LaSSIE: a Knowledge-Based Software Information System

Premkumar Devanbu*
Ronald J. Brachman
Peter G. Selfridge
Bruce W. Ballard

Abstract

Invisibility is an inherent and significant problem in the task of developing large software systems. There are no direct solutions to this problem; however, there are several categories of systems—relational code analyzers, reuse librarians, and project management databases–that can be seen as addressing aspects of the invisibility problem. We argue that these systems do not adequately deal with certain important aspects of the problem, namely, *semantic proliferation, multiple views*, and *the need for intelligent indexing.* We have built a system called LaSSIE, which uses knowledge representation and reasoning mechanisms to directly address each of these issues. LaSSIE provides a knowledge base explicitly representing a large software system, and an interactive interface to give programmers direct access to the software through several semantically-based views. LaSSIE is limited in a number of ways, but it illustrates how representation and reasoning mechanisms can be used to alleviate invisibility and complexity.

1 Introduction

The growing cost of software development, particularly in larger systems, is well documented. Efforts at containing this growth have met with limited success. In his classic paper, "No Silver Bullet" [11], Brooks attempts to explain this; he argues that existing developments in software engineering, such as high-level languages, time-sharing and integrated programming environments have overcome certain *accidental* difficulties in the software development process; certain other problems, that are the *essence* of the software process, still remain. He identifies four essential difficulties in building large software systems:

- *Complexity:* The complexity of the development process, the application domain, the internals of the system, etc.
- *Conformity:* Software cannot have a regular formal structure (like Mathematics or Physics) because it must adapt to interact with people and institutions according to their diverse needs.
- *Changeability:* In any system, the software component is usually the easiest one to modify.
- *Invisibility:* The structure of software (unlike those of buildings or automobiles) is hidden. The only external evidence we have of software is its behavior when executing.

Brooks argues that there is no single, immediate solution to these difficulties, and thus, no "silver bullet" to solve the software engineering crisis. In the case of *Conformity* and *Changeability*, we must agree; it appears that software exists to fill market needs, and there is no denying that these needs change. Thus, the extreme malleability of software presents a compelling reason to change the software part of the system to meet these evolving needs. These two difficulties seem inherent in the interactions of software-driven systems with the marketplace.

In this paper we focus on the other two essential difficulties, *Complexity* and *Invisibility.* We explore the relationship between them, and their causes and effects. We also discuss existing approaches to these problems, and the shortfalls thereof. We then present a system called "LaSSIE" (*La*rge *S*oftware *S*ystem *I*nformation *E*nvironment) that incorporates a large knowledge base, a semantic retrieval algorithm based on formal inference, and a powerful user interface incorporating a graphical browser and a natural language parser. LaSSIE is intended to help programmers find useful information about a large software system—in our case the AT&T Definity^{TM} 75/85[1] [3]. Finally, we evaluate the contributions made by LaSSIE to the technology of information systems that deal with invisibility.

*Also at the Department of Computer Science, Rutgers University, Piscataway, New Jersey 08854, USA.

[1] Definity^{TM} 75/85 is a scalable PBX product with a flexible and powerful feature set; the software controlling this switch is of the order of a million lines of NCSL of C.

2 Complexity and Invisibility

Complexity is a common feature of software systems, especially large ones. It has various causes: a large, complex (perhaps even inconsistent) set of requirements, long system lifetimes (due to prohibitive costs of redevelopment), difficulties in interpersonal and interorganizational relationships (see [14] for a comprehensive survey), etc. For a descriptive analysis of the nature of complexity in large software systems, see Perry [35]. For our purposes, it is instructive to consider how invisibility and complexity interact in large systems.

2.1 Invisibility and the Evolution of Complexity

In many cases, the *initial* design and architecture of a system is derived after careful analysis of the requirements. There are a small group of professionals involved in the beginning, and there is usually good communication amongst them. These are generally experienced designers and architects who have considerable exposure to the application domain; they have a deep understanding of the relationship between the needs of the domain and the design rationale. The initial architecture often structures a system in a "virtual machine" (VM) style [31]: the system is implemented as a layered set of VM's, each VM implementing reusable primitives for the VM's above it. These primitives have a simple, well-defined relationship to the needs of the application domain. The layered structure is conceived to promote the simplicity and understandability of the system (and consequently, programmer productivity). In short, the design embodies certain *architectural principles* that, when carefully followed by developers, keep the system quite simple and elegant, and relative to its size, not all that complex. The design of the AT&T Definity 75/85 is a good example of this.

Unfortunately, the structure and principles underlying large systems are invisible; there is no clear presentation of the architecture for programmers to examine, and thus to honor, as they extend the system. In [11], Brooks points out that in civil engineering, geometric abstractions such as scale drawings and stick-figure models capture the essence of the design for everyone involved in a large construction project. He goes on to point out why this is much harder for software systems:

> "As soon as we attempt to diagram software structure, we find it to constitute not one, but several graphs superimposed one upon another.[2] These graphs represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. ...In spite of the progress in restricting and simplifying the structures of software, they remain essentially unvisualizable..." [11]

[2]Brooks suggests that software must be understood from *multiple points of view* (we amplify this below, in Section 3.)

Invisibility is thus an inherent property of large software projects; programmers find it difficult to get needed information. Because of this, as the project evolves in response to the marketplace, and grows in size (both people and code) and complexity, the initial architecture, as well as the advantages afforded by it, slowly dissipate. Invisibility has other adverse effects, which we outline below.

2.2 Effects of Invisibility – The Discovery Task

Invisibility complicates the task of acquiring the knowledge (a complex melange of application domain knowledge, architectural and design assumptions, and project-specific development practice) that a programmer needs to do his/her job. This phenomenon has recently been the subject of study at AT&T Bell Laboratories [28] and elsewhere [44]. Modica [28] has called this the *discovery* task. Discovery has impacts on both productivity and quality of the software produced.

Modica has found from surveys that the discovery task can take anywhere from 30 to 60% of a developer's time, depending on the nature of the project, and the expertise of the person involved. Thus, discovery adversely impacts a developer's *productivity.* The *quality* impact of discovery arises from a developer's action upon incorrect or insufficient information. Soloway *et al.,* in [44], document cases (actually in a very small task) where programmers made mistakes when they used documentation that presented information in one way, and not when it presented information in another form. In two different studies, Soloway, Adelson, and Ehrlich showed how a lack of knowledge affects the quality of the results in design tasks [1] and in small programming-related tasks [43]. Without proper knowledge of the architecture of the system, developers can make mistakes and produce incorrectly functioning code.

In large systems, the impact of poor discovery is of a different form, but much more insidious. Developers can implement their subsystems in a manner that *violates* the architectural principles of the large system they are modifying. The code may work correctly, and consequently pass system test; however this kind of violation, carried out at various points over a period of time, results in a *loss of architecture* that gradually erodes the original simplicity of the system. Thus, this lack of knowledge among developers leads to a vicious cycle where the system becomes progressively more complex, and thus harder to know. Invisibility and complexity can be seen to "feed" off each other.

