11

Software Engineering

Christoper A. Welty

Abstract

This chapter reviews the application of description logics to Software Engineering, following a steady evolution of description-logic based systems used to support the program understanding process for programmers involved in software maintenance.

11.1 Introduction

One of the first large applications of description logics was in the area of software engineering. In software, programmers and maintainers of large systems are plagued with information overload. These systems are typically over a million lines of code, some approach fifty million. The size of the workforce dedicated to maintaining these enormous systems is often over a thousand. In addition, turnover is quite high, as is the training investment required to make someone a productive member of the team. This seems, on the surface, to be a problem crying out for a knowledge-based solution, but understanding precisely how description logics can play a role requires understanding the basic problems of software engineering "in the large."

11.2 Background

The three principal software maintenance tasks are pro-active (testing), reactive (debugging), and enhancement. Central to effective performance of these tasks is *understanding the software.* In the 1980s, cognitive studies of programmers involved in program understanding [Soloway *et al.*, 1987] revealed two things:

(i) Programmers typically solve problems by realizing "plans" in their programs. This seems to tie the notion of program understanding to plan recognition [Soloway *et al.*, 1986].

(ii) *Delocalized plans* (plans which are not implemented in localized regions of code) are a serious impediment to plan recognition, both for humans and automated methods [Soloway and Letovsky, 1986].

While these observations were interesting, the studies from which they were derived were slightly flawed from the industrial perspective described above: the subjects of these studies were almost exclusively students working alone with small domain-independant programs (i.e., sorting, searching, etc.). It was not clear how these results applied to experienced programmers working in teams with huge domain-specific programs.

An ambitious effort launched by AT&T [Brachman *et al.*, 1990] attempted to address this problem by studying maintainers of a large software system, and measuring the time they spent performing different categories of tasks. What they found was a bit startling: up tp 60% of the time was spent performing simple searches across the entire software system. A part of what was termed *discovery*, and as pointed out later in [Welty, 1997], the need for these searches was the result of the delocalization not only of plans in software, but of information in general; information a maintainer needs to understand a section of code is frequently not found in the vicinity of that section of code, but may be before or after in the file, in a different file, in a different directory, etc. For a large software system whose source code is spread out over a large number of files in a deep and complex directory structure, finding something as simple as, e.g., the definition of a data-type, with tools such as *find* (the Unix program that runs another program on all files recursively down a directory structure) and *grep* (the Unix program that searches files for strings) was both difficult and time-consuming.

Another more comprehensive study was performed by MCC around the same time [Curtis *et al.*, 1988], which concluded, among other things, that prerequisite to understanding the software is understanding the domain in which the software operates and is a part—if you don't know what a "dial-tone" is, you can't be expected to debug the code that generates a dial-tone.

11.3 LaSSIE

In an attempt to have a direct impact on the maintenance group, the researchers at AT&T developed the notion of a *Software Information System* (SIS) [Brachman *et al.*, 1990]. An SIS is basically an information system which treats the software system source code itself as data, and stores relationships that can provide the information maintainers frequently search for during discovery.

The first SIS, LaSSIE [Devambu *et al.*, 1991], was developed to assist the understanding of AT&T's Definity 75/85 software system. Influenced by their own study and that of MCC, it contained two components: a *domain model* and a *code model.*

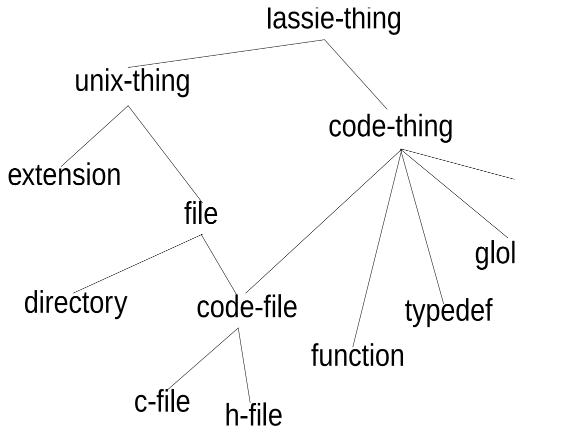

Fig. 11.1. The Lassie code-level ontology.

