Component Search and Reuse: An Ontology-based Approach

Awny Alnusair and Tian Zhao
Department of Computer Science
University of Wisconsin-Milwaukee, USA
{alnusair, tzhao}@uwm.edu

Abstract

In order to realize the full potential of software reuse, effective search techniques are indeed essential. In this paper, we propose a semantic-based approach for identifying and retrieving relevant components from a reuse repository. This approach relies on building a knowledge base according to an ontology model that includes a source-code ontology, a component ontology, and a domain-specific ontology. Due to the indexing and knowledge population mechanisms we used, our proof-of-concept supports various kinds of search techniques. However, our experiments show evidence that only pure semantic search that exploits domain knowledge tends to improve precision. Based on a usability case study, we argue that semantic search is indeed usable and practical.

1. Introduction

Systematic software reuse enables developers to use existing software components for constructing new quality software systems. Thus, speeding up the development process and reducing costs and risks. Failure modes analysis of the reuse process shows that in order to be reused, a software component must be findable and certainly understandable [5]. On one hand, understanding the component's functionality as well as its relationships with other components in a reuse library is usually hampered due to the lack of quality descriptions of library services. On the other hand, finding suitable components is still a significant barrier for exploiting systematic software reuse.

In this paper, we present an approach for describing, retrieving, and exploring the various relationships among software components in object-oriented reuse libraries. In order to provide a formal and precise representation of library code, our approach relies on ontologizing software knowledge.

Ontologies provide means to explicitly describe concepts, objects, properties and other entities in a given application domain, and to represent the relationships that hold among these concepts. Due to their solid formal and reasoning foundation, ontologies can play an important role in domain engineering reuse; they can be used to structure and build a source-code knowledge base that can be used by software agents (e.g., search engine) and certainly serve as a basis for semantic queries [9]. Therefore, ontologies have been successfully used by various researchers to improve many aspects of the software engineering processes [12].

Towards this end, we have developed an ontology model that includes an enhanced software representation ontology. This ontology is further extended with additional component-specific knowledge and automatically populated with ontological instances representing various program elements of a given library. These instances are further annotated with respect to concepts from a domain-specific ontology.

Ontology-based component search is thus performed by the semantic matching of user requests expressed using terms from the domain ontology with component descriptions found in the populated knowledge-base. Furthermore, the knowledge population and indexing mechanisms used in our approach still allows searching the knowledge base using the most familiar keyword search. This is particularly useful when domain ontologies or semantic component annotations are lacking or incomplete. Users are thus able to retrieve components using purely semantic-based queries, keyword queries, type-based queries or a mixture of all.

2. Ontology model for component reuse

At the core of our ontology model for object-oriented component retrieval is a Source Code Representation Ontology (referred to afterwards as SCRO). This ontology provides a base model for capturing the relationships and dependencies among source-code artifacts. It models major concepts and features of object-oriented programs, including encapsulation, class and interface inheritance, method overloading, method overriding, and method signature information. SCRO's knowledge is represented using the

OWL-DL[1] ontology language. OWL is a web-based conceptual modelling language used for capturing relationship semantics among domain concepts, OWL-DL is a subset of OWL based on Description Logic (DL) and has desirable computational properties for automated reasoning systems. OWL-DLs reasoning support enables inferring additional knowledge and computing the classification hierarchy (subsumption reasoning). SCRO defines various OWL concepts that map directly to source-code elements and collectively represent the most important concepts found in object-oriented programs. Furthermore, SCRO defines various OWL object properties, datatype properties, and ontological axioms to represent the various relationships among ontological concepts. SCRO is precise, well documented, and designed with ontology reuse in mind. The availability of SCRO online [1], allows researchers to reuse or extend its representational components to support any semantic-based application that requires source-code knowledge.

2.1. Domain-specific ontology

Domain ontologies describe concepts and structures related to a particular domain (e.g. finance, shopping, medicine, graphics, or the object-oriented programs domain as specified in SCRO). In our approach to component retrieval, we use a domain ontology that conceptualizes each software library we need to reuse. This ontology provides a common vocabulary with unambiguous and conceptually sound terms that can be used to annotate software components. Annotations in this context serves two key purposes. Firstly, both software providers and users can communicate using a shared and common vocabulary provided by this ontology. Thus enabling a precise retrieval of API components. Secondly, it brings different perspectives to typical program comprehension tasks. Users can familiarize themselves with terminology and conceptual knowledge that is typically implicit in the problem domain.

