

# Augmenting Abstract Syntax Trees for Program Understanding

Christopher A. Welty

Vassar College  
Computer Science Dept.  
Poughkeepsie, NY 12604-0462  
*weltyc@cs.vassar.edu*

**Abstract:** Program Understanding efforts by individual maintainers are dominated by a process known as *discovery*, which is characterized by low-level searches through the source code and documentation to obtain information important to the maintenance task. Discovery is complicated by the *delocalization* of information in the source code, and can consume from 40-60% of a maintainer's time. This paper presents an ontology for representing code-level knowledge based on *abstract syntax trees*, that was developed in the context of studying maintenance problems in a small software company. The ontology enables the utilization of automated reasoning to counter delocalization, and thus to speed up discovery.

Subject Areas: Program Understanding, Software Information Systems.

Note: This paper is available at [http://www.cs.vassar.edu/faculty/welty/papers/ase-97\\_1.html](http://www.cs.vassar.edu/faculty/welty/papers/ase-97_1.html) which contains links to some of the background materials.

## 1 Introduction

The software industry is plagued by problems dealing with legacy systems, including re-engineering, maintenance, Y2K, training, etc. Anyone who has spent any time working in or with this industry can not help but be immediately convinced that "upstream" efforts, i.e. research focusing on requirements or development, are not currently significant. The recent near collapse of the CASE industry [Lewis, 1996] is further evidence that this is true.

While this point has been often repeated, the lions share of software engineering research, particularly AI&SE research, continues to be devoted to upstream parts of the software life-cycle. The research community has traditionally turned away from solving real industry problems because they involve too much that is not relevant, or interesting, from a research perspective. In turn, industry

has tended to turn away from research because its application shows little present or near-future value.

A new research paradigm known as *industry as laboratory* [Potts, 1993] is beginning to catch on amongst some software engineering researchers, and was the subject of a recent workshop [Welty and Selfridge, 1997]. One of the keys to research fulfillment under this paradigm is understanding, and making clear, that it is not the same as technology transfer [Henninger, 1997]. This is a subtle and often misunderstood issue that has prevented some researchers from using industry as a laboratory. There is a lot of interesting research that can be based on the real and current problems of the software industry, without having to go through the slings and arrows of creating and supporting a product. Research in program understanding is one example of this, and although the early work pre-dates the *industry as laboratory* term itself, it clearly fits the definition.

This paper reports on a research effort that was motivated by the landmark work on Software Information Systems (SISs) [Devanbu, et al., 1990] [Selfridge, 1990], and our experiences trying to apply this technology to the problems being experienced by a small software company. While we found that the most critical need of the maintainers we studied was domain knowledge, and a clear picture of how the software fit into the domain, that aspect of this research is not discussed here. Instead we focus on the domain-independent framework that was developed, and how it helped maintainers understand the software. The ways in which this framework fits into a solution to the more important domain knowledge problems has been documented somewhat ([Welty, 1995a] and [Welty, 1995b]), and will be the subject of future reports.

The most significant of the domain-independent research that came out of this effort has been the marriage of some

```

void group_deliver (
    MAIL_MESSAGE message,
    GROUP group)
{ LIST members;

  members = get_members(group);
  while (! empty(members)) {
    ind_deliver(message, car(members));
    members = cdr(members);
  }
}

```

of the powerful automated reasoning capabilities of *description logics* [Brachman, et al., 1991] with abstract syntax trees in support of program understanding, yielding a tool for visualizing object-oriented programs. This tool highlights pertinent information, such as identifying side-effects, and facilitates browsing the code in a way that dramatically speeds up the understanding process.

## 2 Background

It has been estimated that \$30 billion is spent annually on software maintenance (\$10 billion in the US), comprising 50% of most data processing budgets and 50-80% of the effort of an estimated one million programmers [Shrobe, 1995]. Despite this, maintenance has not been as closely studied as requirements or design, though it has not been entirely ignored.

### 2.1 Discovering Delocalization

Empirical studies of maintainers have found that they spend at least 40% [Henninger, 1997] and as much as 60% [Selfridge, 1990] of their time repeatedly performing very simple, low level searches through the source code and documentation to find information. This process has been called *discovery*.

