A Knowledge-Based Software Information System

Premkumar Devanbu
Peter G. Selfridge
Bruce W. Ballard
Ronald J. Brachman
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Abstract

The difficulty of maintaining very large software systems is becoming more widely acknowledged. One of the primary problems is the need to access information about a complex and evolving system. We are exploring the contribution to be made by applying explicit knowledge representation and reasoning to the management of information about large systems. LaSSIE is a prototype tool (based on the ARGON system) that uses a frame-based description language and classification inferences to facilitate a programmer's discovery of the structure of a complex system. It also supports the retrieval of software for possible reuse in a new development task. Among LaSSIE's features are an integrated natural language front-end (TELI) that allows users to express requests in an informal and compact fashion. Although not without some limitations, LaSSIE represents significant progress over existing software retrieval methods and strictly bottom-up cross-referencing facilities.

1 Introduction

The problems that arise with large, complex software systems include those of producing the code, managing a multi-person enterprise, testing the system, and assuring its integrity with respect to various specifications and other design documents. In many ways the most difficult problem involves maintenance, which includes fixing explicit bugs and, more importantly, upgrading the system to add new features or adapting the system for slightly different purposes. Some software systems, including those that control the space shuttle, nuclear power plants, and communication networks, have become so large and complex that no one person, or even a small set of people, understand them. This lack of a reliable knowledge source is exacerbated by people moving within an organization or leaving it altogether.

One common aspect of maintenance and other problems with large software is the *discovery* problem, i.e., the process of learning about an existing system in order to use or modify it. A developer must spend a great deal of time "discovering" features of an existing system, ranging from the overall software organization and the conceptual framework that drove that organization to the location and details of specific functions and data structures. All of this is prerequisite to implementing the actual modification for which the developer is responsible. Discovery also has a lot in common with the problem of retrieving code for re-use. One could imagine, for example, a system that could retrieve an existing piece of code that implements a specified function. The discovery process then becomes a process of formulating a series of queries to retrieve information, including actual code, about the system.

We have undertaken the task of building an information system (IS) to aid in the discovery process. This paper first examines the problem of developing such an IS in more detail. An existing system of large software is used as a test case in this work; and four specific "discovery queries" are examined to further motivate our approach. Next, the core system we have developed, called LaSSIE, is described in detail. Then, two extensions to LaSSIE are described: the addition of low-level code knowledge, and the integration of a natural language front end. Finally, we put this effort into perspective by examining the queries that can be currently handled, comparing our effort to previous work on software retrieval and related systems, and outlining directions for future work.

2 The Problem in More Detail

The AT&T System 75^{TM} [AT&T Technical Journal, 1985] is a Private Branch Exchange (PBX) that can handle up to 800 telephone lines. As a modern digital switch, it is controlled by a large and complicated software system that enables it to perform the basic switching functions as well as implement a sizeable collection of somewhat customizable features. This software is complex along several dimensions. It contains about a million of lines of C code; it comprises multiple versions, the latest of which is always in a state of flux; and, most importantly, it is a manifestation of a complex conceptual model of the architecture of the switch and its functionality. For this reason, the code can be understood only with reference to a framework that exists apart from it—a framework that reflects the hardware and software architecture, as well as the various resource and real-time-response constraints that the system is designed to satisfy.

The kinds of questions asked by System75 developers give us some insight into the conceptual model(s) of a large switch. Consider these queries, typical of the ones elicited in extensive discussions with developers:

- **Q1.** How do I allocate an international toll trunk?
- **Q2.** What messages are sent by a process in the network layer when an attendant pushes a button to activate the "Hold" feature?
- **Q3.** What C functions enable the Call Forwarding feature at a phone ?
- **Q4.** What functions in the Line Manager Process access global variables defined in "/usr/pgs/gp/tgpall/profum.h"?