To this end, we have developed a mini-ontology for data retrieval in the Semantic Web applications domain. This ontology (referred to afterwards as SWONTO) has been used during the evaluation of our approach (cf. Section 4) and serves as a proof of concept that component search can be significantly enhanced through the use of domain ontologies. A small fragment of the ontology's taxonomy is shown on the upper right pane of Figure 1 and the complete ontology is found online [1].

2.2. Component-specific ontology

In the context of component retrieval, we certainly need a profound semantic description of the component's inner working structure and its interrelationships with other components in a given domain. Therefore, we extends SCROs semantic representations of API structures and enrich it with additional component-specific descriptions required to uniquely identify and retrieve an API component. The result is a COMPonent REpresentation ontology (referred to afterwards as COMPRE). In addition to the concepts, axioms, and properties inherited from SCRO, COMPRE defines its own class hierarchy and relations for semantic component descriptions. For instance, the `Component` concept represents software components in general and subsumes other OWL classes that represents Java-specific components such as `Method`, `ClassType`, and `InterfaceType`. A fragment of the ontologys taxonomy is shown on the left pane of Figure 1 and the complete ontology can be examined online [1].

Moreover, COMPRE defines various ontological axioms and object properties that represent relationships among software components. These properties link components with their corresponding semantic descriptions specified in the domain-specific ontology. For example, `hasDomainInput` and `hasDomainOutput` and their corresponding inverse properties are used to annotate individual software components with domain terms representing the expected inputs and outputs of the component. Moreover, COMPRE defines the `dependsOn` symmetric object property that defines dependency relationship between components. `describedBy` is also defined to link a component to a domain concept that best describes the purpose or the nature of the component.

In addition to pure semantic search that is based on component annotations with respect to a domain ontology, COMPRE also defines various datatype properties to model other metadata about components. These properties are provided to enable metadata keyword queries that can be used when semantic annotations are lacking or incomplete. For instance, the `hasInputTerms` and the `hasOutputTerms` are used to assign meaningful keywords describing the component's input and its expected output.

2.3. Knowledge population

Once the ontology structure is specified, one next needs to populate the knowledge base with ontological instances that represent various ontological concepts and their corresponding relationships. Therefore, we have built a knowledge extractor subsystem for the Java programming language. Our subsystem performs a comprehensive parsing of the Java bytecode and captures every ontology concept that represents a source code element and generates instances of all ontological properties defined in our ontologies for those program elements. The generated semantic instances

[1] `http://www.w3.org/2004/OWL/`

are serialized using RDF [2]. RDF is web-based language suitable for describing resources and provides an extensible data model for representing machine-processable semantics of data. For each application framework parsed, we thus generate an RDF ontology that represents the instantiated knowledge base for the framework at hand. This knowledge base is managed by Jena [8], an open source Java framework for building Semantic Web applications.

The process of generating semantic instances for the concepts and relations specified in SCRO is completely automatic. However, the process of annotating components according to COMPRE's object properties is currently manual as it is the case for semantic annotations in general. Our tool though provides means for inserting these annotations directly into the knowledge-base, thus gradually building semantic descriptions for a particular API that can be shared, evolved, and reused by a community of users. On the other hand, metatdata modelled by COMPRE's datatype properties is generated automatically via direct parsing of the source-code. We thus capture and normalize method signatures, identifier names, source-code comments, and available Java annotations in order to obtain a meaningful keyword descriptions of components. These descriptions are lexically analyzed, stored, and indexed using the tokenization and indexing mechanisms provided by Apache Lucene [3], an open-source full-featured text search engine.

In the next section, we show how the knowledge generated using this knowledge extractor sub-system can be used for component search. For an extended discussion of our ontologies, complete knowledge population samples, we refer the reader to our ontologies website [1].

