An Ontology-based Approach to Software Comprehension - Reasoning about Security Concerns

Yonggang Zhang, Juergen Rilling, Volker Haarslev

Computer Science and Software Engineering
Concordia University, Montreal, QC, Canada
{yongg_zh, rilling, haarslev}@cse.concordia.ca

Abstract

There exists a large variety of techniques to detect and correct software security vulnerabilities at the source code level, including human code reviews, testing, and static analysis. In this article, we present a static analysis approach that supports both the identification of security flaws and the reasoning about security concerns. We introduce an ontology-based program representation that lets security experts and programmers specify their security concerns as part of the ontology. Within our tool implementation, we support complex queries on the underlying program model using either predefined or user-defined concepts and relations. Queries regarding security concerns, such as exception handling, object accessibility etc. are demonstrated in order to show the applicability and flexibility of our approach.

1. Introduction

Developers today not only develop but also inherit systems that are intrinsically difficult to understand and maintain because of their size and complexity, as well as their evolution history [2, 3]. With applications that become exposed to volatile environments with increased security risks (e.g., distributed environments, web centric applications, etc.), identifying these security flaws in existing software systems becomes one of the major activities in software maintenance phase.

In addition, software maintenance is a task not only time consuming but also error prone. Changes made to existing software systems may likely introduce new security vulnerabilities that are typically caused by "carelessness or lack of awareness about security concerns" [14].

On top of various techniques to automatically identify software vulnerabilities at the source code level, manual code auditing is still considered a necessary approach due to the fact that through manual auditing one can identify security flaws that are otherwise impossible to find automatically. However, manual audits are significantly more expensive than automatic analysis tools since programmers must first know what the security flaw looks like and take time to understand the code under review.

In this research, we present a novel ontology-based program comprehension approach that allows security experts, as well as programmers, to specify their security concerns and identify security flaws based on user-defined specifications. Using our approach, auditors typically start with a hypothesis of what a security flaw looks like, and then apply static code reviewing to determine whether this security risk exists in the code. The approach can be viewed as an iterative comprehension process based on examining and refining a hypothesis.

The remainder of the paper is organized as follows: In section 2, we introduce background relevant to ontology-based program comprehension. In section 3, we introduce our SOUND tool and approach. Section 4 focuses on the use of our software ontology to support reasoning about security concerns in the source code. Section 5 discusses related work followed by conclusions in section 6.

2. Ontology-based Program Comprehension Model

There exists a significant body of research to model program comprehension in terms of mental representations and the process of creating them. *Bottom-up* [1] theories consider that understanding a program is constructed from source code reading and then mentally *chunking* or *grouping* the statements or control structures into higher level information, i.e. from bottom up. Such information is further aggregated until high-level abstraction of the program is obtained. *Top-down* models [2], on the other hand, start the comprehension process with a *hypothesis* concerning a high-level abstraction, which then will be further refined, leading to a hierarchical comprehension structure. The understanding of the program is developed from the confirmation or refutation of hypotheses.

However, it is unlikely that only one comprehension model is exclusively used by software engineers. It has been shown [3] that programmers often switch between bottom-up and top-down models during the comprehension process. In situations where the code is unfamiliar to the programmer, a bottom-up model is preferred; when knowledge of the program and application domain is available, programmers tend to adopt a top-down approach for the assimilation of their program understanding [4].

Based on these models, various tools have been developed to assist programmers during the comprehension process. These tools range from analyzing very low level source code entities such as program dependencies graph or program slicing [5] etc to tools that provide design level concepts such as software architectures [7]. Common to all of these tools is that they are trying to represent software in various forms to facilitate the construction of mental models.

While these tools are often quite successful in achieving a specific program comprehension task, they typically lack flexibility in supporting other comprehension tasks. Also, most of these tools continue to exist in isolation. Software maintainers have to use these comprehension tools independently from each other, therefore requiring additional efforts in manually integrating the results from the different tools [8]. Another major shortcoming of these tools is that analysis techniques supported are often neither intuitive nor flexible enough from a maintainer's perspective to provide the required support to create an appropriate mental model. This is mainly because the foundation models for these tools do not correspond closely to a programmer's own mental programming model or to his/her expertise. Programmers will have to adjust their comprehension strategy to the model provided by the tools.

