David G. Belanger, Ronald J. Brachman, Yih-Farn Chen, Prekumar T. Devanbu, and Peter G. Selfridge

David G. Belanger, ***Ronald J. Brachman,*** ***Yih-Farn Chen,*** ***Prekumar T. Devanbu,*** and ***Peter G. Selfridge*** *are with AT&T Bell Laboratories, Murray Hill, New Jersey. Mr. Belanger is head of the Advanced Software Department. He is responsible for research into concepts and tools to increase productivity and quality in software development. He received a B.S. from Union College, an M.S. from Case Institute of Technology, and a Ph.D. from Case–Western Reserve University, all in mathematics. He joined AT&T in 1979. Mr. Brachman is head of the Artificial Intelligence Principles Research Department. He is responsible for development of the CLASSIC knowledge representation system, basic research into tractable reasoning systems, and applications of artificial intelligence technology to software development.* (continued on page 41)

A software information system collects information about large software systems in knowledge and data bases. It includes applications built on these bases to provide easy access to software information or to automate elements of the software development process. Work in this area is proceeding at AT&T Bell Laboratories in three related directions. The C Information Abstraction system automatically extracts information from C language programs and stores it in a relational database. It includes a growing collection of applications of this information. The MView system provides graphical views of interrelated software and lets a user quickly browse software systems. The LaSSIE system represents large amounts of software information in a formal language and provides intelligent assistance in retrieving software components. Progress in these three directions is reviewed in this paper.

Introduction

The growth in the size and complexity of telecommunication software is fast outpacing the ability of any individual or group of individuals to understand large systems. Many systems contain more than a half million lines of code and range up to large switching systems of 3 to 5 million lines of code each. As the software portion of our products increases, the competitiveness of AT&T products is increasingly determined by the software that runs them. For example, the 5ESS® electronic switching system, AT&T's largest switch (it can serve over 100,000 customer lines), is controlled by complex software responsible for providing plain old telephone service (POTS), a variety of additional features, such as call forwarding and data transmission for businesses, Integrated Services Digital Network (ISDN), and internal functions such as collecting and collating billing information and determining that the switch is functioning correctly.[1]

There are many problems that arise in the development of very large scale software systems. They share some common traits with large projects in other disciplines (e.g., very large construction projects). Among these are the needs for scheduling of people and materials, surmounting unpredictable hurdles, and assuring quality.

Another class of problems is unique to software—those involving extensive and frequent modification.[2] Because the functionality of large software systems is continually evolving, modification activities include the addition of complex new features as well as fixing bugs. The compound annual growth rate of some systems is as high as 40 percent over a life cycle longer than 10 years. Adding new features is especially important in competitive telecommunications software, and the interaction of features greatly complicates maintenance. Other reasons for the difficulty of maintaining software include:

1. Software systems are so large that no one person understands more than a fraction of them.
2. Knowledge about the system is distributed over several sources: documents (which are often out-of-date), human experts, and the code itself.
3. Software evolves over a long period of time, during which the structure that may be initially present may degrade.
4. Structure and conventions are extremely difficult to preserve, because the structure is invisible. The *invisibility* problem was well described by Brooks,[3] who pointed out that, unlike buildings, software has no inherent structure that can be easily represented in blueprints or other documents.

Software Information Systems. As awareness of these problems has grown during recent years, a variety of tools and systems have been built to address them. A *software information system* (SIS) is a computer system that represents information about a large collection of software and provides access to that information in a useful way. The design and appearance of a SIS depends heavily on its applications. For example, one can imagine a software *oracle*, representing vast amounts of information and capable of answering very complicated queries about software. This can be contrasted with a sophisticated software browser where the emphasis is on fast, supportive human interaction. In the latter case, a user may be happy to do the cognitive work, probably involving examination of the code itself, but the SIS provides rapid access to the source files and makes evident relationships between code entities that are not obvious, like function call relationships and data structure dependencies.

Regardless of their design or targeted uses, software information systems have four key aspects:

- *Acquisition.* The information represented in a SIS must be acquired, and should be acquired in as automated a fashion as possible. Acquisition tools are critical if the information is to be reliably up-to-date, and the information that can be acquired will, inherently, define the potential usefulness of the system.
- *Representation.* The information has to be represented, and the expressibility of the representation scheme will affect the power of the SIS.
- *Accessibility.* The information has to be accessed in as useful a way as possible. This does not imply accessibility by a human only; often, a SIS will serve as a back end to other tools.
- *Applications.* A SIS will have a collection of applications, some of which may be straightforward, such as support for browsing code, and some of which may be quite complex, for example automatic extraction of reusable subsets of a software system.

These issues of acquisition, representation, access, and application are interdependent and determine the utility of a SIS.

SIS at AT&T Bell Laboratories. This article describes three related research efforts within AT&T Bell Laboratories to develop software information systems. First, the C Information Abstraction System (CIA) for automatically extracting information from C code is presented, with a set of sample applications that use the data that CIA produces. This set of tools has already proven to be

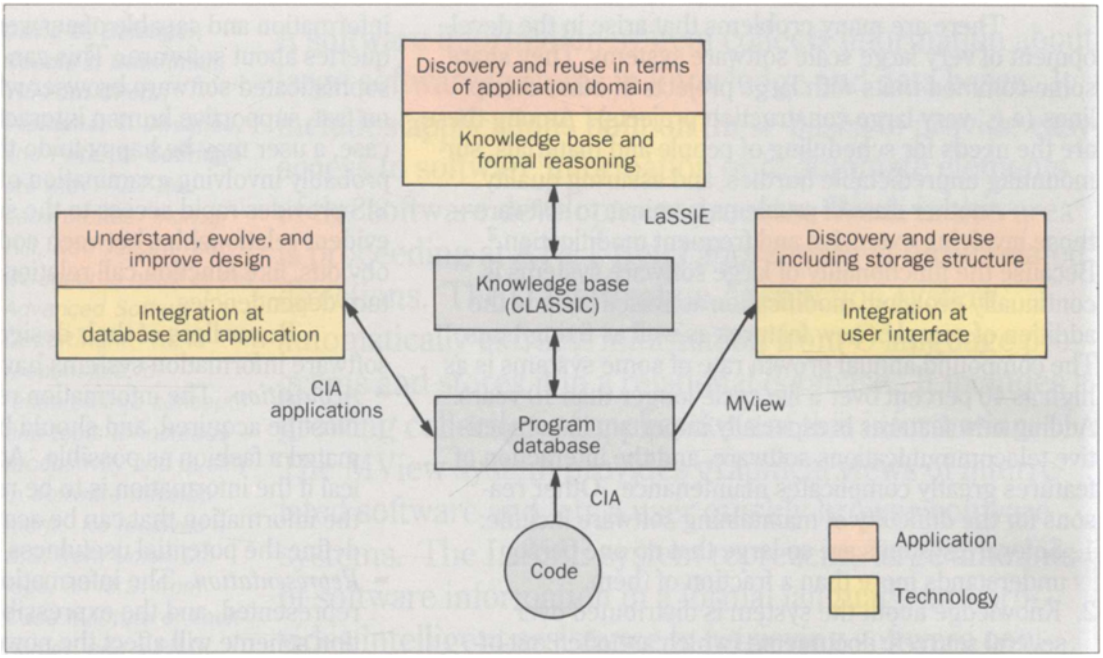


Figure 1. Software information system directions. A program database is the foundation of research efforts. Two directions—LaSSIE and MView— are aimed at improving the process of "*discovery*" *for* software maintenance and reuse. A third direction—CIA—is aimed at understanding, evolving, and improving software designs.

a powerful aid in the analysis of existing C programs. (A version of this capability for C++ has recently been completed.[4]) Next, the LaSSIE system is discussed. Based on formal knowledge representation and inference technology and encoding several "views" of software for the Definity® 75/85 telecommunication systems, LaSSIE allows a user to formulate and answer complex queries about the software. Then we discuss the MView system for providing graphical access. This system features high-bandwidth graphical views of interrelated software objects, and combines fast graphical browsing with display of different levels of related information about a single system. We conclude with a brief discussion of future work in software information systems.