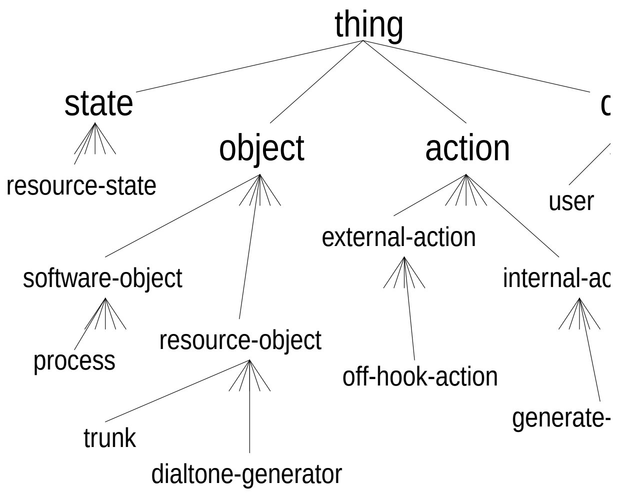

Fig. 11.2. The Lassie telephony ontology.

The code model was implemented with a simple ontology of source code elements, shown in Figure 11.1, which was derived empirically from the basic kinds of searches maintainers performed. The knowlede-base (the actual assertions about individual functions, files, data-types, etc.) was populated automatically from the source code.

The domain model was reverse engineered from the code and contact with the domain experts, it contained knowledge about the *telephony* domain, i.e. the things the software system dealt with. These included entities such as telephones, microphones, cables, cable-trunks, etc. A sample of the ontology is shown in Figure 11.2.

One of the most interesting aspects of this work, and perhaps the most significant from the perspective of exploring description logics, is an analysis of the differences between these two models. The code-model was founded on a very simple ontology, containing perhaps twenty concepts, and was populated with a large number of individuals, on the order of thousands (at least one for each file, data type, function, and variable in the system). The domain model had a large and complex ontology, containing perhaps two hundred concepts, *but very few individuals.*

The reason for this difference was that the trivial searches that characterized software discovery were performed for two reasons:

(i) Discovering specific information about the software, e.g., what is the datatype of the variable `dial-tone`?
(ii) Discovering specific information about the domain, e.g., what is a dial-tone?

In case (i), the maintainer requires specific information about the software, and thus raw data that represents that information is required. For example, by far the most common question asked during discovery is, “Where is this variable used?” [Welty, 1997]. Normally, a maintainer would *grep* for the variable in the rest of the code to find the answers to this question, and as if this didn't take enough time and effort, the results would have to be pruned by hand to remove various kinds of “semantic noise” such as:

(i) variables with longer names that include the desired variable name
(ii) names of functions that include the desired variable name
(iii) comments that include the variable name
(iv) other non-variable string matches

In this particular case, the amount of semantic noise was quite high as a result of mandated naming conventions whose intent was to make the source code easier to understand (semantic noise, also known as *false positives* is a general problem with string-based search methods, and will be discussed further in Chapter 14).

The SIS code model immediately solved these problems by identifying “variable” as a semantic category (as well as file, function, etc. See Figure 11.1). This meant, quite simply, that where a string search for places in which e.g., the variable `error-value` was used might yield such unwanted results as: `compute-error-value`, `display-error-value-result-code`, `error-value-lookup-table`, etc., limiting the search to variables would remove up to 80% of the noise.

In addition to trivially being able to restrict searches to specific categories, other information that could be extracted automatically was mined from the code. For variables, it is simple to automatically determine:

- The file it was defined in.
- Each function in which the variable was used.

