Semantic Web-based Source Code Search

Iman Keivanloo[1], Laleh Roostapour[1], Philipp Schugerl[1], Juergen Rilling[1]

[1] Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada
{i_keiv,l_roost, p_schuge, rilling}@cse.concordia.ca

Abstract. The ability to search for source code on the Internet has proven to be essential for many common software development and maintenance tasks. However, available code search engines are typically limited to lexical searches and do not take in consideration the underlying semantics of source code such as the program structure or language. Especially object-oriented source code, which includes inheritance and polymorphism, is often not modeled to a level in which it can be queried sufficiently. In this paper, we present a Semantic Web-based approach to source code search that uses ontologies to model and connect source code fragments extracted from repositories on the Internet. This modeling approach allows us to reason and search across project boundaries while dealing with incomplete knowledge and ambiguities. In comparison with other search tools, we incrementally build our knowledge base without the need to re-visit fragments or compile the source code. Our search engine is scalable and allows us to process and query a large code base collected from open-source projects.

Keywords: Semantic Web; source code; search engine; ontology

1 Introduction

Today's software development tools perform reasonably well in providing search facilities within the boundaries of the compilable components of a software system under development. However, the need for similar powerful search capabilities on source code available outside this scope, across project boundaries and the Internet, has so far often been neglected. While there exists some work on source code search in general [1,2], only limited research has focused on source code search on the Internet [3,4,5]. Internet-scale source code search differs from traditional search methods, due to (1) the heterogeneity of its resources from where information has to be extracted, (2) the problem of scalability due to the large amount of information that has to be searched and (3) the incomplete data and implicit and explicit dependencies among code segments [6].

The Semantic Web has not yet played the key role in improving Web search due to the unavailability of information resources that support easy automated population of the necessary knowledge bases. Nevertheless, its core technologies pave the way for a better searchable and semantic rich Internet in the near future. In contrast to arbitrary content available in natural language, the automatic population of source code to Semantic Web knowledge bases is relatively straight-forward since data is already represented in a structured way [7]. Recent work has shown that existing Semantic Web approaches can provide online search services for large amounts of data such as DBpedia which covers around 500 million triples [8].

In our research, we introduce a new approach in which we model source code using Description Logic (DL) and use Semantic Web reasoners to complete or deal with missing knowledge. Furthermore, we discuss how our search engine can handle large amounts of data typically found in the Internet. We first identify usage scenarios for source code search and classify the type of queries such scenarios should support. We then show how we can address each type of query and how our results differ from traditional code search engines. Semantic Web reasoners allow for complex queries to support source code searches involving transitive closure, unqualified name resolution and dynamic code analysis. We have implemented a proof of concept implementation that has been integrated within our existing SE-Advisor project [9].

The Semantic Web provides the enabling technologies that allow us to analyze and search for certain semantics in source code and provides the means to deal with transitive closure and incremental data analysis. Since repositories contain both the knowledge and data required for automated reasoning, new information can be incrementally added. Furthermore, the Open World Assumption (*OWA*) has various applications in this global search domain such as “Identify all classes that are not a subtype of a specified interface”. Considering possibly incomplete data, the OWA must hold to avoid any invalid result for such queries.

Our research is significant for several reasons: (1) We model source code and programming language specifics in OWL with the specific usage scenario of source code search and show how this impacts our ontology design. (2) We provide a scalable and incremental approach to build a source code knowledge base that is able to deal with incomplete information and can be used by web crawlers. (3) We use Semantic Web reasoners to complete missing and consolidate ambiguous information. (4) We illustrate how a semantic source code search engine can benefit software developers/maintainers during typical maintenance tasks.

The remainder of this paper is organized as follows. Section 2 defines what is covered by our Semantic Web-based source code search and introduces several usage scenarios and problems with traditional search engines. In Section 3, we first describe the architecture of our search engine and then pick up the previously described usage scenarios to discuss our ontology design in detail. Related work is discussed in Section 4 and Section 5 concludes with a summary of our approach and discusses future work.

2 Semantic Web-based Source Code Search

Different definitions exist as what constitutes a large scale source code search. In [7] the term *Internet-scale Code Search* was introduced and defined as the process of searching over a large number of open source projects and other complementary information resources such as, web pages and wiki pages that include source code. In [10] the term *Next-Generation Code Search* was presented to describe a source code search that considers structural code information and provides services to support other software engineering activities. In the context of our research we combine and extend the functionalities covered by these existing source code search definitions by three major factors: (1) The ability to search over a large amount of source code such as found on the Internet, (2) fine-grained structural search capabilities and dynamic code analysis and (3) the support for incomplete source code fragments. Code fragments, in this work, are defined as physically separated pieces of code (e.g. a class file or a few lines of code within a web page) which are syntactically correct and have a logical dependency to other code fragments.