An ontology is a description of the concepts and relationships that can exist in a domain [10]. In particular, in software program comprehension, a ontology consists of vary concepts from programming languages and techniques such as class, method, algorithm etc. as well as a set of roles (relationships between concepts) characterize the relations between entities occurring in the software program, such as define, implement, etc. Given the fact that a mental model may take many forms, but its content normally constitutes an ontology [9], an ontology-based approach to program comprehension is then a straight forward technique to bridge the gap between software representations and mental model.

In this paper, we present a new perspective of program comprehension, in which we specify it as an iterative process of concept recognitions and relationship discoveries in source code (Figure 1). Such a process typically starts with an initial mental ontology that represents a programmer's knowledge about the programming and application domain. By reading source code and documents, instances of concepts and relationships are identified, and new concepts are discovered from the software artifacts. The result of each of these iterations therefore includes the identified instances of concepts in the program, as well as a richer ontology containing the newly discovered knowledge, i.e. a better mental model (the ontology and its instances) is constructed.

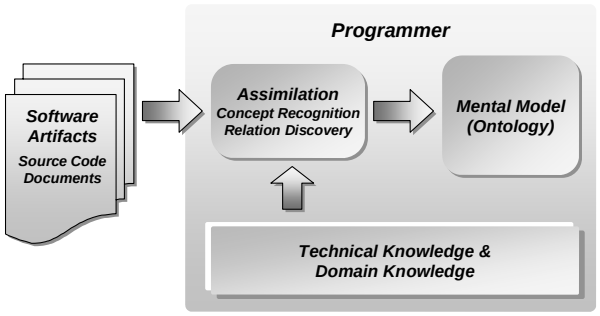

Figure 1 – Ontology-based program comprehension

Concept Recognition – One can consider the following example, in which a programmer might start with a simple ontology about the Java language. Such an ontology might include concepts like *method*, *variable* and *modifier*. During code/documentation review, the programmer tends to first recognize instances of these existing concepts, due to their familiarity. In situations where further analysis is required, for example to study the accessibility of source code entities, programmers would use additional concepts, such as *public method* or *private variable*, to analyze the source code.

In some cases, new concepts may be learned from the source code or documents. For example, a design document may state that a class is a *façade class*. For a programmer who is not familiar with such a concept, further consultation of the documents or analysis of the source code would be required, to understand that the façade class is the public access point of a component. As a result, the newly learned concepts will be used to enrich the ontology.

Relationship Discovery – The comprehension process also includes discovering properties of identified entities, i.e. their relations with other entities. Simple relationships, such as a variable is *defined in* a class or a method *calls* another method, can be recognized instantly. More implicit relations can be constructed from these simple relations. For example, a class C_1 *uses* another class C_2 if either a variable defined in C_1 has the type C_2 or a method defined in C_1 accesses methods or fields defined in C_2.

Result – The result of the ontology-based program comprehension process is an ontology that captures

adequate concepts and relationships required for a particular comprehension task, as well as a set of source code entities and their relations corresponding to an instance of that ontology. The ontology can therefore be considered as the programmer's current mental model of a program. Such an ontology forms the basis for the next iteration of the program comprehension.

3. SOUND – an Ontology-based Program Comprehension Tool

As part of this research, we have developed a tool for the support of our ontology based program comprehension model. Our SOUND (Software Ontology for UNDerstanding) tool is an Eclipse Plug-in that provides ontology management and reasoning service integration for the Eclipse IDE (Figure 2).