Discovery typically takes place as a result of examining a section of source code and asking a question about something in that section. For example, consider the code shown in Figure 1. A maintainer debugging this section of code might ask, “What is the class GROUP?”

While intrinsically a very simple process, discovery takes so much time because of the nature of the source code. Programming languages, while adequate for representing control-flow, tend to *delocalize* other kinds of information [Soloway, et al., 1986]. That is, when examining a piece of code, all that a maintainer can *see* is what is in the current window of the text editor. The likelihood of a maintainer finding the answer to a question decreases with the distance the answer is from the current visual context

[Lampert, et al., 1988]. In other words, maintainers tend to base their understanding on what they can see, what they already know, and what they can *easily ascertain*.

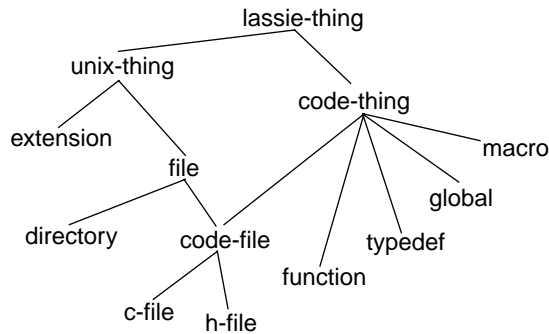
The best known and most closely studied example of this is the *delocalized plan* [Soloway and Letovsky, 1986], in which some abstract goal is achieved by lines of code that are spread out through the source program (probably across different source files). The problem is more than just delocalized plans, however, it’s delocalized information in general. In the C++ method shown in Figure 1, the delocalization of the source code relationships is manifested in numerous ways. Take, for example, the simple question, “What is the class GROUP?” Class definitions are sometimes at the top of a source code file, but more often are located in separate header files. To get the precise answer to this question, the maintainer would have to find the definition. The more effort this search requires, the less likely it is that the maintainer will follow through. If the maintainer does give up before finding the definition, chances are she will proceed trying to understand this method based on whatever information she already knows or can guess, and this incomplete understanding can lead to mistakes [Lampert, et al., 1988].

Furthermore, recent studies have shown that object-oriented languages actually *increase* understanding problems by delocalizing much more information than their imperative predecessors [Huitt and Wilde, 1992]. Inheritance, in particular, spreads method and slot (instance variable) declarations up the class hierarchy, making it harder to find answers to questions about class composition, among other things.

### 2.2 Software Information Systems

Clearly it would be better if maintainers did not carry out understanding tasks with incomplete information. Strict processes and methodologies (that e.g. may require a maintainer not to give up on such a search) don’t hold up under the realities of the software industry. What is needed is a way to *localize* the information that a maintainer requires, and this is precisely the goal of *Software Information Systems* (SISs) [Brachman, et al., 1990], which were developed in order to address the representation problems inherent in programming languages. The goal of an SIS is to provide a representation of the knowledge a maintainer will seek during discovery in a form that makes it easy to find.

The LaSSIE SIS [Devanbu, Selfridge, and Brachman, 1990] used a very simplistic code level ontology, of which the concept taxonomy is shown in Figure 2. This code model was based on the types of objects that could be



**Figure 2.** The LaSSIE code-level taxonomy.

automatically recognized in C language source code by the CIA system [Nishimoto, Chen, and Ramamoorthy, 1990], which was used to populate the knowledge base from a large software artifact. The most significant facility provided by LaSSIE was the localization of information regarding the definitions of the source code objects listed in Figure 2.

Its powerful domain model combined with the efficient inference and expressive query capabilities of the underlying description logic CLASSIC [Brachman, et al., 1991], made LaSSIE an effective tool for assistance during maintenance. We began our research with the goal of studying how the SIS technology might be used in a different environment than that which spawned LaSSIE. Where LaSSIE was the result of studying a huge maintenance group in one of the world’s largest corporations, we chose a maintenance and support group in a small software company.