In addition, for each function, information was extracted regarding the file it was defined in. From this, a simple inference could be made as follows:

$$\begin{aligned}\mathsf{Variable} \sqsubseteq\ & \forall \mathsf{usedInFile.File} \sqcap \forall \mathsf{usedInFunction.Function} \sqcap \\ & \mathsf{usedInFile} = \mathsf{usedInFunction} \circ \mathsf{definedInFile}\end{aligned}$$

In other words, if a function uses a variable then the variable is used in the file that function is defined in; this produces all the locations where the variable is used. In this manner, the LaSSIE system augmented the basic data in a number of ways through inference.

A similar and nearly as common maintenance task was, e.g., after modifying a function, searching for all the places that function is used to see if the changes affect other sections of the code. Information about what functions call others (i.e., the call graph) was also kept in the code model, and an expression similar to the one in the example above can be used to derive all the files in which a function is called.

The code model alone was able to simplify several of the common discovery tasks maintainers experienced during code modification, but as suggested in case (ii) of the reasons for engaging in discovery listed above, there are other reasons for a maintainer to be searching through the code. For these cases, in which domain information is the desired result of a search, a robust description of the domain is required, and was provided by the domain model (see Figure 11.2).

For example, a maintainer may want to know what kinds of actions a user of the system can take by themselves. To answer this question from the code—the usual approach before LaSSIE—would be quite difficult. One method might be to *grep* through the code for the string "user"—hoping of course that the documentation is up to date or consistent with respect to user actions. Clearly the semantic noise would be quite high in such a case.

Another approach might be to start with a peice of code the maintainer is familiar with, and draw some clues from that for where to look next. The point here is that, whereas for code-model queries the goal is quite specific, domain-oriented queries are not, and imply a lot of time browsing, searching for new ideas, etc. The code is organized around specific functions, not around specific domain concepts, and of course multiple "views" of the code is not supported.

To address this type of need, the LaSSIE domain model expressed knowledge about the domain of telephony. It presented numerous key concepts that let maintainers view the knowledge in the code in a variety of different ways. The domain model was mostly terminological, since it was a description of the things that the software could do. An action concept, such as "generating a dial tone" was a description of the action, whereas an individual would be an actual action of generating a dial tone at some fixed time. These individuals did not normally exist in the domain model, except as examples. The concept would roughly be:

$$\begin{aligned}\mathsf{GenerateDialToneAction} \sqsubseteq\; & \mathsf{Action} \sqcap \forall \mathsf{initiatedBy.LatBox} \sqcap \forall \mathsf{follows.OffHookAction} \sqcap \\ & \forall \mathsf{recipient.LocalPhone} \sqcap \geqslant 1\, \mathsf{hasConnection}\end{aligned}$$

In other words, a "generate dial tone action" is an action that is initiated by a local telephone service following an "off hook action." The recipient of the product of the action (the dial tone) is a phone for which a connection has been allocated.

Other domain concepts described things that the software reacted to, such as:

$$\begin{aligned}\mathsf{OffHookAction} \sqsubseteq\; & \mathsf{Action} \sqcap \forall \mathsf{initiatedBy.User} \sqcap \leqslant 0\, \mathsf{follows} \sqcap \\ & \forall \mathsf{recipient.LatBox} \sqcap \forall \mathsf{activates.AllocateConnectionAction}\end{aligned}$$

In other words, an "off hook action" is an action that is initiated by a user (more commonly the result of pressing a button these days than lifting the receiver off the hook). It follows no previous action, and the recipient of the product of the action is the local telephone service (on which the software is running). The action activates a search for a connection.

Returning to the randomly chosen example above, the maintainer looking for all actions that can be initiated by a user would simply enter a query such as

$$\mathsf{Action} \sqcap \forall \mathsf{initiatedBy.User}$$

and the system would find all the concepts subsumed by that expression. LaSSIE contained a facility for defining new domain concepts identified by maintainers during discovery, and adding them to the domain model by simply assigning them a name (e.g., USER-ACTION in this example).

