An Ontology-based Approach for Traceability Recovery

Yonggang Zhang, René Witte, Juergen Rilling, Volker Haarslev

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

Abstract.

Traceability links provide support for software engineers in understanding the relations and dependencies among software artifacts created during the software development process. In this research, we focus on re-establishing traceability links between existing source code and documentation to support reverse engineering. We present a novel approach that addresses this issue by creating formal ontological representations for both the documentation and source code artifacts. These representations are then aligned to establish traceability links at the semantic level. Our approach recovers traceability links by utilizing the structural and semantic information in various software artifacts and the linked ontologies are also supported by ontology reasoners to infer implicit relations among these software artifacts.

Keywords: Traceability, Reverse Engineering, Ontology, Text Mining

1. Introduction

Traceability links help software engineers understand the relations and dependencies among various software artifacts. However, it is a well known fact that even in organizations and projects with mature software development processes, software artifacts created as part of these processes end up to be disconnected from each other [1,2]. This lack of traceability among software artifacts is caused by several factors, including: (1) the fact that these artifacts are written in different languages (natural language vs. programming language); (2) they describe a software system at different abstraction levels (design vs. implementation); (3) the processes applied within an organization that do not enforce maintenance of existing traceability links; and (4) a lack of adequate tool support to create and maintain traceability.

The missing traceability among software artifacts becomes a major challenge for reverse engineering activities. As a result, during the reverse engineering of existing software systems, reverse engineers have to spend a large amount of effort on synthesizing and integrating information from various information sources to establish links among these artifacts. The cost associated with this manual effort is the main motivation for existing research in providing automatic assistance in establishing and maintaining traceability links among software artifacts [1].

Software design documentation and source code are two of the major software artifacts typically used as part of reverse engineering. Existing source-document traceability research [2, 3] mainly focuses on connecting documents and source code using Information Retrieval (IR) techniques. However, these approaches typically ignore structural and semantic information that can be found in both documents and source code, limiting therefore both their precision and applicability.

In this paper, we present a novel approach that explicitly includes structural and semantic information integration, by providing an ontological representation for both types of software artifacts – source code and documentation. Instead of using simple IR, we developed a Text Mining (TM) system for semantically analyzing documents. The discovered concepts and concept instances from both source code and documents are used to establish the links between these two software artifacts. In addition, the formal ontological representation also allows us to take advantage of automated reasoning services provided by ontology reasoners to infer implicit relations (links) between these two types of artifacts.

Our research is significant for several reasons. Firstly, software artifacts other than source code, such as documentation, contain rich semantic information that is not used by existing reverse engineering tools. Introducing an ontological representation for software documentation enables us to utilize Natural Language Processing (NLP) techniques to "understand" parts of the semantics conveyed by these artifacts and to establish additional traceability links among these artifacts.

Secondly, the uniform ontological representation for both source code and documentation allows us to share common concepts between different resources, easing the integration of information by allowing for the recovery and establishment of traceability links among documentation and source code artifacts.

Finally, representing software artifacts in a formal ontology allows programmers to reason about various implicit relations between software artifacts. Taking advantage of existing ontology-based knowledge representation techniques such as Description Logics [4] and ontology reasoners [5], users can define new concepts and roles (types of relations) for specific reverse engineering tasks and query the ontology using either the pre- or user-defined vocabulary.

The remainder of the paper is organized as follows: in Section 2, we provide the background of our research, including formal ontologies and text mining techniques. Section 3 presents our ontology-based reverse engineering environment, which has been utilized to provide ontological representations for both source code and documentation. In Section 4, a detailed discussion concerning recovering and maintaining the traceability links between source code ontology and documentation ontology is given, followed by an initial evaluation of our research in Section 5. Related work is discussed in Section 6, and conclusions and future work are presented in Section 7.

2. Background

In this section, we introduce the background of our research, including ontologies and their formalisms – Description Logics, as well as text mining techniques and ontology population approaches.

2.1 Ontology and Description Logics

Ontologies are often used as a formal and explicit way of specifying the concepts and relationships in a domain of discourse. Meanwhile, Description Logics (DL) [4], as a family of Knowledge Representation formalisms, has been long regarded as a standard ontology language. DL is also a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [6]. DL represents the knowledge of a domain by first defining the relevant concepts of the domain in a taxonomy, and then using these concepts to specify properties of individuals occurring in the domain. The use of DL allows us to formally characterize subsumption relationships between concepts: A concept C is considered a sub-concept of D if all instances of C are also instances of D.