Figure 1 indicates the directions each of the above projects is taking and the primary motivations for those directions. All the projects are driven, at the elementary level, from information about the source code of a software system, i.e., a *program database*. The formats of the program databases vary by project, as described in later sections.

Two of the directions taken are aimed at improving the process of "discovery" for software maintenance and reuse. (Discovery is the ubiquitous process of finding out enough about a software system that one can proceed with confidence to change it.) LaSSIE is concerned with adding concepts and domain (e.g., PBX) knowledge to the process so that users can interact with the system in a language related to their application. It accomplishes this by the use of formal knowledge representation and inference technology. MView, on the other hand, seeks to broaden the scope of objects that can be investigated (e.g., including file directories) and to integrate this information by means of a fast, expressive, user interface. Finally, CIA applications are oriented to understanding, evolving, and improving

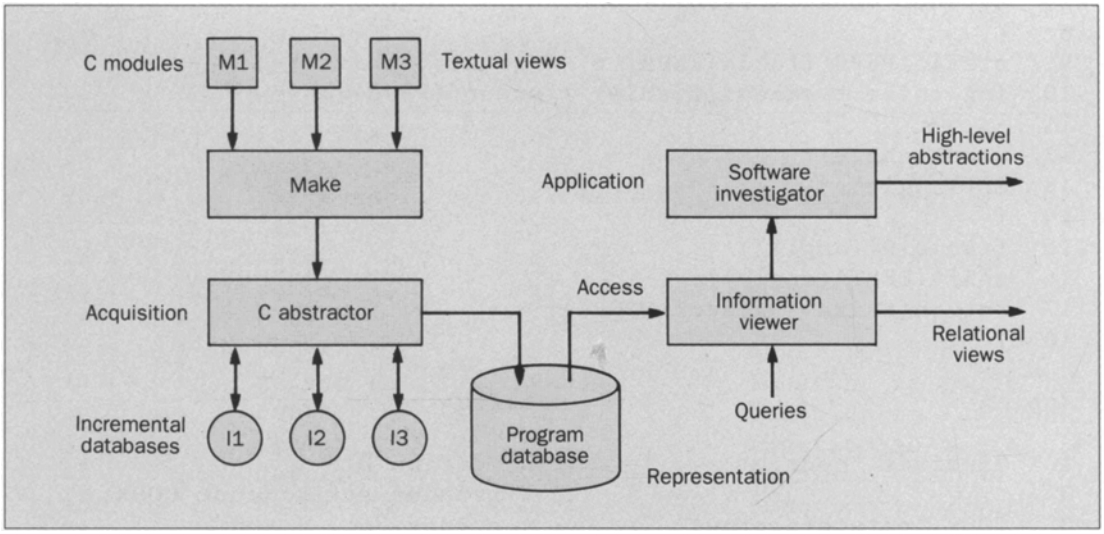


Figure 2. The C Information Abstraction System. The C abstractor converts textual representation of C programs into a relational database. Developers use the information viewer to query the database. The software investigator is a collection of tools that presents graphical views and high-level information about program structures.

software designs. Some applications automate previously manual tasks, while others provide support for the software designer. Integration in CIA is done by broadening the information in the program database (e.g., by annotation), by integration with related software databases, and by the individual CIA application programs.

Automatic Acquisition

Ideally, a software information system should have its information collected through an automatic process. Because program text is usually the only reliable information source that can be processed automatically, extracting program information from text is a *sine qua non* of software information systems. This section describes the C Information Abstraction System.[5,6] a collection of tools that extract and process relational information from C programs and make the information available to applications.

The Database Approach. The CIA system contains three major components: the C abstractor, the information viewer (InfoView), and the software investigator, as shown in Figure 2. The C abstractor is an *acquisition* tool. It converts the text of a set of C source files (modules) into a set of entities and relationships according to a simple model of the key relationships in code. A relational description of each source file is recorded in its own incremental database. The incremental databases are then linked together to construct the complete program database. *Make*[7] checks the timestamps of each file and invokes the C abstractor on a file only if that file has been modified since the creation of the last program database.

We chose a relational database *representation* for the program information so that it can be *accessed* by a variety of database management systems and tools, including Awk,[8] INGRES,[9] and our InfoView system. A collection of tools has been built to process the relational views to provide graphical views and some high-level information about program structures. These tools are collectively called the software investigator and are considered the primitive *application* tools that can be used to construct other applications.

This database approach has several advantages:

- The partition between information abstraction and

Panel 1. A Textual View of `Trans`

```
<trans.c>
1   #include "coor.h"
2   #define FNUM 2
3   #define DELTA(x) (x + 1/x)
4
5   extern COOR *rotate();
6   extern COOR *shift();
7   typedef COOR *(*PFPC)();
8
9   static PFPC ftable[FNUM] =  { rotate, shift } ;
10  int tsize = sizeof(ftable) / sizeof(PFPC);
11
12  trans(angle)
13  int angle;
14  {
15  ftable[0](angle);
16  shift(DELTA(angle));
17  return(tsize * sizeof(COOR));
18  }
```

```
<op.c>
1   #include "coor.h"
2
3   COOR *rotate(degree)
4   int degree;
5   {  /* ... */  }
6
7   COOR *shift(distance)
8   int distance;
9   {  /* ... */ }
```

```
<coor.h>
1   #define DIM 2
2   typedef struct coor COOR;
3   struct coor {
4   int point[DIM];
5   COOR *next;
6   };
```


presentation allows applications to skip the time-consuming parsing process and concentrate on analyzing and presenting the information in various forms.

- A program database can be processed easily by standard query languages such as SQL on most relational database systems.
- Linking a program database with the databases for other languages or software documents is a natural extension.

We are currently in the process of building information abstractors for C++, Makefiles, and Korn Shell scripts[10] using the same approach.

CIA's Model of Code. The most critical design decision to be made about the program database is its ontology, or schema, which defines the software entities, attributes, and relationships to be stored in the database. It serves as a specification for the C abstractor, and determines the knowledge that all CIA-related tools can provide to users. A carefully designed schema will reduce the storage requirement while satisfying the needs of most applications.

The C abstractor records information about five kinds of global entities in C programs: files, macros, variables, types, and functions. An entity is global if its identifier can be referenced across boundaries of entity definitions in C programs. We refer to these five types of entities as *CIA entities*. Local entities are ignored because they tend to increase the size of a database substantially and they affect only a small context. Moreover, their information can easily be generated on the fly if necessary.