These queries require different kinds of answers, which depend on knowledge associated with at least four different views of the system:

- A *functional* view—what is the code doing relative to the switching function? Our IS should know how internal operations, or *actions*, relate to external events such as a user picking up a phone. For Q1, some code might be described as "allocating a trunk", which is an operation internal to the switch.
- An *architectural* view—what is the hardware and software configuration? System 75 has a number of layers in its software architecture, each of which presents a "conceptual base" for the layers above it. For Q2, one needs to know what processes are in the network layer.
- A *feature* view—how are basic system functions associated with features such as "Call Forwarding"? For Q3, we must capture the way in which a feature cuts across a number of basic functions and has ramifications on all layers.
- A *code* view—how do the code-level components (source files, header files, functions, declarations, etc.) relate to each other? Functions call functions, source files include header files, functions and declarations are defined in source files, etc. For Q4, these relationships need to be represented.

In addition to these somewhat independent views, some additional issues must be addressed in a software information system capable of handling queries like those above. The views must be integrated in order to answer queries like Q2 that combine them. The system must also allow queries about the structure of the knowledge base itself, in addition to individual facts in the domain. How the queries are asked is very important if the system is going to be useful; the use of a formal query language will be much less effective than being able to query in a subset of English. Finally, the role of classification (discussed in detail later) will be important if the system is to out-perform a static keyword approach.

3 The LaSSIE System

LaSSIE is an experimental knowledge-based IS running on a Symbolics 3600, under ZetaLisp/Flavours. It consists of a knowledge base (KB), a window interface (based on ARGON [Patel-Schneider *et al.*, 1984]), a graphical browsing tool (based on the ISI-GRAPHER [Robins, 1988]), and a customized version of the TELI natural language interface [Ballard and Stumberger, 1986]. The system is designed to be used in a formulate-retrieve-reformulate cycle. If the answer to an initial query is unsatisfactory, the user can reformulate the query in a variety of ways, and try again. The reformulation step can be carried out using descriptions of retrieved individuals, or by exploring the knowledge base for related concepts. Natural language can be used to formulate a query or to reformulate part of a previous query.

In all modes of querying, the KB plays a key role in processing queries and in assisting the user in reformulating a query when necessary. The design of this KB is therefore crucial. We now describe the perspective from which the KB was constructed. The LaSSIE KB primarily captures the *functioning* of the system, from a conceptual viewpoint, with some information about its *architectural* aspects.

3.1 Functional Knowledge

Most of the functions of System 75 can be described in terms of *operations*, or *actions* that it performs. Some examples are

- Connect a user to a call.
- Initialize a call control process.
- Audit the digit translation database.
- Release buffer space to free some memory.
- Light up an LED when a call is terminated at a station.
- Allocate a touch tone recognizer because of a pickup by a user.

Corresponding to each of these actions are segments of code and the files that contain them. Notice also that these actions can be cast into the general form *Actor does Action on Object to Recipient using Agent because-of Action.* This general form was used to formulate descriptions of a wide range of actions in the call processing area of System 75. It also motivated the design of LaSSIE's natural language interface. We then coded these descriptions in the KANDOR knowledge representation system [Patel-Schneider, [1984], which classifies them into a conceptual hierarchy using a formally defined subsumption inference operation. This hierarchical KB is the core of LaSSIE, consisting of about 200 frames and 3800 individuals, which describe System 75 using functional, architectural, and code-level concepts, and their inter-relationships.

3.2 The Knowledge Base

As shown in Figure 1, the four principal object types of concern in our domain are `OBJECT`, `ACTION`, `DOER`, and `STATE`. The edges of the taxonomy have their common "IS-A" meaning[1]. `DOER` represents those `THING`s in the system that are capable of performing actions. Nodes below `DOER` and `OBJECT` represent the architectural component of the system, i.e., its hardware and software components. Nodes below `ACTION` represent the system's functional component, i.e., the operations that are performed to or by the system. The relationship between the two system components is captured by various slot-filler relationships between `ACTION`s, `OBJECT`s and `DOER`s. Each action description

[1]In particular a `TRUNK` IS-A `RESOURCE-OBJECT`, a `COMMUNICATIONS-DEVICE`, and a `DOER`.