Basic elements of DL are atomic concepts and atomic roles, which correspond to unary predicates and binary predicates in First Order Logic. Complex concepts are then defined by combining basic elements with several concept constructors. For example, in the domain of software design technique and documentation structure, having atomic concepts such as DesignPattern and Paragraph, as well as an atomic role contains that describes a relation between these two concepts, a new concept DesignPatternDoc can then be defined by a conjunction constructor and existential qualifier:

$$\text{DesignPatternDoc} \equiv \text{Paragraph} \sqcap \exists\ \text{contains.DesignPattern}$$

Individuals existing in the domain and their relations can be specified as instances of their corresponding concepts and roles. For example, the following DL expressions define *p* as a paragraph instance, *abstract_factory_pattern* as a design pattern instance, and the body of *p* contains *abstract_factory_pattern*.

p:Paragraph, abstract_factory_pattern:DesignPattern, (p, abstract_factory_pattern): contains

Having DL as the specification language for a formal ontology enables the use of reasoning services provided by DL-based knowledge representation systems. Our Racer system [5] is an ontology reasoner that has been highly optimized to support very expressive DLs. Typical services provided by Racer include terminology inferences (e.g., concept consistency, subsumption, classification, and ontology consistency) and instances reasoning (e.g., instance checking, instance retrieval, tuple retrieval, and instance realization). For example, given the above concept definition of DesignPatternDoc, as well as the assertions about instance *p* and *abstract_factory_pattern*, the ontology reasoner can automatically infer that *p* is also an instance of DesignPatternDoc.

For a more complete coverage of DLs and Racer, we refer the reader to [4, 5].

2.2 Text Mining and Ontology Population

Text Mining (TM) is commonly known as a knowledge discovery process that aims to extract non-trial information or knowledge from unstructured text. Unlike *Information Retrieval* (IR) systems, TM does not simply return documents pertaining to a query, but rather attempts to obtain *semantic* information from the documents themselves, using techniques from Natural Language Processing (NLP). For example, our TM subsystem obtains information about individual software entities mentioned in the documents, like the *architecture*, its *components*, and relationships with *packages* or *classes.* These so-called *Named Entities* (NEs) are exported into the documentation ontology, which can then be loaded into a visualization tool or a reasoning system like Racer.

We implemented our Text Mining subsystem based on the GATE *(General Architecture for Text Engineering)* framework [7], one of the most widely used NLP tools. Within the text mining process, we make use of a number of standard NLP techniques. These include first dividing the textual input stream into individual tokens with a *Unicode tokeniser,* using a *Sentence Splitter* to detect sentence boundaries and running a statistical *Part-of-Speech* (POS) *tagger* that assigns labels (e.g., noun, verb, and adjective) to each word. Larger grammatical structures, *Noun Phrases* (NPs) and *Verb Groups* (VGs), are created based on these tags using *chunker* modules.

Text mining results are exported by instantiating a pre-modeled ontology in a so-called *ontology population* step. This facilitates linking results from document analysis (represented by ontology instances) with source code analysis results (which are also stored in an ontology). Details on this step are provided in Section 3.2.2.

An example system for ontology population from natural language texts is the KIM platform described in [20]. For more details on these steps, we refer the reader to [7] and the GATE user's manual.

3. Ontological Representation for Software Artifacts

Software artifacts such as source code or documentation typically contain knowledge that is rich in structural and semantic information. This information is not used by existing IR-based traceability research [2, 3]. On the other hand, formal ontologies, as the successor of *semantic networks*, have been long regarded as standard techniques to capture semantics in a domain of discourse. Providing uniform ontological representations for various software artifacts enables us to utilize semantic information conveyed by these artifacts and to establish their traceability links at semantic level. In this section, we introduce our SOUND program comprehension environment [8], which provides ontological support for various software maintenance tasks.

3.1 Overview

In order to utilize the structural and semantic information in various software artifacts, we have developed an ontology that captures major concepts and relationships in the software domain. An ontology-based program comprehension environment –

SOUND (Software Ontology for UNDerstanding) [8] has been developed to extract concept instances and their relations from source code and documents. The SOUND environment facilitates reverse engineers in both discovering concepts and relations within a software system, as well as automatically inferring implicit relations among different artifacts (Figure 1).