Panel 2. A Relational View of `Trans`

Entities

file:	`trans.c, coor.h, op.c`
macro:	`DIM, FNUM, DELTA`
type:	`struct coor, COOR, PFPC`
variable:	`ftable, tsize`
function:	`rotate, shift, trans`

Attributes of `ftable`

file:	`trans.c`
data type:	`PFPC`
name:	`ftable`
storage class:	`static`
beginning line:	`9`
ending line:	`9`

Relationships

entity 1	relationship	entity 2	comment
trans.c	includes	coor.h	file to file
PFPC	refers to	COOR	type to type
ftable	refers to	PFPC	gbvar to type
ftable	refers to	FNUM	gbvar to macro
ftable	refers to	rotate	gbvar to function
ftable	refers to	shift	gbvar to function
tsize	refers to	ftable	gbvar to gbvar
tsize	refers to	PFPC	gbvar to type
trans	refers to	ftable	function to gbvar
trans	refers to	shift	function to function
trans	refers to	DELTA	function to macro
trans	refers to	COOR	function to type
trans	refers to	tsize	function to gbvar
COOR	refers to	struct coor	type to type
struct coor	refers to	COOR	type to type
struct coor	refers to	DIM	type to macro
op.c	includes	coor.h	file to file
rotate	refers to	COOR	function to type
shift	refers to	COOR	function to type

To illustrate how the C abstractor converts the textual representation of C programs into a relational database, Panel 1 shows the text of a simple C program `trans`. After the analysis, the C abstractor discovers 14 CIA entities as shown in the first part of Panel 2. The attributes of these entities and the relationships among them are stored in the program database. For example, the declaration of the global variable `ftable` shown in Panel 1 resides in `trans.c`, has the data type `PFPC`, starts at line 9, and ends at line 9. Therefore, the C abstractor creates the attribute list of `ftable` shown in the second part of Panel 2. Each of the 14 entities has a similar attribute list. Reference relationships extracted between these entities are shown in the third part of Panel 2, under "Relationships."

The C program database can be processed by any relational database system. We also provide a simple information retrieval system called InfoView that allows users to perform simple queries on the database. The main InfoView commands are

- `def`: display the attributes of entity definitions or declarations
- `viewdef`: display the text of entity definitions or declarations
- `ref`: display the relationships between entities
- `viewref`: display the text of entity references

These commands are based on the concepts of entities, attributes, and relationships described above. A user needs to understand only this simple model to use the InfoView commands. A knowledge of the underlying relational database schema or a complex query language is not required.

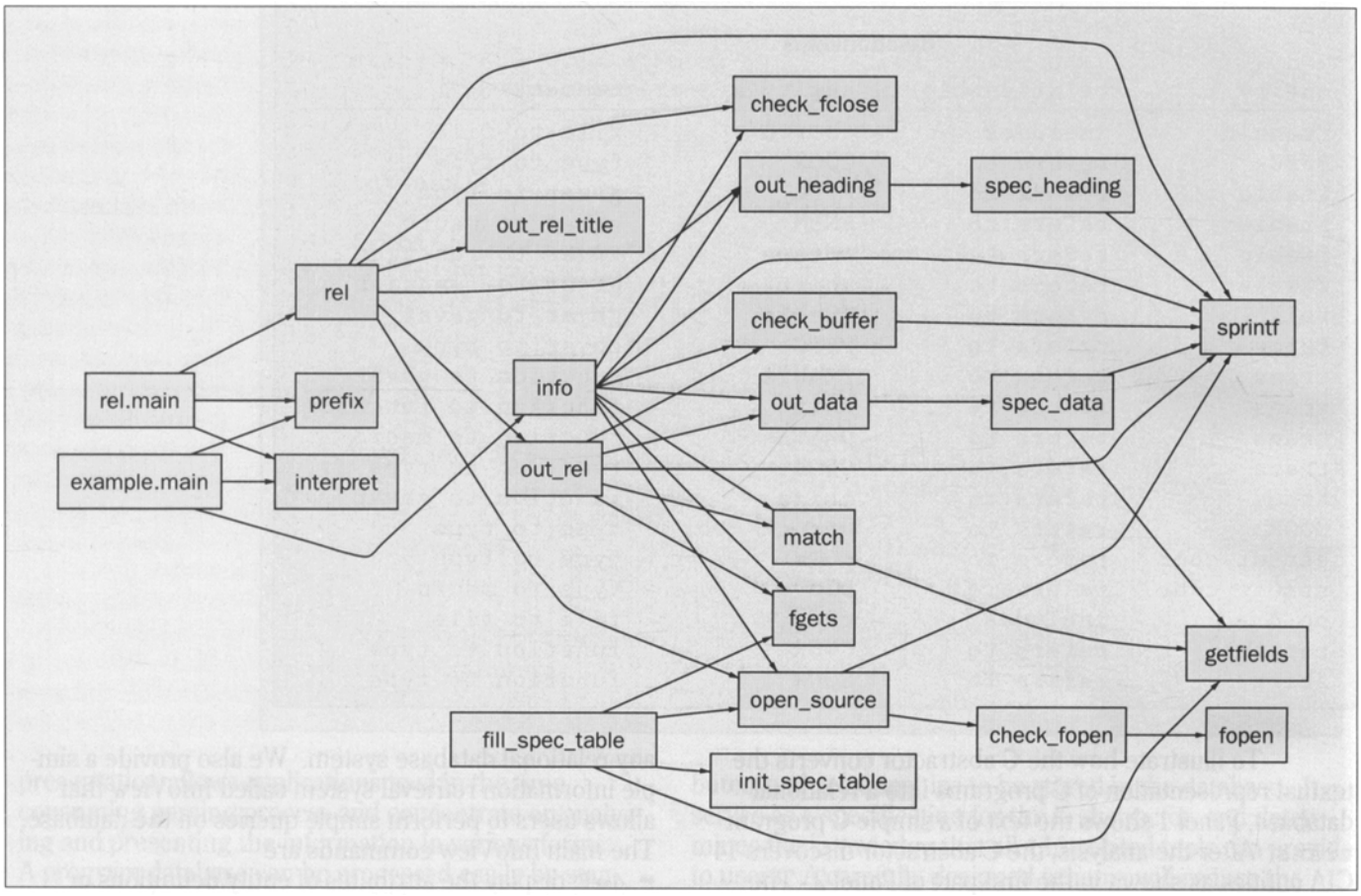


Figure 3. A function call graph. The C Information Abstraction System automatically generates this picture of the function layers and fan-ins and fan-outs of a small program. No human intervention is needed.

Applications of CIA Program Databases. The main reason to build program databases is to support applications that need to access program structure information. This section demonstrates the power and extensibility of a program database by showing how it can been used to draw graphical views of program structures, extract self-contained subsystems, eliminate unused program entities (dead code), restructure software, and compare program structures. These tasks used to be considered difficult and time-consuming, but they are now being carried out easily through automatic tools that take advantage of the program database information that CIA provides.

Graphical views. Information stored in the program database is often used to generate pictures that show interesting aspects of the internal program structure.