2.3 Effects of Invisibility – Reduced Reuse

As we described earlier, the initial layered design of large systems is conceived to provide a large number of reusable primitives. Unfortunately, a developer whose task it is to implement, modify or add a special operation to the system often cannot find out if it has already been done, and if not, whether there is a particular way of doing it that would conform to the initial, intended architecture of the system. Because of this difficulty, instead of reusing existing primitives, programmers reimplement them; this results in lowered productivity.

Reduced reuse can also have a negative impact on quality. Conversations with developers revealed several cases where programmers, unaware of a "virtual machine primitive" for an operation, repeatedly re-implemented the same operation (in one case, ten times!). When a bug was found in the operation *every single implementation* had to be successively found and fixed. This is a case where invisibility led to reduced code reuse, and then to increased complexity. Proper knowledge of the system could have prevented this. Thus, we would like to build into our system a library of reusable parts, along with a helpful access mechanism. These points are discussed in more detail in [16].

In this section, we have described the problem of invisibility and its deleterious effects. There have been some attempts to deal with this problem; before we go on to explore them, we first describe an important aspect of invisibility that many systems fail to address.

3 Multiple Points of View

The discovery process can be thought of as a series of questions that developers must ask and answer, to gain a proper understanding of the part of the system that they are working on. Here are some typical questions, gathered from conversations with developers in the AT&T Definity 75/85 project:

- **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 a Call Forwarding feature activation?
- **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 *domain model* view (Q1)—What is the code doing relative to the conceptual objects (trunks) and actions (allocating, releasing, connecting, signaling) in the switching software domain?
- An *architectural* view (Q2)—Q2 makes reference to the "network layer", which is a part of the architecture of Definity 75/85.
- A *feature* view (Q3)—How are basic system functions associated with customer features such as "Call Forwarding"?—Q3 combines the feature view with the code view:
- A *code* view (Q4)—How do the code-level components (source files, header files, functions, declarations, etc.) relate to each other?

To maintain the system competently, a developer must be able to explain (to herself) the structure and behavior of the system; in order to construct such an explanation, the developer has to draw upon at least the four different kinds of knowledge outlined above—how does this piece of the system relate to other components, fit in with the architecture, reuse the built-in primitives, and satisfy the customer's need? The pursuit of this sort of knowledge is hindered by the complexity and invisibility of the system's architecture. Field studies by Curtis, Krasner and Iscoe [14] have shown that programmers by and large fail to develop this kind of understanding; they talk about the problem of "thin spread of application domain knowledge". They find that a deep understanding of the application domain and its relationship to system architecture is not widespread in the programming workforce. The majority of programmers don't understand the system from "multiple points of view"[3].

Invisibility can then be thought of as the problem of getting knowledge from the few experts that have it, to the large number of developers who do not[4]. As we shall now see, some attempts have been made to address this information management issue.

4 Software Information Systems

There are several systems that provide a variety of information about an existing system to a programmer. They seem to fall into three main categories: relational code analyzers, project management databases, and reuse librarians.

4.1 Relational Code Analyzers

Relational Code Analyzers (RCA's) such as MasterScope [47], CScope [46], and CIA [12] derive certain code-level relationships (such as function-calls-function, function-uses-variable, etc.) directly from source code. This information is generally stored in a relational database. One can query this database using a typical relational language. Questions such as "What functions that call the function `flash-button` refer to a global variable that is also referred to by the function `display-number?`" can be asked.

RCA's can answer a number of useful questions for programmers, and are widely used. The information that is retrieved is current, and reflects the actual state

[3] This should not be confused with the "multiple views" problem in databases, where different views are provided for different users; our goal here is to illustrate the fact that different kinds, or *ontologies* of knowledge, are needed to understand how a large system works.

[4] Often-asked questions in the Definity 75/85 project, such as the ones above, can usually be answered by only a few human experts, whose time is both limited and valuable.

of the code base. However, they fail to address certain important aspects of invisibility. First, they simply do "string-matching" during retrieval; they do not capture the *semantics* of the strings used in the query. For example, in the previously cited query, the meaning of `flash-button` or `display-number` (or indeed, their relationship to `hold-button` and `display-name` respectively) would be opaque to an RCA. Users cannot query RCA's using *descriptions* of components; if one gets a name wrong, there is no recourse.

Secondly, the query processing ability of RCA's is strictly limited to the *code* view discussed in Section 3. They cannot answer queries that incorporate other views. Though they capture the syntactic structure of the code, they do not describe the relationship between the structure of the code and the architecture of the system or the domain of application. As Brooks points out, the difficulty of integrating these different views is one reason why the structure of large systems remains invisible. In the following section, we discuss project management databases, which address the issue of integrating different kinds of information about a software development project into one information base.

4.2 Project Management Databases

Project management databases (PMDB's) are primarily designed to serve as repositories of all the artifacts generated and used during the life of a software development project, such as documents, programs, test scripts, problem reports, personnel, tools, milestones, and accounting charts. They are intended to provide database support for all lifecycle activities, beginning with requirements specification right through to system test.

PMDB [34] is a typical example of such a system. It aims to collect almost all of the information about a project into one central information system, which can be queried and updated. The information system is based on the entity-relationship model to characterize the artifacts and activities of a software project, with their attributes and inter-relationships. Another example is the SODOS system [21], which is intended to "support the definition and manipulation of documents" used in developing software. For this purpose, source code files are considered documents. Features include the management of documentation templates and cross-indexing documents based on their life cycle interrelationships[5]. There is a query-by-example retrieval mechanism which combines the use of relations such as *"implemented-by"* with keyword matching.

ALMA [48] is a generic (or meta-level) kernel that can be used to implement customized database support for a variety of software engineering environments. It comprises an entity-relationship meta-model that can be instantiated to produce a schema for a specific project database. It has facilities to generate various tools for manipulating lifecycle objects, such as syntax-directed editors, updaters, reporters etc. It includes version management facilities. ALMA can thus be viewed as an "applications generator" type tool that can be used to generate a range of different kinds of PMDB's, suited to the needs of particular projects.

None of the existing project management systems, however, attempt to capture knowledge about the application domain of the system, or how the needs of the application domain are manifested in the architecture. The type of information captured in PMDB's is more or less independent of the exact nature of the system under development. They have been used to provide facilities to track the various activities, resources, and artifacts of the software development process; they usually do not provide a description of the structure of the system itself, which programmers can use to combat invisibility. Also, the query mechanisms used in these systems are usually quite simple; they do not include any inferential ability. We argue the importance of inference later in the paper (Section 6), after describing how LaSSIE works.

4.3 Reuse Librarians

Reuse librarians retrieve software components for possible reuse. These systems usually comprise a library of components (object modules, source modules, specification documents, etc.) and a query/retrieval mechanism. A variety of libraries and retrieval methods are available. The retrieved components can either be directly reused, or adapted as necessary. The retrieval mechanisms in these systems are usually one of three types: *keyword*, *faceted-index*, or *semantic-net*. CATALOG [20] is an example of the keyword-based approach typical of standard information retrieval systems. Each library item has an associated set of keywords; the retrieval mechanism takes a set of keywords that the user specifies and matches it with the stored keywords in various ways, and retrieves a matching set. Depending on the choice of keywords used in storage and retrieval, keyword systems may provide adequate performance. However, the *semantics* of the keywords used in retrieval is unavailable to either the storage or retrieval algorithms of such a system. Because of this, they can neither organize the components in a way that facilitates query reformulation or browsing, nor can they in any way infer the "meaning" of the special set of keywords used in the query. In Definity 75/85, for example, we found that the words "connect", "cut-through", and "terminate" are all used to mean the same thing; more interestingly, combinations of keywords can mean something substantially different in different contexts. For example, the meaning of "termination" is quite different in "process termination" (stop a running process) than it is in "call termination" (connect a call to the designated extension). Given the size of software systems, and the variety of naming styles, it is important for a library of reusable components of a large system to provide *semantic retrieval*, *i.e.*, retrieval

[5] Thus, a feature description in a requirements document might be tied by an *implemented-by* relation to a source code module.

based on meaning.