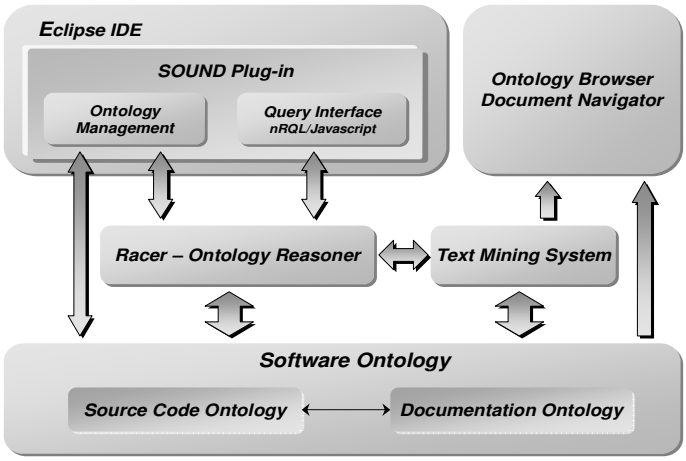

Figure 1 – Overview of SOUND Environment

Instances of concepts and roles in the software ontology can be populated by our Eclipse plug-in or text mining system. The discovered instances from different sources can be automatically linked through ontology alignment [9]. Based on the software ontology, users can define new concepts/instances for particular reverse engineering tasks through an ontology management interface. The ontological reasoning services within the SOUND environment are provided by our ontology reasoner – Racer [5].

3.2 Software Ontology

The software ontology in our system consists of two sub-ontologies: 1) The source code ontology represents the syntactic and semantic information of source code; 2) the documentation ontology represents semantic information extracted from software documentation.

3.2.1 Source Code Ontology

The source code ontology has been designed to formally specify major concepts of Object-Oriented Programming languages. In our implementation, this ontology is further extended with additional concepts and roles needed for some specific languages (in our case, Java). Table 1 shows part of the taxonomy of the source code ontology.

Within this ontology, various roles are defined to characterize the relationships among concepts. For example, two instances of SourceObject may have a definedIn relation indicating one is defined in the other; or an instance of method may read an instance of Field indicating the method may read the field in the body of the method.

Using DL, if a role R is defined as a transitive role, and if instances of the role (a,b)∈R and (b,c)∈R are specified, then (a,c)∈R is also implied. Transitive roles are especially useful for specifying part-of relations between source code entities (through definedIn role), inheritance relations between classes (through hasSuperType role), and indirect calling relations (through indirectCall role).

Table 1 – Concept Names in the Source Code Ontology

Concept Name	Description and Examples
Thing	everything, top concept
JavaThing	things in Java
SourceThing	things in source code
SourceAction	actions in source – declaration, invocation, etc.
SourceObject	objects in source
Package	Java packages – *java.lang*
SourceFile	Java source files – *String.java*
Class	Java classes – *String*
Comment	inline comments – */**…*/*
Variable	variables – *System.out, temp*
Field	class variable – *System.out*
LocalVariable	local variable – *temp*
Member	class member
Field	class variable – *System.out*
Method	class method – *print(…)*
Type	types in Java – *int, float, String*
PrimaryType	primary types in Java – *int*, …
Class	abstract types – *String*

Concepts in the source code ontology typically have a direct mapping to source code entities, and thus instances of these concepts can be automatically recognized by our SOUND plug-in, by utilizing the JDT compiler provided by Eclipse. The SOUND plug-in also identifies instances of roles (i.e., relations between source code entities) by statically analyzing the source code.

3.2.2 Documentation Ontology

The documentation ontology consists of a large body of concepts that are expected to be discovered in software documents. These concepts are based on various programming domains, including programming languages, algorithms, data structures, and design decisions such as design patterns and software architectures.