In this paper, we are interested in improving both the completeness and precision of search results compared to existing source code engines available on the Web. It has to be pointed out, that dealing with source code fragments is inherently more difficult than source code search within a project that can be build (has all source code available e.g., within Eclipse). Dependencies among code fragments (e.g. library dependency represented by import statements in Java) need to be resolved without holding any assumption about the order and the availability of other fragments and without compromising the scalability of the engine. This is one of the main challenges of source code search on the Internet and a motivating start point for our research. In what follows we will discuss the type of source code search scenarios that should be supported.

2.1 Usage Scenarios

Source code search in the Internet is motivated by both, the possibility to re-use high quality code fragments from open source projects as well as gaining a general understanding of a piece of code (code comprehension). A typical search scenario is the identification of code fragments that describe how a class is instantiated. While this might seem trivial, today's complex software designs and layers of libraries that build on each other make the search for correct parameters a rather regular activity. For example, the new object could be instantiated by calling several methods of a library in a predefined order specified by the design of the target software. To be able to support a developer in such scenarios the engine must extract different types of information from source code. This following section discusses these types of information.

Existing research in the source code search domain [4,7,11] and related research in regards to maintenance tasks [1,2] was able to identify a limited number of frequently asked basic core queries. In what follows, we provide a categorization of these core queries into three categories based on their search requirements and various query examples.

Structural Analysis. Queries that focus on structural aspects of available source code typically provide information available at the annotated AST (Abstract Syntax Tree) level. Examples are queries related to method definitions, variables of a specific type and specific scope or searches to find all packages imported by a particular class. Even with this category being the simplest form of queries, there are cases in which many existing publicly available search engines will fail to provide precise results.

```
import packageA.*;
public class ClassX {
    TargetType var;
}
```

Fig. 1. The visibility and fully qualified type name of the `TargetType` is determined by the programming language and underlying semantics. The ontology and reasoner must handle such ambiguity.

Given the query *"Find classes that have a variable of a type packageA.TargetType"* and the code fragments shown in Fig. 1, most source code search engines will fail to provide results. We chose this rather primitive motivating example to demonstrate the limitation of search engines that do not analyze the semantics of source code and only perform a lexical search.

In this particular scenario the search engine has to be aware of programming language semantics (Java uses package visibility as the default visibility) and must be able construct the fully qualified type name out of the import statement in the beginning of the file. Also note how in the example the `import packageA.*` statement makes it ambiguous to resolve the type of `TargetType` as it could be part of the package `packageA` or the current package. This is further referred to as the *unqualified name resolution* problem.

For a more complex structural analysis it is also required that search engine can handle transitive closures as part of a search. An example for a transitive search is the following involving object-oriented inheritance structures. Suppose there are three classes of *A*, *B* and *C* and an instance *x* which belongs to *C*. *A* is the super-type of *B* and *B* is super-type of *C*. In this case, if a user searches for those variables that belong to *A*, *x* must be considered as one of the answers. Similar queries could involve class level information involving all sub-types or super-types of a specific class. During program comprehension, an interesting application for this static analysis is the ability to return all package dependencies that a specified class/package depends on (including imports and imported by dependencies).

Dynamic Code Analysis. Much of the encoded dynamics available from source code fragments are not identifiable by traditional search engines. While a simple textual search might reveal the usage of a static type, such search engines do not consider the dynamic type of a variable. For example, if a chained method call `method1().method2()` is used, the return type of `method1` is never mentioned explicitly such as the given example in Fig. 2. In object oriented programming languages this problem becomes even more complex due to polymorphism. In order for a search engine to be able to answer this type of questions, it is necessary to analyze the program flow prior to returning the results. Dynamic code analysis represents a key challenge by requiring e.g. method interfaces to be matched between method callers and callees. Queries like *"Identify all uses of a method"* are a common [1] and can only be answered correctly by considering the program flow.

```
ClassX var;
var = new ClassY();
var.method1().method2();
```

Fig. 2. Example source code which benefits from a dynamic code analysis. An instance of `var` is defined as a type `ClassX` and the method `method1` is called on the variable, its dynamic type being `ClassY`. Also note how the class of `method2` cannot be determined with a simple regular expression (lexical search).

A semantic source code search engine must support some form of transitive resolving mechanism to provide more complete and precise results for method invocations. Obtaining complete results from a structural analysis is a non-trivial and often impossible task [12] but we do expect that the applied techniques should provide improved precision compared to traditional search engines.

Metadata Analysis. Common to all previous query categories is that they operate on source code as the sole information source. For metadata queries additional information that is not directly available in the source code, such as project information or related artifacts, is considered. Such queries will be typically combined with other query categories to restrict their result sets. A typical query could be *"Identify all source code belonging to a certain project"* or *"List source code available under the GPL license"*.