The classification and retrieval library scheme suggested by Prieto-Diaz and Freeman [37] involves the construction of a domain model and its subsequent use in query formulation/reformulation. In their paper, Prieto-Diaz and Freeman developed a taxonomic domain model for the set of data operations embodied in a library of software components, categorized along different *facets*. For example, the facet *"Function"* can have values such as *add, append, create, and evaluate*, and the facet *"Object"* values such as *array, expression, and file*. The library is queried by specifying facets and values; it is more amenable to a "query–modify" retrieval cycle than a pure keyword description. Once designed, however, the classification scheme is static and fixed. Additionally, the representation scheme used here is rather weak; one cannot express constraints (*e.g.*, "password files can *only* be changed by a process with root privileges.") in this faceted language.

The abstract data-type (ADT) library proposed by Embley and Woodfield [18] uses aspects of faceted indexing as well as keywords—they propose a software library consisting of a collection of general purpose ADT's, each with a special descriptor. The descriptor includes facets such as *Domain* and *Operations*; it lists several aliases, and descriptive keywords. The user can define explicit relationships between ADT's in the associated descriptors; certain relationships such as *close-to*, *depends-on*, or *generalizes* can also be derived automatically from the values of their facets, and the given keywords. Unfortunately, these derivations are not *inferences*, in the sense that they are not formally defined, or based on semantics.

The AIRS [30] system, and other semantic-net based systems such as that of Woods and Somerville [50], and the RLF [41] work at UNISYS, all provide some version of structural representation of knowledge. In the case of AIRS and RLF, the representational framework is based on KL-ONE [10]; the RLF work is based on a similar language called KNET.

All of the systems described in this section are intended to be general purpose software librarians; they are not specifically intended to promote reusability within the framework of a large system; as such, they do not address the issue of invisibility. In addition, none of these systems support a *classification*-style inference. As we shall argue in Section 6, classification is helpful in dealing with invisibility[6].

An important, often-ignored, aspect of reuse libraries is that the index of reusable components of a large software system must reflect the programmers' view of the domain of application of the library. As Curtis argues:

> "The effective use of a reusable library will require an indexing scheme similar to the knowledge structures possessed by most programmers working in an application area". [13]

Thus, a library of reusable components within a large system needs an *intelligent indexing scheme* that includes *specific* knowledge about the architecture within which these components are embedded. In addition, Bellin [6] suggests that the description of each component should include constraints on how it should be reused; without this information, the reuser is liable to violate the architecture and contribute to increased complexity of the system. In all of the systems mentioned above, with the exception of [41], the languages can be used to specify mere associations; they are not intended to specify constraints. For example, in these schemes, one cannot specify that update operations must be done *only* by the database manager, or that password files can *only* be changed by a process with root privileges. These constraints are needed to specify architectural principles, as discussed in Section 2.1.

The knowledge-based approach of the LaSSIE system is intended to meet these various needs; we capture, in a knowledge base, descriptions of reusable objects, along with the architectural constraints embodied in the design of a sizable body of software controlling the AT&T Definity 75/85; then, we use inference to answer the programmer's queries about reusable components. In the following sections, we discuss how the LaSSIE system approaches the invisibility problem.

5 An Attack on Invisibility – LaSSIE

The LaSSIE system is an attempt to build a software information system that integrates *architectural, conceptual and code views* of a large software system into a knowledge base (KB) for use by developers. The KB is built using a classification-based knowledge representation language, KANDOR [32], to provide *semantic retrieval.* Besides serving as a repository of information about the system, the KB serves as an *intelligent index* for reusable components. LaSSIE also provides a user interface with a graphical browser and a natural language query processing system.

In this section, we first describe the knowledge base, beginning with the rationale for its design; we then describe the classification-based query processing done by KANDOR, and the reformulation/browsing facility; we then discuss the importance of frame-based classification systems in attacking the invisibility problem.

The first task we have to undertake is to find a body of knowledge that embodies the architecture and function of a target software system. Since this is a task of formalizing knowledge about a pre-existing system, we call this process *reverse knowledge engineering.* After the knowledge is identified, we would then try to embed this knowledge in a suitable formal knowledge representation language.

[6]We defer this discussion until after we have described our classification-based knowledge representation technique and its application in LaSSIE.

5.1 What Goes in the Knowledge Base

The first task in building any knowledge base, of course, is to determine the form of knowledge to be represented—one has to identify the *ontology* of the domain. In other words, what will the KB that is produced by reverse knowledge engineering talk about? In the LaSSIE system, we have chosen an ontology that focuses on the *actions* and *objects* of the Definity 75/85 architecture. Our choice was influenced by three different factors: existing work in the field of Domain Analysis, empirical studies of programming knowledge, and vocabulary used by programmers in Definity 75/85[7].

Reverse knowledge engineering is closely related to the notion of Domain Analysis [2; 36; 29]. There are some differences: the domain analyst looks at a *variety* of systems that service the same application domain and produces an *external* description of a set of reusable components, which can be used to build applications for that domain; the reverse knowledge engineer, on the other hand, looks at the architecture of a *specific* large system and produces an *internal* description of the components of the system, and the architectural framework that unites them[8]. Despite the differences, the result of both kinds of analyses includes a *description of a set of components*. Thus the terminology used by domain analysts to describe the results of their efforts is pertinent to a reverse knowledge engineering effort. Both Neighbours [29] and Prieto-Diaz [36] suggest a model based on *objects* and *actions*.

Our KB is concerned with storing knowledge about a large software system, and using this knowledge as an index into a library of reusable components. As Curtis [13] suggests, *the way programmers think about the system* is an important determinant of the structure of the KB. Shneiderman [40] has conducted empirical studies of programmers, and developed a *syntactic/semantic* model of programming knowledge. The syntactic part is mainly the structural details of the programming language itself. For the semantic part, he proposes a cognitive model based on the *action-object* paradigm. This model has also been suggested by educators responsible for training new developers on large software projects within AT&T Bell Laboratories. They have suggested that a common, unified view of the architecture should be developed and taught to all the developers in the project. Thus, a KB describing the architecture in these terms could serve as a common baseline of information for all developers; such a view has also been advocated by Fischer and Schneider [19].

Descriptions that programmers generate and use also reflect this model. As a normal coding practice, many source files in Definity 75/85 include a short one-line description of the function implemented in that source file. Here are some sample comments:

- This function terminates a call to a user.
- Conference the held user to the current call.
- Update the attendant's status light for the night service feature.
- Cut-through a station user to a call because of answer, originate, or unhold.