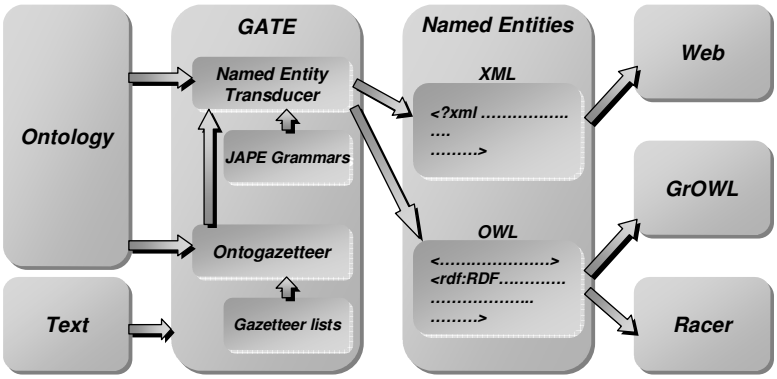

Figure 2 – Text Mining and Ontology Population Process

The documentation ontology governs the identification of relevant named entities. With each concept in the ontology, a *gazetteer* list of terms is connected, which allows an *OntoGazetteer* NLP component to semantically tag individual words in the document, linking them to one (or multiple) point(s) in the ontology. Complex named entities can then be detected in another step using a cascade of finite-state transducers implementing custom grammars written in the JAPE language, which is part of GATE. For example, we can find through the OntoGazetteer that the word *layer* can be part of an architectural description. The NP analysis step will mark up the text fragment *the controller layer* as a single noun phrase, with *layer* being the head noun and *controller* a modifier, specifying exactly what layer is meant. By combining the syntactical with the semantic information, we can detect named entities, which correspond to ontology concepts. In another step, we determine relationships between detected entities, e.g., class *belongs_to* layer. This is again achieved with a combination of two techniques: A number of pre-defined patterns are detected using additional JAPE grammar rules. Additionally, we compute predicate-argument structures using the SUPPLE parser that allow us to determine subject-object-predicate relationships, which are further filtered using the documentation ontology, restricting the syntactically possible relations to semantically valid ones.

In a final step, the analysis results are exported. Besides storing the marked-up documents as XML files, we can add detected instances and relations (i.e., object properties) to a pre-defined ontology in an ontology population step. For example, each detected textual entity of the semantic type *method* becomes an instance of the ontology class Method. This requires an additional *normalization* step prior to export, as textual descriptions for the same semantic entity can differ. For example, a method named "myMethod" can be referred to in a text as *"the myMethod method provides...", "myMethod provides...",* or even *"this method provides"*. Automatic normalization ensures that only a single instance *myMethod* is created in the ontology in this case, while still referring to the various textual references. Additionally, document-specific information is recorded as well, e.g., in which sentence an entity was found. An example for the documentation ontology with populated instances is shown in Figure 6.

3.3 Query Interface

Users of our SOUND environment can use the Racer query language nRQL [10] to retrieve instances of concepts and roles in the ontology. An nRQL query uses arbitrary concept names and role names in the ontology to specify properties of the result. In a query, variables can be used to bind to instances that satisfy the query.

However, the use of nRQL queries is still largely restricted to users with a high mathematical/logical background due to nRQL's syntax, which, although comparatively straightforward, is still difficult for programmers to understand and even more difficult to apply. To bridge this gap between practitioners and Racer, we have used an additional scripting language – JavaScript as a query language. We introduce a set of built-in functions and classes in the JavaScript interpreter – Rhino*, to simplify user querying on the ontology for users.

Within the JavaScript interpreter, we provide a set of logic functions for formulating complex concepts. Using these logic functions, users can construct their own concepts. For example, the concept ClassMember discussed in Section 2.1, can be specified using the build-in functions as:

```
ontology.define_concept("DesignPatternDoc", AND("Paragraph, EXIST("contains", "DesignPattern")))
```

Two classes, Query and Result, are provided to assist users in composing queries and manipulating the results. Users can arbitrarily use the vocabulary in the ontology to retrieve instances with specified properties. The typical procedure of composing a query is as follows: (1) query variables are declared; (2) restrictions that apply to the variables are specified using concepts, roles, and instances in the ontology; and (3) the query is submitted to the built-in JavaScript object called "ontology".

The result of the query is a set of tuples that satisfy the specified restrictions. For example, the following query/script retrieves all paragraphs that contain design pattern instances from the documentation ontology:

```
var disign_pattern_doc = new Query();                  // create an new query
disign_pattern_doc.declare("P", "DP");                 // declare two query variables in the query
disign_pattern_doc.restrict("P", "Paragraph");         // restrict P to be bound to a paragraph instance
disign_pattern_doc.restrict("DP", "DesignPattern");    // restrict DP to be bound to a design pattern instance
disign_pattern_doc.restrict("P", "contains", "DP");    // restrict P contains DP
disign_pattern_doc.retrieve("P");                      // the query will only retrive instances of P
var result = ontology.query(disign_pattern_doc);       // perform the query
```

The query first declares two variables M and P, and then specifies that P shall be bound to an instance of Paragraph, and DP to an instance of DesignPattern. The third restriction specifies that P and DP shall have a contains relation. The next statement states that this query only retrieves instances bound to P.

The scriptable query language allows users to benefit from both the declarative semantics of Description Logics as well as the fine-grained control abilities of procedural languages.

