CodeLink: Using a Semantic Wiki for Code Documentation

CS886: Natural Language Computing Project (Spring 2007)

Steven She (shshe@uwaterloo.ca)

August 10, 2007

Abstract

Documentation maintainence is a difficult and costly process. Existing forms of software documentation exist independent from the code it describes. As a result, problems arise when source code evolves since there is no traceability between the documentation and code domains.

CodeLink, a semantic wiki designed for code documentation attempts to address this problem. CodeLink provides a platform for establishing traceability links between developer documentation and code concepts. Traceability links between documentation and code can be explicitly created by the user through annotations, or inferred through the use of natural language analysis. CodeLink employs an ontology as its knowledge model and a semantic wiki as its user interface. Annotations on traceability relations in the ontology serve to connect natural language phrases with formal ontology concepts.

In this paper, the motivation, design and implementation of CodeLink are described. Particular focus is placed on the natural language processing (NLP) component of CodeLink. The process in which the NLP component infers traceability links from the natural language text in the semantic wiki is described in depth. Several methods of querying and retrieving information from the knowledge model are discussed, followed by a summary of related and future work.

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

Software documentation serves as an important artifact in which domain knowledge and business requirements are stored. Current forms of documentation exist independently from the code it describes. As a result, documentation maintenance is often neglected and not maintained as software evolves. The knowledge contained in outdated documentation can no longer be trusted as the requirements and implementation have changed.

In addition to documentation maintenance, changes to software are often captured in a variety of forms such as change request documents, code comments, or informal e-mails. Traceability between these artifacts exists as natural language phrases and sentences, however, it is up to a human to mentally extract the traceability links from the natural language descriptions.

CodeLink attempts to address the issues of documentation maintenance and traceability between software artifacts. At its core, CodeLink is a documentation tool. Through the use of a semantic wiki, CodeLink provides a collaborative environment for establishing traceability links between software artifacts. Traceability links from unannotated documentation are detected by applying natural language analysis to the text. An ontology is used to formalise the code model and the traceability links between code and documentation.

CodeLink is intended for developer documentation. One particular form of developer documentation is a software cookbook [1, 2]. A cookbook contains a set of recipes that informally describe how to adapt the application framework to solve a specific problem. A recipe is typically structured into three section: purpose, procedure and source code examples. Cookbook documentation is ideal for CodeLink since the documentation concepts are closer to code in abstraction than other forms of documentation.

CodeLink employs a *semantic wiki* as its platform for documentation. A *wiki* is a web application that allows users to collaborate on the creation and modification of the content. Links are created to establish relationships between pages in the wiki. A semantic wiki extends a traditional wiki with a form of knowledge representation. The knowledge model enables the system to reason about the data contained within the semantic wiki, thus providing a richer set of relations and querying capabilities than a traditional wiki.

A semantic wiki is well-suited for the collaborative documentation creation and maintenance process. Many open-source projects, such as the projects in the Apache Foundation[1], Mozilla[2] and the Eclipse Foundation[3], employ a traditional wiki for documentation. While a traditional wiki provides a collaborative means of maintaining documentation, there is no traceability between documentation and code.

In this paper, CodeLink, a semantic wiki for code documentation is presented. In Section 2, the implementation, architecture and details of the CodeLink components are described. Section 3 presents several methods of navigating and querying the knowledge base. Section 4 describes related work, Section 5 describes short and long term future work and Section 6 concludes.