Most of these describe a simple *action*, sometimes *caused* by other actions, with *operands*, *actors*, and *contexts*. In the examples above, the contexts simply amount to the customer feature that this action supports or plays a part in. In the following section, we describe the representational framework used to build a knowledge base based on the view we just discussed, and then we describe the LaSSIE KB itself.

5.2 KANDOR, the Knowledge Representation Language

Any knowledge-based system needs a language in which the knowledge base is to be expressed. Since the system's knowledge is to be made explicit and available for inspection by the programmer, as well as used for computing inferences, the *knowledge representation language* must be precise and formal—any language with the same type of precision afforded by classical predicate logic is a potential candidate. It is important that the language be expressive enough to allow us to say exactly what needs to be said about a large software system. It is equally important that the computation of inference with the language be tractable, and thus the language will need to be restricted in some way, so as to avoid theorem-proving that takes exponential or infinite time.

There are a number of currently popular styles of knowledge representation language. Standard logical languages are good in situations in which knowledge is incomplete, and in which one needs to express mainly factual assertions. Production Systems are useful when the knowledge is heuristic and can best be expressed in a condition/action style (*e.g.*, in medical diagnosis). *Frame systems* are best when we want to describe sets of objects with complex relational structure, and when the domain exhibits strongly hierarchical, taxonomic categories of objects.

Given our emphasis on retrieving descriptions of parts of a large software system, and the centrality of the object/action paradigm, we have chosen a frame-based representation for the LaSSIE knowledge base. In this style of representation, *frames* represent classes of objects; they do so in an "object-centered" way, by clustering all information about a given object in one place. In KANDOR [32], our language, a frame is considered a complex *description*, which expresses constraints on members of the class that it denotes. A frame's definition is very much like a standard dictionary definition, starting with a set of super-frames (more general classes) and then adding restrictions to those parent frames, thereby creating a more specialized class. Note that frames formed

[7] The discussion on ontology is abbreviated here; a longer discussion can be found in [9].

[8] The precise role played by the domain model in the knowledge base is described later, in Section 5.4.

in this way induce a *taxonomy* of classes, with the most general class THING at the "top", and the most specific classes at the "bottom".

The restrictions in frame definitions are usually specified in terms of *slots*, which are two-place relations that describe the attributes of class members. Restrictions can be formed by limiting the type of slot-filler to be expected (*e.g.*, "a directory *all* of whose files are header-files") or by specifying the maximum and minimum number of fillers expected (*e.g.*, "a function with *exactly two* arguments"). In order to be considered a member of the class defined by a frame, an individual object must provably be describable by all of the super-frames and must provably satisfy each of the restrictions.

In KANDOR, *individuals* are the objects that represent individual items in the domain. Individuals have their own identities, regardless of their descriptions, but, as implied above, an individual will be described by a set of frames. In addition to the general description of an object by its set of parent frames, the representation of an individual is completed by expressing the relationships between it and other individuals (this is done using instantiated versions of the slots). Thus, a particular file might be considered to be describable by the frames C-FILE, SOURCE-FILE, and CALL-PROCESSING-FILE, and might have the name slot filled in by Capro.c, and the directory slot filled in by /Usr/Callpro. The latter two items themselves would be KANDOR individuals with their own descriptions.

Frame systems like KANDOR perform two key inferences that allow them to be used in retrieval applications like LaSSIE. First, and most commonly, there is *inheritance of properties*, which allows a sub-frame to "inherit" all properties of its super-frames (this is a simple consequence of the sub/super-frame relationship representing a universally quantified conditional). Typically inheritance is considered to contribute efficiency of representation, since one need only represent in a single place a general property that applies to many classes. The second key inference, performed by only a small number of systems like KANDOR, which interpret frames as descriptions (rather than as sentences), is *classification*, in which, for any individual, all descriptions that can provably apply to that individual are found. If KANDOR descriptions can be interpreted as representing necessary and sufficient conditions for class membership, then inheritance can be considered to use the necessary conditions in a "forward" direction, and classification can be considered to use the sufficient conditions in a "backward" direction. Classification can also be applied to frames, wherein all appropriate super- and sub-frames for a frame are determined automatically.

5.3 The Knowledge Base – Actions, Objects, and Doers

Figure 1 shows the root of the LaSSIE taxonomy (THING) and the two levels immediately below it. The four principal object types of concern in our domain are ACTION,

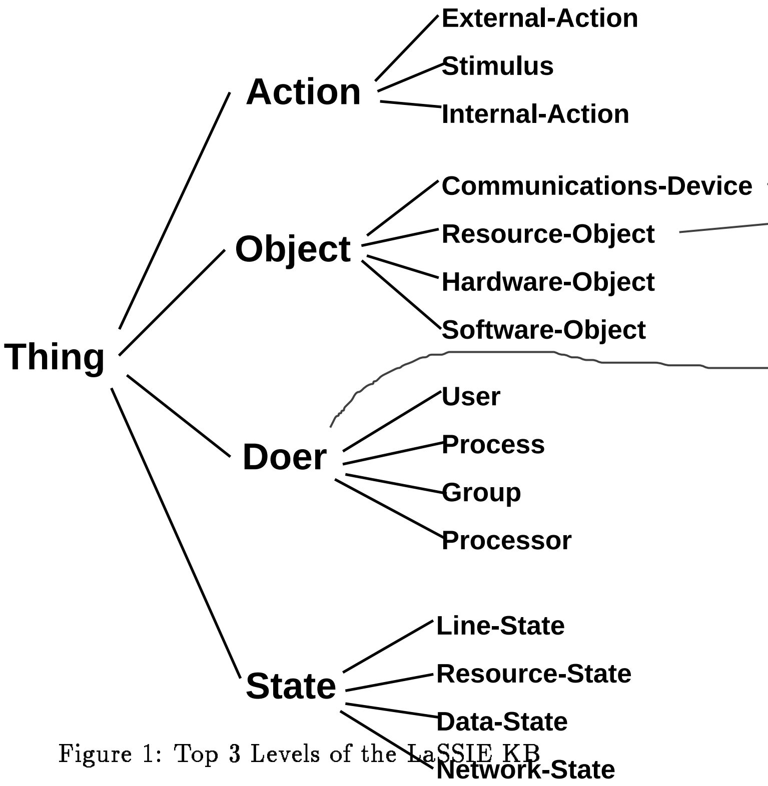

Figure 1: Top 3 Levels of the LaSSIE KB

OBJECT, DOER, and STATE. The edges of the taxonomy have their usual "IS-A" interpretation[9]. DOER represents those THINGs 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 on or by the system. The relationship between the two components is captured by various slot-filler relationships between ACTIONs, OBJECTs and DOERs. Each action description combines the top level concepts in various ways using KANDOR descriptions. Here is a typical one:

```
1(define concept USER-CONNECT-ACTION
2    NETWORK-ACTION CALL-CONTROL-ACTION
3    defined
4    (exists has-actor (generic PROCESS))
5    (exists has-agent (value Bus-Controller))
6    (all has-operand (generic USER))
7    (exists has-environment (generic CALL-STATE))
8    (exists has-result (value Talking-State)))
```

In this example, 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. For example, (*exists* has-agent (*value* Bus-Controller)) says that the bus controller process is always an agent for every USER-CONNECT-ACTION. Among the other items, those in all capitals are *frames*, which represent general concepts 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 lower case are *roles*, which represent relationships between individu-

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

als. Thus, the interpretation of this frame is as follows: USER-CONNECT-ACTION [Line 1] is by definition [3] a network action [2] and a call-control-action [2] that is done by a process [4], using the bus-controller [5] on a user [6], and which takes the user from some call state [7] to the talking state [8]. LaSSIE's KB contains 102 action concept descriptions of this type, which are classified into a conceptual hierarchy. Farther down in the hierarchy, the action concepts become very specific.