3 Semantic Web-based Search Infrastructure

Our Semantic Web application is mostly using standardized components, such as the persistent storage, reasoners and the Sparql-Endpoint. The main ideas and concepts introduced in this paper are realized in the *ontologizer* component. Fig. 3 shows a high-level overview of all components.

We distinguish two forms of reasoners due to the type of inference required: DL and light-weight reasoners. We consider those reasoners that do not fully support OWL-Lite (the simplest subset of OWL 1) as light-weight reasoners as discussed in [13]. A DL reasoner is responsible for complex analysis such as required by our dynamic code analysis. Light-weight reasoners cover transitive closure and classification tasks required by the structural or metadata analysis. While light weight reasoners typically are attached to the RDF repository, it is reasonable to detach a DL reasoner and only apply it to filtered data from the repository in order to cope with scalability issues. Based on the input query, the engine must select the appropriate reasoner.

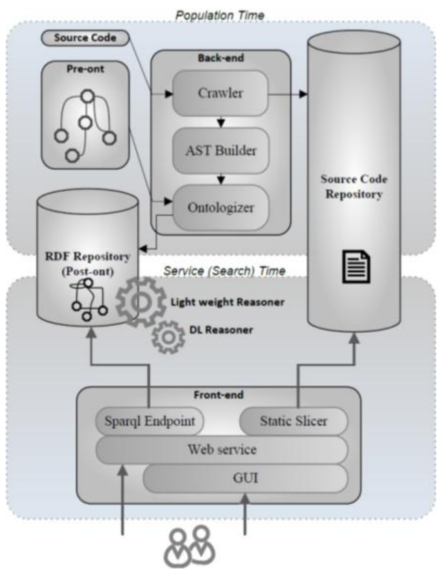


Fig. 3. The high-level schema of our Semantic Web-based source code search engine. The described core infrastructure does not yet provide additional processing such as result ranking or data mining.

The architecture provides two approaches for executing source code queries. The GUI is used to execute Sparql queries directly on the repository from an Eclipse plug-in while the web service provides programmatic access to the data. A source code slicer is required for presentation of the cached results and to select those lines of code that are related to the answer statement (similar to the text preview shown in Google searches). Other major components that are part of our infrastructure are: an AST builder, a Web crawler and the source code repository itself. The crawler connects to source code resources on the Web and populates the source code repository. This repository contains source code in its original format. The AST builder analyzes the fragments available in the source code repository and creates a separate AST for each source code fragment. This approach is based on the no-ordering assumption discussed earlier in this paper. To avoid unnecessary resource consumption in terms of processing and memory, we created an optimized ontology for the architecture. The core ontology pre-ont, is the basis for the run-time populated ontology post-ont. The output ontologies will be saved into RDF repository component.

3.1 Ontology Design Discussion

The Semantic Web provides several languages for knowledge representation and sharing. During the language selection in the ontology design phase, the designer must consider both functional and nonfunctional requirements, a tradeoff between expressivity and computational performance. Moreover, each language provides a set of properties with possible side effects of which the designer must be aware to avoid any unexpected result from both a logical and computational point of view [14]. In the following, properties from Semantic Web languages for our pre-ont and post-ont ontologies are discussed in detail.

The metadata analysis of source code requires *rdfs:Class*, *rdfs:subClassOf* and *rdf:type* [15,16] properties to build e.g. a taxonomy of projects. Therefore RDFS is sufficient to answer queries related to this kind of analysis. Considering object-oriented languages, the two most coarse-grained properties are package and class definitions for which a hierarchy needs to be modeled. Moreover, each class has some property and method definitions including their signatures and local variables for which both *Is-A* and *Part-Of* relationships need to be modeled. Modeling object-oriented source code is a kind of metamodeling, as defined in [17], which is not easy to achieve considering both expressivity and performance and may lead to undecidability [1,17].

There exist some specific types of OWL that deal with metamodeling problem such as OWL FA [18]. However in this paper we focus on the challenges of scalable metamodeling of source code (specifically object-oriented source code) from an implementation point of view. Our goal is to satisfy the requirements of such search engines by a limited subset of Semantic Web languages. In the following three possible metamodeling approaches for object oriented programming languages using current Semantic Web languages are discussed:

- Tbox-based metamodeling (i.e. mainly concepts, roles and possible restrictions)
- Mixed metamodeling (Tbox and Abox)
- Abox-based metamodeling (i.e. mainly individuals, roles and restrictions)

The underlying rationale here is to avoid the possible overlap between concept and individual which may lead to OWL Full [13]. The regular way to model a Is-A relationship in the Semantic Web is the subtype property (*rdfs:subClassOf)*. However, writing logical constructions for reasoning on Part-Of relation is more challenging to model since either an *existential quantification* or the *owl:hasValue* property may be used in addition to available roles. Contrary to the two first solutions, the last approach does not use concept hierarchies for Is-A modeling and simply uses roles to model all kind of relations.