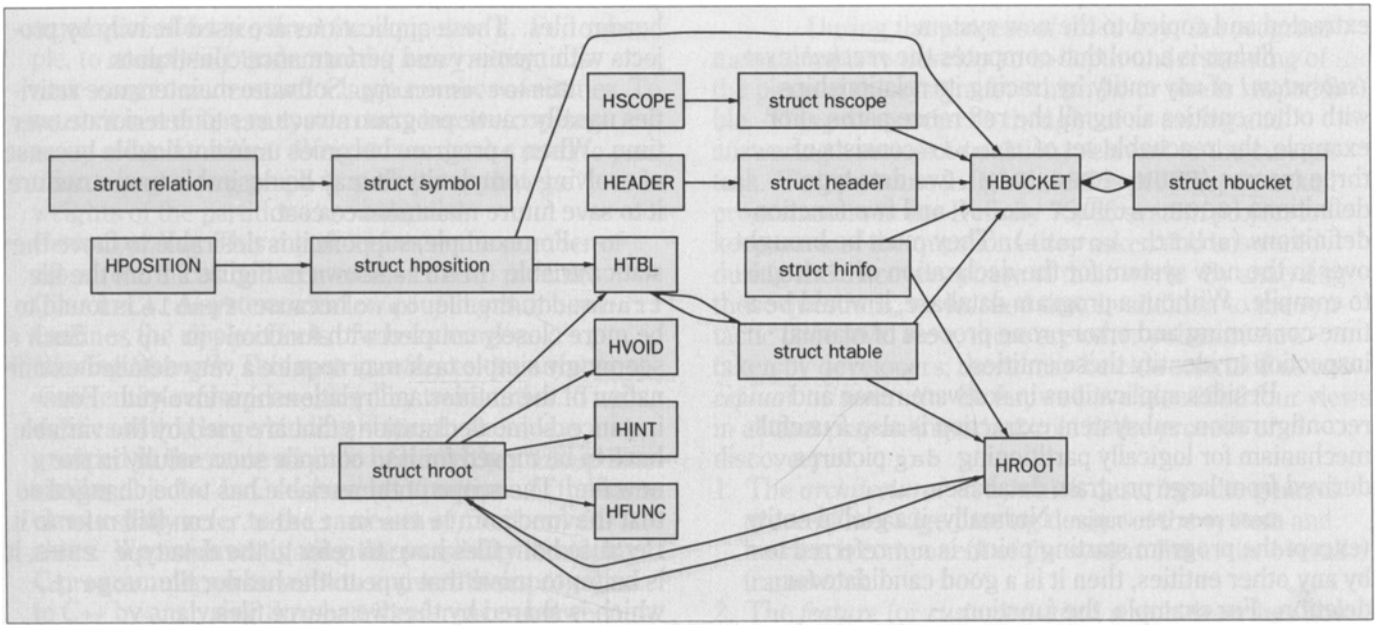


Figure 4. A data structure map. This is another type of picture that the C Information Abstraction System generates automatically. It shows data structures and their relationships in a library. CIA handles larger pictures by partitioning systems into pages.

Tools have been developed to convert relationships in the C program database to graph descriptions that can be laid out automatically by `dag`, a tool for drawing directed graphs.[11]

We illustrate two types of automatically generated pictures. Figure 3 shows the function call graph of a small program. The relational information was retrieved from the database, and the picture layout was done automatically by `dag` without any human intervention. It shows clearly the function layering structure and the fan-ins and fan-outs of each function.

Figure 4 shows a set of data structures and their relationships in a library. Cycles in the picture usually represent recursive references occurring in data types such as linked lists.

For large programs, the directed graphs generated by `dag` do not fit on a single sheet of paper. Fortunately, `dag` provides a pagination feature for partitioning the directed graphs. Graph filters that hide unnecessary details in various ways have also been used. The next section shows how to retrieve a logical subset of a software system for display.

Subsystem extraction. Often it is useful to extract and reuse one or more self-contained subsystems from an existing system. A subsystem can be analyzed and reused in a context independent of other components in the original software system. For example, suppose the static variable declaration `ftable` in Panel 1 is to be extracted and reused in a new system. Note that the variable declaration alone will not compile in the new system because there are missing references. To solve the problem, all entities referred to directly or indirectly by `ftable` need to be

extracted and copied to the new system.

Subsys is a tool that computes the *reachable set (subsystem)* of any entity by tracing its relationships with other entities along all the reference paths. For example, the reachable set of `ftable` consists of three macros (`FNUM`, `PFPC`, `DIM`), two data-type definitions (`COOR`, `struct coor`), and two function definitions (`shift`, `rotate`). They must be brought over to the new system for the declaration of `ftable` to compile. Without a program database, it would be a time-consuming and error-prone process of manual inspection to identify these entities.

Besides applications in software reuse and reconfiguration, subsystem extraction is also a useful mechanism for logically partitioning `dag` pictures derived from large program databases.

Dead code elimination. Normally, if a global entity (except the program starting point) is not referred to by any other entities, then it is a good candidate for deletion. For example, the function `fill_spec_table` shown in Figure 3 has a fan-in of zero and should probably be deleted. (To be precise, all kinds of reference relationships defined in the CIA conceptual model should be considered in computing the delete sets.)

Before deleting any entity, we need to determine its *delete set*, which specifies all the dead code associated with it. The delete set of an entity a, D_a, consists of all the entities in its reachable set, R_a, that do not belong to the reachable set of any entity outside R_a. This ensures that all unused entities are deleted in a single process and that the program behavior is not affected after dead code elimination. For example, There are six functions in the reachable set of `fill_spec_table`, but four of them (`open_source`, `check_fopen`, `fopen`, and `getfields`) are in the reachable sets of other functions and therefore cannot be deleted.

Besides dead code elimination, the program database has also been used to detect unnecessary header files. These applications are used heavily by projects with memory and performance constraints.

Software restructuring. Software maintenance activities usually cause program structures to deteriorate over time. When a program becomes unmaintainable because of evolving complexity, it may be desirable to restructure it to save future maintenance cost.

For example, suppose it is desirable to move the static variable `ftable` shown in Figure 2 from the file `trans.c` to the file `op.c` because `ftable` is found to be more closely coupled with functions in `op.c`. Such a seemingly simple task may require a very detailed examination of the entities and relationships involved. For instance, some declarations that are used by the variable have to be moved for it to compile successfully in the new file. The scope of the variable has to be changed so that the function `trans` in `trans.c` can still refer to it. Because both files have to refer to the data type `PFPC`, it is better to move that type to the header file `coor.h`, which is shared by the two source files.

A tool can use the program database to do this checking and then take proper actions automatically or provide suggestions to programmers. For a human programmer to do all this checking in a large programming project is almost inconceivable. For this reason, most project managers have not liked the idea of restructuring dormant code even if the code contains obvious design deficiencies. One tool to do part of this now exists in prototype form. Considerable work still remains to be done on robust metrics-guided restructuring.

Metrics. One of the uses of a program database is to support the analysis of software systems. This can take the form of structural analysis, as above, or of metrics to describe properties of the system. Besides obvious statistical metrics, we describe a few interesting metrics here to show the potential of the database:

- *Weight*: This metric refers to the number of entities that an entity depends on, i.e., members of its subsystem as defined in "Applications of CIA Program Databases." The *weight* of an entity usually indicates the

amount of effort required to manipulate it. For example, to completely understand a *heavy* entity, it is necessary to understand a large number of entities. To reuse it in a different system may involve copying many related entities. Weight can also be used to partition a large `dag` graph in such a way that the weights of the partitions are balanced.

- *Cross Coupling*: This metric refers to the number of cross references between entities in two modules (files in C). A system with low coupling numbers confines the ripple effects of most changes.
- *Binding Strength*: This metric refers to the number of same entity references shared by two entities. Two entities with a large binding strength should be grouped in the same module because they manipulate similar objects. In C++, member functions of the same class usually refer to the same set of member variables. We are investigating the possibility of helping C programmers who want to convert their programs to C++ by analyzing the shared references between pairs of C functions.

Much of the work here is aimed at understanding the value of individual metrics and the information they convey. The above metrics have the potential of being used in analyzing the evolution of the design of a software system.