The most specific action types, which each correspond to a particular function/source file,[10] are coded as individuals. For example,

```
1(define 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 /Usr/Pgs/Gp/Tgpall/Profum.c)
10   (calls-function Signal-Add-Error-Action)
11   (accesses-variable *Call-Record*))
```

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]; it is implemented by the source file /Usr/Pgs/Gp/Tgpall/Profum.c [9], calls the function Signal-Add-Error- Action [10], and uses the global variable *Call-Record* [11]. It should also 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.

Concepts in the KB, such as the above examples and USER, RESOURCE-OBJECT, GROUP, etc., are concepts that are specific to Definity 75/85 and have a well-defined meaning within the architecture. They are referred to repeatedly in the design and specification documents. We show further details of the portion of the LaSSIE KB under the OBJECT concept in Figure 2. Notice that some concepts appear in **boldface**. These indicate that a concept has more than one parent in the taxonomy (*i.e.*, the taxonomy is a tangled hierarchy); For instance, PROCESS is both a RESOURCE-OBJECT and a SOFTWARE-OBJECT.

The operations that can be performed on these objects are also well-defined: they constitute some of the reusable primitives of the architecture. The descriptions of these operations, the way in which they are implemented in the system, and the conditions under which they are performed constitute some of the *architectural principles* that promote simplicity and understandability of the system. The *domain model* for Definity 75/85 plays a crucial role in structuring this knowledge.

[10] Definity 75/85 has only one function per source file.

Figure 2: The LaSSIE taxonomy under OBJECT

5.4 The Role of the Domain Model

As intimated above, Definity 75/85 and other large software systems are often designed in a layered virtual machine style, each layer providing reusable primitives for the layers above. In Definity 75/85, the key ingredients of these layers are communicating independent processes; the reusable primitives are the functions that can be performed when messages are sent to these processes. Examples include "Connect an ISDN trunk to this call," "Translate the dialled digits," and "Display a number on the Attendant's console." These key primitive building blocks are combined and sequenced in various ways to implement the customer features; the descriptions of these operations, with their operands and precise effects constitute the "domain model" (in the spirit of Prieto-Diaz [36]) of the PBX switching features supported by the Definity 75/85.

Though there was no explicit discussion of the domain model anywhere in the documentation, a detailed study of the architecture indicates that the designers carefully considered the feature set and isolated a set of primitives that could be used to implement them; the architecture is designed to implement these primitives in a reusable manner. The architecture also includes some constraints (strict layering, deputation of certain tasks to certain processes, etc.) that govern the manner in which the primitives in the domain model are embedded in the architecture. Definity 75/85 and related products are typically described at the level of the process/message architecture, or at the level of the customer services performed by the switch[11]. The domain model is only implicit in the system, and while it may constitute an important part of the programmer's understanding of the system,

[11] For example, "plain old telephone service" (POTS), call forwarding, call waiting, conference calls—these services are usually referred to as "features".

it is left for each programmer to divine in his/her own way.

It is our view that in an integrated Software Information System, this key level of description should be made explicit, as should its connections to the features built from its primitives and its connections to the architecture that implements it. Thus, in LaSSIE, the domain model serves a key integrative function. It can also help the programmer access the rest of the knowledge base, since it maps most closely onto the common conception of the overall function being performed by the switch. Queries about various functions of the switch—especially by programmers newer to the project—will naturally be formed in terms of actions like connecting two trunks rather than in terms of certain messages received by the "connection manager" (these latter types of queries will also need to be answered, of course). In LaSSIE a concept like `TRUNK-CONNECT-ACTION` denotes a domain model primitive, *i.e.*, a common, important operation that can be reused in different features. In the Definity 75/85 architecture, it is executed by the `Timeslot-Controller`, which is a process in the `Network-Layer`. The definition of this concept will include all this information, as well as a pointer to the code object (subroutine or function) implementing this operation, and a list of customer features that use this operation.

The domain model and its relationship to the architecture of the system is built manually into the LaSSIE knowledge base. In addition to this, we also capture the *code view* of the system, which is, happily, mostly automatically derived, using CScope [46], as described below.

5.5 Integrating Code-level Information

We enhanced the conceptual action-object-actor level of representation with certain simple but important code-level relationships such as function-*calls*-function, sourcefile-*includes*-headerfile, etc., which are syntactically derived by scanning the code. Thus questions such as "what functions call 'apost' and include 'errproc.h'?" can be processed.

To accomplish this, we developed a general representation of code objects and their inter-relationships and added them, codified in KANDOR, into the knowledge base. The instances of these concepts were derived as follows: first the source files were scanned with CScope and several relations generated. Then these relations were organized around specific instances of functions and source files. These collections were translated directly into KANDOR function calls to add the appropriate information to existing information from the manually generated KB concepts. Thus the syntactically derived code-level information is smoothly incorporated with the conceptual and architectural views, so one can pose queries such as "What global variables are accessed by a function that flashes a display lamp at an attendant's console?"

A frame-based description of a function can combine the architectural/domain aspects as well as its code-level relationships into one unit that can be manipulated by the classification algorithm. This facility for combining different kinds of information about an object provides a way to integrate the code, conceptual and architectural views of the system.

5.6 Querying and Browsing

As explained above, most of the LaSSIE KB describes actions; the paradigmatic usage is a query from a user that describes an operation that she wants to see examples of, to understand in more detail, or to reuse; the LaSSIE system will retrieve instances. LaSSIE also has an interactive graphical user interface that includes a modified version of the ARGON [33] system, and the ISI-Grapher [38]. In this section, we present examples that illustrate how the retrieval system works. The core of the retrieval system is the classification algorithm; it simplifies the task of querying a large knowledge base.