[1] Apache Foundation, `http://wiki.apache.org/general/`
[2] Mozilla, `http://wiki.mozilla.org/Main_Page`
[3] Eclipse Foundation, `http://wiki.eclipse.org/Main_Page`

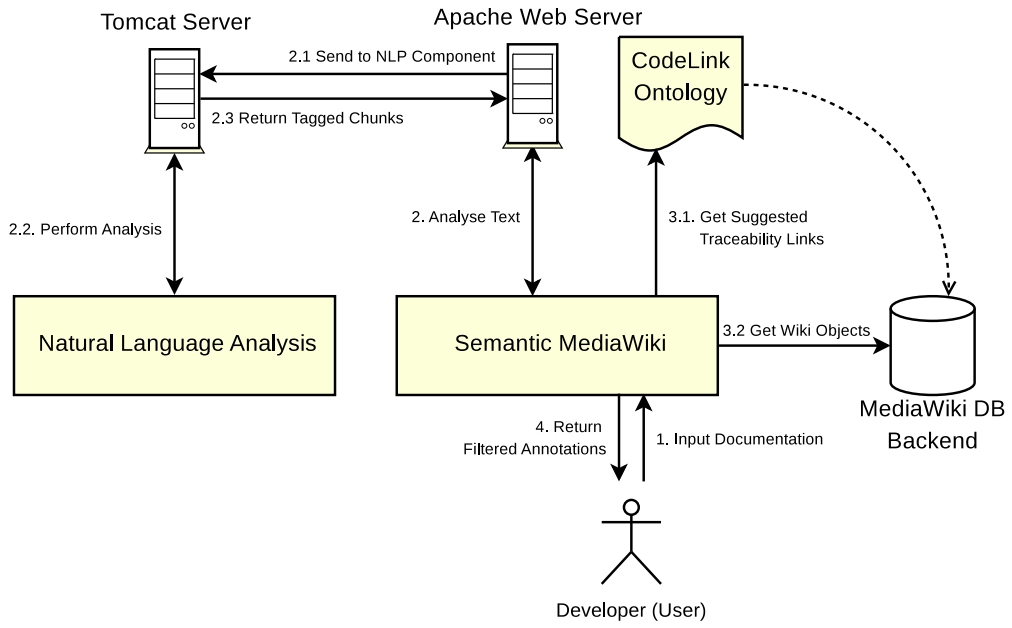

Figure 1: Sketch of the CodeLink Documentation Process

2 Implementation

2.1 CodeLink Architecture

CodeLink consists of three components: knowledge model, semantic wiki, and natural language processing. The three components are described in further detail below.

The knowledge model component formalises the documentation and code domain concepts. CodeLink uses an ontology for its knowledge model. An ontology is defined as "a formal explicit specification of a shared conceptualisation" [3]. The code domain is formalised as a sub-ontology based on the language constructs of the Java Programming Language. In the documentation sub-ontology, a single generic documentation concept was created. CodeLink uses the ontology to store associated predicates (verbs) and objects (nouns) for traceability relations between code and documentation domains. The associated words are used by CodeLink during its natural language analysis. The CodeLink ontology is further described in Section 2.2.

The semantic wiki component provides users with an user interface to annotate traceability links between documentation and code concepts. CodeLink uses Semantic MediaWiki (SMW) [4] for this component. SMW is a web application written using the PHP[4] language. A MySQL database is used to store relations and attributes. The features of SMW are further describe in Section 2.3.

CodeLink provides a natural language processing (NLP) component in order to identify potential trace-