### 2.3 The Laboratory

The company we chose to study is a small software company consisting of roughly 70 people, with twenty involved in the maintenance and management of a single piece of software, and ten involved in customer support. We have agreed not to mention the name of the company or its primary business focus, as we mention briefly here, and plan to document more specifically elsewhere, many of the internal problems they had. This turns out not to be a problem, since all the prototype work was done on a hypothetical email delivery system [Welty, 1995a], and that will be used in examples where the domain is relevant.

Aside from being small, this company was of interest because they had already made a commitment to use Smalltalk, and to at least try and impose more strict methodologies on their maintenance team. The company was, however, very strapped for resources and their most experienced developers were rarely available. This was creating a snowballing maintenance problem which in turn affected all aspects of their business.

We began our study in the usual ways, interviewing management and staff in groups and one-on-one and observing the maintainers at work.

## 3 Initial Goals and Analysis

We entered into this project expecting to employ SIS technology to help make information that the maintainers needed more accessible. All research in program understanding that we were aware of, however, had resulted from studies of student programmers [Soloway and Ehrlich, 1984] [Quilici, 1994], or maintainers in large companies [Selfridge, 1991] [Ning, et al., 1994], so we first wanted to verify that the same kinds of problems existed in this environment.

### 3.1 Confirmation of Previous Results

We focused our attention on the less experienced members of the maintenance group who did the bulk of the maintenance work (enhancements and testing – most of the debugging work was done during crises by the experts).

We used the contextual inquiry interviewing technique [Holzblatt and Jones, 1993] for studying the maintainers at work. There were some flaws in this from an empirical perspective, since the maintainers admitted they worked much harder while we were present, and frequently followed through on tasks that normally they would have waited to ask someone about. Nevertheless, we quickly confirmed that these maintainers spent close to 60% of their time engaged in discovery.<sup>1</sup>

We were also able to confirm that the delocalization of information in the source code was the biggest factor contributing to the time spent in discovery. For example, the most common activity was “grepping” the source code for all uses of a variable in order to see the places it was changed. The second most common activity was browsing the class hierarchy looking for a method or slot definition to better understand the use of a variable in the code.

### 3.2 Common Discovery Questions

Obtaining precisely the results we expected was most encouraging, and our next step was to ask the maintainers to describe what they were doing in terms of the questions they were finding answers to. Once they became accustomed to thinking about what they were doing that way,

1. It should be noted that this is the percentage of the time spent while we were observing them at their workstations actively working on the software, not the percentage of their total time spent at work. It would have required far more elaborate (and intrusive) monitoring to accurately estimate what percentage of their total time was spent in this activity.

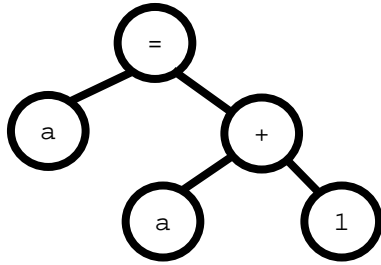


Figure 3. A simple Abstract Syntax Tree

we stopped the direct observations and had them all write down the questions that they were finding answers to during the course of their maintenance activities, and the frequencies of each. Among the most common questions, in order of their reported frequency during the period we studied, were:

1. Where is this variable modified?
2. What are the available slots and methods on this instance?
3. What is the data-type of this variable or function?
4. What are the superclasses of this class?
5. What does this function return?
6. Does this function have side-effects?
7. Is this data-type used?

### 3.3 Representing the Code

The LaSSIE code-model ontology only provides answers to questions regarding the location of definitions, and is clearly too course-grained to cover these more specific questions. In addition, LaSSIE was never intended for use with an object-oriented language. In response to the list above, we sought to develop a finer-grained ontology – an ontology that was capable of capturing all the information right down to the line by line specification of the source