The LaSSIE System

As we have seen, the CIA system constructs a program database that reflects the syntax of the code of a large software system. Thus information that has a syntactic manifestation in the source code, such as what functions a given function calls, can be acquired automatically. This *code level* knowledge plays a vital role during the discovery process. (With large software systems of the sort we are discussing, discovery—finding out enough about a system— can take more than 50 percent of the average developer's time.) As we have mentioned, discovery is one of the key tasks to be supported by software information systems.

During the process of discovery, the programmer attempts to develop an in-depth understanding of the piece of the system for which he or she is responsible. This process can be thought of as asking and answering a series of questions relevant to the current task. To gain some insight into this process, we asked programmers in the Definity 75/85 system project to keep track of the questions they asked and answered during the discovery phase of their work. By analyzing these questions, we learned that, in addition to the syntactic code level view, there are other points of view taken by developers; these may be characterized as *conceptual* or *semantic*. So far, we have identified four views in all that play an important role in the process of discovery:

1. The *architectural view*, which describes the general process/message-passing design of the system and how each component of the system fits in the overall framework.
2. The *feature* (or *customer*) *view*, which describes how the system looks to its user and what customer feature each component supports and how it supports it.
3. The *code view*, which characterizes the structures of directories, files, functions, data, etc., in the system.
4. The *domain model view*, which captures the basic key operations of the domain (e.g., switching), such as allocating a trunk or terminating a call, and how these operations are implemented in the system.

In a typical discovery scenario, questions will be asked that combine more than one of these views, as for example, "What messages are sent by a process in the network layer when an attendant pushes a button to activate the 'hold' feature?" and "What C functions cause an international toll trunk to be allocated?"

It is our long-term research goal to capture these different views, integrate them, and make them available to programmers for general-purpose question answering. The LaSSIE (large software system information environment) system is our first attempt to do this.[12] In LaSSIE, we use a formal knowledge representation (KR) lan-

guage to describe potentially reusable parts of a large software system, and we answer queries from programmers using a formal inference algorithm that runs over these KR structures. A programmer describes an operation that he or she is interested in reusing or just learning more about, and the inference algorithm compares the query and the stored descriptions in a general way, ultimately retrieving those operations that match the query. The user can make the request in several different ways, and can iteratively refine the query on the basis of the results of previous retrievals.

While code-level knowledge can be conveniently stored in traditional database formats, knowledge about the purpose of the code, and about the structure of the domain itself, does not fit well in standard data models. LaSSIE uses a sophisticated representation language based on a concept of structured object-oriented descriptions called "frames" (see Appendix A) to capture its knowledge of software systems. The formal nature of this type of language allows us to define a complete and correct inference procedure called "classification," by which LaSSIE can match large classes of descriptions without needing to compare each individual in the class with a query. The frame language also lets us represent the same objects from multiple points of view and integrate these views in both the knowledge base and in queries. The user queries the LaSSIE knowledge base using a powerful user interface, which incorporates a graphical browser and a natural language interface (TELI).[13] (This interface is currently not related to MView, but we expect that in the future we will incorporate insights from that system's ability to provide multiple integrated graphical views.) To elucidate further how LaSSIE works, we take a brief look at its knowledge engineering approach and then illustrate how the system is used with an example.

The LaSSIE Knowledge Base. The LaSSIE prototype has an extensive representation of the call processing subsystem of the Definity 75/85 system, a medium-sized PBX. The LaSSIE knowledge base describes the *actions* and *objects* in the switching domain and how the actions are implemented in the Definity 75/85 system architecture. Each action class (e.g., `CALL-CONTROL-ACTION`) and each object class (e.g., `CALL-STATE`) is described by a frame.

A typical LaSSIE frame description might look like this:

```
1 (define concept USER-CONNECT-ACTION
2  defined
3  NETWORK-ACTION CALL-CONTROL-ACTION
4  (fills has-actor Bus-Controller-Process)
5  (fills has-layer Call-Operation-Layer)
6  (exists has-operand USER)
7  (exists has-environment CALL-STATE)
8  (fills has-result Talking-State))
```

In this description, words in italics represent reserved words in the formal representation language. These perform logical functions like stating that a certain individual plays a certain role with respect to a general class of items [e.g., (*fills* `has-actor Bus-Controller-Process`) says that the bus controller process is the actor for every `USER-CONNECT-ACTION`]. Among the other items, those in all capitals are *frames*, which represent general concepts of things like users and call states. Items with initial capitals are *individuals*, which stand directly for individual objects in the domain, like the bus controller process. Items in all lowercase are *roles*, which represent relationships between individuals. Thus, the interpretation of this frame is as follows: `USER-CONNECT-ACTION` [line 1] is by definition [2] a network action [3] and a call control action [3], which is done by the bus controller process [4] in the call operation layer of the architecture [5], on a user [6], which takes the user from some call state [7] to the talking state [8].

It should be noted that in this language there is an alternative to *defined*. Frames specified as *defined* are given complete necessary and sufficient conditions.

Thus, in this case, if something is found to be a USER-CONNECT-ACTION then it is assumed to be a NETWORK-ACTION, its has-actor role is assumed to be filled by the Bus-Controller-Process, etc. Additionally, if something is known to be a NETWORK-ACTION with all the other properties specified above, then it is provably a USER-CONNECT-ACTION. The alternative, *primitive* frames, have only necessary but not sufficient conditions.

In all, LaSSIE's knowledge base contains 102 action concept descriptions of this type, which are classified into a hierarchy. The hierarchy ranges from very general action concepts, like CONNECT-ACTION, down to very specific ones corresponding to particular functions or messages. An example of the latter is the individual, Attendant-Add-User-Action:

```
1  (define individual Attendant-Add-User-Action
2   ACTION
3   (has-actor Bus-Controller-Process)
4   (exists has-operand USER)
5   (exists has-recipient CALL)
6   (exists has-environment CALL-STATE)
7   (has-result Talking-State)
8   (has-cause Attendant-Add-Button-Push)
9   (implemented-by /Usr/Pgs/Gp/Tgpall/Profum.C)
10  (calls-function Signal-Add-Error)
11  (accesses-variable *Call-Record*))
```

This structure means the following: Attendant-Add-User-Action [line 1] is an action [2] that is performed by the bus controller process [3], which adds a user [4] to a call [5]; it takes its operand (the user) from some call state [6] to the talking state [7]; it is caused by an attendant pushing an "add" button [8]; it is implemented by the source file /Usr/Pgs/Gp/Tgpall/Profum.c [9], calls the function Signal-Add-Error [10], and uses the global variable *Call-Record* [11]. By using the classification procedure one can determine, purely from their descriptions, that Attendant-Add-User-Action is an instance of USER-CONNECT-ACTION.

Using LaSSIE. To use LaSSIE, a user describes an operation of interest in a formal query structure resembling a frame. On the basis of the meaning of that description and that of the stored frames, the classification process finds all objects that match the query description. In addition, if there are too many matching objects (or there aren't any), the LaSSIE interface lets the user modify the query, in whole or part. The user can "cut and paste" pieces of descriptions of retrieved individuals to modify the query. The user can also explore the knowledge base using a graphical browser to select concepts to modify the query.