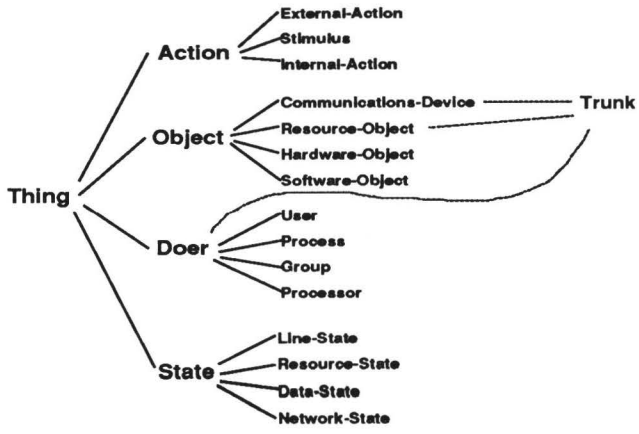

Figure 1: Top 3 Levels of the LaSSIE KB

combines the top level concepts in various ways using KANDOR descriptions. A typical one is:

```
1   (knd-de frame USER-CONNECT-ACTION
2        NETWORK-ACTION CALL-CONTROL-ACTION
3        defined
4        (exists has-actor (generic PROCESS))
5        (exists has-agent
6                    (value BUS-CONTROLLER)))
7        (all has-operand (generic USER))
8        (exists has-environment
9                    (generic CALL-STATE))
10       (exists has-result
11                   (value TALKING-STATE))
```

In other words, USER-CONNECT-ACTION [1] is by definition [3] a network action [2] and a call-control-action [2] which is done by a process [4], using the bus-controller [5,6] on a user [7], which takes the user from some call state [8,9] to the talking state [10,11]. LaSSIE's KB contains 102 action concept descriptions of this type, which are classified into a tangled hierarchy. Farther down in the hierarchy, the action concepts become very specific.

The most specific action types, each of which correspond to a particular function/source file[2], are coded as individuals. For example,

```
1   (knd-de individual ADD-USER-ACTION
2        (ACTION)
3        (has-actor CALL-CONTROL-PROCESS)
4        (has-agent BUS-CONTROLLER))
5        (has-operand GENERIC-USER)
6        (has-recipient GENERIC-CALL)
7        (has-environment GENERIC-CALL-STATE)
8        (has-result TALKING-STATE)
9        (implemented-by
10           /usr/pgs/gp/tgpall/profum.c)
```

In other words, ADD-USER-ACTION [1] is an action [2] that is performed by the call control process [3] using the bus-controller [4]; its operand is any user [5], and its recipient is a call [6]; it takes its operand from any call state [7] to the talking state [8]; and it is implemented by the source file /usr/pgs/gp/tgpall/profum.c [9,10]. It should be noted here that the KANDOR classification algorithm will ensure that this individual gets classified under the frame USER-CONNECT-ACTION mentioned above. It is this kind of classification that organizes the large number of frames and individuals in LaSSIE into a usable form.

[2]System 75 has only one function per source file.

3.3 Why Classification is Essential

In a large software system, it is very difficult to know how one part of the system relates to another. Our approach is to build explicit descriptions of the actions performed by different parts of the system, then use formal inference to build a taxonomic hierarchy, where all IS-A links are derived from the descriptions themselves. The formal, logical nature of the inference, which is based on an intuitive set-theoretic semantics, ensures that action descriptions are classified where one would expect to find them. (The inference procedure that accomplishes this is described in [Patel-Schneider, [1984].) Thus, programmers working on distinct components of the system can describe the operations performed by their specific components and be sure that their work is properly organized and categorized with other components for retrieval and re-use by later programmers.

The taxonomy can also be useful in query formulation and reformulation. When querying the database, if there are no answers, or if there are too many answers, a tool like ARGON [Patel-Schneider *et al.*, 1984] can be used to specialize, generalize, or look for alternatives for an appropriate portion of the query, modify that portion, and retry the query. For example, a programmer may query the system for an action that reinitializes a trunk. This query may be stated as "a process-operation whose operand is a trunk and whose result is the initialized-state". If no such action exists, the the user can use the taxonomy to generalize either TRUNK, INITIALIZED-STATE or PROCESS-OPERATION, to see whether any matching instances are retrieved.