A LaSSIE query[12] is simply a description of an action, such as the following:

CONNECT-ACTION
has-actor `DOER`
has-operand `OBJECT`
has-cause `ACTION`
  has-actor `USER`

This query will retrieve all individual `CONNECT-ACTION`s performed by a `DOER` on an `OBJECT`, because of an `ACTION` by a `USER`. The query processing is carried out in two stages. First the query is placed in the LaSSIE taxonomy by the classification algorithm, using the description in the query and descriptions of the frames in the taxonomy; then, the matching instances are the instances of those frames that are subsumed by the classified query. The user who generated the query does not have to know that `PROCESS`Ses, `USER`s and `TRUNK`s are `DOER`s (see Figure 1); neither does she need to know how many processes there are, or what their names are. The classification algorithm takes care of all of that. Thus, it is possible for a user to retrieve answers to a query without knowing the structure of the taxonomy of a KB, or the individuals therein, using the *semantic* retrieval provided by the classification algorithm. For example, the following individual would be retrieved by the above query:

```
(define individual Attd-Merge-Call-Action
        (CALL-MERGE-ACTION ATTD-CAUSED-ACTION)
        (has-actor Attd-Monitor-Process)
        (has-agent System-Fabric-Manager-Process)
        (has-operand Generic-Call)
        (has-recipient Generic-Call)
        (has-environment Generic-Call-State)
```

[12]The actual query language in ARGON [33], the system that LaSSIE is based on, includes negation, disjunction, etc; we illustrate just a simple query here.

```
8       (has-result Talking-State)
9       (has-cause Attd-Button-Push)
10      (implemented-by /Src/Ugp/Attd/Atd-Au-Im.c)
11      (calls-function Signal-Error-Action)
12      (accesses-variable *Call-Record*))
```

The advantages of semantic retrieval are illustrated in this example[13]. Though the action is called a MERGE-ACTION, the classifier recognizes it as a CONNECT-ACTION based on the descriptions of each. Additionally, from the description of Attd-Button-Push (not shown here) it realizes that this is an ACTION by an ATTENDANT, which is defined to be a specialization of USER.

The ARGON interface displays all the retrieved individuals, and the user can select any one of them for detailed display, with all its slots and fillers; each of these fillers and slots can, in turn, be selected, and further information about them is available. Thus, the user can find out that this action is performed by the Attd-Monitor-Process (selecting that will retrieve its description, which tells the user that this process is in the Service-Layer of the architecture), and that it uses the System-Fabric-Manager-Process as an agent. Furthermore, if one selects the concept ATTD-CAUSED-ACTION (which denotes any action by the system caused by an action by an attendant), one would find that it is always performed by the Attd-Monitor-Process, *i.e.,* this process, and only this process, responds to actions by the attendant.[14]

Thus classification makes it possible that *a merge action caused by a button push by an attendant* gets retrieved by a query that asks for *a connect action caused by an action by a user*; furthermore, the retrieved individual carries with it a wealth of information about the architectural, conceptual, and code aspects of the operation. So, retrieval based on the semantics, along with integration of multiple views helps in the discovery of useful information about Definity 75/85.

In the event that the query retrieves too many, or too few relevant instances, the user can reformulate it, taking advantage of the information in the taxonomy. For example, assume the query above retrieved too many instances—there may be too many DOERs doing too many things to too many OBJECTs. The user can specialize the above query by replacing either OBJECT or DOER with their children in the taxonomy, either by using a menu choice, or by looking it up in the graph. This is the advantage gained by having an *intelligent index.* Thus, if there are no (or an insufficient number of) answers, the user can generalize parts of the query.

In addition to its graphical interface, LaSSIE also has a natural language query interface based on TELI [4], which we now describe.

5.7 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 [4]. 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 compatibility tuples as described above. For example, consider the frame definition[15] shown below, with its associated verb *connect*:

```
(verbframe CALL-CONNECT-TRUNK-ACTION
   (connect) (ACTION)
   (exists has-operand (generic TRUNK))
   (exists has-recipient (generic CALL))
   (exists has-actor
          (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 was automatic. The resulting compatibility

[13] The retrieved individual is an instance of a CALL-MERGE-ACTION [2] done by the Attd-Monitor process [3], using the System-Fabric-Manager [4] process, which connects one CALL [5] to another [6], leaving both calls in a Talking-State; this action is caused by the attendant pushing a button [9], it is implemented by the source file listed in line 10; it calls the function Signal-Error-Action and uses the global variable *Call-Record*

[14] This would be specified by the inclusion of the restriction (*all* has-actor (*value* Attd-Monitor-Process)) in the concept definition of ATTD-CAUSED-ACTION.

[15] Actually, the form shown generates a table entry for the action, associating it with a verb name, and generates a standard function call to define a KANDOR frame.

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 bus 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); this query can also be directly edited if reformulation is necessary.

For example, TELI's output for the above query is:

```
(set A1 (ACTION A1)
      ((ACTION BY AGENT) A1 Bus-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 **Bus-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 example, the query *What actions by a process reserve a touch tone recognizer because of a pickup by a user?*, would be translated to

ALLOCATE-ACTION
    has-actor `PROCESS`
    has-operand `TOUCH-TONE-RECOGNIZER`
    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 `ACTION`s respectively are specified by using the `HAS-ACTOR` and `HAS-CAUSE` slots.

6 Why Frame-Based KB's with Classification are Important

LaSSIE accrues several key advantages by using a frame system like KANDOR, which incorporates classification as its foundation. These include the following:

- *Aggregation of information about individuals*—The object-centered approach of a frame language allows all information about an individual to be concentrated in one place. This allows us to integrate information obtained from different sources and different points of view into one description.
- *Semantic retrieval*—Classification helps during query processing by reducing the amount of information that the user has to know about the knowledge base. In purely syntactic information systems, where the terms used have no descriptions (*e.g.,* relational databases or keyword systems), the user has to know the exact terms that are used in the database. Classification reduces the number of terms that the user needs to know. For example, when searching for a general category of items (*e.g.*, `FILE`), all subcategories (as expressed in the generalization hierarchy induced by classification of frames; *e.g.*, `HEADER-FILE`) can also be retrieved. This point has been made by Beck, Gala and Navathe (and others):

  > "... the user can describe a query in terms which may be different from the exact terms under which the desired information is stored, as long as the meaning is similar". [5]

  Essentially, classification is a logically defined operation that allows *semantic retrieval* in a simple yet principled manner.
- *Use of classification and inheritance to support updates*—When new information is added to the knowledge base, the system automatically classifies the new items, thereby giving the developer an integrity check. Since all categories implied by the new description are found, the developer can see if there were any omissions or accidental inferences to be made. Inheritance also makes the addition of new information easier, since many of the properties of a new item can be derived from its membership in existing classes.
- *Use of the KB as an index*—In a way, the frame knowledge base acts as a general schema over the particular code data stored in the system. The general knowledge encoded in the frames can be used to guide the user in his/her search for particular items of interest. LaSSIE has several means for using the frame knowledge base in this way (including a graphical display of the frame hierarchy).

We can now compare our work to the other semantic-net reuse library systems listed in Section 4.3. RLF does not have a built-in classifier; the KNET language is used to describe the components, and the retrieval/browsing algorithm is left to the application developer. The AIRS [30] system uses a heuristic retrieval algorithm based on a numerical "conceptual-distance" measure, which the user has to specify. Woods and Somerville use conceptual dependency (CD) diagrams. Their query mechanism is based on a set of *verbs*. They use word associations specified by the user to identify the conceptual dependency graph for a given verb. The user is then prompted for further information based on the selected conceptual dependency graph to further narrow the search for components. For example, the verb *send* might be associated with the CD graph named *communicate*. Then the structure of the *communicate* concept would be displayed on the terminal, and user would be prompted to fill in keywords for objects associated with this structure; these would then be matched with stored descriptions to find close matches. The algorithm is implemented in Prolog; the retrieved components are listed in order of closest match, depending on how many keywords matched.