Since Semantic Web reasoners are mostly optimized for Tbox reasoning due to their historical background, the two first approaches could be considered more reasonable than the last one. However, Tbox metamodeling has deficiencies in the source code search domain if we want to avoid the undecidability problem. Some of the solutions lead to models which are more complicated than our requirements (for example modeling a class declaration and implementation using two concept and individual entities respectively). While complex models may support more details about the domain, they also require more space to store and time to process affecting the scalability of our approach. Furthermore, the core model must be able to support requirements defined by other groups of queries or client-side applications. To avoid any unnecessary modeling complexity, one solution is to use Tbox for both class declaration and implementation level modeling but there are some applications for source code search and specifically for software maintenance which cannot be satisfied. For example, it is not possible to express some restrictions on the super-types of an entity using Description Logic. An alternative solution is the Abox-based approach. It satisfies the previous requirements and requires fewer triples to be modeled by omitting unnecessary details (i.e. differentiation between the type declaration and implementation). The following language properties are required for Abox-based metamodeling in this domain:

- Concept: For domain modeling, e.g., package, class and method concepts.
- Individual: For entities such as a specific Java class, a local variable, etc.
- Role (owl:ObjectProperty): For any kind of relationship (incl. Part-Of and Is-A)

So far we discussed how metamodeling could be implemented for the structural analysis but failed to discuss the *unqualified name resolution* mentioned earlier. Two solutions have been designed in this paper to address the resolution problem. The first approach is a verbose modeling approach which is called *loose unqualified name resolution*. The other approach uses a DL reasoner to resolve the names accurately and is further referred to as *full unqualified name resolution*. It is important to mention that there is no contradiction between them and that they are actually complementary. *Loose unqualified name resolution* can be applied by default and then, depending on the available processing resources, a DL reasoner may be used to make the result of the first approach more accurate. The reason for this separation is the computational overhead associated with *full unqualified name resolution*. *Loose unqualified name resolution* uses similar properties as in the metamodeling section while *full unqualified name resolution* uses *rdfs:subClassOf, owl:hasValue* and *owl:equivalentClass properties*. The rationale of these two approaches is described in next section as part of our description of the ontology.

Although OWL offers many interesting properties that can be used here such as complement, union, disjoint, inverse property and universal quantification, we have decided not to include them in our core ontology because of their computational overhead. This decision may restrict the expressivity of the ontology but has been taken out of our experience with very large ontologies which is supported by the discussion in [19]. The core ontology is modeled only based on the specified properties for each analysis. Sophisticated related tasks could be implemented at the client-side using any other properties from Semantic Web languages. Union and inverse properties are valuable especially for the software comprehension and maintenance domain [1]. To address them in the core level, query language facilities can be used. In addition, the inverse property can be represented explicitly within the populated ontology. Our resulting ontology language is a subset of OWL-Lite except for dynamic code analysis and the optional *unqualified name resolution* approach that is reasoner-based since these two sections require the *owl:hasValue property* that leads to OWL-DL. Considering OWL 2.0, the OWL 2 EL [19] covers all the required properties which also supports the design decisions that have been made regarding the exclusion of disjunction, inverse role and negation for computational performance reasons.

3.2 Source Code Ontology

Our ontology includes some very high-level concepts based on the two core source code ontologies found in [9,20]. Based on the high-level concepts, other details have been added to support previously discussed usage scenarios. The basic ontology with populated samples is available online[1]. The populated samples are based on the given techniques discussed in this section. In the following, those parts that are more interesting and related to this paper are discussed.

Type hierarchy. We use individuals to represent types (i.e. class and interfaces in object-oriented language paradigms) and their implementation details, according to the A-Box metamodeling approach discussed in the previous section. To simulate the transitive closure that might exist between these types, the OWL transitivity property is applied on the relevant roles. As a result, the object-oriented classes defined in the source code are represented as individuals of *Class*/*Interface* concept. Variables are also defined as individuals belonging to the *Variable* concept. In order to be able to represent inheritance hierarchies and the relationships between the instances and the classes, we use a single transitive role which is *hasType*. Fig. 4 shows the very high-level overview of the ontology focusing on metadata queries.

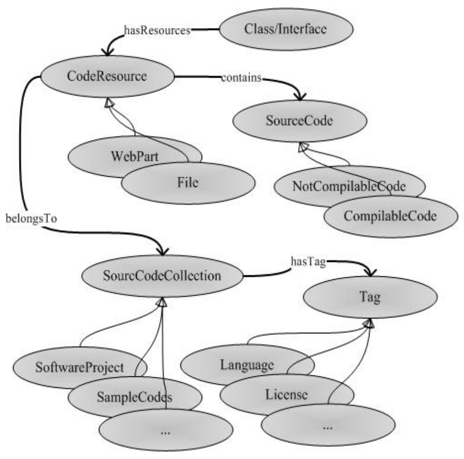


Fig. 4. High-level concepts of our ontology related to the metadata section that is mostly being used to restrict query results.