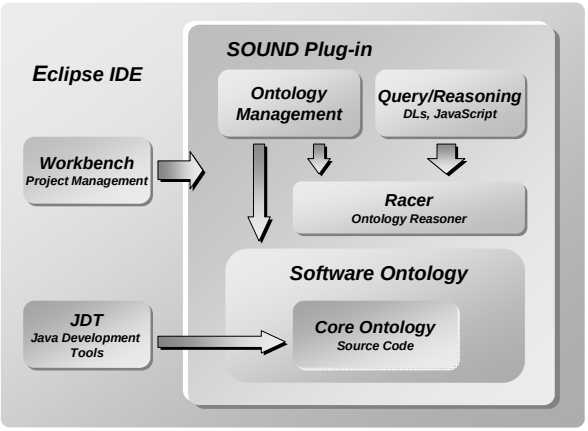

Figure 2 – SOUND tool overview

The software ontology consists of a core source code ontology and a set of user-defined concepts and relations. The core ontology, which is extracted from Eclipse, has a direct mapping to the Java source code. The ontology management interface provides functionalities such as defining concepts/relations, specifying instances, and browsing the current ontology. An ontology reasoner – Racer [11] is used to provide reasoning services on the software ontology.

3.1 Description Logics and Racer

Introducing tool support for ontology-based program comprehension benefits programmers during the mapping of mental concepts to source code entities, and it also allows users to take advantage of the existing expressiveness of ontology languages and reasoners. They allow users to construct complex concepts and queries to derive implicit facts from the ontology.

In order to precisely characterize concepts and their relations in our software ontology, an ontology language with well-defined semantics is essential. Description Logics (DLs) have been long regarded as standard ontology languages. DL is also a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [20]. DLs, a family of knowledge representation formalisms, represent the knowledge of an application domain by first defining the relevant concepts of the domain and then using these concepts to specify properties of objects and individuals occurring in the domain [12].

Basic elements of DLs are atomic concepts and atomic roles, which correspond to unary predicates and binary predicates in First Order Logic. More complex concepts are then defined by combing basic elements with several concept constructors. For example, having atomic concepts in Java language, such as `Method` and `Exception`, as well as a atomic role `throw`, which indicates a method may throw an exception, a new concept `MethodThrowException` can be then defined as –

$$\texttt{MethodThrowException} \equiv \texttt{Method} \sqcap \exists \texttt{throw.Exception}$$

For a more complete and detailed coverage of DLs, we refer the reader to [12].

Combined with state-of-the-art ontology reasoners, such as Racer [11], various kinds of queries concerning the software ontology can be answered. Racer is a knowledge representation system that has been highly optimized for very expressive Description Logics. Typical reasoning services (types of queries) [13] provided by Racer includes –

- Concept consistency – is the set of instances described by a given concept empty?
- Concept subsumption – is there a subset relationships between the set of instances described by two concepts?
- Ontology consistency – find all inconsistent concepts names in the ontology.
- Classification – determine all the subsumption relationship between concepts in the ontology.

Given a set of facts in the domain as instances of concepts and roles in ontology, Racer can answer following types of queries –

- Consistency checking – are the restrictions in the ontology too strong with respect to the facts?
- Instance checking – is a specified individual in the domain an instance of a given concept description?
- Instance retrieval – retrieve all instances of a given concept description.
- Tuple retrieval – retrieve tuples of instances that satisfy given constraints.
- Instance realization – compute the most specific concept of an individual.
- etc.

3.2 Core Ontology

Having a sufficiently expressive ontology language, such as Description Logics in our case, one can design a core ontology to capture major concepts of Object Oriented Program languages. This core ontology can then be further extended with some Java specific concepts and roles. Figure 3 shows major concepts used in our core ontology.

The use of DLs allows us to formally characterize subsumption relationship between concepts. A concept C is considered as a sub-concept of D if all instances of C are also instances of D. Therefore, if an individual is specified as a Method in our ontology, it will be automatically recognized as a Member, and further, as a SourceObject.

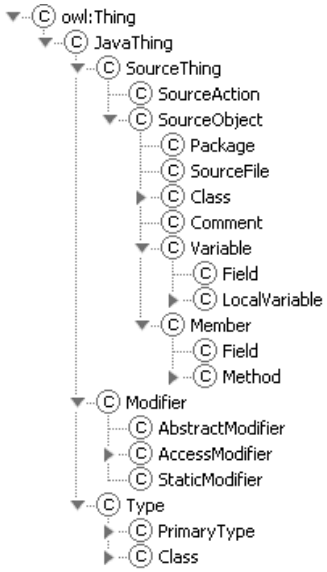

Figure 3 – Concept names in core ontology

Within our core ontology many roles are also defined to specify the relationships between various concepts. Part of the role names are shown in Table 1