[4]The PHP Group, `http://www.php.net/`

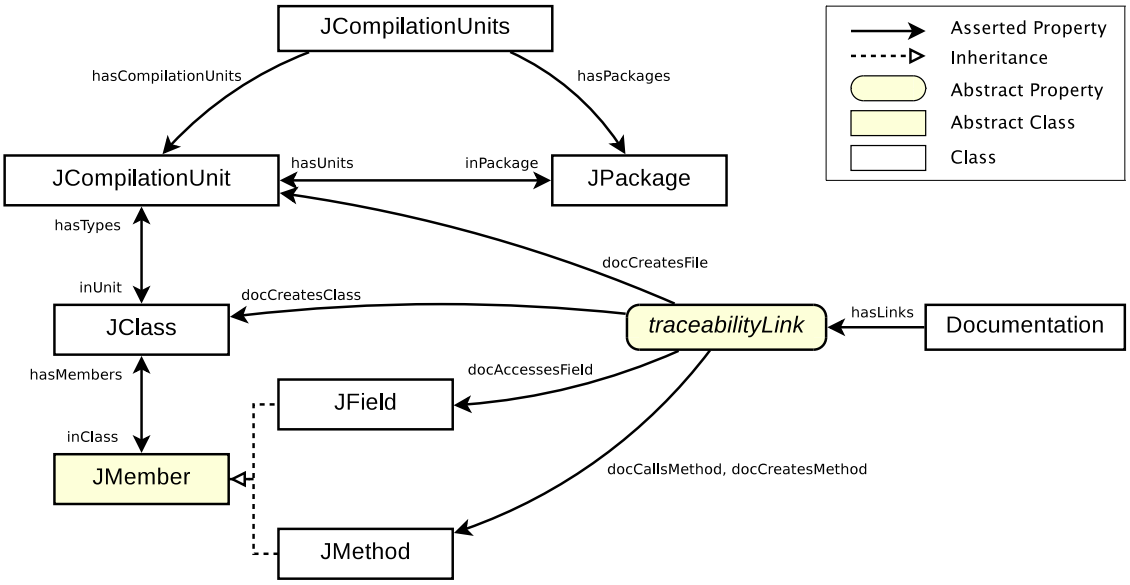

Figure 2: CodeLink Ontology

ability links in the documentation. LingPipe[5], a linguistic analysis framework written in Java, was used as the basis of the NLP component. A web service for the NLP component using Apache AXIS was created in order to interface with SMW. The NLP component performs shallow parsing on the documentation text and suggests traceability annotations to the user. A sketch of the CodeLink documentation process is shown in Figure 1. The details of each step of the natural language analysis are further describe in Section 2.5.

2.2 CodeLink Ontology

The CodeLink ontology consists of the code and documentation sub-ontologies. The CodeLink ontology is shown in Figure 2. Object properties between code and documentation concepts are the traceability links between the two domains. On the code side, the concepts were formalised based on Java, by using the EMF Java metamodel as the basis of the code sub-ontology. On the documentation side, a single generic documentation concept exists. Expanding the documentation ontology to encompass the different types of documentation is future work. Predicate and object annotations are placed on the object properties relating the code and documentation sub-ontologies. The predicate and object annotations are used during the natural language processing stage described in Section 2.5.

The CodeLink ontology was formalised using the Web Ontology Language (OWL) [5]. OWL is an ontology language that is a syntactic extension of RDF-S, however, RDF-S is not fully compatible with OWL [6]. CodeLink avoids this incompatibility by converting from OWL, the more restrictive language, to RDF-S when importing the ontology into SMW.

[5] alias-i, `http://www.alias-i.com/lingpipe/`

One particularly useful feature of OWL is its ability to reason existentially about the data contained, resulting in the ability to derive facts that are not explicitly present in the ontology. An *equivalent class axiom* to capture deprecated documentation in the CodeLink ontology can be written as follows:

$$\mathsf{DeprecatedDocumentation} \sqsubseteq \mathsf{Documentation} \sqcap \exists\, \mathsf{traceabilityLink}\ (\mathsf{JMethod} \sqcap \mathsf{isDeprecated} \in \mathbf{true})$$

In the above class axiom, the DeprecatedDocumentation class is defined as Documentation that contain some traceabilityLink relation with its value being a JMethod that isDeprecated. By using this class axiom, DeprecatedDocumentation is not explicitly created, rather, it is derived from the data stored in the knowledge base. Section 5 describes future extensions to CodeLink that will take advantage of the reasoning capabilities provided by the ontology.

2.3 Semantic MediaWiki

Semantic MediaWiki (SMW) is an extension of the MediaWiki web application with an underlying knowledge model. SMW's knowledge model contains wiki pages, categories, relations and attributes. The SMW knowledge model is similar to the Resource Description Framework (RDF) [7] and RDF-Schema (RDF-S) [8]. In SMW, a wiki page is a RDF resource, a category is similar to a RDF-S class, a relation is similar to a RDF predicate where the object is another resource, and an attribute is similar to a RDF predicate where the object is a literal. The similarities and differences between SMW and RDF/RDF-S are described in further detail below.