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

4 Linking Software and Documentation Ontology

Having both source code and documents represented in the form of an ontology allows us to link instances from source code and documentation using existing approaches from the field of ontology alignment [9]. Ontology alignment techniques try to align ontological information from different sources on conceptual or/and instance levels. Since our documentation ontology and source code ontology share many concepts from the programming language domain, such as Class or Method, the problem of conceptual alignment has been minimized. This research therefore focuses more on matching instances that have been discovered both from source analysis and text mining.

Our text mining system can additionally take the results of the source code analysis as input when detecting named entities. This allows us directly connect instances from the source code and document ontologies. For example, our source code analysis tool may identify c_1 and c_2 as classes, and this information can be used by the text mining system to identify named entities – c'_1 and c'_2 and their associated information in the documents (Figure 3). As a result, source code entities c_1 and c_2 are now linked to their occurrences in the documents (c'_1 and c'_2), as well as other information about the two entities mentioned in the document, such as design patterns, architectures, etc.

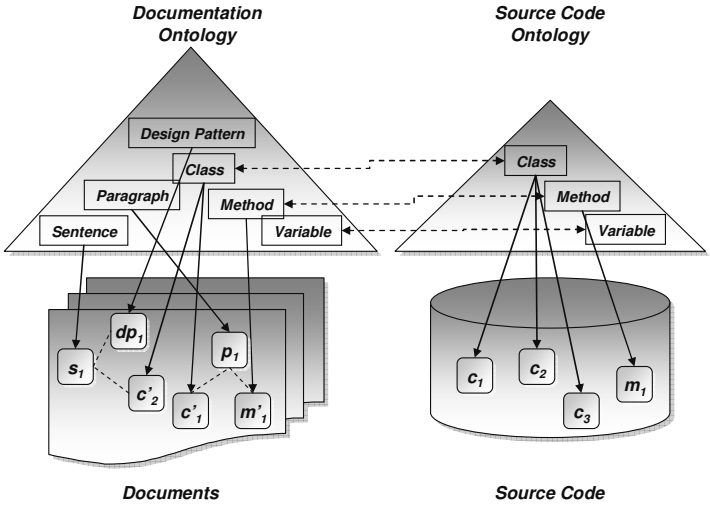

Figure 3 – Linking Instances from Source Code and Documentation

After source code and documentation ontology are linked, users can perform ontological queries on either documents or source code regarding properties of c_1 or c_2. For example – retrieve document passages that describe both c_1 and c_2, or retrieve design pattern descriptions referring to the class that contains the class currently analyzed. Note that the alignment process might also identify inconsistencies – the documentation might list a method for a different class, for example – which are detected through the alignment process and registered for further review by the user.

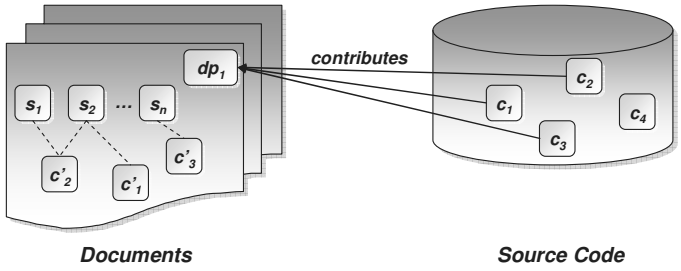

Figure 4 – Retrieve Implicit Information from Documents

In addition, users can always manually define new concepts/instances and relations in both ontologies to establish the links that cannot be detected by the automated alignment. For example, as Figure 4 shows, the text mining system may detect an instance of DesignPattern – dp_1 and users can create the relations between the pattern and classes that are contributed the pattern (e.g., c_1, c_2, and c_3) through our query interface. The newly created links then become an integrated part of the ontology, and can be used to, for example, retrieve all documents related to the pattern (i.e., s_1, s_2, …, s_n).