Table 1 – Role names in core ontology

Role Name	Description
definedIn	SourceObject A is defined in another SourceObject B
hasSuper	Class A has super class Class B
call	Method A calls Method B
read	Method A read Variable B
write	Method A write Variable B
readField	Method A read a Field B, sub-role or read
writeField	Method A write a Field B, sub-role of write
hasModifier	SourceObject A has a Modifier B – such as public, protected, private, etc.
throws	Method A throws an Exception B

We further enriched the vocabulary by providing for each role in our ontology also its corresponding inverse. For example, the inverse role of hasSuper is hasSub, i.e. if class C_1 hasSuper class C_2, then C_2 also hasSub class C_1. Similarly, if method M write a variable V, then V is writtenBy M.

One of the advantages of using the Racer system is the ability to define transitive roles. If a role R is defined as a transitive role, and if (a,b)∈R and (b,c)∈R are specified, then (a,c)∈R also holds. Transitive roles are especially useful for specifying part-of relation between source code elements (through definedIn role), inheritance relation between classes (through hasSuper role), and indirect calling relations (through indirectCall role).

It is also possible to define subsumption relationships between roles. If role R is a sub-role of S, then if (a,b)∈R, then (a,b)∈S also holds. For example, in our core ontology, readField is a sub-role of read, let M and F are instance of Method and Field respectively, and if relation (M,F)∈readField is discovered, then (M,F)∈read will be automatically recognized by Racer.

Concepts in the core ontology typically have a direct mapping to source code elements, and thus instances of these concepts can be automatically discovered by a Java compiler provided by Eclipse – the JDT (Java Development Tools). Instances of roles are obtained by static analysis of source code. The advantages of static the source code analysis over dynamic analysis include its light weight and its claiming about all possible program executions rather than being limited to several test cases [14]. However, one major drawback is that many relations discovered are only *potential* relations. For example, static analysis might report a method calls another method based on the source code, but that invocation might never happen because of the program's run-time configuration.

3.3 Query Interface

As part of our tool, users can use a expressive query language provided by Racer – nRQL[19] to retrieve instances of concepts and roles in the ontology. An nRQL query uses arbitrary concept and role names in the ontology to specify the properties of the result, in which query variables can be used to bind to instances that satisfy the query.

However, the use of nRQL is still much restricted to users with high mathematical background because Description Logics formulae, although comparatively straightforward, are still difficult for programmers to understand and even more difficult to create. To bridge this gap between practitioners and Description Logics systems, we have utilized a script language – JavaScript

as the query language, and provided a set of build-in functions and classes in the JavaScript interpreter Rhino* to simplify user querying on the software ontology.

Some of the built-in functions for formulating complex concepts are summarized in Table 2, in which C and R are string parameters that denote concept and role names respectively.

Table 2 – Build-in functions

AND (C_1, C_2, …)	conjunction of C_1, C_2, …
OR(C_1, C_2, …)	disjunction of C_1, C_2, …
NOT(C)	negation of C
EXIST(R, C)	concepts that exist a relation R whose filler is type of C
KNOWN(R)	concepts that have explicit specified relation R

Users can construct their own concepts using these build-in functions. The concept MethodThrowException, discussed in section 3.1, can for example be specified as

AND("Field", EXIST("throw", "Exception"))

Two classes – Query and Result are provided to assist users in composing queries and manipulating results. Table 3 shows the major functions provided by the Query class.