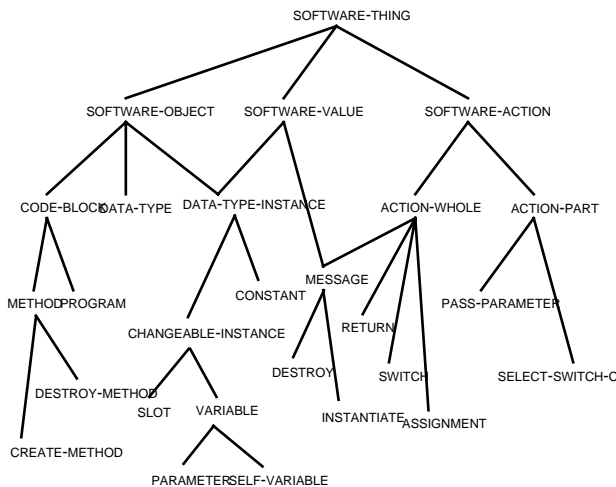


Figure 4. The taxonomy of code-level concepts.

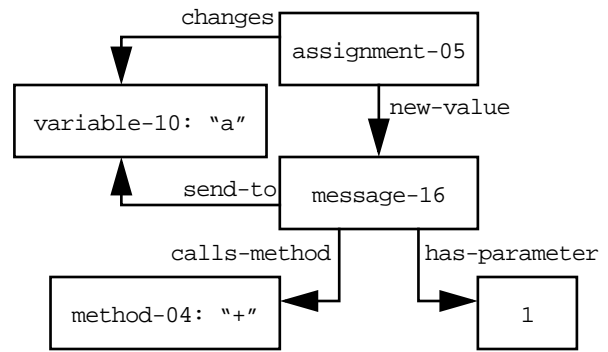


Figure 5. A simple Augmented AST for a=a+1.

code. Inspired by the well-known programming understanding work based on cliché recognition [Quilici, et al., 1996], we were quickly led to *abstract syntax trees* (ASTs).

ASTs have been used in program synthesis and understanding for some time, but previous work in program understanding proceeded with ASTs that had little semantic structure [Quilici, 1994]. For example, the AST for the C expression `a=a+1;` is shown in Figure 3. This tree represents only the syntactic structure of the code. We felt that the power of the SIS approach lay in the use of semantic information. For example, the LaSSIE ontology was able to represent the fact that a variable definition was in a specific file. This relationship between a variable and a file requires more semantics than an AST can provide.

## 4 Augmenting Abstract Syntax Trees

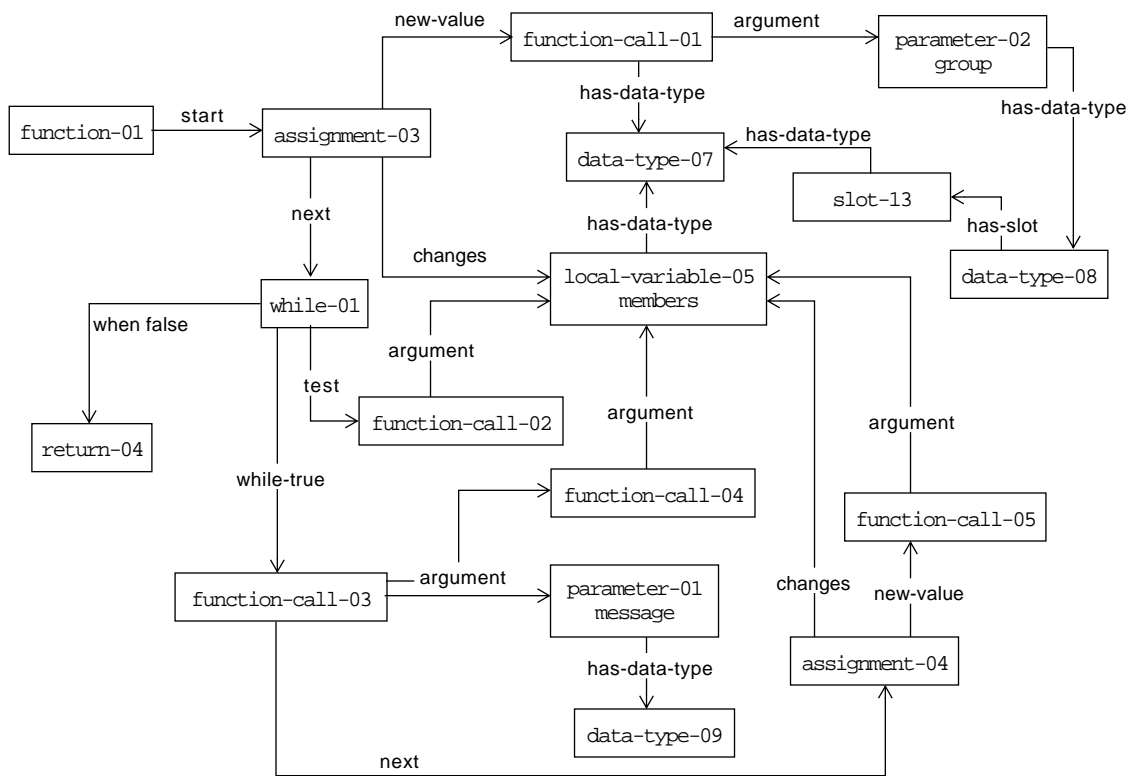
We chose to add semantics to ASTs and develop an ontology for software understanding that could answer the kinds of questions we were dealing with. This new ontology redefines delocalization and brings automated reasoning to bear on the problem of localizing information.