Package hierarchy and dependencies. Packages or namespaces are a way to categorize source code in programming languages. There are two kinds of relationship that can be distinguished: The ownership relation among classes/packages themselves and the dependency relationships between classes and packages. The ownership relationship is represented using *belongsTo* and *hasSubPackage* roles. These roles allow a user to search for packages and review the classes belonging to a current package or all its sub packages (by using transitive closure). The dependency relationship is modeled through the *imports* role and establishes the relationship to both classes and packages. Considering software maintenance requirements, *uses* is one role representing a typical dependency among software entities. A typical application example in the software maintenance domain would be a Semantic Web application that requires a set of related information with

1 http://aseg.cs.concordia.ca/ontology/#sourcecode

respect to a target entity [2]. Additional support for other maintenance applications is provided by another version of the *uses* role in the ontology which has a transitivity property.

Class details. Fig. 5 shows the corresponding concepts and roles required for the source code implementation representation. The ontologizer creates the post-ont ontology by adding new individuals based on the predefined concepts in pre-ont for each code fragment (e.g. a Java class). Following this approach, the roles in the ontology are used to represent the implementation details for the available source code. The method signatures are modeled using a series of roles like *hasFirstParam* and *hasSecondParam*. Although there are other approaches to model the input parameters of a method using Semantic Web languages, the proposed solution is sufficient for our usage scenarios.

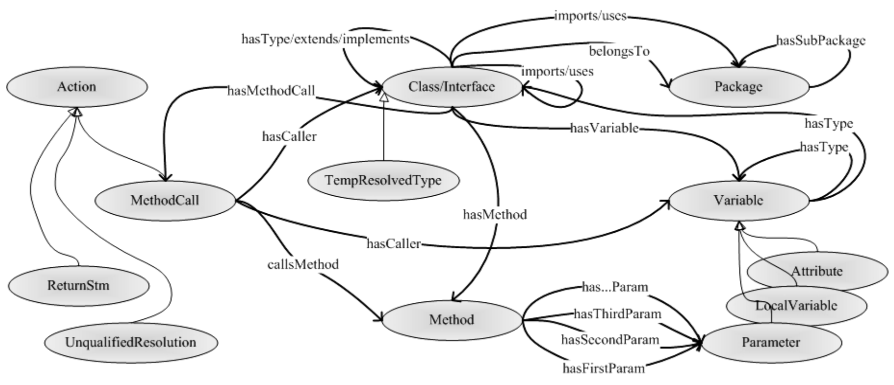


Fig. 5. Concepts and roles that are part of the discussed ontology design. There exist some less important detailed entities in the ontology such as *extends*, *implements* and *hasDefiendType* which are not shown in the overview.

Methodcalls and return statements. At the source code level, method call statements are of significance. We represent method calls using individuals of the *MethodCall* concept defined in the ontology. Each class or method is an owner of such individuals using the *hasMethodCall* role. However, each method call statement has two important pieces of information associated, the method and the caller. These two are required in order to perform a dynamic code analysis, especially for object-oriented code since the caller type determines the actual callee (i.e. method implementation). According to the challenges mentioned in the previous sections, the ontologizer cannot specify the receiver at post-ont creation time in most cases. Consequently, another component is needed to resolve this problem. The ontologizer only creates an individual of the *MethodCall* concept and models the caller and callee information using *hasCaller* and *callsMethod* roles. If a method has input parameters, these parameters must be modeled using the available roles to support method overloading.

The caller individual can be a class, a variable or even another method call with a type that is being determined by the return statement of the method being called. The *ReturnStm* concept is a key concept for solving many of our modeling challenges. The ontologizer is responsible to create a new sub concept from *ReturnStm* for each return statement in the code. Note that this is the first step where the ontologizer must create concepts instead of individuals. Each new sub concept contains two pieces of information which are an OWL equivalent restriction and an OWL subclass restriction. The equivalent restriction is filled using the pattern (presented using Protégé [21] OWL terminology): `hasCaller some (hasType value [Owner class of the method]) and callMethod value [Method specification]`. In addition, the subclass restriction will be populated using: `hasType value [Corresponding individual for the right hand side of the return statement]`. Fig. 6 presents a sample for this pattern. It states that whoever calls the method and *hasType* of the specified class, must have the type of the right hand side of the return statement. It is the reasoner's responsibility to infer the method call types using the added return statements.