3. Ontological search

Listing 1 shows a partial RDF description obtained during the knowledge population phase for a Jena API method, the `create` method. This method belongs to the `QueryFactory` class and usually used to create a `Query` object given the specified input. This RDF description clearly captures the component's metadata at the semantic and syntactic level.

The underlying data structure of RDF is a labeled directed graph. Each node-arc-node in this graph represents a triple that consists of three parts, *subject*, *predicate* and *object*. Consider Listing 1 for example, the described method in this snippet, `create[..]`, is always the subject, ontology properties are predicates, and objects are either a resource, unlabeled node (blank node) or a literal value. For example, the first triple below uses a property from SCRO to assert that the method has an input parameter of type `String`. The second triple associates this parameter with few terms describing its purpose. The third triple, however, tags the same input parameter with a meaningful concept (`QueryText`) from the domain ontology. Thus, giving the parameter an agreed-upon and meaningful description other than terms or the semantically vague String type.

```
1. create[..] scro:hasInputType String
2. create[..] compre:hasInputTerms
              "query string"
3. create[..] compre:hasDomainInput
              [ a swonto:QueryText]
```

```
@base           <http:.../ontologies/kb.n3>
PREFIX scro:    <http:.../ontologies/scro.owl#>
PREFIX compre:  <http:.../ontologies/compre.owl#>
PREFIX swonto:  <http:.../ontologies/swonto.owl#>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>

<#QueryFactory.create[String,String,Syntax]>
 a scro:StaticMethod ;
 scro:hasInputType <#String> ;
 scro:hasInputType <#String> ;
 scro:hasInputType <#Syntax> ;
 scro:hasOutputType <#Query> ;
 scro:invokesMethod <#parse[Query,String,Syntax]>;
 scro:hasSignature "create[String,String,Syntax]";

 compre:describedBy    [ a swonto:QueryCreation];
 compre:hasDomainInput [ a swonto:QueryText];
 compre:hasDomainInput [ a swonto:URI];
 compre:hasDomainInput [ a swonto:QueryLanguageSy-
 ntax];
 compre:hasDomainOutput[ a swonto:ExtendedQuery];

 compre:hasInputTerms "query string ...";
 compre:hasInputTerms "base URI ...";
 compre:hasInputTerms "query syntax URI ...";
 compre:hasOutputTerms "query ...";
 dc:description "create query ...";
                    ....
```

Listing 1. RDF descriptor for an API method

This multi-faceted description of components enables four different types of queries against the knowledge base: *a*) type or signature-based queries; *b*) metadata keyword queries; *c*) pure semantic-based queries; or *d*) blended queries of the previous three types.

However, we focus the discussion on pure semantic-based queries that rely on domain-specific knowledge. Primarily, search techniques that rely on variations of keyword-based search suffer from synonymity and polysemic ambiguity that often lead to low recall and precision. On the other hand, signature matching techniques cannot distinguish between components that have the same signature but serve different purposes, e.g. using Jena API to `create` a new query vs `read` a query from a file. In semantic search, however, these limitations are completely dealt with since the semantics of each of the types in sig-

[2] `http://www.w3.org/TR/rdf-primer`
[3] `http://lucene.apache.org/`

natures are encoded and processed during search. Besides addressing knowledge representation effectively, semantic search offers extensible solutions to component retrieval. Since we are focusing on API usage and reuse, the descriptions shown in Listing 1 capture the component's interface and its relationships with other components. However, these description can be easily extended to capture other facets (e.g., component's environment) via introducing additional ontological properties.

Reasoning is one of the primary added benefits in semantic search. In addition of classifying and checking the consistency of our ontologies, a DL reasoner can also be used to inferring and thus enriching the knowledge base with additional knowledge that is not explicitly stated. Thus, playing a vital role in improving search precision and recall in comparison with other search techniques. DL Subsumption reasoning, for example, is typically used to establish subset inclusion relationships between different concepts and properties in the ontology. Consider the descriptions in Listing 1 for example, when pure semantic-based queries are used, users need only to provide domain concepts describing the component's interface. Therefore, if the user provides `SemanticQuery` as a domain output of the requested component, the method shown in the listing would still match this request since `SemanticQuery` subsumes `ExtendedQuery` as specified in the subsumption hierarchy of our domain ontology. Thus, automatically enabling an implicit form of query expansion.