### 4.1 An Ontology for Code-Level Knowledge

The heart of the new ontology is the concept taxonomy shown in Figure 4. These concepts represent nearly all the syntactic constructs available in Smalltalk [Welty, 1995a]. It is important to realize that while a normal AST also includes all syntactic objects in a language, it does not differentiate between them – instances of each concept in this taxonomy represent different kinds of nodes in the augmented AST.<sup>1</sup>

For example, the augmented AST for the statement

1. Although all our work is in Smalltalk, we have chosen to present the simple examples in this paper as C++ instead, under the assumption that it is more widely understood.



**Figure 6.** An Augmented Abstract Syntax Tree representing a C function.

`a=a+1;` is shown in Figure 5, and it is clear that our AST has now become a semantic network. Note that in our ontology we have chosen to treat basic arithmetic operators as methods, not as syntactic elements of the language.

In addition to instances of a few concepts in the taxonomy, Figure 5 shows some of the relationships defined in the ontology, such as that between an assignment operation and the variable it changes (*changes*), an assignment operation and the new value to be assigned (*new-value*), a message and the object it is invoked on (*send-to*), a message and a parameter (*has-parameter*), and a message and the method it invokes (*calls-method*). Figure 5 uses the convention that individuals<sup>1</sup> are given names which include the concept they are individuals of followed by a random number to make each name unique.

The ontology, including all the relationships and constraints, is described fully elsewhere [Welty, 1995a], and though there is clearly more here than in an AST, the true augmentation has yet to be introduced.

## 4.2 Redefining Delocalization

When all the code is represented as a large semantic network, and the maintainers are presented that representation of the program instead of the line by line representation manifested in traditional source code, delocalization and localization take on new meanings.

Relationships between the code-level objects are no longer scattered through a text file, but are represented as links between the objects. Consider a slightly larger, though still trivial, example, shown in Figure 6. This is the augmented AST for the C++ code shown in Figure 1. Note the addi-

1. We must be careful here to keep track of the terminology, since this is an ontology for representing object oriented programs. The term *individual* will be used to refer to objects in the Classic knowledge-base. Individuals are instances of the concepts shown in Figure 4, but they should not be confused with instances of the object-oriented classes used in the software this knowledge-base represents. For example, the object `variable-10` in Figure 5 is an individual of the Classic concept `variable`, and it represents the variable “a”, which in the software is an instance of some class. This rather odd (but necessary) “double instance” representation, and the trouble it causes, is discussed further in [Welty, 1995b].

tion of flow-control (`start`, `next`, `while-true`, and `when-false`) and data-type definition (`has-data-type`) relations to those shown in Figure 5.

With the augmented AST, the distance between pieces of information is no longer measured in terms of lines of code, or different files, but in terms of the number of *links* that must be traversed. For example, in Figure 5 to get from the object `assignment-05` to `method-04`, we must traverse the `new-value` link to get to `message-16`, and then the `calls-method` link. These two objects, then, are separated by two links, and we will say they are two *hops* apart.