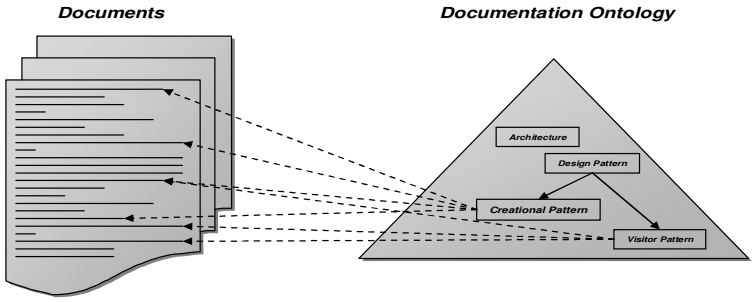

Figure 5 – Classification of Documentation based on Ontology

Furthermore, documents can not only be linked to source code, but also to design-level concepts that relate to particular reverse engineering tasks. For example, in contrast to the serialized view of software documents, i.e., sentence by sentence, or paragraph by paragraph, the formal ontological representation of software documentation also provides the ability to create hierarchical documentation views. Using the classification service of the ontology reasoner, one can classify document pieces that related to a specific concept or a set of concepts (Figure 5). For example, the Visitor Pattern [11] documents can be considered as all text paragraphs that describe/contain information related to concept Visitor pattern. The following new concept VisitorPattern-Doc can be used to retrieve paragraphs that related visitor pattern. Similarly, a new concept HighlevelDoc can be also defined to retrieve all documents that contains high level design concept Architecture or DesignPattern. The ontology reasoner can automatically classify documents according to these concept definitions.

$$\text{VisitorPatternDoc} \equiv \text{Paragraph} \sqcap \exists\ \text{contains.Visitor}$$

HighlevelDoc ≡ DocumentFile ⊓ ∃ contains.(Architecture ⊔ DesignPattern)

5. Evaluation

As part of this research, we have extended our SOUND environment by several ontology alignment rules to link the documentation ontology and source code ontology. The implementation is based on the Eclipse platform. An initial evaluation has been performed on a large open source Geographic Information System (GIS) – uDig*. The uDig system is a set of Eclipse plug-ins that provides geographic information management integration for the Eclipse platform. The uDig documents used in the study consist of a set of JavaDoc files and a requirement analysis document.†

Links between the uDig implementation and its documentation are recovered by first performing source code analysis to populate the source code ontology. The resulted ontology contains instances of Class, Method, Field, etc, and their relations such as inheritance, invocation, etc. Our text mining system takes the identified class names, method names, and field names as input to populate the documentation ontology. Through this text mining process, a large number of Java language concept instances are discovered in the documents, as well as design level concept instances such as design patterns or architecture styles [12]. The ontology alignment rules are then applied to link both the documentation ontology and the source code ontology. Part of our initial result is shown in Figure 6, and the contents of the related sentences are:

Sentence_2544: *"For example if the class FeatureStore is the target class and the object that is clicked on is a IGeoResource that can resolve to a FeatureStore then a FeatureStore instance is passed to the operation, not the IGeoResource".*

Sentence_712: *"Use the visitor pattern to traverse the AST"*

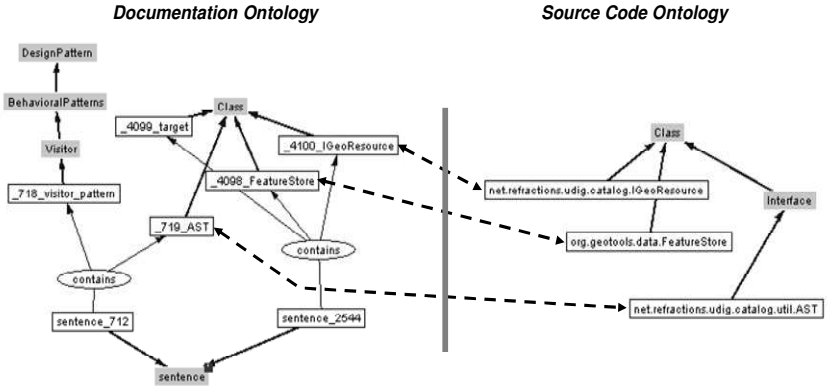

Figure 6 – Linked Source Code and Documentation Ontology

* http://udig.refractions.net/confluence/display/UDIG/Home
† http://udig.refractions.net/docs/

Figure 6 shows that in the uDig documents, our text mining system was able to discover that a sentence (*sentence_2544*) contains both class instance *_4098_FeatureStore* and *_4100_IGeoResource*. Both of these instances can be linked to the instances in source code ontology – *org.geotools.data.FeatureStore* and *net.refractions.udig.catalog.IGeoResource*, respectively. In addition, in another sentence *(sentence_712)*, a class instance (*_719_AST*) and a design pattern instance (*_718_visitor_pattern*) are also identified. Instance *_719_AST* can then be linked in a similar manner to the *net.refractions.udig.catalog.util.AST* interface in the source code ontology.