While these two models independently solved existing problems, it soon became clear that integrating the two models was an important requirement. Using the tool exposed the fact that most domain queries were followed by code-queries. For example, after exploring the domain model to discover the significance of a "connect action", the maintainer will typically ask, "What are the functions that implement it?" In addition, classifying software components by their relevance in the domain was viewed to be a very significant bit of functionality, as this permitted components to be found and retrieved with this information—something that was not previously possible.

This integration between the two models made it possible to use subsumption to find different software objects. For example, all functions that implement connect actions would be:

$$\mathsf{Function} \sqcap \mathsf{ConnectAction}$$

Variables used in functions that implement user actions would be:

$$\mathsf{Variable} \sqcap \forall \mathsf{usedInFunction.UserAction}$$

The LaSSIE system underwent steady development for several years at AT&T, and was shown to cut down on the time maintainers spent in discovery. In order to further improve the process, it was observed that:

- The connection between the domain and code models needed to be made by hand. This was time-consuming to create, and difficult to maintain since the domain model changed over time as new features were added to the software system. Maintainers began to lose faith in the domain model, as a result, and usage deteriorated.
- The code model, though incredibly simple, was used far more frequently than the domain model, and became an important part of every maintainer's tool set. It did not, however eliminate the searches maintainers made, and therefore did not completely replace *find* and *grep*.

11.4 CodeBase

Because the code model proved quite useful and easy to maintain, the demand for it began to increase. This introduced two problems for the LaSSIE SIS:

- Like all description logic systems, it was main memory based. The software contained many thousands of functions, variables, and files. More importantly, the complexity of the function call graph, variable usage graph, and location maps, exceeded one million. It was not possible to store this amount of information in main memory of any computer at that time.
- The natural language interface, while simple and easy to understand, did not *facilitate* using the system quickly. One still had to compose a proper query and type it in. If the result of one query were to be used in another, the maintainer had to re-type the name(s) of the concepts or individuals involved. Increased usage demanded a better user interface.

The CodeBase system [Selfridge and Heineman, 1994] offered solutions to both of these problems. Perhaps the most significant achievement was the development of a system for off-line storage of individuals. The relatively small code-model TBox was always kept in memory, but individuals were kept on a disk, in a technique similar to virtual memory. The difference was in the heuristics used for predicting what portions of the ABox to pre-load.

Whereas a virtual memory system normally uses heuristics based on temporal or spatial proximity, for a knowledge base like LaSSIE, this was not relevant. The location of an individual in physical memory was no indication of its relevance to other individuals near it in physical memory.

The heuristics for virtual memory are based on the empirical observation that when one location is accessed, it is probable that the next access will be to a nearby location in memory. The LaSSIE developers observed that, in a description logic ABox, when an individual is accessed it is probable that the next access will be *to one of its role fillers*, or to objects along some role path from the accessed individual. Because a role may have many fillers, and because an individual may have many roles, there is no way to arrange the individuals in memory so that the normal virtual memory heuristics will be efficient.

CODEBASE also provided numerous graphical tools for viewing and browsing the information in the knowledge base. While this is less significant from the general perspective of description logics, it is important from the standpoint of developing knowledge-based systems. One must never forget that these systems interact with people, and can not be considered as viable systems unless the human is "in the loop."

11.5 CSIS and CBMS

Development of LaSSIE was eventually halted by the trivestiture of AT&T in 1995. Research into software information systems did not stop, however, and description logics have played an important role in this continued development.

Two issues were brought to light by the LaSSIE system:

- The deterioration of the domain model over time was another manifestation of the classic software documentation problem: the same information being stored in different ways. The code model stayed relevant because it was automatically generated from the only thing that *had* to be maintained: the software. It did not, therefore, need to be maintained separately to remain accurate. The documentation and the domain model were different representations of the knowledge that was, perhaps implicitly, in the code. These representations always lagged the "real" one, since they had to be maintained independently.
- The delocalization of information in software, which is the central obstacle to code understanding, required new ways of viewing the code. Looking at code on the screen, analogously to the heuristics for operating system virtual memories, is inherently two-dimensional. It does not allow for relationships between code-level entities to be viewed, or localized.