Our interface, which is discussed in [Welty, 1996], supports this notion by presenting hypertext links for every relationship an object is involved in. This approach clearly changes the meaning of delocalization, and it also reinforces the point that normal source code delocalizes everything but control flow. In our representation, *far more information is localized*.

In addition, any relationships that would prove useful in discovery (such as answers to the questions listed in Section 3.2 on page 3), could be added as links. For example, `parameter-02`, as an instance of `data-type-08`, has a slot represented by `slot-13`. The individuals `parameter-02` and `slot-13` are two hops apart in Figure 6, but adding a `has-slot` relationship between the two would cut that down to one, thus localizing slot information on class instances.

Such added semantic links would certainly speed up understanding if present, but the data shown in Figure 6 is only that which can be extracted automatically from the code by parsing it. These other links are not explicitly there, and requiring a maintainer to put them in would make the discovery process more time consuming than not having this representation at all.

### 4.3 Simple Reasoning

The essential aspect of the ontology is that it adds semantic information to an AST representation in a description logic (CLASSIC), and this information can now be used in automated reasoning. It is through reasoning that extra semantic links, not present in the simple parse tree, can be added to our augmented AST representation without any burden placed on the maintainer.

We believed early on that we would get a lot of use out of simple path-tracing grammars, once the code was represented as a semantic net. We were quite surprised to find that even simpler forms of inference yielded a big payoff as well.

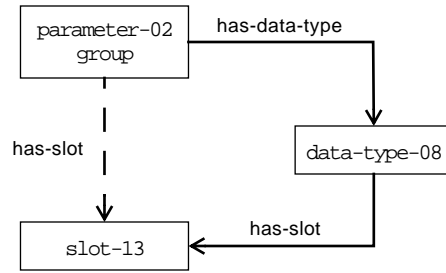


Figure 7. Localizing the `has-slot` relation.

The simplest inference the system provides is through relation inverses. Every code-level relation has an inverse, and this provides a tremendous amount of cross-reference information. Consider the simple code shown in Figure 5; the inverse of the `changes` relationship is `changed-by`, and this now gives us the ability to localize within one hop *all places a variable is changed*. It would be impossible to over-emphasize the importance of this simple inference, as it localized information regarding the most common question the maintainers reported asking during discovery (Question 1.), and it was considered by that group to be the most significant achievement of this work.

The `data-type-of` relation, which is the inverse of the `has-data-type` relation, was used to localize to within one hop answers to Question 7. (Is this data-type used?) The answer to the question is “yes” if the `data-type-of` relation exists on the data-type, and “no” if it does not.

### 4.4 Path Tracing

Another relatively simple class of inferences that yield useful results is path tracing. This fairly common inference mechanism was used to localize answers to Question 2. For this question (What are the available slots and methods on this instance?) the inference merely had to make the `has-slot` and `has-method` relations local to each instance of a data-type.

The localization of the `has-slot` relation on `parameter-02` from Figure 6 is illustrated in Figure 7. In this case the inference specification in Classic is:

```
has-slot <- (has-data-type has-slot)
```

Where the right side of the specification is the *path*, and the left side is the relation to be derived. This notation would correspond to the following Prolog assertion:

```
has-slot(?x,?z) :-
    has-data-type(?x,?y), has-slot(?y,?z).
```

We chose to call this class of inferences “path-tracing” because its implementation required just that: tracing paths through the semantic network of individuals. When inverses are brought into the picture, we found that all path-tracing inferences were expressions of transitivity. In the example above, `slot-of` (the inverse of `has-slot`) is transitive over `data-type-of` (the inverse of `has-data-type`). Expressing the rule that way, of course, would lead to the standard kinds of infinite loops that occur when a relation is the first step in its own derivation.

#### 4.5 Detecting Side-Effects

The most significant inference we discovered was the automatic identification of side-effects. It was here that the true value of Classic as a medium for augmenting ASTs was really brought to light, since detecting side-effects in our representation turned out to be a fairly simple application of subsumption. While the reasoning presented in the previous two sections were clearly very useful, and showed off the benefits of our augmentation to ASTs, they used inferences found in most KR systems. It is subsumption reasoning that distinguishes description logics such as Classic from other KR systems [Brachman, et al., 1991].