The key advantage of this solution, compared to traditional ones used in object-oriented source code analysis, is that source code can be analyzed in no order and in one-pass as it is required by a search engine crawler that visits one web-site at a time without prior knowledge of the web structure. This also allows the parallelization of crawling and increases the scalability of our infrastructure.

```
<rdfs:subClassOf rdf:resource="http://...#return"/>
<rdfs:subClassOf>
     <owl:Restriction>
            <owl:onProperty rdf:resource="http://...#hasType"/>
            <owl:hasValue rdf:resource=" http://...#v34451"/>
     </owl:Restriction>
</rdfs:subClassOf>
```

Fig. 6. A sample populated partial ontology for a return statement in Java code.

Unqualified name resolution. Based on our previous discussion, one of the main issues in source code processing is the need to handle unqualified names that may occur. A user must be able to search by the qualified name of a class instead of its unqualified name to increase the result accuracy. Considering that we have to deal with typically incomplete data, this requirement, while a relatively trivial problem for local source code searches, is inherently more challenging for Internet-based source code search approaches. As mentioned in the previous section, we introduce two approaches to address this problem: *loose unqualified name resolution* and *full unqualified name resolution* (which requires a DL reasoner).

Loose unqualified name resolution tends to predict all the possible qualified names for each unqualified name instance based on local information available for each segment. Given this approach, for each potential result, one individual of type *TempResolvedType* is instantiated. While it might seem that by using this approach we are adding invalid information to our repository, this is not the case, since we separate this temporarily information of resolved types from the actual Class or Interface information. This allows us, even without using the second approach, which is complementary to this one, to answer backward queries correctly. For example a query that asks for *"Variables that belong to class java.io.File"*. Since such queries use a valid qualified name as the input they will never be affected by some of the potentially invalid qualified names inserted temporarily into the repository.

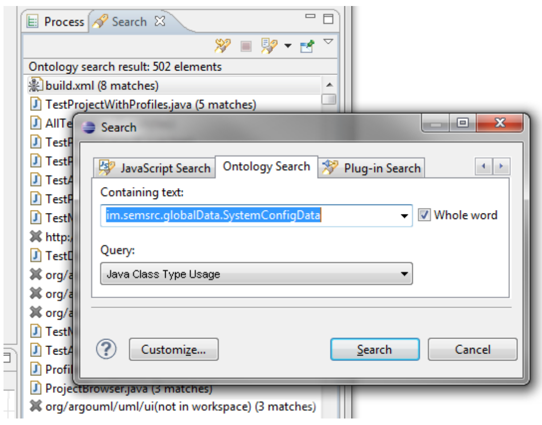
Full unqualified name resolution is designed using inter-segment inferences. In this reasoner-based approach, a sub concept of *UnqualifiedResolution* is created for each qualified name. Then, using OWL equivalent and subclass restrictions, it is stated that there is an individual belonging to a *TempResolvedType* concept. If it has the specified qualified name then it actually belongs to Class/Interface concept. In this way once a type resolved successfully then all the loose resolutions will be resolved using the reasoner. The interesting aspect of this solution is its capability of resolving names even if the actual type implementation has never been analyzed prior by the ontologizer as long as it has occurred at least once as a qualified name through the whole repository.

3.3 Implementation

We developed a Java application as a proof of concept based on the discussed architecture and ontology. Since one of the main applications is in software maintenance and comprehension, it is also integrated with our current SE-Advisor, an ongoing project on semantic modeling of software artifacts [9], process modeling, and semantic analysis [22] to support the evolution and maintenance of software systems. A snapshot of our current search plug-in is shown in Fig. 7. The SE-Advisor provides a user not only access to source code but also to the related information extracted from other resources such as bug trackers.

Most of our tool chain is adopted from available software components. Javaparser [23] is used for the AST construction and ontology manipulation tasks respectively. RacerPro [24], is used to run Sparql queries against the populated post-ont ontology. While advanced users may run their customized query directly using the provided web services or download the knowledge base for further processing, a set of pre-defined core functionalities are provided by our GUI. Using the GUI there is no need to write any kind of query which is helpful for those users without prior knowledge of ontology query languages.

While we previously discussed all the pre-ont and post-ont details, we have not yet addressed the way identifiers (e.g. URI) must be created during the post-ont population. There are three kinds of identifiers: Some must be created randomly since they have to be unique (considering the complete knowledge base) such as return statements. Others, such as types, must have a qualified name inside the post-ont ontology. Methods, on the other hand, are independent from the owner classes according to the ontology design. This approach ensures that entities will have consistent identifiers even when they are produced completely separately. This also enables the parallel processing of web content through multiple crawlers. Although an identical behaviour could be implemented using *owl:sameAs* in a post-production and integration step, we decided not to use it because of its memory and processing overhead.