It is notable that semantic-based retrieval alleviates many problems typically faced by tools that rely on exact keyword or type matching. One of the strengths of our approach, however, is the ability to utilize our various ontologies in order to perform blended search against the knowledge base. In particular, this is helpful when components in the knowledge base are not completely annotated or when users are still in the process of becoming familiar with the ontology. Consider for example a user who wishes to find a component in which the component's domain output type (`SemanticQuery`) and one of the actual input types (`Syntax`) are known. Furthermore, since the user is not sure about the other input types, she wants to provide a few terms to filter out the results. This request can be expressed using the following query in DL-like syntax:

$$
\begin{aligned}
&Query \equiv compre : Component \sqcap \\
&(\exists compre : hasDomainOutput \,. \\
&\quad swonto : SemanticQuery) \sqcap \\
&(\exists scro : hasInputType \,.\, kb : Syntax) \sqcap \\
&(\exists compre : hasInputTerms\ value\ ''base\ \ uri'')
\end{aligned}
$$

As expected, executing this query returns not only the method shown in Listing 1 but also other unrelated methods. It turns out that the input terms specified in the query are very popular and are used to describe API methods that are used to `create`, `read` or even `parse` a semantic query. Nevertheless, this search mechanism is flexible since it allows a wide range of queries to run against the knowledge base. In fact, the expressive power provided by our ontologies allows users to express their queries in more details than would otherwise be expressed with any alternative method. For instance, assume that the user was able to obtain a `Query` object as described in the previous example. The next natural step is to find a component that can take this query as input, execute it, and return the required results. Browsing the Jena API looking for such a component or even querying using typical keyword-based queries would not return an answer since there is an intermediate query execution object that must be obtained to complete the task. This appears to be a dead end. However, using semantic search, this request can be expressed fairly easily as follows:

$$
\begin{aligned}
&Query \equiv compre : Component \sqcap \\
&(\exists compre : hasDomainInput \,. \\
&\quad swonto : SemanticQuery) \sqcap \\
&(\exists compre : hasDomainOutput \,.\, \exists\, compre : \\
&\quad hasDomainOutput \,.\, swonto : ResultSet)
\end{aligned}
$$

This query expresses the fact that we are looking for a component that takes a semantic query as input and returns another component that returns a query solution. Thus, querying for multiple components at the same time.

3.1. Implementation and ranking mechanisms

We have implemented this approach in a tool called CompRE, conveniently named after the main ontology in our model. CompRE is deployed as a plug-in for the Eclipse Integrated Development Environment (IDE). Figure 1 shows a snapshot of CompRE's main views in the Eclipse workbench. When loaded for the first time, CompRE processes the library code and the component ontology in order to generate the initial knowledge base as described in Section 2.3. This process is completely automatic and efficient (it only took 4.5 seconds for parsing and processing the Jena framework). CompRE also includes a module that allow users to tag components with semantic references that corresponds to concepts from the domain ontology. These annotations entered via drag and drop mechanisms, captured by the storage module, and stored automatically in the knowledge base. Upon the conclusion of the knowledge population process, a knowledge repository is created and becomes ready for answering user requests.

CompRE provides two separate views for formulating queries. The first view is provided as a simple data entry form as shown in the figure. In each entry box, users need to provide search restrictions that are either prefixed with an ontology name or provided as plain keywords enclosed