The first step in determining how to address these problems was to perform further studies of programmers involved in discovery to gain more detailed insight into specifically what they were doing. One such study, in this case of programmers maintaining a moderate-sized object-oriented software system, found that the most common high level queries were:

(i) Where is this variable modified?
(ii) What are the available slots and methods on this instance?
(iii) What is the data-type of this variable or function?
(iv) What are the superclasses of this class?
(v) What does this function return?
(vi) Does this function have side-effects?
(vii) Is this data-type used?

Clearly, to provide answers to questions like these requires far more fine-grained information about the software than simply the locations of the definitions. Furthermore, this study confirmed 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.

These issues spurred research into *Comprehensive Software Information Systems* [Welty, 1995], which soon became *Code-Based Management Systems.* The idea of CBMS was to define the most precise level of granularity of representation needed to have *complete knowledge of the software system in the knowledge-base.* In other words, to have the knowledge-based representation be the artifact that is maintained.

From a description logics perspective, such a comprehensive representation of software in a knowledge base required the ability to deal with large amounts of information efficiently. In addition, such a deep representation made it possible for a wide range of inferences that were well-suited for subsumption reasoning.

A CBMS is based on a full-scale parse of the code to construct an *abstract syntax tree* (AST), which is basically the parse tree. The AST has all the information of the source code, such that the source code can be completely generated from the AST. The AST is augmented with semantic information that can be derived automatically from the syntax. In C++, for example, we know that the left side of an assignment operator is the variable to be changed, and the right side is the new value.

The ability to represent everything in the code requires a deeper ontology of code-level software elements than the original LASSIE ontology, that includes statements, blocks, conditions, etc. In fact, every syntactic element of the programming lan-

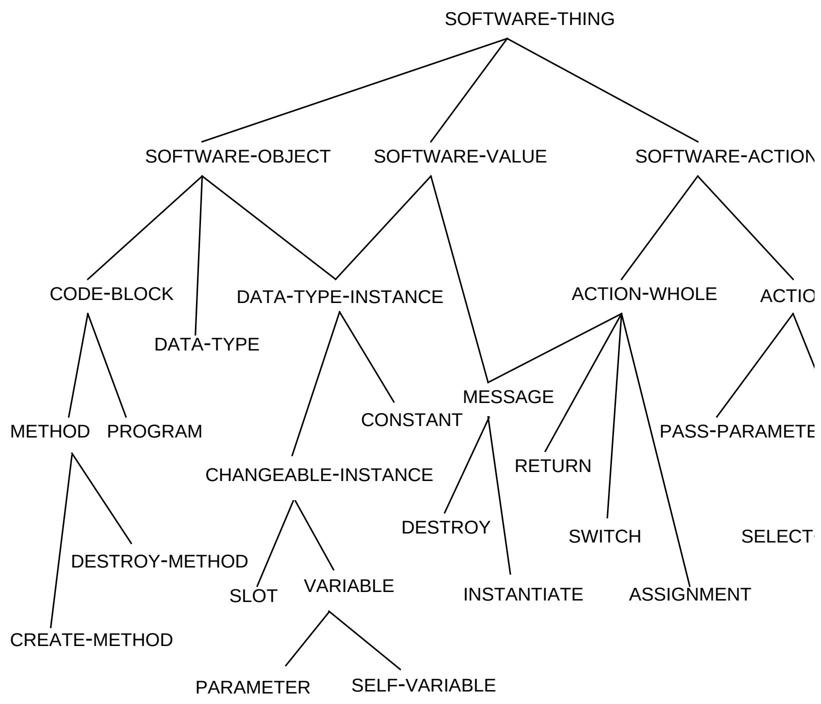

Fig. 11.3. A simplified code-level ontology.

guage is in the ontology. A simplified ontology for an object-oriented language is shown in Figure 11.3.