Ontology search result: 502 elements
build.xml (8 matches)
TestProjectWithProfiles.java (5 matches)
ProjectBrowser.java (3 matches)
org/argouml/uml/ui(not in workspace) (3 matches)

Containing text: im.semsrc.globalData.SystemConfigData
Whole word [✓]
Query: Java Class Type Usage

Fig. 7. Eclipse plug-in integration that enables code search directly out of the IDE. Using this form of global source code search, a user may search for e.g., all usages of a certain type, not only locally but globally in all available source code on the Internet. Resources that are within the currently opened Eclipse projects are automatically linked to the results so that double-clicking a file opens the local copy. Other results (marked X) are opened in the internal Eclipse web-browser from the source code repository.

Our current implementation does not support queries related to versioning. That is, if there are two versions of a specific source code fragment, the engine considers them as separated entities in the repository and also within the ontology. From an ontological point of view, basic versioning knowledge could be modeled easily but the major roadblock is the required input data. Extracting versioning information from source code found on the Internet is a challenging and ongoing research topic in source code analysis which is out of scope of this paper. However, if such information is extracted (or is added manually) the underlying architecture of our search engine can support versioning queries using the chosen expressivity.

4 Related Work

Description Logics and the Semantic Web have been used to represent a software project formally. Although the infrastructure of development environments is not based on DL, the conversion mostly has been done for software maintenance purposes. One of the early applications of DL surfaced in the software engineering domain in program comprehension and maintenance. This kind of formalism is used for two main purposes which are domain knowledge maintenance and source code comprehension. LaSSIE [25] is one the successful applications of the former. By modeling the domain knowledge the maintainer was presented with the ability to locate related information about a specific concept. As an example for the latter, [2] used DL for source code modeling to support program comprehension considering specifically the ability of backward trace (i.e. inverse roles), path tracing (i.e. having all related information in one place) and classification. Recently, those three applications also have been explored using Semantic Web languages. In [26], ontologies are used to represent source code formally and to find security related vulnerabilities. It has been implemented by defining those patterns formally and using a reasoner for classification. Another sample of Semantic Web applications in Software Engineering is presented in [9] and mainly focuses on representing the links required for software maintenance tasks. However, all of these works mainly focus on establishing links between source code and other artifacts or represent localized models that only focus on project specific artifact searches.

Previous work on Internet-scale code search has focused on different search aspects. Reiss [27] emphasized on a source code search approach that returns a complete working code segment rather than a set of (often partial) code fragments. Following this approach, the S6 project [27] requires users to specify test cases and a list of keywords, which are used to retrieve the closest match to a given piece of code. As mentioned in [27], a general challenge is to derive strategies on how to avoid an exponential number of results. Within our work we address this challenge by applying our two step processing strategy. Similar to the S6, the PARSEWeb project [28] tries to find examples for complex object creation by modeling them as a method invocation sequence problem. PARSEWeb's main contribution is that it also supports some source code analysis during the search by resolving the receiver of method calls. However, since PARSEWeb is using an external, traditional code search engine it has not always access to all source code segments and their dependencies and therefore will not always be able to retrieve correct results.

Assieme [4] is another source code search project which has two interesting contributions: (1) The ability to find different kinds of resources related to a given query and (2) being able to resolve possible implicit references. The intention to include implicit reference resolution as part of the query engine is similar to our *unqualified name resolution*. However, the Assieme project uses a complier based approach which has two major drawbacks compared to the work introduced in this paper. Firstly, it requires that each source code has to be compilable which makes it not suitable in cases when only partial or incomplete code is available. Secondly, the compilation itself results in an additional cost and performance overhead.

Other source code engines use repositories based on relational models to store and query information. Sourcerer is a relational model based repository [10] and search engine [3] that supports fine-grained structural queries using textual and structural source code information. Strathcona [5] is a local search engine that supports searches for code samples. During the search, it relies on structural data stored in a relational database. The interesting feature of Strathcona is its ability to restrict a method call query by the receiver type. However, in its current implementation it does not support hierarchical transitivity.

In addition, there are some other publicly available code search engines on the Web such as Google Code Search [29], Koders [30], Codase [31], Krugle [32] and JExample [33]. While these engines cover different open source project and source codes, they do not provide results for most of the usage scenarios listed in this paper. A main reason for this deficiency is that they do not apply enough syntax and semantic-based code analysis.

At last, there has been a significant body of work on local search techniques. While they are focusing on project specific queries, our objectives are completely different in respect to openness where queries can be geared towards a non-complete search space across project and even organizational boundaries.

5 Conclusion and Future Work

In this paper, a novel Semantic-Web enabled source code search is introduced. Our approach does not only scale in regards to the large amount of source code available on the Internet but is also optimized to work with web crawlers to incrementally build a semantic knowledge base. Our approach is scalable as we support one-pass parsing and do not make assumptions on the order of fragments analyzed. This makes our Semantic Web based approach unique among other currently adopted approaches for Internet scale source code search research. We have pointed out various usage scenarios of source code search in which traditional search engines fail to provide (complete) results and show how we out-perform them by incorporating source code semantics into the knowledge base. We have discussed our ontology design in regards to its strengths and weaknesses and pointed out common pitfalls in designing source code ontologies in general, and our highly optimized ontology for source code search in particular. Furthermore, we have analyzed various source code metamodeling approaches under the aspect of source code search to make our approach scalable.