The key to side effect detection was in classifying the different kinds of side-effects that occur in software: I/O related side-effects, and changes to global variables. Any method that calls an I/O method or that changes a global variable is therefore a method with a side-effect. In addition, a method that does not call an I/O method nor changes a global variable can have an *indirect* side-effect if it calls a method with a side-effect. Finally, a side-effect (direct or indirect) is *conditional* if it occurs in a conditional branch of the control flow.

The facility for detecting side-effects is described completely in [Welty, 1995a]. Although it only requires a few lines of specification in Classic, the full capability involves eight levels of inference and its explanation (in English) is quite involved. We will instead focus here on the central part of this reasoning, upon which all the different kinds of side-effect related information is based: the identification of an `assignment-side-effect`.

Assignment side-effects are direct side-effects involving a change to a global variable. Referring back to the code-level ontology shown in Figure 4, the only way a variable can be changed is in an assignment statement, which is represented by an individual of the concept `assignment`. Not every assignment is a side-effect, only those which change a global variable. Description logics are designed to make it easy to specify the sufficient conditions for distinguishing a concept from its sub-concepts. In fact, to say that “a side-effect is a kind of assignment in

which the variable being changed is a global” requires only

```
assignment-side-effect::  
  (and assignment  
    (all changes global-variable))
```

In other words, any assignment whose `changes` relation points to an individual of `global-variable` is automatically classified as a side-effect. The two assignments shown in Figure 6 are not side effects, since they change *local* variables.

The full side-effect facility produces similar inferences for propagating the existence of side-effects up to the methods they are contained in, out to methods that call them, and so forth. The facility is fully automated, and the result is the localization of information about side effects. Each method invocation (individuals of the concept `message`) is identified as a call to a method with a direct side-effect, or a call to a method with an indirect side-effect. This, again, brings the answers to a common discovery question to within one hop.

## 5 Results and Conclusion

We studied maintenance problems in a small software company and found that the maintainers spent most of their time searching through the code for answers to questions, in order to assist their understanding. These searches were time consuming because the answers were typically delocalized – spread out through the source code. We recorded the questions and then developed an ontology for code-level knowledge based on adding semantic information to Abstract Syntax Trees. We added specific kinds of automated reasoning that localized answers to all the most common questions we recorded.

Significant steps remain to completely transfer the techniques reported here to industry. In particular, a system for translating back and forth between the Smalltalk source code and the code-level representation of that source code. Such a translation would be easy in principle: there is a one to one correspondence between the syntactic elements of the language and the ontology. This did not seem to be a particularly interesting research problem, so we did not pursue it. The company we worked with is looking into developing this.

There were so many interesting aspects of this project, and it was difficult to decide which to place in this paper. The dynamic between the research group and the employees of the company was quite complex, and many things reported here have been over-simplified due to space constraints. It may seem strange, for example, that a company so strapped for resources would let us in to study what they

were doing without expecting us to apply our research directly to their software system. This was a problem, in fact, but we solved it by bartering some of our own experience with solving some of their smaller, more mundane, problems.

The results were, however, very well received. The management and the maintenance group remain quite impressed with the capabilities these techniques demonstrated, and were amazed that with each discovery question the maintenance team came up with, we were easily able to specify a mechanism to localize the answers. They are looking forward to eventually being able to employ a full-blown system for performing maintenance. It was interesting to us that some of the simpler things to do (such as tracking variable usage through relation inverses) were the most impressive to them.