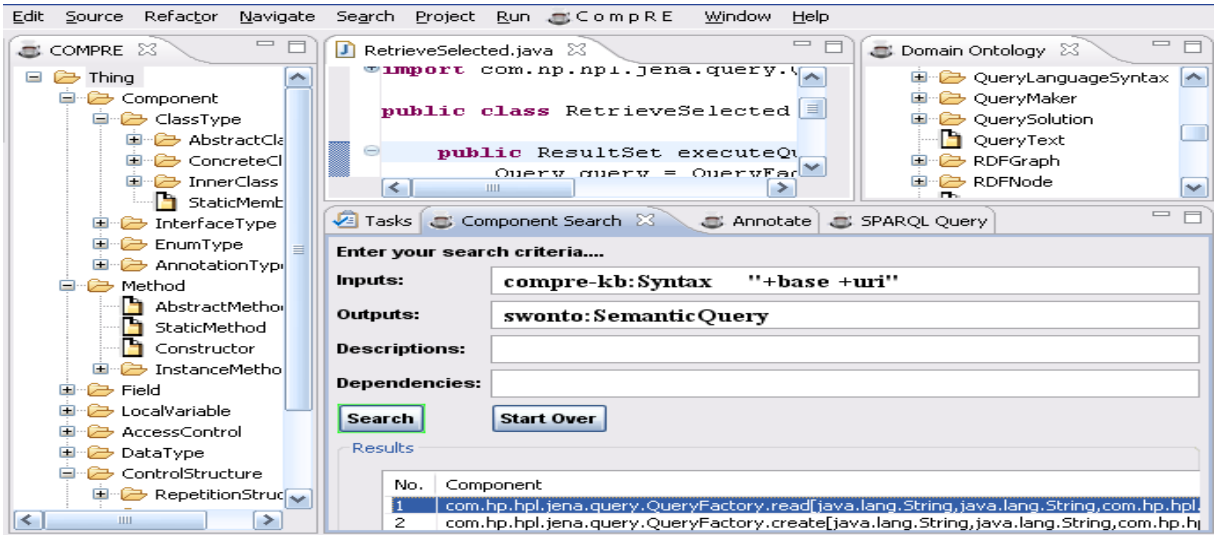

Figure 1. CompRE: showing the component ontology, domain ontology, and the main search view

within quotes. As described in the previous section, using the `compre-kb` prefix tells the system that this is in fact an actual API type specified in the knowledge base. However, the `swonto` prefix refers to a concept from the currently active domain ontology. Since the domain ontology can be different for each API, its name is provided as an external configuration parameter. Free-text requirements in each query may optionally utilize all fuzzy extensions supported by the Lucene's query parser, thus allowing a full-featured keyword-based search. Once the form is filled, CompRE collects the search requirements and automatically generate a query using the SPARQL [4] query language, it then executes this query against the knowledge base. CompRE also provides a query answering view for advanced users who wish to edit their own SPARQL queries directly, therefore, gaining full control over various aspects of the component ontology. For instance, users may wish to specify that the desired component `extends` a particular component or perhaps `usedBy` a certain number of components as a measure of its popularity in the target library. Regardless of the data entry mechanism used, CompRE executes the query, ranks the retrieved instances, and presents the result in a viewer that enables further exploration of each recommended component.

Ranking the retrieved candidates according to their relevancy to the user needs saves time and efforts. While allowing blended queries in our approach ensures flexibility and robustness, it however complicates the ranking process. When blended or pure syntactic queries are submitted, we initially rely on the traditional, however solidly proven, scoring mechanisms supported by Lucene. In order to improve this initial ranking, we refine this initial order based on suitability measures that consider the current user context. We thus parse the code that is currently being developed and create a profile that includes all visible types that are either declared by the programmer or inherited in the user's context. We further analyze each retrieved candidate's signature in terms of the new input types that this candidate will introduce into the current context if selected by the user. Naturally, candidates that introduce more types should be assigned a lower rank value. However, finding the newly introduced type in the context profile, will not count against this candidate's score.

With the absence of keywords in user queries, we apply only context-based heuristics such that candidates with exact matches are put at the top of the list while other candidates are ranked based on the number and type of their input and output types. For example, consider a user who is trying to search for a component that requires two particular input types, namely I1 and I2. Assume that the repository contains three components, namely C1, C2, and C3. Lets also assume that C1 is an exact match, C2 has only one input type (I1), and C3 requires three input types (I1, I2, and I3). The system is then ranks C1 first, C2 is ranked second, and C3 is in fact the least desired since it will introduce a new type to the user context, it is thus included in the result set to improve recall, however, ranked last. These heuristics are simple, easy to implement, and work surprisingly well.