None of these systems has a well-defined notion of classification; this makes them ill-suited for our application, as we now explain. Building a KB of the size required for LaSSIE (about 200 frame descriptions) involves constructing a large number of KANDOR descriptions (several example concept descriptions can be found in Section 5.3). Once a description is input to KANDOR, the classification algorithm kicks in and "places" the concept in the proper relation to other descriptions already in the KB. Without classification, the knowledge engineer would have to figure out the proper placement by hand—a daunting task with such a large knowledge base. Thus, classification plays a crucial role in assembling the taxonomic KB that provides intelligent indexing, as called for by Curtis [13]. Constructing a very large KB in the KL-ONE style language used in AIRS and RLF would be easier with a classifier. Unfortunately, because of the power of KL-ONE [39], a complete classification algorithm is impossible. The CD diagrams used in Woods and Somerville do not have a taxonomic organization; the matching is done by keyword associations. Thus, in this case, besides constructing the CD diagrams, it is also necessary for the knowledge engineer to specify keyword associations. Likewise, with AIRS, in addition to describing the components in a KL-ONE like language, the knowledge engineer has to specify a numerical measure of "conceptual distance". The simplicity of the KANDOR language, and the built-in classifier, simplifies the knowledge engineer's task, while still providing semantic retrieval as well as a rich knowledge structure for browsing, navigation, and query reformulation.

7 Limitations of LaSSIE

LaSSIE is mainly intended to process queries about actions. Its KB consists of a large collection of action descriptions in KANDOR, and the user can query this collection by specifying an action description that she is interested in; the classification algorithm retrieves the instances thereof.

The strengths of LaSSIE are discussed in the previous section—the essential limitations of LaSSIE are those that are caused by the limitations of KANDOR. These limitations can be divided into two classes.

The first class of limitations is rooted in the fact that KANDOR is a domain-independent language, not specifically designed to represent knowledge about real-time software in terms of objects and actions. So there are various aspects of each that cannot be expressed adequately within its representational framework. With respect to `ACTION`s, one must express all aspects using the same undifferentiated slot mechanism. This means that special inferences based on particular slots such as `has-cause`[16], `has-result` and `has-environment`[17], or `has-service`[18] cannot be performed. With respect to `OBJECT`s, it would be useful to represent part-of hierarchies. For example, a light is a `part-of` a button which in turn is a `part-of` a telephone. KANDOR does not support reasoning based on meronymic (part-of) hierarchies. Meronymic reasoning is also important for dealing with plan-like groups of `ACTION`s. We elaborate on plan-based reasoning in the following section.

The second class of limitations are due to certain expressive limitations made in KANDOR to make the classification algorithm faster and easier to implement. For example, one cannot specify a relationship between the fillers of different slots in a concept. Thus, in the concept describing the call waiting feature, one might want to specify that the person being called (and currently off-hook) is the same person who receives the call-waiting signal. KANDOR is too weak to specify this precisely.

8 Further Work

8.1 Planning Knowledge

The action-based representation used in LaSSIE has one crucial failure; the *contexts* within which operations are performed are not well established. Thus, the following kinds of questions cannot be answered:

- Why is this action being performed?
- How can this operation be performed?
- Can this operation be done in that context?
- Is this operation involved in more than one customer feature?

Anyone familiar with the planning literature in AI will recognize that all of these questions relate to plan-like information. There is a considerable body of literature linking plan knowledge to the cognitive processes involved in software design/development. Soloway, *et al.*, [44; 43; 42] have established the importance of plans in programming knowledge; Letovsky [26; 25] has shown that plans are important for program comprehension and

[16] *e.g.*, transitive closure over causality

[17] *e.g.*, relating the `result` post-condition of one action to the `environment` pre-condition of another

[18] Establishing the context or rationale for an action

inspection. To answer "Why" questions[19], programmers give answers that involve describing a plan that contains the operation; "How" questions[20] are answered by describing a plan that satisfies the goals of the given operation.

How can a LaSSIE-like KB be enhanced by including plan knowledge? With more information about the context of an action within a plan that implements either a higher-level goal, or a customer feature, more questions could be answered. Thus, one can imagine a KB that contains a large number of plans, describing various operations, features, etc., of Definity 75/85. These plans would each be composed of a series of actions. Inference algorithms would run over these plan data structures that would answer questions of the sort posed at the beginning of this section; others may assist a user to insert a new plan into the KB and properly classify it in relation to previously stored plans. Knowledge about the software *process,* that is, information about the tools and methods used in development, which is not currently available in LaSSIE, could also be stored as plan knowledge (see Huff [22] for a plan-based approach to software process representation).

We have built a plan classification system [15] to experiment with various relationships that can be defined between plans, and inference algorithms that can derive these relationships.

8.2 Knowledge Acquisition

We have discussed how the invisibility problem can be addressed by properly storing and retrieving knowledge of various kinds about the architecture of a system. In the current LaSSIE system, the concepts describing the actions and objects in the domain were generated by reading and understanding the architecture documents, and the comments in the source files. This is an entirely manual process. Though the end result of this effort is a knowledge base that can be queried by developers in a manner that has an impact on the visibility of a system, building this KB is an intensive task, the automation of which would be a difficult but worthy goal.

Various researchers [7; 49; 24] have addressed the task of scanning code automatically and deriving various kinds of information about the function of the code. Letovsky [24] and Wills [49] are concerned with recognizing pieces of code that correspond to simple algorithmic fragments, such as loops, accumulations, etc. Biggerstaff [7] attacks a different problem; he seeks to mimic the process by which a programmer experienced in writing, say, device drivers, might scan the code for a new device driver (previously unknown to him/her) and use various clues such as typical variable names, characteristic data structures, etc., to explain the function of the new device driver.

Even if this problem remains unsolved, we believe that the amelioration of the invisibility problem, the consequent reduction in architectural erosion, and the possible increase in code reuse should offset the cost of building the KB by hand.

9 Summary

We have argued that the problem of invisibility, and the related problem of complexity are caused by information barriers. We briefly discussed the limitations of existing attempts to solve these problems. We then presented a system called LaSSIE, which uses explicit knowledge representation and reasoning to address these limitations. The LaSSIE system is unique in its ability to provide *semantic retrieval* from an information base that *integrates multiple views.* Additionally, LaSSIE's KB provides an *intelligent index* (over a library of reusable components) that is attuned to the programmer's view of a large system. This approach shows promise, and also shows up some inherent limitations in the classification approach. The work is continuing; we are now focusing on both the representation and the knowledge acquisition issues.

Acknowledgements

The LaSSIE work is a multi-person effort, involving (besides the authors) Diane Litman, Jim Piccarello, and Bob Kayel. Several discussions with Bill Frakes in the initial stages of this project helped clarify the problem. This paper has benefited greatly by helpful comments from Alex Borgida, Dewayne Perry, Pamela Zave, Ed Pednault, and anonymous referees.

References

[1] Adelson, B. and Soloway, E.. The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, 11(11), 1985.

[2] Arango, G., Domain Analysis: From Art Form to Engineering Discipline, *Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, May, 1989.

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

[4] 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.