## REFERENCES

- [Brachman, et al., 1990] Brachman, R., Devanbu, P., Selfridge, P., Belanger, D. and Chen, Y. Toward a Software Information System. *ATT Technical Journal*. 69(2). Pp. 22-41. March, 1990.
- [Brachman, et al., 1991] Brachman, R., et al. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. *Principles of Semantic Networks*. Morgan Kaufman. Pp. 401-456. May, 1991.
- [Devanbu, et al., 1990] Devanbu, P., Selfridge, P., Brachman, R., and Ballard, R. The LaSSIE Software Information System. *Communications of the ACM*. September, 1990.
- [Devanbu, Selfridge, and Brachman, 1990] Devanbu, P., Selfridge, P., and Brachman, R. LaSSIE - A Classification-Based Software Information System. *Proceedings of the 12th International Conference on Software Engineering*. 1990.
- [Henninger, 1997] Henninger, S. Case-Based Knowledge Management Tools for Software Development. *Journal of Automated Software Engineering*. 4(3). Kluwer Academic Press. July, 1997.
- [Holzblatt and Jones, 1993] Holzblatt, K. and Jones, S. *Contextual Inquiry: A Participatory Technique for System Design*. Lawrence Erlbaum. 1993.
- [Huitt and Wilde, 1992] Huitt, R., and Wilde, N. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*. 18(12), December, 1992.
- [Lampert, et al., 1988] Lampert, R., Littman, D., Pinto, J., Soloway, E. and Letovsky, S. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*. 31(11). Nov, 1988.
- [Lewis, 1996] Lewis, T. The big software chill. *IEEE Computer*. 29(3), Pp. 12-14. March, 1996.
- [Ning, et al., 1994] Ning, J., Engberts, A., and Kozaczynski, W. Automated Support for Legacy Code Understanding. *Communications of the ACM*. 37(5), Pp. 50-58. May, 1994.
- [Nishimoto, Chen, and Ramamoorthy, 1990] Nishimoto, M., Chen, Y. and Ramamoorthy, C. The C Information Abstraction System. *IEEE Transactions on Software Engineering*. March, 1990.
- [Potts, 1993] Potts, C. Software Engineering Research Revisited. *IEEE Software*. 10(5): 19-28. 1993.
- [Quilici, 1994] Quilici, A. A Memory-Based Approach to Recognizing Programming Plans. *Communications of the ACM*. 37(5), Pp. 84-93. May, 1994.
- [Quilici, et al., 1996] Quilici, A., Yang, Q., and Woods, S. Applying Plan Recognition Algorithms to Program Understanding. *Proceedings of KBSE-96*. IEEE Computer Society Press. September, 1996.
- [Ross and Schoman, 1977] Structured Analysis: A language for communicating ideas. *IEEE Transactions on Software Engineering*. SE-3(1):16-34. 1977.
- [Selfridge, 1991] Selfridge, P. Knowledge Representation Support for a Software Information System. *Proceedings of the Seventh Conference on Artificial Intelligence Applications*. Pp. 134-140. IEEE Computer Society Press. March, 1991.
- [Soloway and Ehrlich, 1984] Soloway, E. and Ehrlich, K. An Empirical Investigation of Tacit Plan Knowledge in Programming. *Human Factors in Computer Systems*. Ablex Publishers. 1984.
- [Soloway and Letovsky, 1986] Soloway, E. and Letovsky, S. Delocalized Plans and Program Comprehension. *IEEE Software*. 3(3). May, 1986.
- [Shrobe, 1995] Shrobe, H. *DARPA Evolutionary Design of Complex Software - Additional Program Information*. Available at [http://www.ito.darpa.mil/ResearchAreas/EDCS\\_Detail.html](http://www.ito.darpa.mil/ResearchAreas/EDCS_Detail.html).
- [Welty, 1995a] Welty, C. *An Integrated Representation for Software Development and Discovery*. PhD Thesis. Rensselaer Polytechnic Institute. Troy, NY. 1995. Available at <http://www.cs.vassar.edu/faculty/welty/papers/phd/>.
- [Welty, 1995b] Welty, C. Towards an Epistemology for Software Representations, in Setliff, D. ed, *Proceedings of the Tenth Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press. November, 1995.
- [Welty, 1996] Welty, C. An HTML Interface for Classic. *Proceedings of the 1996 International Workshop on Description Logics*. AAAI Press, November, 1996.
- [Welty and Selfridge, 1997] Welty, C., and Selfridge, P. Breaking the Toy Mold. In *The International Journal of Automated Software Engineering*. 4(3). Kluwer. January, 1997.