Let us illustrate with a brief example how LaSSIE helps the user formulate and reformulate queries, to narrow in on the answers he or she desires. Suppose that a developer wants to find an operation performed by the console controller process, where a telephone user is connected to a call when a console user flashes the hook. This query can simply be typed in English, and the TELI interface would translate it to appear as below. The developer would confirm this translation by requesting a retrieval:

```
CONNECT-ACTION
    has-actor  Console-Controller
    has-operand   USER
    has-cause  FLASH-HOOK-ACTION
       has-actor  CONSOLE-USER
```

The developer's request represents the query "What are the connect actions performed by the console controller process that connect a user when a console user flashes a hook?" The lines above illustrate how that query would be displayed on the LaSSIE screen. If there are no matching actions, the LaSSIE user can generalize this query in various ways. One way to do it might be to click on CONSOLE-USER and browse the knowledge base, graphically, in the vicinity of that concept. The user

might find that `USER` is a more general concept, and try that. In the same way, `FLASH-HOOK-ACTION` could be replaced by the concept `STIMULUS`, which is more general. To make the query even less restrictive, the user might remove the `has-actor Console-Controller` restriction. Now the query would be, informally, "What is an action that connects a user because of a stimulus given by a user." In this case, there might too many matching individuals retrieved; however, the developer might find, among the retrieved individuals, one that is caused by an action by an attendant; the developer might use this description to modify the query and retry the retrieval. At this point, `Attendant-Add-User-Action` (see above) would be one of a smaller set of matches.

This example illustrates, first of all, how retrieval is based on the meanings of the descriptions, rather than the exact syntax of the query or the contents of the knowledge base; thus, a function that corresponds to *an action that adds a user to a call because of a button push by an attendant* is retrieved for a query that asks, "What is an action that connects a user because of a stimulus from a user?" This ability to retrieve on the basis of meanings of structured descriptions is very important, given the predilection of programmers in large software systems to use different terms to mean the same thing. (For example, a file that allocates a trunk may be called `trunk_seize.c`; one that allocates a touch-tone recognizer might be called `ttr_grab.c`.) It also illustrates how the LaSSIE interface assists the user in reformulating the query, both by using parts of retrieved individuals, and by exploring the knowledge base. It should be noted that the retrieved descriptions shown above combine the domain perspective (what operations are performed on what objects, to what effect), the architectural perspective (what processes, layers, etc. are involved), and the code level (what variables are referred to, what other functions are called, etc.). By inspecting the description, and the code for the function itself, the developer can either reuse the code directly, or at least gain information relevant to the task at hand. And besides helping in query processing, classification can also assist in organizing large numbers of descriptions.

In the initial LaSSIE prototype, we have explored how formal knowledge representation and reasoning techniques can be used to address some problems in software information systems; the work is continuing. We are currently exploring basic research issues in knowledge representation that have been raised by this work, and are working with potential customers in developing what we have learned by evolving the research prototype into a production version of LaSSIE.

The MView System

Clearly, a software information system must provide access to the information it represents. The kind of access will depend on the purpose of the SIS, as discussed in the introduction to this paper. This section describes a system, MView, that provides graphical access in an integrated way to several kinds of software information.

Overview. Several kinds of software information are naturally graphlike, including information about code, like a graph of function call relationships, module-to-module communication paths in a multiple-process system, and the organization of multiple derived versions of a software system. In addition, a structured knowledge base like the one used in LaSSIE can be represented and accessed as a graph. The MView system concentrates on displaying code-level information as separate graph structures, and allows the user to browse these structures for discovery purposes. These different views are integrated so that changing the display of one view can change that of another view, in order to highlight relationships between the views. The current MView prototype supports four views: a directory view, a function call view, a process and message view, and a function view.

MView's Model of the Code. MView's model of the code is reflected in each of the views it presents. First of

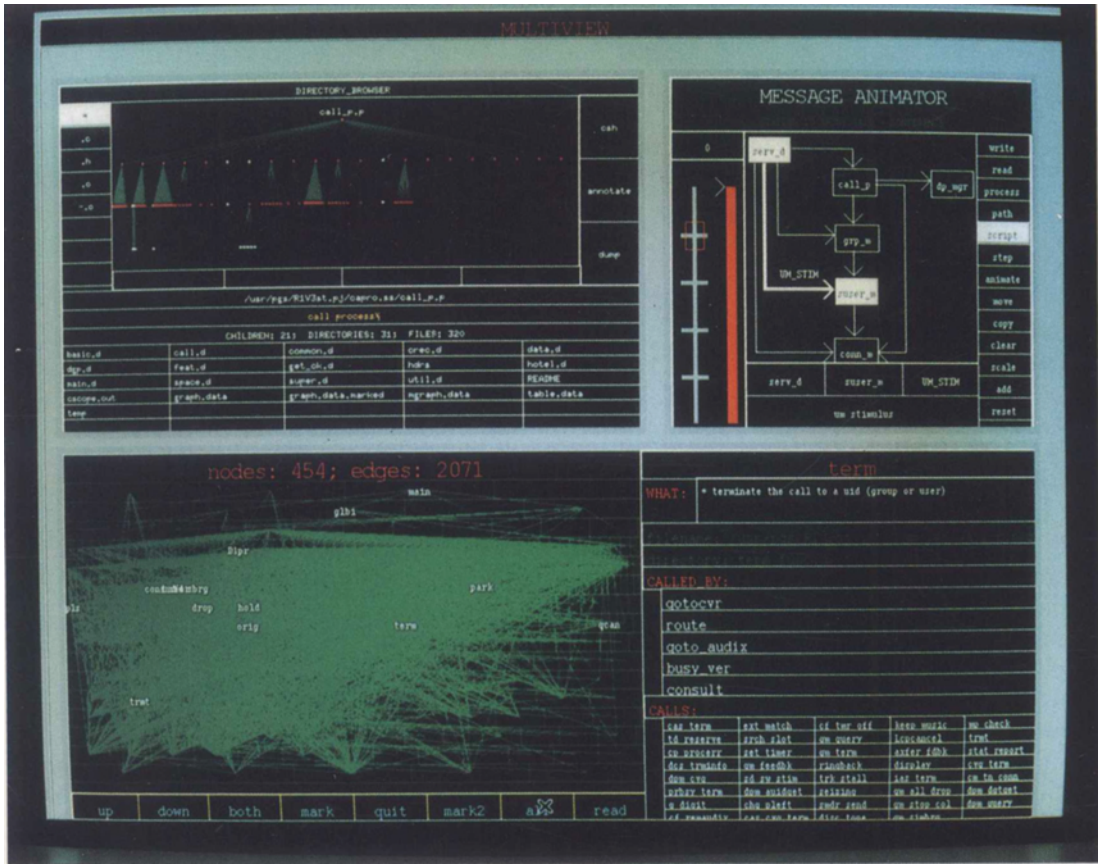


Figure 5. The MView screen. Four views appear: file/directory (top left), message/process (top right), function (bottom right), and call graph (bottom left). MView reveals important structures in a large software system. Developers use it to explore call graphs and examine source files.

all, the model assumes a well-organized directory structure where related source files are grouped together in subdirectories; the directory view takes advantage of this organization. Second, MView assumes that function-calling relationships are the dominant reflection of control flow in the program; the function call view allows these relationships to be browsed. Third, functions must be defined in source files, and those files contain a structured comment that can be identified by a prefix keyword and extracted syntactically. The function call view and the function view take advantage of these assumptions. Fourth, the process and message view assumes that the system under investigation is built from separate UNIX processes that communicate by small, typed messages, and that a transcript of message traffic is available. Finally, MView assumes that the source code is available on-line and that being able to examine this code is critical to the discovery process.

Figure 5 shows the MView screen running on the call processing portion of the Definity 75/85 system software base. The upper left-hand corner is the *file/directory view*, showing the directory structure of the

call process, one of the processes at the heart of the Definity 75/85 system. To the right of that is the *message/process* view. In the lower left is the *call graph view*, showing the set of call graphs of the functions that make up the call process. To the right of that is the *function view.*

MView provides a graphical view of the directory structure. This view shows the directory tree as a graphical object that is easily perceived and manipulated. The user can browse up and down the tree, examine the source files, and filter the tree for specific files or classes of files. The user can enter annotations that are attached to specific nodes and appear in the display when revisited. For example, in Figure 5 the annotation indicates that the current tree is for the call process.