Categories in SMW are similar to a RDF-S class. A subclass is represented as a categorisation of a category. An individual is represented as a categorised wiki page. SMW does not have a representation for RDF-S subproperties. The `Category` namespace in SMW is used to store all categories.

A SMW relation is similar to a RDF predicate, however, SMW relations require the object of the triple to be another resource, or wiki page in SMW terms. SMW relations are added to the knowledge model by creating a page in the `Relation` namespace. Establishing a relation between two wiki pages is done by placing a relation annotation on one of the wiki pages. For example, in order to establish a relation in SMW on the page ThisPage, the following annotation is added to the wiki page:

```
[[TestRelation::TargetPage]]
```

The above annotation would establish the TestRelation relation from ThisPage to TargetPage. In terms of RDF, the following triple would be created:

Subject: <http://www.example.com/semanticwiki/ThisPage>
Predicate: <http://www.example.com/semanticwiki/Relation:TestRelation>
Object: <http://www.example.com/semanticwiki/TargetPage> .

SMW attributes are similar to relations, except the object of the triple must be a literal, or built-in type in SMW terms. SMW contains several built-in types such as integer, string and enumeration. SMW attributes are added to the knowledge model by creating a page in the `Attribute` namespace. Similar to a relation, attributes are established by placing an attribute annotation on a wiki page. For example, in order to set the attribute Name to "Hello World!" on the page ThisPage, the following annotation is added:

```
[[Name:=Hello World!]]
```

2.4 Populating the Knowledge Model

The CodeLink ontology is formalised using an OWL ontology, however, SMW requires the use of its own knowledge model. The current release of SMW (version 0.6) provided a built-in OWL import, however, it was not capable of parsing and importing the CodeLink OWL ontology correctly[6]. A custom-built import tool was written for SMW in order to create the wiki categories, relations and attributes specified in the CodeLink ontology. The import tool creates a wiki page for an individual, a category for a class, a relation for an object-property, and an attribute for a datatype property. An annotation in the OWL ontology is translated into an attribute-value pair on the annotated concept's page in SMW. For example, a predicate annotation of "create" on the DocCreatesClass object property in the OWL ontology is translated into setting the Predicate attribute to "create" on the DocCreatesClass relation page in SMW.

On the code domain, application source code is parsed and analysed to extract the source code concepts. The current implementation of CodeLink constructs a separate source code model instead of directly populating the OWL code sub-ontology. Populating the CodeLink code sub-ontology directly is described further as future work in Section 5. The source code model is constructed using the Eclipse Modeling Framework (EMF) in conjunction with the QDox[7] JavaDoc parser. EMF is a modelling and code generation framework for the Eclipse IDE[8]. EMF generates a class library to manipulate and serialise instances of a metamodel. QDox provides an object interface for interacting with Java source files. The source code model is stored as a XML [9] file.

The XML source code model is loaded into SMW through the CodeLink code initialisation interface. CodeLink parses the XML model and generates wiki pages in the for code concepts in the `Code` namespace. Pages for compilation units, packages, classes and methods are generated. The generated pages contain SMW annotations specifying the attributes and relations of the code concept, thus, creating the attributes and relations in the SMW knowledge model. For example, the initialised wiki text for the page Code:java.util.ArrayList appears as follows:

[6] Ontoworld.org, `http://ontoworld.org/wiki/Help:Ontology_import`
[7] QDox, `http://qdox.codehaus.org/`
[8] The Eclipse Foundation, `http://www.eclipse.org/`

```
* Name: [[Name:=ArrayList]]
* Has Compilation Unit: [[HasCompilationUnit::Code:Java.util.ArrayList.java]]
* [[HasMethod::Code:Java.util.ArrayList:size()]]
* [[Category:JClass]]
...
```

In the above wiki text for the Code:java.util.ArrayList page, a Name attribute set to "ArrayList". A HasCompilationUnit relation is established to the `Code:Java.util.ArrayList.java` page. The ArrayList page also contains several HasMethod relations, which only one is shown. The Code:java.util.ArrayList is also categorised as a JClass, making it an instance of the JClass concept in the CodeLink ontology. Several other attributes and relations were omitted to save space.