Large AT&T Switches like System 75 and 5ESS^{TM} are actually structured to support re-use. The layered architecture is intended to promote the re-use of primitives from lower levels to construct higher-level operations. Although this is intended to simplify construction and maintenance, identifying the appropriate primitives when they are needed can be difficult. When primitives are not used as they were intended, the original simplicity of the system is lost; in addition to needless re-coding, the system becomes harder to maintain and understand.[3] The LaSSIE KB will help prevent this *loss of architecture* by explicitly codifying the primitives supported by the architecture into a formal, taxonomic knowledge base and making it available for browsing and querying with a powerful user interface.

4 Incorporating the Code View

Representing the "code view" of System 75 means developing a general representation of code objects and their inter-

[3]It is dangerous for the same operation to be reimplemented several times by different programmers in different subsystems; besides the wasted work, when a bug develops in this operation, *every single implementation thereof* must be found and fixed.

relationships, then populating this generic taxonomy with instances from the system. The goal is to facilitate answering queries that contain requests for general information about file structure ("what extensions do source files have?"), general information about System 75 software ("where are System 75 header files located?"), and information about specific code objects ("what functions call 'apost' and include 'errproc.h'?").

We have designed a taxonomic and relational model of the C language and C programming conventions and implemented most of this model in a KANDOR knowledge base. This knowledge base, which is integrated with the functional and architectural knowledge described in the previous section, represents the Unix file structure, including directories, C source files, header files, object files, makefiles, and their inter-relationships; and cross-reference information, including source files and functions, header files, macro definitions, and type (struct) declarations. The relationship knowledge includes both "defined-in" (as in what function is defined in what source file, or what macro is defined in what header file), and "referenced-in" (as in what functions reference (call) what other functions) relationships.

We have added to this generic knowledge base information specific to System 75 and its own software methodology. This information includes directory and file naming conventions as well as conventions about the file structure itself.

This conceptual framework of about 40 frames has been populated with individuals generated automatically from 310 System 75 source and header files. This generation was done in a three-stage process starting with the data file created by CScope [Steffen, 1985], which is then further analyzed to generate two-place relations between code objects, which are then grouped together and used to generate legal KANDOR definitions. The resulting knowledge base includes, besides the 310 files, 27 directories, 433 functions, 39 structure definitions, and 1416 `#defines` (macros). These objects are very richly interconnected; a fairly typical function will call a dozen others and use several dozen `#defines`.

5 Adding a Natural Language Interface

To provide a natural language interface for LaSSIE, we customized the TELI system, which maintains data structures for each of several types of knowledge [Ballard and Stumberger, 1986, Ballard, 1988]. This information includes (1) a *taxonomy* of the domain, which enables the parser to perform several types of disambiguation; (2) a *lexicon*, which lists each word known to the system, along with information about it; and (3) a list of *compatibility tuples*, which indicate plausible associations among objects and thus reflect the semantics of the domain at hand. For example, an agent can perform an action on a resource, but actions cannot be performed on agents, resources cannot perform actions, etc.

In LaSSIE, KANDOR individuals generally correspond to proper nouns (i.e., names), while a frame may correspond to either a verb or a common noun. Generally, frames under `ACTION` correspond to verbs describing actions, while nodes under `OBJECT` or `DOER` correspond to nouns. For example, the frame `ALLOCATE-ACTION` maps to "allocate", "reserve", and "grab", and `PROCESS` maps to the noun "process". Individuals are usually associated with one or more proper nouns in an obvious way. For example, the individual process `BUS-CONTROLLER` is named "bus controller".