The call graph view displays a call graph where each node (represented by a box) is a function and arcs represent static calling relationships. The graph is sorted so that called functions are always lower than calling functions (unless a cycle exists), but the display is not further optimized. This allows the graph to be displayed and browsed quickly. The set of functions available consists of those functions in the directory structure currently being displayed in the directory view. The call graph view can be filtered and browsed in various ways, but mostly in conjunction with the next view, the function view. In particular, by selecting the buttons on the bottom part of the view, the frontier of the call graph can be expanded. Thus, one can start with a single function, selected via the function view, and then explore the functions that are called, and so on. This provides a controlled process of exploration that can be extremely useful.

At any one time, there is a "current" function that is displayed textually in the function view. The user can change the current function by pointing to a node in the call graph. This view displays everything that is known about the function, including the complete path name, line of structured comment if available, and the functions that call it and that it calls. The labeled parts of the function view in Figure 5 illustrate this for the function *dialtone.* Although the display is textual, the "panes" of this view are mouse-sensitive. The user can do the following:

- Search for a function by name
- Restrict the call graph view to that function only, and then use the browsing ability in the call graph view to "push out" the frontier of the call graph in a controlled manner
- Restrict the call graph to only those functions defined inside a particular file or a particular directory
- Search for functions whose structured comments contain a keyword (e.g., *trunk*, *dialtone*, and the like) and restrict the call graph to those functions; these can then be individually examined
- Open a textual editor window to examine and change the source file itself.

The Process and Message View. The process and message view is a relatively independent view within MView. It is an incarnation of the Animator system, which can run as a stand-alone program. This view displays a diagram representing the process structure of a multiple process system and message paths between the processes, similar to the Software Development Assistant.[14] The user can read in a script or trace of message traffic, which is then displayed as a *scroll bar* form. The user can step through the script and the sending process, receiving process, and message type will be highlighted. The user can also filter the display to restrict it to messages sent by a given process, received by a given process, and of a given message type.

Using MView. MView has been used to explore the call processing software of System 75 (now the Definity 75/85 system). It has proven very useful for a novice trying to discover important structures in a large software system. Its primary benefit is that it lets user explore the call graph interactively and examine source files in a controlled fashion. The ability to locate files on the basis of name patterns and documentation keywords is also very useful (for Definity system software, a line

Figure 6. The MView screen after discovery. Here, the message/process view (top right) shows only those messages sent by the call process. The user has interactively filtered the call graph view (bottom left) by examining source code and calling relationships so that it shows how the messages were sent.

beginning with "WHAT" indicates a description of the function defined in that file).

We have used MView in the following informal experiment. Given a message trace, where each process is a black box that sends and receives messages, could the other views of MView allow someone to determine the sequence of functions responsible for generating a set of messages? This experiment was, at least partially, a success. Figure 6 shows the MView screen after determination of the functions in the call process responsible for the initial set of messages from that process (the messages `CM_ORIG`, `GM_ORIG`, `GM_COLL`, and `CM_TN_CONN`). In Figure 6, the message view has been filtered to show only those messages sent by the call process. The call graph has been interactively filtered by examining the source code and the calling relationships to determine how these four messages were sent. The flow of control is as follows. The function `mainlp` is a busy loop that receives messages and passes them to `cpdist`, which then distributes them to different "supervisors" depending on the state of the call, which is initially in the `idle` state. Thus, the first message is sent

to the idle supervisor, `idle_sup`. Then, `orig` is called because a call is being originated, and `orig` calls `cm_orig` and `gm_orig`, which sent messages to the connection manager and the group manager respectively. After that, `collect` is called to initiate the collection of digits (`collect` is shown in the function view). `Collect` then calls `gm_coll_dg` to tell the group manager to begin collecting digits. Then, `collect` calls `dialtone`. `Dialtone` calls `cmtnconn`, which sends the fourth message; this message tells the connection manager to actually connect a dialtone to the call.

Conclusion

Software information systems are computer-based tools that represent information about large software systems in ways that allow it to be used to aid software development and maintenance. This technology has already had substantial impact on software projects at AT&T. CIA, along with its application systems, has been used in over 200 projects ranging in size from small projects of a few thousand lines of code to giant projects of over a million lines of code. It is not uncommon to see system graphs, as in Figures 3 and 4, on developers' walls. In addition, users are building their own applications.[14] LaSSIE has addressed a large PBX System and is being ported to new platforms. MView, the newest of the three, is just starting experimental use.

There are a number of directions that this work will take. We need to find techniques to increase the power of the systems in specific applications and the breadth of the systems to address a larger variety of applications. We need also to pursue better understanding of the impact of this technology on the software development process.

A first step is to investigate integration of the systems that are now available. As we have seen, the individual systems stress different capabilities and have different strengths in terms of information acquisition, representation, access, and application. There is a clear opportunity for LaSSIE and MView to use the database created by CIA. In addition, the MView system, with its emphasis on different code views, and LaSSIE, which is concerned with higher-level concepts, can be integrated.

Beyond this, there are a number of opportunities, and difficult tasks, which can increase the impact of this technology on our software development process. One is to increase the abstraction capability of the information system. For example, users have added concepts such as "process" to CIA databases. There are also several levels above the object and action level represented in LaSSIE. Specifically, there are what we might call the *feature* level and the *plan* level. The feature level is oriented toward specific features, like *call forwarding* or *conferencing.* At an even higher level, one can view software at a level of functionality that could be called the *plan* level. For example, "establish a connection" is a goal that much call processing software is trying to achieve, independent of the nature of the calling parties or the calling features invoked.

Another opportunity is to increase the breadth of coverage of the SIS. A variety of data can be obtained about software as it executes, in addition to what has been discussed. Integration is also achievable between software knowledge and databases associated with other phases of the software life cycle. In particular, computer-aided software engineering (CASE) systems and systems for test and documentation management are good candidates for integration. This will increase the scope of usefulness of the information system.

The concept of a software information system as described here is not yet completed, but is already generating productivity and quality gains for software developers within AT&T. The next few years should see exciting new capabilities emerging.

References

1. G. D. Bergland et al., "Improving the Front-End of the Software Development Process for Large-Scale Systems," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 7-21.
2. B. W. Boehm, "Improving Software Productivity," *IEEE Computer*, September 1987, pp. 43-57.

3. F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, 1987, pp. 10-20.
4. J. Grass and Y.-F. Chen, "The C++ Information Abstractor," *USENIX Second C++ Conference*, San Francisco, April 1990.
5. Y.-F. Chen, "The C Program Database and Its Applications," *Summer USENIX Conference Proceedings*, Baltimore, 1989.
6. Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, March 1990.
7. Glenn Fowler, "A Case for Make," *Software: Practice and Experience*, accepted for publication, 1990.
8. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The Awk Programming Language*, Addison-Wesley, Reading, Massachusetts, 1988.
9. M. Stonebraker et al., "The Design and Implementation of INGRES," *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976, pp. 189-222.
10. M. I. Bolsky and D. G. Korn, *The Korn Shell Command and Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
11. E. R. Gansner, S. C. North, and K.-P. Vo, "DAG—A Program that Draws Directed Graphs," *Software: Practice and Experience*, Vol. 18, No. 11, 1988.
12. P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "A Knowledge-Based Software Information System," *Proceedings 12th International Conference on Software Engineering*, Nice, France, 1990.
13. B. W. Ballard, "A Lexical, Syntactic, and Semantic Framework for User-Customized Natural Language Question-Answering System," *Lexical-Semantic Relational Models*, Martha Evans, editor, Cambridge University Press, 1988.
14. C. C. Hayden et al., "The Software Development Assistant," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 76-90.
15. A. Borgida et al., "CLASSIC: A Structural Data Model for Objects," *Proceedings ACM-SIGMOD International Conference on Management of Data*, Portland, Oregon, 1989, pp. 59-67.