2.5 Analysing the Documentation

In order to create a traceability link between documentation and code, a traceability annotation must be added to a wiki page. CodeLink discovers traceability annotations for a page using natural language analysis. Early versions of CodeLink automatically applied inferred annotations. While an automatic approach was transparent to the user, the annotation precision was not perfect. As a result, CodeLink adopted a cooperative approach to page annotation. CodeLink uses natural language processing techniques to discover potential annotations. The annotations are presented to the user, and the user may select or modify the annotations to be added to the page. In this section, the natural language analysis process is described.

CodeLink adds the `Code` and `Doc` namespaces to the wiki in order to separate the code and documentation domains. As described in Section 2.4, the `Code` namespace is used to store attributes and relations on the associated source code. The `Doc` namespace is used to store code documentation. CodeLink performs natural language analysis on wiki pages in the `Doc` namespace.

CodeLink begins by extracting valid natural language elements from the wiki page. List items and paragraphs are extracted from the documentation while code blocks are filtered out. Due to the parsing complexity of wiki text, the wiki text is first converted to HTML. Regular expressions are used to extract the appropriate HTML structures. The HTML tags are then removed, leaving only plain text.

The next stage separates the extracted text into sentences through a sentence chunker. Each sentence is then placed into a phrase chunker, which separates phrases based on their part-of-speech (POS) tag into four tag categories: *common nouns*, *proper nouns*, *modal auxiliaries*, *verbs*, and *others*. The first category, *common nouns*, attempts to detect object references to code concepts in the documentation. For example, the word "file" is tagged as a common noun. The *proper nouns* category is used to detect object names in a sentence. CodeLink extends the LingPipe noun chunker in order to detect proper nouns for code concepts. Code concepts follow a syntactic pattern that is distinct from natural language. The chunking algorithm was extended to detect references to compilation units, packages and methods in the documentation. This was done by using a heuristic that chunks words which are connected together by a period with no white space in between. For example, the class java.util.ArrayList is recognised and returned as a single proper noun

Traceability Relation	Predicate Annotation	Object Annotation
DocCreatesClass	create, make	class
DocCreatesCompilationUnit	create, make	file
DocCreatesMethod	create, make	method
DocCallsMethod	call	method
DocAccessesField	access, read, write	field

Table 1: Annotated Object Properties in the CodeLink Ontology

by the extended chunker. The third category, *modal auxiliaries*, attempts to detect the strength of necessity that a predicate within a sentence carries. This is an important distinction in documentation as optional or suggested criteria should not be annotated as an assertion. The identification of modal verbs was inspired by the REVERE system by Sawyer et al. [10]. The fourth category, *verbs*, contains verbs other than modal auxiliaries. This group is used to determine the action to be performed in a sentence. Verbs are especially useful in cookbook documentation [11] since they convey actions that are should be performed in the recipe. The *others* tag category encompasses all other POS tags.

Next, CodeLink attempts to discover traceability links within the sentence. In this stage, the subject, predicate and object are identified in the tagged sentence. In the current version of CodeLink, the subject of the sentence is assumed to be the current documentation page. The predicate of a sentence is assumed to be a word that is tagged as a *verb* by the phrase chunker. The object of a sentence is assumed to be the *common noun* of the sentence. Applicable traceability relations are identified through a mapping from the predicate and object words to traceability relations in the CodeLink ontology. The mapping is specified through predicate and object annotations that are added to the traceability relations in the CodeLink ontology. The predicate annotation contains verbs for which the traceability relations is applicable. The object annotation contains nouns which are used to filter the set of possible traceability relations. For example, the traceability relation DocCreatesCompilationUnit has a predicate annotation of "create" and "make", and a object annotation of "file." The DocCreatesCompilationUnit relation is only applicable if the sentence contains a predicate of "create" and an object of "file". The predicate and object annotations for the traceability relations in the CodeLink ontology are shown in Table 1.