As explained above, action frames include slot restrictions corresponding to case roles including the actor, the operand, the recipient, the cause of the action, etc. To each of these, there naturally correspond one or more English *prepositions*. Thus, each slot associated with an action frame gives rise to compatability tuples as described above. As an example, consider the frame definition[4] shown below, with its associated verb *connect*:

```
1 (verbframe CALL-CONNECT-TRUNK-ACTION
2    (connect) (ACTION)
3    (exists has-operand (generic TRUNK))
4    (exists has-recipient (generic CALL))
5    (exists has-actor
6            (value CALL-CONTROL-PROCESS)))
```

For this frame and its slots, the following compatibility tuples are generated:

```
CALL-CONTROL-PROCESS connect TRUNK
CALL-CONTROL-PROCESS connect to CALL
```

The "annotation" of the knowledge base was done manually, after which the conversion to the TELI data structures is automatic. The resulting compatability tuples for LaSSIE include 167 verb case frames, corresponding to a total of 40 verbs. The lexicon contains 882 entries, including 193 common nouns and 260 proper nouns.

To process a query such as *What actions by the line controller are caused by an action by an attendant?*, TELI parses the input, making intimate use of the compatibility tuples and the taxonomy to insure globally consistent case bindings. The final parse tree is then converted into a semantic structure resembling a first-order logical form, which is sent to a LaSSIE-specific filter to strip out quantifiers associated with words such as "a" and "the". The resulting structure is then passed back to LaSSIE for translation into a query that is executed (thus performing a retrieval) but which also provides an editable ARGON expression. For example, TELI's output for the above query is:

```
(set A1 (ACTION A1)
      ((ACTION BY AGENT) A1 Line-Controller)
      ((ACTION CAUSE ACTION) A2 A1)
      ((ACTION BY AGENT) A2 P1)
      (ATTENDANT P1))
```

This is then translated into the following editable ARGON query:

ACTION
  HAS-ACTOR **LINE-CONTROLLER**
  HAS-CAUSE *ACTION*
    HAS-ACTOR *ATTENDANT*

Note that the user of LaSSIE need not know the details of the underlying KB in order to pose questions in English but, by seeing the associated ARGON query, may well learn something about the KB when the input is processed. For

[4] Actually, the form shown generates a table entry for the action, associating it with a verb name, and generates a standard `KND-DE` call to define a KANDOR frame.

example, a query *What actions by a process reserve a touch tone recognizer because a pickup by a user ?*, would be translated to

ALLOCATE-ACTION
HAS-ACTOR PROCESS
HAS-CAUSE *OFF-HOOK-ACTION*
HAS-ACTOR *USER*

In this case, the user would learn that the action verb "reserve" corresponds to ALLOCATE-ACTION, "pickup" to OFF-HOOK-ACTION, and also that actors of and causes of ACTIONs respectively are specified by using the HAS-ACTOR and HAS-CAUSE slots.

6 Discussion

6.1 Results

The overall goal of this project was to build an information system that represents a significant amount of knowledge of a large software system. Our motivating problem was that of *discovery*: the need of developers to be able to access existing knowledge of the system prior to extending it. As we built LaSSIE, we were forced to elucidate the *kinds* of knowledge that we needed to represent, as well as how it was to be represented. LaSSIE represents hundreds of interrelated facts about the call-processing part of System 75, including a taxonomic breakdown of high-level actions that drive the system, and low-level knowledge about the code structure. The knowledge of code structure was generated automatically from source files. We added a natural language interface that allows many queries to be formulated in English, and uses the underlying knowledge base to help resolve lexical and syntactic ambiguities. The use of the existing ARGON system allows a very powerful form of exploration.

LaSSIE can answer hundreds of different queries about System 75, including queries about actions, architecture, code, and combinations of the three. ARGON or TELI is used to formulate these queries, which are answered by showing a list of matching instances. These instances can be used to generalize or specialize the query, and the process continues. With regard to the discovery queries presented in Section 2, which are illustrative of some important classes of queries, the current version of LaSSIE successfully answers Q1, Q3, and Q4 exactly as stated. Q2 is an interesting case: while it cannot be handled exactly as stated, LaSSIE can be used to home in on the answer. Q2 is: "What messages are sent by a process in the network layer when an attendant pushes a button to achieve the 'Hold' feature?". The problem is that the sending of a message is not represented at a fine enough grain, so that "message sent when an attendant pushes a button" cannot be directly retrieved[5]. However, LaSSIE can be used to answer the related query, "what functions are called when an attendant pushes a button to activate the 'Hold' feature". At this point, the user can inspect the functions' source code manually to determine which messages could be sent under actual running conditions. Even for queries that cannot be handled exactly, LaSSIE's mode of interaction is rich enough to provide at least a partial answer.