4. Experiments and results

Due to the lack of independent and standard benchmark test data, search tools evaluation is, to some degree, subjective. However, we designed our experiments such that it

[4] `http://www.w3.org/TR/rdf-sparql-query/`

increases our confidence level of a fair evaluation. We have selected the Jena framework for testing CompRE, the domain ontology described in Section 2.1 fits naturally in the Jena's application domain.

4.1. Experiment: component search

This experiment is designed to reveal the overlap between various search methods that are supported by CompRE. The fundamental guiding hypothesis we test in this experiment is that pure semantic-based representation and annotation of library components improve search precision when compared with other techniques. Precision is defined as the ratio of the number of *relevant* component instances that are recommended by the tool to the total number of recommended instances. Recall is the other commonly used metric in evaluating search systems, it is defined as the ratio of the number of *relevant* component instances that are recommended to the total number of relevant components in the repository. However, in these experiments we fix recall since we are searching for distinct components, i.e, the component we are searching for is either found or not found.

We have selected twelve programming tasks, six of these tasks were carefully designed by us and the remaining tasks were collected from the Jena developers forum[5]. Each of these tasks requires a query to be fired in order to search for a component that is required to complete the task. These tasks are diverse enough and cover various aspects of the problem domain. For space limitations, we do not include these tasks here, rather, an extended discussion of the tasks and results can be found online [1].

We then prepared the necessary coding environment and formulated four search queries for each task, i.e, one query for each kind of search supported by CompRE. Precision summery graph for running these queries is shown in Figure 2.

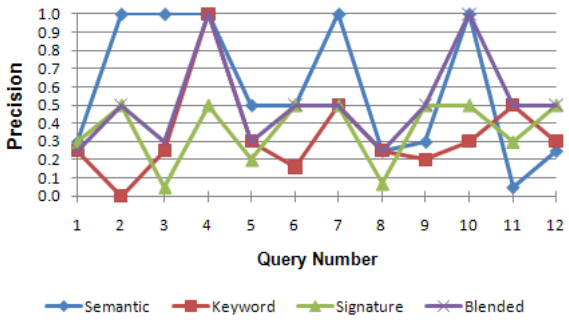

Figure 2. Precision graph for Jena queries

As expected, semantic search tends to perform poorly when the components in the knowledge base are incompletely or incorrectly annotated. In Q11 for example, a required Jena method, `makeRewindable`, was not completely annotated with the proper return type. Thus, search produced spurious components since only the input type was used during search. However, in most cases when proper tags exist, semantic search can precisely describe the needed component and improve overall precision values as seen in Figure 2. Metadata keyword-based search performs poorly due to the two well-known fundamental issues of polysemy and synonymy. These two problems become even more evident when searching for software components. This is due in part to inconsistent and often incomplete API descriptions of library code. Nevertheless, keyword search tends to return an exact match when a particular keyword is used to describe only a single component in the library (e.g., *clone* in Q4).

Signature based queries tend to yield low precision in cases where the component signature includes one or more semantically vague types such as the Java String type. The best example to illustrate this notion is Q3. This task requires accessing a query service over a HTTP connection, therefore, one needs to provide, among other things, the URL of this service and the text representation of the query; both of which are specified as String objects. Unless there is a clear semantic descriptions for such parameters, the matchmaking process would return many false positives (low precision). Blended search, however, performed surprisingly well. We believe that it is often the case that the user is certain about a single API type that is used in the component's signature or a certain keyword that precisely describes some aspect of the component. These descriptions can also be coupled with semantic annotations to produce higher precision. These results indicate that blended search needs to be formally investigated in more details.

Ranking and running time analysis have been computed as well. On average, semantic search achieved 1.75 rank over twelve queries, i.e., the desired component was ranked either at the first or at most the second position. This ranking score is relatively comparable with other search schemes in which they achieved 2.27, 1.8, and 1.66 for keyword, signature, and blended search, respectively. However, the average observed response times were 15.5, 9, 2.5, for semantic, blended, and signature and keyword searches, respectively. All experiments were performed on a Windows XP machine with 1.8GHZ Intel processor and 1GB memory. This relatively lower performance for semantic queries is a result of having our queries run through the reasoner. In general, search time for all search methods is susceptible to increase as the size of the knowledge base increases; in the case of semantic search, speed is continuously improving as reasoners evolve. Achieving perfection in component search is near impossible, however, we believe that CompRe's internal mechanisms proved effective, and in the majority of cases, show a clear support for our hypothesis.