In addition to these concepts representing the syntactic elements of the source language, roles were use to relate instances of these concepts to each other for control flow, data flow, call-graphs, etc. For example, take the following C++ code fragment:

```
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);
  }
```

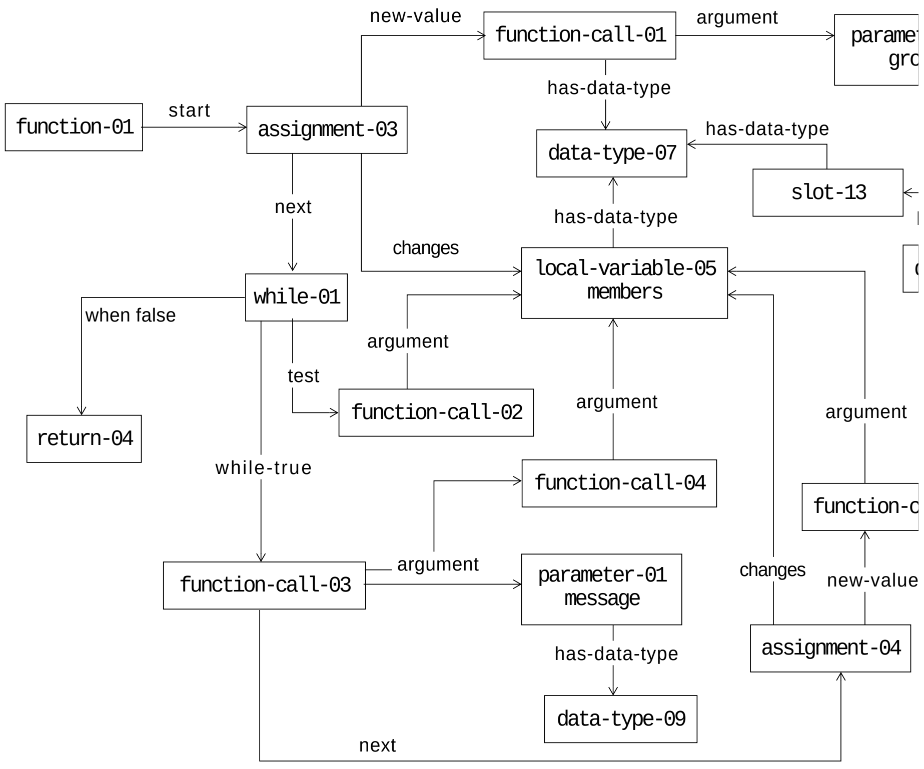

Fig. 11.4. A CBMS representation of the code fragment.

```
}
```

The CBMS representation of that fragment is shown in Figure 11.4. Note that Figure 11.4 shows only the ABox corresponding to the small code fragment, and that role fillers are shown as binary relations.

With an interface that showed individuals in the code representation with role fillers displayed as hypertext links (see [Welty, 1996a]), this ontology alone localizes far more information than the standard text view of software displayed in an editor window. Again, an editor window localizes only the control flow information; a maintainer looking e.g., at the code fragment shown above, only sees the text. The lines are arranged in roughly control-flow order.

Using a CBMS representation, a maintainer's view is focused on a particular object, such as the assignment statement on the first line of the function. This view would be:

```
ASSIGNMENT-STATEMENT-23:
  implementation-of: {FUNCTION-03: group_deliver}
  next: {WHILE-STATEMENT-14}
  changes: {LOCAL-VARIABLE-16: members}
  new-value: {FUNCTION-INVOCATION-34: get-members(group)}
```

In this kind of view, anything in `{...}` is a hypertext link to a similar description of the individual named in the link, and localization takes on a new meaning: the number of hypertext links a desired piece of information is from the current context (individual being viewed). For example, information about control flow is accessible through a chain of `next` links, but in addition, information about data flow is accessible through the `new-value` link, about the function being implemented, about the variable being used, etc.