[5]To answer this question precisely, the code has to be actually run or simulated; this means that computing a correct answer would be undecidable.

6.2 Related work

Traditional approaches to software retrieval fall into two complementary categories: high-level classification techniques, which emphasize retrieval by software category; and low-level cross-reference tools, which facilitate various kinds of browsing at the code level.

The goal of high-level classification techniques is usually to create a database of programs and program parts that can be retrieved for re-use. Two methods of indexing are normally used. In the first, keywords are used to describe and classify software components and keywords are used for retrieval in the traditional fashion: a user will list a set of uninterpreted terms that "describe" the desired component and the system will retrieve all components that are close in some multi-dimensional space defined by the keywords. The CATALOG system [Frakes and Nejmeh, 1987] is of this type. Clearly, the utility of a keyword system will depend on how well the keywords describe the components and how well they match those keywords normally thought of by a user. The further issue of generating the database arises here, as it does with any such database.

Prieto-Diaz expanded the notion of strict keyword retrieval by forming a static taxonomy of concepts or "facets" that impose an organization on the set of keywords. [Prieto-Diaz, 1987] For example, the facet "Function" includes the terms *add, append, create, evaluate*, and the facet "Objects" includes the terms *arrays, expressions, files*, and *functions*. The system is queried much like a keyword system, but may be more amenable to a "query–modify" retrieval cycle than pure keyword description. Once designed, however, his classification scheme is static and fixed.

At the low level, there are a number of tools derived from the notion of a cross-reference listing, which indexes two code components with each other, for example, files and function calls. MasterScope [Teitelman, 1974] was one of the earliest such tools; it was integrated with the InterLisp environment. CScope [Steffen, 1985] and CIA [Chen and Ramamoorthy, 1986] are tools that run in the C environment; they both automatically generate a database of two-place relations (essentially, the "defined-in" and the "referenced-in" relations) and allow a user to query or browse the relationships of a large software system. CIA, the more comprehensive of the two, is based on the relationships between five code objects: files, functions, global variables, type definitions, and macro definitions ("#define" statements), and it allows limited two-place queries. For example, one can ask for all functions that call a given function, or all macros used in a given file. The current implementation is unable to handle queries with conjunctions, negation, or quantification.

Neither of these two approaches—software classification techniques and cross-reference tools—comes close to achieving the power of LaSSIE, due in part to the fact that they do not provide inference capabilities. They could not handle the classes of queries illustrated by Q1–Q4 in Section 2. They do not address the issue of integrating high-level functional knowledge and code knowledge, attempt to model

the underlying domain, or capture more than a single view of software.

6.3 Directions for future research

LaSSIE has reached a plateau of accomplishment, but there is a long way to go before it is the ideal software Information System. For example, we need to incorporate more of the architectural view of System 75. This involves a more detailed examination of the process-level functioning of the system, including details on the purpose of specific processes, the messages they send, and the meanings of those messages.

On a more practical level, we are re-designing LaSSIE to use the CLASSIC knowledge representation language [Borgida *et al.*, 1989]. Present plans also include porting the system from the Symbolics machine to run on SUN workstations and other Common Lisp environments. This involves a re-design of the ARGON interface.