Appendix A. "Frame" Description Languages

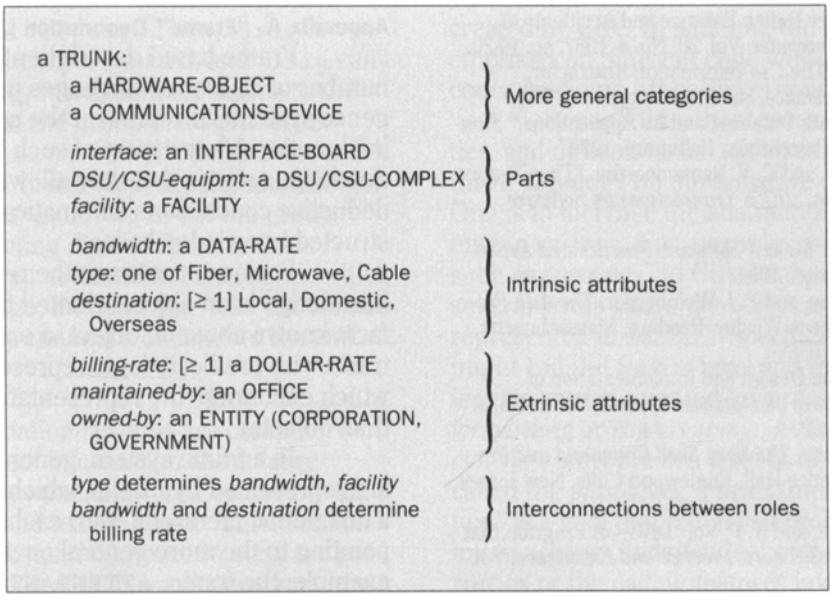
Frame-based description languages are one of a number of classes of languages used in artificial intelligence systems to represent the computer's knowledge of its domain. A key aspect of such "knowledge representation languages" is that they allow the machine to draw deductive conclusions automatically from formally constructed knowledge bases.

Frames encourage the representation of knowledge in an object-oriented fashion, clustering all facts known about an object in one structure (this contrasts with predicate logic representations, for example, which encourage the representation of sentences rather than objects).

In a frame system, general categories of objects are represented by *frames*, which are usually arranged in a taxonomic hierarchy, with each specialized frame pointing to the more general ones that it implies. For example, the frame `ATTENDANT-CONNECT-ACTION` (one in which the attendant is connected to something) could specialize the more general frame `CONNECT-ACTION` (in which any object is connected to another), which in turn would specialize `ACTION`. This is illustrated by the simple taxonomy in the box on page 40.

Each frame is a formal description that, when appropriate, can be applied to some individual objects (or just *individuals*). For example, in LaSSIE, Definity 75/85 system functions are represented by individuals. Those functions that connect the attendant to some resource (e.g., a trunk) would point, either directly or indirectly, to `ATTENDANT-CONNECT-ACTION`. This connection signifies that the individual is appropriately described by the frame, and is the principal way to assert a proposition in a frame system. The individual will then "inherit" all properties the members of the class generally have.

Each frame is itself composed of a number of parts, reflecting an intuitive breakdown of descriptions of domain objects. Besides specifying more general categories implied, a frame description will specify the parts that the type of object has, as well as its properties



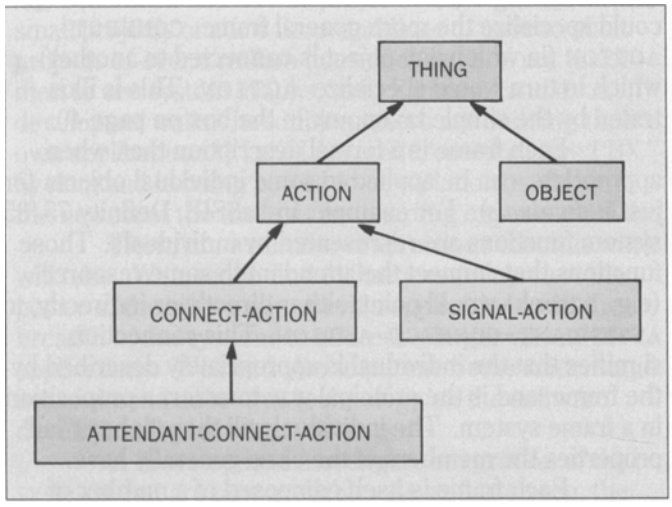


Figure A-1. Frame defining the concept of a `trunk`. Parts of a figure correspond to the parts of a normal dictionary definition.

(both internal, like the contents of a file, and external, like the author of a file). In frame languages like the one underlying LaSSIE, the parts and attributes are represented by structures called *roles* (e.g., `bandwidth`, `billing-rate`, in Figure A-1). The final part of the formal description is a set of constraints on the roles: how many fillers each has (e.g., how many callers can be connected to a call at one time?), what type of filler each requires (e.g., some actions operate only on resources), relationships among different roles, and so forth.

Because frames are formal objects, certain relationships between them can be deduced on the basis of their structure. A key deductive operation, from which LaSSIE gets much of its power, is one in which a new frame is placed in the taxonomy below every frame that represents a more general description, and above every frame that is more specific. The same operation, called *classification*, is used to determine when an individual satisfies a description. Not only does classification allow us to determine every individual that is described by a

user's query, it is also helpful in keeping the knowledge base properly organized. The classifier can detect inconsistencies and can show the user when a new description inadvertently falls below a more general frame that perhaps was not intended. The logical properties of the language means that the classification of a frame is always correct, and thus can point out subtle errors made by the user.

The Artificial Intelligence Principles Research Department at AT&T Bell Laboratories has developed a new frame-based description system, called CLASSIC,[15] which will be used in the next generation of the LaSSIE system.

Biographies (continued)

He has a B.S.E.E. from Princeton University and an S.M. and Ph.D., both in applied mathematics, from Harvard University. He joined AT&T in 1985. Mr. Chen is a member of technical staff in the Advanced Software Department, where he works on modeling and integration of software databases. He has a B.S. in electrical engineering from National Taiwan University, an M.S. in computer science from the University of Wisconsin, Madison, and a Ph.D. in computer science from the University of California, Berkeley. He joined AT&T in 1987. Mr. Devanbu is a member of technical staff in the Artificial Intelligence Principles Research Department. He works on acquisition, management, and retrieval of knowledge for software information systems. He has a B.Tech. in electronic engineering from the Indian Institute of Technology and an M.S. in computer science from Rutgers University, where he is a candidate for a Ph.D. He joined AT&T in 1982. Mr. Selfridge is a member of technical staff in the Artificial Intelligence Principles Research Department. He works on applying artificial intelligence technology to problems in software and on developing graphical discovery tools. He has a Ph.D. in computer science from the University of Rochester. He joined AT&T in 1982.

(Manuscript received December 13, 1989)