As part of our future work, we plan to add the versioning information extracted from metadata to the repository and deal with contradictory information (e.g. through levels of trust). In addition, more detailed source code query capabilities will be added to our Eclipse plug-in. We also plan to make our tool available to the community in form of an open source source code search engine and are currently working on the web interface for this project.

References

1. Erdos, K., Sneed, H.M.: Partial comprehension of complex programs (enough to perform maintenance). In: 6th International Workshop on Program Comprehension (1998)
2. Welty, C.: Augmenting abstract syntax trees for program understanding. In: 12th IEEE International Conference Automated Software Engineering (1997)
3. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. In: 21h ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (2006)
4. Hoffmann, R., Fogarty, J.: Assieme: finding and leveraging implicit references in a web search interface for programmers. In: 20th annual ACM Symposium on User Interface Software and Technology (2007)
5. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: 5th International Conference on Software Engineering (2005)
6. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (1997)
7. Gallardo-Valencia, R.E., Sim S.E.: Internet-scale code search. In: 1st ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (2009)

8. Auer, S., Lehmann, J.: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In: European Semantic Web Conference (2007)
9. Rilling, J., Witte, R., Schuegerl, P., Charland, P.: Beyond Information Silos - an Omnipresent Approach To Software Evolution. J. Semantic Computing. 2, 431--468 (2008)
10. Bajracharya, S., Ossher, J.: Sourcerer: An internet-scale software repository. In: 1st ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (2009)
11. Reiss, S.P.: Specifying what to search for. In: In: 1st ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (2009)
12. Binkley, D.: Source Code Analysis: A Road Map. In: 7th Future of Software Engineering (2007)
13. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Semantics: Science, Services and Agents on the World Wide Web. 3, 15--182 (2005)
14. Graua, B.C., Horrocksa, I., Motika, B., Parsia, B., Patel-Schneiderc, P., Sattlerb, U.: OWL 2: The next step for OWL. J. Web Semantics: Science, Services and Agents on the World Wide Web. 6, 309--322 (2008)
15. RDF/XML Syntax Specification, http://www.w3.org/TR/REC-rdf-syntax, last visited January 2010.
16. RDF Schema, http://www.w3.org/TR/rdf-schema, last visited January 2010.
17. Motik, B.: On the Properties of Metamodeling in OWL. In: Internationa Semantic Web Conference (2005)
18. Pan, J.Z., Horrocks, I.: OWL FA: a metamodeling extension of OWL D. In: 15th International Conference on World Wide Web (2006)
19. OWL 2 Web Ontology Language Profiles, http://www.w3.org/TR/owl2-profiles, last visited January 2010.
20. Kiefer, C., Bernstein, A., Tappolet, J.: Analyzing Software with iSPARQL. In: 3rd International Workshop on Semantic Web Enabled Software Engineering (2007)
21. Protégé Ontology Editor and Knowledge Base Framework, http://protege.stanford.edu, last visited Januart 2010.
22. Meng, W.J., Rilling, J., Zhang, Y., Witte, R., Charland, P.: An ontological software comprehension process model. In: 3rd Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engeenirng (2006)
23. Javaparser - Java 1.5 Parser with AST generation and visitor support, http://code.google.com/p/javaparser, last visited January 2010.
24. Haarslev, V., Moller, R.: RACER System Description. In: International Joint Conference on Automated Reasoning. R. Goré, A. Leitsch, T. Nipkow (eds.). pp. 701--705. Springer-Verlag, Berlin (2001)
25. Devanbu, P., Brachman, R., Selfridge, P., Ballard, B.: LaSSIE: a knowledge-based software information system. In: 12th International Conference on Software Engineering (1990)
26. Yu, L., Zhou, J., Yi,Y., Li, P., Wang, Q.: Ontology Model-Based Static Analysis on Java Programs. In: 32nd Annual IEEE International Computer Software and Applications Conference (2008)
27. Reiss, S.P.: Semantics-based code search. In: 31st IEEE International Conference on Software Engineering (2009)
28. Thummalapenta S., Xie, T.: PARSEWeb: programmer assistant for reusing open source code on the web. In: *International Conference on Automated Software Engineering* (2007)
29. Google Code Search, http://www.google.com/codesearch, last visited January 2010.
30. Koders, http://www.koders.com, last visited January 2010.
31. Codase, http://www.codase.com, last visited January 2010.
32. Krugle, http://www.krugle.com, last visited January 2010.
33. JExamples, http://www.jexamples.com, last visited January 2010.