Table 3 – Methods of build-in class Query

declare(var_1, var_2, ….)
Declare query variables in the query, which will be bound to instances that satisfy the restriction.
restrict(var, C)
Specify a concept restriction to the query result – the query variable has to be an instance of C
restrict(var, R, object)
Specify a relation restriction to the query result – the query variable has an R relation with the object. The object can be either a variable or an instance.
not_concept(var, C)
Specify a negated concept restriction – the query variable must NOT be an instance of C
no_relation(var, R, object)
Specify a negated relation restriction – the query variable has NO explicit R relation with the object.
retrieve(var_1, var_2, ….)
Specify the result will only include instances bound to specified query variables

The typical procedure of querying the ontology is as follows: (1) query variables are declared, (2) restrictions are specified, and (3) queries are submitted to the built-in *ontology* object that represents the software ontology. The query results are a set of tuples that satisfy the specified restrictions. An alternative to defining a new concept MethodThrowException is the following query script that directly retrieves *all methods that may throw exceptions* –

```
var method_throw_exception = new Query();
method_throw_exception.declare("M", "E");
method_throw_exception.restrict("M", "Method");
method_throw_exception.restrict("M", "Exception");
method_throw_exception.restrict("M", "throw", "E");
method_throw_exception.retrieve("M", "E");
var result = ontology.query(method_throw_exception);
```

By introducing the scripting language, users can benefit from both the declarative semantics of Description Logics as well as the fine-grained control ability of procedural languages. It must be noted that users are not limited to the set of predefined scripts; they can extend the scripting library depending on the comprehension task and the vocabulary of the ontology. For example. the following script

```
for(i = 0; i < result.size(); i++){
   out.println("Method " + result.get(i, "M") + " throws  " + result.get(i, "E");
}
```

will print out a detailed report of the above query result.

In Figure 4, we illustrate a typical query-answer scenario using our SOUND tool.

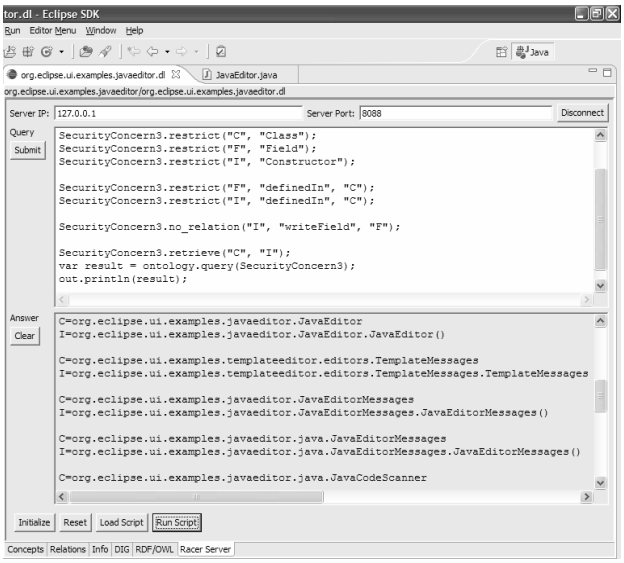

Figure 4 – The SOUND scripting and querying interface

4. Reasoning about Security Concerns

In what follows, we discuss how our ontology-based comprehension approach can be applied to provide programmers or security experts with the ability to query and reason about different security concerns.

4.1 Object Accessibility

Typical Object Oriented Programming languages provide object modifiers such as *public*, *private*, and *protected* to restrict the accessibility of objects and method. However, improper use of these access modifiers may cause security risks. For example,

* available online at http://www.mozilla.org/rhino/

making class variables (fields) public may cause vulnerable data exposure and may cause undetermined behaviors if the public field has not been initialized.

Within our ontology, we provide a set of atomic concepts and roles to facilitate querying and reasoning about such security concerns. The following scenario illustrates how our approach can facilitate security experts or programmers in identifying potential vulnerabilities caused by unexpected object accessibility.

The user first creates a basic scenario for the code auditing task by defining a public field as a field that has public modifier. This new concept, for example called PublicField, can be defined in our tool by using the native Description Logic formula:

$$\text{PublicField} \equiv \text{Field} \sqcap \exists\ \text{hasModifier.PublicModifier}$$

or alternatively, by using our scripting interface:

```
var SecurityConcern1 = new Query();
SecirityConcern1.declare("F", "MP");
SecurityConcern1.restrict("F", "Field");
SecurityConcern1.restrict("MP", "PublicModifier");
SecurityConcern1.restrict("F", "hasModifier", "MP");
SecurityConcern1.retrieve("F");
var result = ontology.query(SecurityConcern1);
```