The final step in CodeLink's natural language analysis is the identification of target pages for the traceability relations. The target page is determined by the object name of the sentence. The object name is assumed to be the first *proper noun* following the object. In the case that an object name is not detected in the sentence, the target page is unknown and it is up to the user to provide a page for the traceability relation. On the contrary, when an object name is detected, CodeLink searches through the wiki for pages with a similar Name attribute. All matching pages, if any, are then provided to the user.

Figure 3 provides an example of the natural language analysis performed by CodeLink. This example begins after sentencing chunking with the following sentence extracted from the wiki page: "Create a file named Test.java." The input sentence is then processed by the phrase chunker. The phrase chunker tags "create" as a *verb*, "file" as a *common noun*, and "Test.java" as a *proper noun*. The next stage

Input sentence:
Create a file named Test.java.

Tagged text after phrase chunking:
Create[*verb*] a file[*common noun*] named Test.java[*proper noun*].

CodeLink analysis:

Predicates	Object	Object Names in Wiki
DocCreatesClass DocCreatesCompilationUnit DocCreatesMethod	file	`Code:Com.example.Test.java`

Suggested annotation:
`[[DocCreatesCompilationUnit::Code:Com.example.Test.java]]`

Figure 3: CodeLink Example

attempts to discover traceability relations. The predicate "create", matches three traceability relations in the CodeLink ontology: DocCreatesClass, DocCreatesCompilationUnit, DocCreatesMethod. The traceability relations are then filtered using the object of the sentence. In this case, the object "file", reduces the applicable traceability relations to only one, DocCreatesCompilationUnit. The last step is the identification of the target page for the traceability relation. In this example, "Test.java" is found to be the object name. CodeLink searches through the contents of the wiki and discovers that only one concept, the `Code:Com.example.Test.java` compilation unit, has a matching Name attribute. Now that all the analysis steps are completed, CodeLink will compile a list of suggested annotations. In this case, there is only one: `[[DocCreatesCompilationUnit::Code:Com.example.Test.java]]`.

A screenshot of CodeLink is shown in Figure 4. In this screenshot, CodeLink has performed its natural language analysis on a simple two-step recipe. The tagged phrases and suggested annotations are shown under the "Phrase Chunking" section. Highlighted, non-italic words were tagged as common or proper nouns and words in *italics* were tagged as verbs. The suggested annotations are shown below the tagged phrase. In both cases, only a single relation and target page was suggested. The user has the choice of selecting *other* in order to modify the suggested annotation.

3 Querying the Knowledge Base

In order to retrieve information from the knowledge base, CodeLink uses the semantic query language of SMW. The SMW query language is similar to other RDF query languages, such as SPARQL [12], however, the SMW query language is less expressive than SPARQL. The SMW query language allows for conjunctions, disjunctions and sub-queries using relations and attributes as predicates.

Listing 1 is a SMW query to retrieve all classes within the package `java.util`. The <ask> tag is used to denote the start of a query and the <q> tag is used to denote a sub-query. It is important to note that the

Figure 4: CodeLink Screenshot

```
<ask>
    [[HasCompilationUnit::
        <q>[[HasPackage::java.util]]</q>
    ]]
</ask>
```

Listing 1: SMW query to retrieve all classes in a package

```
<ask>
    [[DocCallsMethod::
        <q>[[HasClass::java.util.ArrayList]] [[Name:=size]]</q>
    ]]
</ask>
```

Listing 2: SMW Query to retrieve documentation that contain a call to a method

```
PREFIX wiki: <http://www.example.com/semanticwiki#>
SELECT ?class
WHERE
  {
    ?class wiki:HasCompilationUnit ?compUnit .
    ?compUnit wiki:HasPackage "java.util" .
  }
```

Listing 3: SPARQL query to retrieve all classes in a package

JCompilationUnit concept is the domain of the HasPackage relation, and the JClass concept is the domain of the HasCompilationUnit relation in the CodeLink ontology. The sub-query is executed first, returning all JCompilationUnit pages contained in the `java.util` package. Each page returned in the sub-query is then used as a value in the parent query. In this case, the parent query returns all JClass pages that have a JCompilationUnit in the `java.util` package.