[5] `http://tech.groups.yahoo.com/group/jena-dev/`

4.2. User study

This experiment is designed to asses the usability of semantic search and to understand the possible difficulties faced by end users in learning and using domain ontologies for successfully completing a particular search task against an unfamiliar API.

Six graduate-level MIS students from Northwestern University have voluntarily agreed to participate in this study. Although all students have at least seven months of Java experience and a good working knowledge of Semantic Web technologies, no student has been directly exposed to the Jena API. We delivered a one-hour tutorial that includes a brief introduction to the Jena API, brief introduction to the domain ontology, and a sample training task that explains CompRE's semantic search features. We then charged each student with other four independent Jena API programming tasks that vary in scope: *T1*) data set creation and handling of multiple RDF gtaphs; *T2*) query construction and execution over a given ontology model; *T3*) result manipulation of query solutions; and *T4*) access and treatment of remote services. On average, two distinct components are needed to successfully complete a given task. An environment is setup for each task with a skeleton code, each student is then asked to finish three *consecutive* tasks using only CompRE's pure semantic search while the last task in the sequence must be completed using other alternative methods of student's choice.

Table 1 shows task completion time measured from the time in which the task is presented to the student until the final correct answer is submitted. Numbers appeared in boldface represents tasks completed using semantic search while other numbers are underlined.

Table 1. User study statistics

	Time (Minutes)			
	T1	T2	T3	T4
S1	35	**22**	**27**	**13**
S2	33	**35**	**20**	**11**
S3	**45**	43	**31**	**33**
S4	**32**	40	**22**	**15**
S5	**52**	**12**	43	**8**
S6	**21**	**16**	**18**	40
Avg (Semantic)	37.5	21.25	23.6	16
Avg (Alternative)	34	41.5	43	40

The most significant conclusion we can draw from these numbers is the correlation between the time taken by students to complete the first semantic task and the last one in the sequence. In most cases, there was a significant reduction in time as students became more familiar with the domain ontology and thus more able to construct more precise queries. This is confirmed by the responses we obtained upon the conclusion of the experiment, four out of six students indicated that there is a small initial learning curve that was reduced fairly quickly as they became more comfortable with the API vocabulary represented in the domain ontology. Since the last task has to be done without CompRE's assistance, most students argued that this task could have been completed faster had CompRE's assistance been allowed. A domain ontology provide a concise description of API content and vocabulary. This knowledge can be used also to successfully finishing a coding task that may require more than one component. Consider task T2 for example, this task requires instantiating an intermediate object of type `QueryExecution`, we suspect that the coherent representation of ontology concepts and axioms aid users in arriving at such conclusions during the initial time invested in understanding and learning the taxonomy.

One may conclude that completing the last task (alternative task) should be relatively easier. After all, students have been using the same API, thus, the knowledge gained about this API after completing the first three tasks can be helpful. However, when examining the results, there is no dramatic improvement in response time for using alternative methods. We suspect that these alternative methods (e.g., exploring documentation, searching for code in the Web, etc.) do not provide a systematic and focused learning experience for programmers. In this study however, we did not intend to make a systematic comparison between semantic search mechanisms with normal search practices used by programmers. However, the obtained results clearly support our hypothesis and show that semantic search, in most cases, yield better API learning experience and can certainly increase programmer's productivity.

Overall, students provided positive comments about CompRE and semantic search. Two students indicated that the SPARQL query view was indeed helpful and used in formulating more complex queries. However, these students requested a thorough integration of the domain ontology including its object properties into the CompRE's domain ontology view. Only one student reported a relative difficulty adapting to a new search approach after being familiar with other alternative methods. This user also requested a Tooltip feature such that when the user hovers over an ontology concept in the domain ontology view, a hover box appears with class description. Based on this sound and helpful feedback, we are currently adding new features as well as modifying CompRE's interface so it becomes more expressive.