In this scenario, allowing public and non-final fields in the code (indicating value of the field can be modified out side the class it defined) might be a security risk. Therefore, SecurityConcern1 can be further refined by adding the following statements:

```
SecurityConcern1.restrict("MF", "FinalModifier");
SecurityConcern1.no_relation("F", "hasModifier", "MF");
```

In order to extend the query for more specific tasks, such as: *Retrieve all public data of Java package "user.pkg1" that may potentially be accessed (read or write) by package "user.pkg2"*, users can further refine the previous query by adding –

```
SecurityConcern1.restrict("F", "definedIn", "user.pkg1");
SecurityConcern1.restrict("M", "Method");
SecurityConcern1.restrict("M", "definedIn", "user.pkg2");
SecurityConcern1.restrict("M", "access", "F");
```

It should be noted that fields or methods in Java are defined in classes, and classes are defined in packages. The ontology reasoner will automatically determine the transitive relation definedIn between the concepts Field/Method and Package. In addition, read and write relations between method and field are modeled in our ontology by the readField and writeField roles, which are sub-role of access. The ontology reasoner is also capable of providing automatically reasoning on this kind of classification.

4.2 Exception Handling

Exceptions correspond to events that can occur during the execution of a program that disrupt the normal flow of instructions. In Java, when an error occurs within a method, the method may throw an exception object that contains run-time information associated with the error. The caller method can then catch that exception and perform recovering from the error. While exception handling mechanisms have greatly simplified error management, security concerns still may arise – an unhandled exception may cause programs to fail. Even worse, if such an exception occurs during file access, it may cause unexpected data exposure.

In Java, a method may arbitrarily throw a RuntimeException in its sub classes without necessarily being caught. For example, the *get* method of java.util.ArrayList in the Java JDK might throw an IndexOutOfBoundsException without forcing the caller method to catch that exception. Although runtime exceptions rarely occur, there are potential situations in particular in multi-thread programs, when an object in one thread accesses the ArrayList object and at the same time its content may be modified by another thread. Therefore, this situation might lead to an unhandled exception. In the following example, we illustrate how our tool can be used to identify these types of problems caused by unhandled exceptions.

The following query retrieves all methods that may throw a RutimeException object:

```
var SecurityConcern2 = new Query();
SecurityConcern2.restrict("M", "Method");
SecurityConcern2.restrict("E", "Exception");
SecurityConcern2.restrict("M", "throw", "E");
SecurityConcern2.restrict("E", "hasSuper", "java.lang.RuntimeException");
SecurityConcern2.retrieve("M");
var result = ontology.query(SecurityConcern2);
```

The first two restrictions in the above SecurityConcern2 query state that M and E are a Method and an Exception respectively. The third restriction states that method M throws an Exception E, and the last restriction expresses that E is a subclass of java.lang.RuntimeException. It has to be noted that the hasSuper role in our ontology represents the inheritance relation between two classes. The transitivity of this role will be automatically handled by the ontology reasoner.

For the next level of analysis, we restrict the query further to *only retrieve those methods that may invoke method M*, we can add

```
SecurityConcern2.restrict("Caller", "Method");
SecurityConcern2.restrict("Caller", "invoke", "M");
```

and change the retrieval statement to –

```
SecurityConcern2.retrieve("Caller", "M");
```

More interesting results can be obtained by using the Negation As Failure (NAF) semantics provided by nRQL. For example, in order *to ensure all runtime exception thrown by M are handled (caught) by its Caller*, we could add one new restriction –

```
SecurityConcern2.no_relation("Caller", "catch", "E");
```

This restriction will return the complementary (NAF) of Callers that catch exception E. In combination with other restrictions, the ontology reasoner will then retrieve each method that does not catch runtime exceptions a caller may throw. It also has to be pointed out that by applying negated restrictions potential performance issue may be introduced, thus their frequent use are not encouraged [19].

4.3 Security Enforcement

Many security flaws are preventable through security enforcement. Common vulnerabilities such as buffer overflows, accessing un-initialized variables, or leaving temporary files in the disk could be avoided by programmers with strong awareness of security concerns.