[5] Beck, H.W., Gala, S. K., Navathe, S. B., Classification as a Query Processing Technique in the CANDIDE Semantic Data Model *Fifth International Conference on Data Engineering*, Los Angeles, CA, 1989.

[6] Bellin, S, Personal Communication.

[19] *e.g.,* Why is a *busy-wait-alert* operation being performed here?

[20] *e.g.,* How is a *busy-wait-alert* operation implemented?

[7] Biggerstaff, T.J., Design Recovery for Maintenance and Reuse, MCC Technical Report STP-378-88, Austin TX, 1988

[8] Borgida, A., Brachman, R. J., Alperin Resnick, L. McGuinness, D. CLASSIC: A Structural Data Model for Objects. *Proc. ACM SIGMOD-89*, Portland, OR, May-June, 1989.

[9] Brachman, R., and Devanbu, P., Domain Modeling in a Software Information System, *OOPSLA Workshop on Domain Modeling in Software Engineering*, 1989, New Orleans, LA.

[10] Brachman, R.J., and Schmolze, J. G., An overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9:171-216, 1985.

[11] Brooks, F. P., No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer Magazine*, April, 1987.

[12] Chen, Y.-F., M. Nishimoto, and C.V. Ramamoorthy, The C Information Abstraction System, IEEE Transactions on Software Engineering, March, 1990.

[13] Curtis, W., Cognitive Issues in Reusing Software Artifacts, in Vol II, *Software Reusability*, T. J. Biggerstaff and A. J. Perlis, Editors, ACM Press, New York, 1989.

[14] Curtis, W., Krasner, H., and Iscoe, N., A Field Study of the Software Design Process for Large Systems, *Communications of the ACM*, 31(11), November, 1988.

[15] Devanbu, P., and Litman, D. A Plan and Scenario Classification System, AT&T Bell Laboratories Technical Report, 1990.

[16] Devanbu, P., Brachman, R., and Selfridge, P., Inference in Support of Retrieval for Reuse in Large Software Systems, *IEEE/SPS Workshop on Software Reuse*, 1989, Indialantic, FL.

[17] Doyle, J. and Patil, R. S., Language Restrictions, Taxonomic Classification, and the Utility of Representation Services. MIT/LCS/Technical Memo 387, 1989.

[18] Embley, D.W., and Woodfield, S. N., A Knowledge Structure for Re-Using Abstract Data Types, *Proceedings, Ninth Annual Software Engineering Conference*, 1987, Monterey, CA.

[19] Fischer, G., and Schneider, M., Knowledge-Based Communication Processes in Software Engineering, *Proceedings, Seventh International Software Engineering Conference*, Orlando, FL, 1984.

[20] 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.

[21] Horowitz, E., and Williamson, R. SODOS – A Software Documentation Support Environment: Its Use, *Eighth International Conference on Software Engineering*, London, UK, 1985.

[22] Huff, K. E., Software Process Instantiation and the Planning Paradigm, *Proceedings of the Fifth International Software Process Workshop*, Kennebunkport, ME, October 1989.

[23] Johnson, W., and Soloway, E., PROUST: Knowledge-Based Program Understanding, *IEEE Transactions on Software Engineering*, March, 1981.

[24] Letovsky, S., Plan Analysis of Programs, Ph.D. Thesis, Yale University, New Haven, CT, 1988.

[25] Letovsky, S., Cognitive Processes in Program Comprehension in *Proceedings of the Second Workshop on Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Editors, Ablex Publishers, Norwood, NJ, 1986.

[26] Letovsky, S., Pinto, J., Lampert, R., and Soloway, E., A Cognitive analysis of a Code Inspection, in *Proceedings of the Second Workshop on Empirical Studies of Programmers*, Washington, DC, E. Soloway and S. Iyengar, Editors, Ablex Publishers, Norwood, NJ, 1986.

[27] Levesque, H. J., and Brachman, R. J., Expressiveness and Tractability in Knowledge Representation and Reasoning. *Computational Intelligence*, 3:78-93, 1987.

[28] Modica, L. Personal Communication, 1989.

[29] Neighbors, J., Software Construction using Components, Ph.D. Thesis, University of California, Irvine, CA, 1981.

[30] Ostertag, E., and Hendler, J.A. AIRS: An AI-based Ada Reuse System. Technical Report CS-TR 2197, Computer Science Center, University of Maryland, 1987.

[31] Parnas, D.L Designing Software for Ease of Extension and Construction, *IEEE Transactions on Software Engineering*, Vol SE-5, No. 2, March, 1979.

[32] Patel-Schneider, P. F., Small can be Beautiful in Knowledge Representation. In *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, December, 1984, Extended version appears as AI Technical Report No. 37, Schlumberger Palo Alto Research, Palo Alto, CA, October, 1984, pp. 11-16.

[33] 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.

[34] Penedo, Maria H. Prototyping a Project Master Database for Software Engineering Environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software*

Development Environments, Palo Alto, CA, December 1986.

[35] Perry, D. E., Industrial Strength Software Development Environments, *Proceedings of IFIP Congress '89, The 11th World Computer Congress*, August/September, 1989, San Francisco CA.

[36] Prieto-Diaz, R. Domain Analysis for Reusability, *Proceedings IEEE COMPSAC-87*, Tokyo, Japan, October, 1987.

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

[38] Robins, G., The ISI Grapher Manual, *University of Southern California Information Sciences Institute*, Technical Manual ISI/TM-88-197, February, 1988.

[39] Schmidt-Schauss, M., Subsumption in KL-ONE is Undecidable, *Proceedings, First International Conference on Principles of Knowledge Representation and Reasoning*, 1989, Toronto, Canada.

[40] Shneiderman, B., Empirical Studies of Programmers, in in *Proceedings of the Second Workshop on Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Editors, Ablex Publishers, Norwood, NJ, 1986.

[41] Solderitsch, J., Wallnau, K., Thalhamer, J., Constructing Domain-Specific Ada Reuse Libraries, *Proceedings, Seventh Annual National Conference on ADA Technology*, March, 1989, Atlantic City, NJ.

[42] Soloway, E., Learning to Program = Learning to Construct Mechanisms and Explanations, *Communications of the ACM*, 29(9), 1986.

[43] Soloway, E., and Ehrlich, K. Empirical Studies of Programming Knowledge *IEEE Transactions on Software Engineering*, Vol SE-10, No. 5, September, 1984.

[44] Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. Designing Documentation to Compensate for De-localized Plans, *Communications of the ACM*, 31(11), November, 1988.

[45] Smoliar, S.W., and Swartout, W., A Report from the Frontiers of Knowledge Representation, Draft Manuscript, USC/ISI, 1988.

[46] J.L. Steffen, "Interactive examination of a C program with Cscope", Proc. USENIX Assoc. winter Conference, Jan, 1985.

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

[48] Van Lamsveerde, A., Delcourt, B., Delor, E., Schayes, M-C., and Champagne, R., Generic Lifecycle Support in the ALMA environment, *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[49] Wills, L.M., Automated Program Recognition, MIT Technical Report 904, Cambridge, MA, 1987.

[50] Woods, M., and Somerville, I., An Information System for Software Components, *ACM SIGIR Forum*, 22:3, Spring/Summer, 1988.