After the source code ontology and documentation ontology are linked, queries regarding the source code entities, design level concepts, and their occurrences in documents can be performed using the reasoning services provided by out ontology reasoner – Racer. For example, during the comprehension of the class *FeatureStore*, a reverse engineer may want to study the classes that are related to *FeatureStore*. Within the source code ontology, a query similar to the following script (Script 1) can be performed to retrieve all classes that contain methods that have called class *FeatureStore*.

```
var query = new Query();                                          // define a new query
query.declare("M1", "M2",  "C");                                  // declare three query variables
query.restrict("M1", "Method");                                   // M1 is a method
query.restrict("M2", "Method");                                   // M2 is also a method
query.restrict("C", "Class");                                     // C is a class
query.restrict("M1", "definedIn", "C");                           // M1 is defined in C
query.restrict("M2", "definedIn", "org.geotools.data.FeatureStore"); // M2 is defined in FeatureStore
query.restrict("M1", "calls", "M2");                              // M1 calls M2
query.retrieve("C");                                              // this query only retrieve C
var result = ontology.query(query);                               // perform the query
```

Script1 – Query on Source Code Ontology

Unfortunately, the class *IGeoResource*, which has a documented relation with *FeatureStore* (Figure 6), will not be returned by such a query, because *IGeoResource* has no explicit invocation relations with *FeatureStore* in the uDig implementation. In addition to these types of source code queries, the reverse engineer can perform queries that are across the boundaries between source code and documentation. Such type of queries are enabled due to the already established links between the source code and documentation ontology. For example, the following query (Script 2) retrieves all classes that occur in the same sentences as class *FeatureStore*. At this time, class *IGeoResource* will be returned because both classes occur in *sentence_2544*. The retrieved classes as well as the associated sentences therefore provide additional information useful for reverse engineers to understand the class *FeatureStore*.

```
var query = new Query();                                          // define a new query
query.declare("S", "C");                                          // declare two query variables
query.restrict("S", "Sentence");                                  // S is a Sentence
query.restrict("C", "Class");                                     // C is a Class
query.restrict("S", "contains", "org.geotools.data.FeatureStore"); // S contains FeatureStore
query.restrict("S", "contains", "C");                             // S also contains C
query.retrieve("C", "S");                                         // retrieve C and the sentence S
var result = ontology.query(query);                               // perform the query
```

Script 2 – Query on Documentation Ontology

The linked source code and documentation ontologies also provide us with the capability to combine semantic information from both software implementation and documentation. For example, our text mining system has detected that class *AST* is potentially a part of a Visitor pattern (Figure 6). In order to retrieves all documented information related to the detected pattern, the following query (Script 3) can be used to retrieve all text paragraphs that describe the sub classes of *AST*.

```
var query = new Query();                                              // define a new query
query.declare("P", "C");                                              // declare two query variables
query.restrict("P", "Paragraph");                                     // P is a paragraph
query.restrict("C", "Class");                                         // C is a class
query.restrict("C", "hasSuper", "net.refractions.udig.catalog.util.AST"); // C is a sub-class of AST
query.restrict("P", "contains", "C");                                 // P contains C
query.retrieve("P");                                                  // this query only retrieve P
var result = ontology.query(query);                                   // perform the query
```

Script 3 – Query Across the Source Code and Documentation Ontology

This query utilizes both, the programming language semantics, such as the inheritance relation between query variable *C* and the class *AST*, and the structural information of documentation, such as the containing relation between *P* and *C*. The result of this query therefore contains all text paragraphs that describe the sub classes of *AST*, i.e. the Vistor pattern. It has to be noted that the role contains is a transitive relation to describe the document structure. The ontology reasoner can automatically resolve the transitivity from Paragraph to Sentence, and from Sentence to Class.

In this section, we presented an initial evaluation of recovering traceability links between source code and documentation on a large open source software system. We have demonstrated the use of automated reasoning to retrieve documented information with regard to a specific reverse engineering task and infer implicit relations in the linked ontologies.

6. Related Work and Discussions

There exists some research in recovering traceability links between source code and design documents using Information Retrieval techniques. The IR models used include traditional vector space and probabilistic models [2] and advanced latent semantic indexing model [3]. In contrast with these IR approaches, our work also utilizes structural and semantic information in both the documentation and the source code by means of text mining and source code parsing. This additional information allows us to recover links that would not be discovered using traditional IR techniques.