In order to deliver more secure software, many development teams have guidelines for coding practice to enforce security. Our tool supports developers and security experts to enforce or validate whether these programming guidelines are followed. For example, to prevent access to un-initialized variables, a general guideline could be: *all fields must be initialized in the constructors.* The following query can retrieve all classes that did not follow this specific guideline.

```
SecurityConcern3.restrict("F", "Field");
SecurityConcern3.restrict("I", "Constructor");
SecurityConcern3.restrict("C", "Class");
SecurityConcern3.restrict("F", "definedIn", "C");
SecurityConcern3.restrict("I", "definedIn", "C");
SecurityConcern3.no_relation("I", "writeField", "F");
SecurityConcern3.retrieve("C", "I");
```

In Java, all classes without a constructor will have a no-argument constructor by default. These classes therefore can be initialized by any part of the program. A good security enforcement guideline is that *each class has to provide at least one constructor*. These classes without any constructors can be retrieved by the following script

```
SecurityConcern4.restrict("C", "Class");
SecurityConcern4.not_concept("C", KNOWN("hasConstructor"));
SecurityConcern4.retrieve("C");
```

The next example demonstrates how our tool can enforce the following security practice: *all methods should take the responsibility to close the file(s) they have opened.* Such a guideline can be enforced by defining two new concepts such as FileOpenMethod and FileCloseMethod. These new concepts correspond to methods that are used to open or close files. Those methods have to be specified as instances of two concepts respectively. In a typical Java program, such methods include:

```
FileOpenMethod = {
java.io.File.createNewFile(),
java.io.File.createTempFile(),
java.io.FileInputStream.FileInputStream(), …}
```

and

```
FileCloseMethod = {
java.io.FileInputStream.close(),
java.io.Writer.close(), …}.
```

Based on these new two concepts, the following query script shown in SecurityConcern5 can be applied to *detect potential methods that only invoke a file open method, but not a corresponding file close method*.

```
SecurityConcern5.restrict("M", "Method");
SecurityConcern5.restrict("O", "FileOpenMethod");
SecurityConcern5.restrict("C", "FileCloseMethod");
SecurityConcern5.restrict("M", "Invoke", "O");
SecurityConcern5.no_relation("M", "Invoke", "C");
SecurityConcern5.retrieve("M");
```

SecruityConcern5 also exposed some of the limitations of our ontology-based approach. At the current stage, the ontology language lacks the ability to represent temporal properties in the domain such as sequence of actions. This might lead to situations where multiple occurrences of a relation are only captured once. In the SecurityConcern5 (file open/close) query, methods that invoke file open methods twice but call the corresponding close method only once are also returned. Furthermore, because our ontology is based on static source code analysis, the query cannot ensure the file open/close methods are actually invoked or the proper sequence (first open then close) is followed during execution.

However, from a code auditing and program comprehension perspective, our tool still provides valuable information to help auditors or security experts to identify flaws with regarding to specified security concerns.

5. Related Works

Existing research on applying Description Logics or formal ontology in software engineering have been addressed in early works of the LaSSIE [15] and CBMS [16] systems. Compared with our approach, these systems are however much more restricted by the expressiveness of their underlying ontology languages. In addition, these systems also lack the support of optimized DL reasoners, such as Racer in our case.

Unlike previous works that utilize only informal ontologies for tool integration [6] and software artifact organization [10], a *formal ontology* allows us to bring *automated reasoning* through DL theorem provers to the field of program comprehension, which is a significant improvement for the software engineering.

Various static analysis approaches for detecting security vulnerabilities have been reviewed in [16], ranging from lexical, syntactical, and even binary code

flaw searching tools to more advanced model verification approaches. Techniques for source code querying and searching are typically limited by their expressiveness and do not provide reasoning capabilities. Other static analysis approaches include program verifiers that are normally very expensive and difficult to use [14]. Our approach benefits from both the expressiveness of the ontology language and the reasoning capabilities of reasoner. The ease-of-use issue is also addressed by providing a scriptable and flexible query language. Our approach distinguishes itself from other techniques by facilitating a representation that attempts to match closely the mental model a programmer creates during a comprehension task with the underlying source code model used to facilitate tool support.