Another advantage of the CBMS approach is that reasoning can be employed to augment the data and automate the localization of even more information. In existing work in CLASSIC, three types of reasoning were employed:

Role inverses. Every role in the ontology has an inverse, and this provides a tremendous amount of simple bookkeeping information useful to maintainers. In the example above, the changes role is filled through parsing with `members`, and the inverse relationship, that the variable `members` is changedBy `ASSIGNMENT-STATEMENT-23` is added as well. The power of this simple inference can not be under-stated. Studies showed that this was the most useful kind of information the system provided, as it answered the most common question asked by maintainers.

Path tracing. Many useful pieces of information were a few clicks away, but would be more useful if brought within one click (i.e., one link). A simple set of forward chaining "filler" rules in CLASSIC are capable of handling this. For example, it is also useful to know within which functions a variable is changed. Without inference, the maintainer must click on the changedBy role for a variable to get to the statement that changes it (or statements), and then must click on the implementationOf role for the statement to get to the function. Instead, with "path tracing rules," we can fill the changedInFunction role automatically with all the values from the path (changedBy implementationOf). Thus in our example we can conclude that `members` is changedInFunction `group_deliver`.

Subsumption. With subsumption reasoning, membership in a number of useful classes can be inferred for individuals representing pieces of the code. For example, the concept GlobalAssignmentStatement is defined:

$$\begin{aligned}\mathsf{GlobalAssignmentStatement} \sqsubseteq\ & \mathsf{AssignmentStatement} \sqcap \\ & \forall \mathsf{changes}.\mathsf{GlobalVariable}\end{aligned}$$

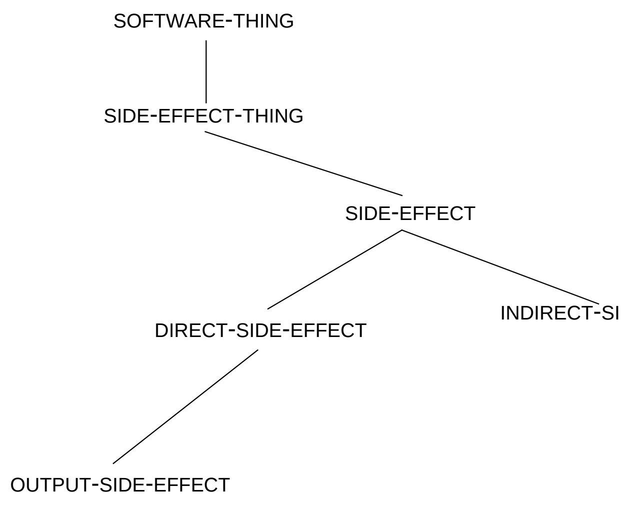

Fig. 11.5. The side-effect ontology.

which allows all the assignment statements that modify global variables to be identified.

The most compelling result that came out of the CBMS work so far has been the automatic detection of side effects, answering the sixth most commonly asked question. This detection was not originally believed to be possible. To simplify the discussion, we assume a pure object-oriented language without pointers or call-by-reference parameters. The latter can be handled in a similar way, the former is still believed to be undecidable.

There can be two kinds of *direct* side effects in a method: a change to a global variable, and any sort of output. A third kind of side-effect is a call to a method that has a side effect. In this case, the side effect does not actually occur within the calling method, yet a side effect will occur when the calling method itself is invoked, so it can be important to discover it. A change to a global variable occurs whenever that variable appears in an assignment statement as the variable to be changed.

The CBMS ontology contains a fairly simple extension which can automatically detect global variable change side effects and calls to methods with side effects. Output methods must be specifically identified as such in order that calls to them may be recognized. This is not really a problem, since output functions are generally part of a support library which would be provided to any developer. The extension begins with a new part of the code-level taxonomy, shown in Figure 11.5. This new taxonomy of primitive concepts fits under the SoftwareThing concept. Next, any individual of GlobalAssignmentStatement (defined above) is a side effect—an AssignmentSideEffect.