Very little previous work exists on text mining software documents. Most of this research has focused on analysing texts at the specification level, e.g., in order to automatically convert use case descriptions into a formal representation [13, 14] or detect inconsistent requirements [15]. In contrast, we aim to support the complete software documentation life-cycle, from white papers, design and implementation documents to in-line code texts (e.g., JavaDoc). To the best of our knowledge, there

has been so far no attempt to automatically cross-link entities (e.g., methods, design patterns, architectures) detected by text mining software documents with corresponding entities found by source code analysis, which is an important contribution of our work.

Existing research on applying Description Logics or formal ontology in software engineering have been addressed in early works of the LaSSIE [16] and CBMS [17] 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.

In our previous work, we have already demonstrated the ontological model of source code and documentation supporting various reverse engineering tasks, such as program comprehension, architectural analysis [8], and security analysis [18]. In another work, we have examined the requirements for software reverse engineering repositories [19], where we focused on dealing with incomplete and inconsistent knowledge on software artifacts obtained from different sources (e.g., conflicting information delivered by source code and document analysis).

7. Conclusions and Future Work

The presented research addresses an important issue in the reverse engineering domain, the recovery and maintenance of traceability links among existing documents and source code artifacts. We present a novel approach that provides formal ontological representations for both source code and document artifacts. The ontologies capture structural and semantic information conveyed in these artifacts, and therefore allow us to recover the traceability links between software implementation and documentation at semantic level.

In addition, utilizing state-of-the-art ontology reasoners such as Racer, our approach also allows inferring implicit relations between discovered concept instances. The linked ontologies provide the capability to perform queries across the boundary between programming language and natural language.

Furthermore, our documentation ontology identifies a large number of design-level concept instances such as design patterns and architectural styles. These identified instances are linked to source code entities, and therefore allows users to discover relations between source code and its design information at different levels abstraction.

As part of our future work, we will be exploring a hierarchical linking strategy, starting from code, including inline comments (like JavaDoc), over implementation, design, and specification documents to domain-specific knowledge, to allow us to offer a truly holistic process for an automated support of traceability links.

ACKNOWLEDGEMENTS

Qiangqiang Li contributed to the text mining system.

REFERENCES

[1] P. Arkley, P. Mason, and S. Riddle, "Position Paper: Enabling Traceability," Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland (September 2002), pp. 61–65.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation". In Proceedings of IEEE International Conference on Software Maintenance, San Jose, CA, 2000

[3] A. Marcus, J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing". In Proceedings of 25th International Conference on Software Engineering, 2002

[4] F. Baader et al., "The Description Logic Handbook", Cambridge Univ. Press, 2003.

[5] V. Haarslev and R. Möller, "RACER System Description", In Proc. of International Joint Conference on Automated Reasoning, IJCAR'2001, Italy, Springer-Verlag, pp. 701-705.

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

[7] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan. "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications." Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia, July 2002.

[8] Y. G. Zhang, R. Witte, J. Rilling, and V. Haarslev, "Ontology-based Program Comprehension Tool Supporting Website Architectural Evolution", Proceedings of the 8th IEEE International Symposium on Web Site Evolution, 2006

[9] N. F. Noy and H. Stuckenschmidt, "Ontology Alignment: An annotated Bibliography – Semantic Interoperability and Integration" Schloss Dagstuhl, Germany, 2005

[10] 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.

[11] E.Gamma, R.Helm, R.Johnson, and J.Vlissides "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994

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

[13] V. Mencl. "Deriving Behavior Specifications from Textual Use Cases". Proc. of Workshop on Intelligent Technologies for Software Engineering (WITSE'04), Austria, 2004.

[14] M.G. Ilieva and O. Ormandjieva. "Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation". 10th Intl. Conf. on Applications of Natural Language to Information Systems (NLDB), Alicante, Spain, 2005.

[15] L. Kof. "Natural Language Processing: Mature Enough for Requirements Documents Analysis?" 10th Intl. Conf. on Applications of Natural Language to Information Systems (NLDB), Alicante, Spain, June 15-17, 2005.

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

[17] 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.

[18] Y.G.Zhang, J.Rilling, V.Haarslev, "An ontology based approach to software comprehension – Reasoning about security concerns in source code". In Proc. of 30th Int. Computer Software and Applications Conference, 2006.

[19] Ulrike Kölsch and René Witte, "Fuzzy Extensions for Reverse Engineering Repository Models". 10th Working Conference on Reverse Engineering (WCRE), Canada, 2003.

[20] Kiryakov A., Popov B., Terziev I., Manov D., and Ognyanoffe D. "Semantic Annotation, Indexing, and Retrieval". Journal of Web Semantics, vol. 2(1), 2005.