6. Conclusion and Future Work

During security analysis of a complex system, rigorous code review can only reduce the number of flaws, not eliminate every single one [15]. Currently, programmers are lacking tool support that not only can detect software vulnerabilities but also facilitate reasoning about their specific security concerns. In this paper, we have presented a novel approach in which source code under review is represented by an ontology model, and security concerns can be specified as part of this ontology. Through integration with a state-of-the-art ontology reasoner, security flaws can be identified and more implicit facts concerning the flaws can be derived. It is in particular the flexibility and adaptability of our approach that support an iterative comprehension process in which security concerns in the ontology can be further enriched and reused later for specific tasks.

Several limitations of our ontology-based approach are also discussed, including the missing support for the representation of temporal properties of the domain and potential performance issue on very complex queries. These drawbacks will be addressed in our future work. We also plan to provide a more comprehensive set of predefined queries to capture knowledge of security experts.

REFERENCES

[1] B. Shneiderman, “Software Psychology: Human Factors in Computer and Information Systems”. Winthrop Publishers Inc. 1980.

[2] R. Brooks, “Towards a theory of the comprehension of computer programs”. International Journal of Man-Machine Studies, 18:543-554, 1983.

[3] A. von Mayrhauser and A. M. Vans, “Program comprehension during software maintenance and evolution”. IEEE Computer, pp 44-55, Aug. 1995.

[4] S. Letovsky, “Cognitive processes in program comprehension”. In Empirical Studies of Programmers, pp58-79, Ablex Publishing Corp. 1986.

[5] M. Weiser, “Program Slicing”, IEEE Transactions on Software Engineering, 10(4):352-357, June 1984.

[6] D. Jin and J. R. Cordy. “A Service Sharing Approach to Integrating Program Comprehension Tools”. In Proceedings of the European Software Engineering Conference, Helsinki, Finland, 2003.

[7] M. Shaw and D. Garlan, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall Publisher, 1996.

[8] J. Ebert, B. Kullbach, and A. Winter. “GraX – An Interchange Format for Reengineering Tools”. In Proc. Of the 6th Working Conference on Reverse Engineering, 1999.

[9] P. N. Johnson-Laird. “Mental models: towards a cognitive science of language, inference, and consciousness”. Cambridge, Mass. : Harvard University Press, 1983.

[10] T. R. Gruber. “Toward principles for the design of ontologies used for knowledge sharing”. Presented at the Padua workshop on Formal Ontology, 1993.

[11] V. Haarslev and R. Möller, “Description of the RACER System and its Applications”, In proceedings of the International Workshop on Description Logics, Stanford, USA, 2001.

[12] F. Baader et al., “The Description Logic Handbook”, Cambridge University Press, 2003.

[13] R. Möller and V. Haarslev, “Description Logics for the Semantic Web: Racer as a Basis for Building Agent Systems”, In: KI – Zeitschrift für Künstliche Intelligenz (special issue on Semantic Web), No.3, July 2003.

[14] D. Evans and D. Larochelle, “Improving Security Using Extensible Lightweight Static Analysis”, IEEE Software, Vol 19, No. 1, Jan/Feb 2002.

[15] U. Lindqvist and E. Jonsson, “A Map of Security Risks Associated with Using COTS”, IEEE Computer, Vol 31, Issue 6, June 1998.

[16] B. Chess and G. McGraw, “Static Analysis for Security”. IEEE Security & Privacy, Vol2, No. 6, Nov 2004.

[17] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, “LaSSIE: a Knowledge-based Software Information System”, Communications of the ACM, 34(5):36–49, 1991.

[18] C. Welty, “Augmenting Abstract Syntax Trees for Program Understanding”, Proceedings of The 1997 International Conference on Automated Software Engineering. IEEE Computer Society Press. P. 126-133. November, 1997.

[19] V. Haarslev, R. Möller, and M. Wessel, “Querying the Semantic Web with Racer + nRQL”, In Proc. of the KI-2004 International Workshop on Applications of Description Logics (ADL'04), Ulm, Germany, September 24, 2004.

[20] OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, URL: http://www.w3.org/TR/owl-ref/