In order to put individuals of AssignmentSideEffect into the side effects taxonomy shown in Figure 11.5, a forward chaining rule is added:

$$\mathsf{AssignmentSideEffect} \;\Rightarrow\; \mathsf{DirectSideEffect}$$

This rule is required because if the relationship it specifies were part of the defined concept, being a direct-side-effect would become a sufficient condition for recognizing assignment side effects, and they would never be found automatically. In other words, the rule says "once an assignment side effect is recognized, it should be also be classified as a direct side effect", whereas putting direct-side-effect after assignment in the defined concept definition would say, "An assignment side effect must already be known to be a direct side effect to be recognized." The latter is not productive.

At this point we can classify all assignments that change global variables as assignment side effects and direct side effects. The next addition is a set of roles that will help identify the methods that contain these side effects: hasDirectSideEffect, its inverse directSideEffectOf, and their role parents hasSideEffect and SideEffectOf. With these roles defined, a path tracing rule is added for DirectSideEffect that says directSideEffectOf = implementationOf. In other words, the directSideEffectOf role should be filled with the value in the implementationOf role of the assignment. Through the role hierarchy, this also adds the SideEffectOf role, and through the inverse, the individual of Method that fills this role gets the hasDirectSideEffect and hasSideEffect roles pointing back to the assignment.

With these inverse roles filled in, we can create a new defined concept to recognize methods with side effects:

$$\mathsf{MethodWithSideEffects} \;\equiv\; \mathsf{Method} \sqcap \geqslant 1\, \mathsf{hasSideEffects}$$

and a more specific one for methods with direct side effects:

$$\mathsf{MethodWithDirectSideEffects} \;\equiv\; \mathsf{Method} \sqcap \geqslant 1\, \mathsf{hasDirectSideEffects}$$

Note that the second concept will automatically be classified under the first. Now, as a result of the rules that added the hasSideEffect links, every method that has in its implementation a slot assignment side effect will have at least one filler in its hasDirectSideEffects role, and will be classified as a method with direct side effects.

The next case is detecting indirect side effects, which first requires recognizing invocations of methods that have side-effects (in OO terms, a method invocation is a message):

$$\mathsf{MessageSideEffect} \;\equiv\; \mathsf{Message} \sqcap \forall \mathsf{callMethod}.\mathsf{MethodWithSideEffects}$$

Individuals of this new concept can be recognized since *all methods with side effects have been found with the previous two defined concepts.* A simple forward chaining rule then links these message side effects back into the side effect taxonomy:

$$\mathsf{MessageSideEffect} \;\Rightarrow\; \mathsf{IndirectSideEffect}$$

Next we define two more roles: $\mathsf{hasIndirectSideEffect}$ and its inverse $\mathsf{indirectSideEffectOf}$, and make them children of $\mathsf{hasSideEffect}$ and $\mathsf{SideEffectOf}$, respectively. Once these roles have been defined, and the message side effects have been found, we can identify all the methods that have them in a similar manner to assignment side effects. First, create a path tracing rule for $\mathsf{IndirectSideEffect}$: $\mathsf{indirectSideEffectOf} = \mathsf{implementationOf}$ which will fill in roles. Now we identify all these methods with indirect side-effects with the concept:

$$\mathsf{MethodWithIndirectSideEffects} \;\equiv\; \mathsf{Method} \sqcap \geqslant 1\, \mathsf{hasIndirectSideEffects}$$

The final step is simply to link methods with side effects into the side effect taxonomy with one last forward chaining rule:

$$\mathsf{MethodWithSideEffects} \;\Rightarrow\; \mathsf{SideEffectThing}$$

The addition of this rule basically creates the side effect taxonomy shown in Figure 11.5.

Not only do these definitions identify functions with side-effects, but they also lead a maintainer directly to the side-effect itself. The point here, from a software understanding perspective, is that subsumption makes it possible to *localize* information that otherwise would be difficult (or at least time consuming) to discover.

The inferences for finding side effects are clearly very deep, yet the developer or maintainer need not be aware of them. All these side effect inferences come with no extra work by the developer or maintainer at all. In fact, answers to *all* of the top questions asked by maintainers during discovery can be localized to within one link, therefore one mouse click in the simple hypertext interface described.