We must also continue to address the problem of knowledge acquisition. Constructing a knowledge base is labor intensive, and we need to examine the possibility of doing some of it automatically. The acquisition of the code knowledge in the current version of LaSSIE was done automatically; acquiring other kinds of knowledge in a similar manner is a research project in itself. There is reason to believe that some large software systems include enough highly standardized comments that this can be done. There has recently been some promising research in the area. Biggerstaff [Biggerstaff, 1988] has proposed an approach to reconstructing the lost design of software from a variety of sources, including source code, design documents, using a domain model, mimicking the process by which an expert who is well acquainted with, for example, windowing systems in general, might reconstruct the design of a new windowing system using his/her knowledge of the general structure of such systems. On a more formal (and somewhat closer the code) level, Letovsky [Letovsky, 1988] and Wills [Wills, 1988], have used formal methods to discover algorithmic patterns (loops, tests, accumulations, etc) in programs.

7 Summary

Our approach to the problem of maintaining and extending large software systems is to employ explicit knowledge representation and reasoning technology. This has led us to formulate complementary models of a software system in terms of its function, architecture, features, and code. To this end, we constructed a knowledge base that captures critical aspects of three of these four views of the System 75 switching system. We also customized and incorporated a natural language component to be used either alone or in conjunction with the ARGON interface.

As a result of these efforts, LaSSIE is the first information system to incorporate multiple views of a large software system embedded in an environment that lets a user query the system and explore the knowledge base. Although much remains to be done, LaSSIE can handle successfully many classes of queries about a large software system.

References

[AT&T Technical Journal, 1985] *AT&T Technical Journal*, Special Issue on the System 75 Digital Communications System, Vol. 64, No. 1, Part 2, January 1985.

[Ballard and Stumberger, 1986] Ballard, B. W., and Stumberger, D.E., Semantic Acquisition in TELI: A Transportable, User-Customized Natural Language Processor, *24th Annual Meeting of the ACL*, New York, June 1986, pp. 20-28.

[Ballard, 1988] Ballard, B. W., A Lexical, Syntactic, and Semantic Framework for a User-Customized Natural Language Question-Answering System, Lexical-Semantic Relational Models, Martha Evens, Editor, *Cambridge University Press*, 1988, pp. 211-236.

[Biggerstaff, 1988] Biggerstaff, T. J., Design recovery for Maintenance, MCC Technical Report Number STP-378-88, November, 1988.

[Borgida *et al.*, 1989] Borgida, A., Brachman, R. J., McGuinness, D., and Resnick, L. A. CLASSIC: A Structural Data Model for Objects. *Proc. ACM SIGMOD-89, on Management of Data*, Portland, OR., May-June, 1989

[Chen and Ramamoorthy, 1986] Chen, Y. F. and Ramamoorthy, C. V., The C Information Abstractor, *COMPSAC*, Chicago, October 1986.

[Steffen, 1985] The CScope Program, Berkeley UNIX Release 3.2, originally written by Joe Steffen.

[Frakes and Nejmeh, 1987] Frakes, W. B. and Nejmeh, B. A., An Information System for Software Reuse, *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, p. 142-151, 1987.

[Letovsky, 1988] Letovsky, S.I., Plan Analysis of Programs, Ph.D. Thesis, Yale University, December 1988.

[Patel-Schneider, [1984] Patel-Schneider, P. F. Small can be beautiful in knowledge representation. In *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, December, 1984.

[Patel-Schneider *et al.*, 1984] Patel-Schneider, P. F., Brachman, R. J., and Levesque, H. J. Argon: Knowledge representation meets information retrieval. In *Proc. First Conference on Artificial Intelligence Applications*, 1984, pp. 280–286.

[Prieto-Diaz, 1987] Prieto-Diaz, R. and Freeman, P. Classifying Software for Reusability, *IEEE Software* 4: 6-16, January, 1987.

[Robins, 1988] Robins, Gabriel., The ISI Grapher Manual, *USC Information Sciences Institute*, Technical Manual ISI/TM-88-197, February 1988.

[Teitelman, 1974] Teitelman, W., The INTERLISP Reference Manual, *Bolt, Beranek and Neuman*, 1974. Section 20 describes MasterScope written by L. M. Masinter.

[Wills, 1988] Wills, L., Automated Program Recognition, Technical Report 904, MIT AI Labs,