5. Related work

Due to the benefits acquired by systematic reuse, many researchers have proposed solutions and tackled the reuse

problem from various perspectives. Many approaches (e.g., [3]) employ traditional knowledge representation and variations of signature matching or keyword-based retrieval. Similar to CompRE, other tools (e.g., [11]) leverage software understanding by being embedded in the development environment. However, unlike CompRE, these tools rely on a local repository of sample client code to search for components. CodeBroker [11] for example, use a combination of free-text and signature matching techniques. In order to retrieve appropriate matches, the user must write high quality doc comments that precisely describe functionality. If the user comments did not retrieve satisfactory results, the system considers the signature of the method immediately following the comments. Finding a well documented code to populate the repository with is highly unlikely, especially in open-source and legacy software.

Other component retrieval approaches (e.g., [6], [7]) apply automated testing techniques to analyze a corpus of client code harvested from the Web. Code Conjurer[6] for example, helps agile development users in finding suitable components on the basis of unit test cases. Therefore, users of the system has to write such test cases in order to invoke the system. Once invoked, the system contacts a remote server that finds suitable candidates based on the component's interface specified in the test case.

Other semantic-based approaches have also been proposed. However, the full potential of utilizing domain knowledge was not explored. Sugumaran and Storey[10] proposed an approach that utilizes domain models; a domain ontology is used mainly for term disambiguation and basic query refinement for keyword-based queries; these keywords are then mapped against the ontology to ensure that correct terms are being used. However, no semantic-based descriptions of components have been used. Other proposals ([2] and [4]) employ ontologies to addressing the knowledge representation problem found in previous approaches. In [4], software assets are classified into domain categories (I/O, GUI, Security, etc.) and indexed with a domain field as well as other bookkeeping fields to facilitate free text search. Although the SRS [2] proposal uses the same indexing mechanism, it maintains two separate ontologies; an ontology for describing software assets as well as a domain ontology for classifying these assets. However, the structure of the source code assets and the semantic relationships between those assets via axioms and role restrictions were not fully utilized.

6. Conclusions and future work

We proposed an approach for component reuse. In addition to supporting pure semantic-based search, our approach also supports other kinds of search techniques. However, our studies showed evidence that pure semantic search that utilizes domain knowledge not only usable and achievable, but also improves precision of search results. Our results also showed that blended search has a great potential, we are currently conducting more case studies to asses the value of blended search. There are also two other future work directions. Firstly, ranking reused candidates has always been a challenge, therefore, we are currently investigating how could ranking be improved using semantic technologies. Secondly, we have not yet investigated how could one motivate library providers to ship domain ontologies with their software, or how could individually created ontologies be shared by a community of users.

References

[1] A. Alnusair and T. Zhao. Ontology models, framework ontolgies, and CompRE evaluation. Available online at: `http://www.cs.uwm.edu/~alnusair/compre`.

[2] B. Antunes, P. Gomez, and N. Seco. SRS: A software reuse system based on the semantic web. *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2007.

[3] S. Bajracharya, O. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *First International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE'09)*, 2009.

[4] F. A. Durao, T. A. Vanderlei, E. S. Almeida, and S. R. Meira. Applying a semantic layer in a source code search tool. In *Proceedings of the 23rd ACM Symposium on Applied Computing*, pages 1151–1157, Fortaleza Ceara, Brazil, 2008.

[5] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.

[6] O. Hummel, , W. Janjic, and C. Atkinson. Code Conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[7] O. Hummel and C. Atkinson. Extreme Harvesting: Test driven discovery and reuse of software components. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI'04)*, 2004.

[8] B. McBride. Jena: a semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[9] F. N. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. *Stanford Knowledge System Technical Report KSL-01-05*, 2001.

[10] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *ACM SIGMIS DATABASE*, 34(3):8–24, 2003.

[11] Y. Ye and G. Fischer. Reuse-conductive development environments. *The International Journal of Automated Software Engineering*, 12(2):199–235, 2005.

[12] Y. Zhao, J. Dong, and T. Peng. Ontology classification for semantic web based software engineering. *IEEE Transactions on Services Computing*, 2(4):303–317, 2009.