In Listing 2, a SMW query to retrieve all documentation pages that contain a method call to java.util.ArrayList.size(...) is shown. The sub-query of this example contains a conjunction between two predicates. The first predicate, `[[HasClass::java.util.ArrayList]]`, returns all JMethod pages contained in the JClass java.util.ArrayList. The second predicate, `[[Name:=size]]` returns all pages with a Name attribute of "size". The conjunction, or intersection of the two predicate selects all java.util.ArrayList.size(...) method pages.

CodeLink provides an interface for an alternate query language, SPARQL. A SPARQL query to retrieve all classes in the `java.util` package is shown in Listing 3. SPARQL is more expressive than the SMW query language. Some of the limitations of the SMW query language include: allowing only a single return variable, lack of a negation operator and optional predicates cannot be easily written. While SPARQL provides an improvement over the SMW query language, it is still incapable of using all the reasoning capabilities provided by an ontology. Section 5 discusses several description logic query languages that could be used in the future.

The knowledge model can also be viewed using a relation and attribute browser in SMW. The SMW browser displays the values of relations and attributes for a given page. Using the SMW browser, associated documentation and code concepts can be easily retrieved and navigated.

The querying and browser tools mentioned only work with annotated documentation. The use of automated natural language analysis tools in CodeLink provides a means of performing a semantic search on

unannotated text. Using the CodeLink annotation discovery for search is further discussed in Section 5.

4 Related Work

Literate Programming by Donald Knuth [13] combines documentation and program code into a single source file. In literate programming, the source file is organised in a manner that is meant to be read by a human, rather than a computer. The source file is *tangled* in order to produce compilable code, or *woven* to create documentation. Traceability links between documentation and code concepts are easily established since both are written in the same source file. Literate programming is a very heavyweight approach. Documentation and code languages are both restricted by the literate programming language. Literate programming also assumes that the documentation are written by the same people that write code. In larger projects, this assumption does not hold. Despite these disadvantages, literate programming addresses similar issues as CodeLink. A further look into literate programming is future work.

AbstFinder by Goldin and Berry [14] uses natural language analysis in order to identify *abstraction identifiers* within the text. An abstraction identifier is defined as the set of chunks within a sentence that define an abstraction. AbstFinder treats a sentence as a stream of bytes, ignoring the semantics of a sentence. Abstractions are identified by finding common substrings among the different sentences. The characters in a sentence are circularly shifted in order to account for different orderings of words. CodeLink differs in that its design attempts to take advantage of sentence semantics by using POS tagging and noun and verb chunking. CodeLink also uses an ontology to formalise documentation and code domain concepts. AbstFinder relies on the assumption that some representation of an abstraction will appear as a concept in a single sentence. CodeLink also relies on the same assumption.

In a paper by Gervansi and Nuseibeh [15], shallow parsing techniques are applied on a specification in order to detect and extract concepts. The authors used a set of document-specific rules in order to extract concepts using a domain-based parser. The resulting parse tree is structured in terms of domain concepts as oppose to the document structure. CodeLink currently uses a small set of heuristics in order to extract concepts out of sentences. A rule-based approach will be explored further in the future.

The REVERE system by Sawyer et al. [10] is a support tool for extracting requirements from natural language documents. REVERE relies completely on probabilistic natural language processing techniques. The natural language document is first tagged by a part-of-speech tagger. A semantic tagger is then applied to the tagged text in order to assign a semantic category to single words and a list of common idioms (e.g. “as a rule” or “keep tabs on”). The user is then able to interact with the abstracted view of the text. Although CodeLink and REVERE share a similar process, there is a major difference in the way domain concepts are identified. REVERE relies on using statistical techniques on a training corpus while CodeLink relies on a domain-specific ontology to identify concepts in the natural language text.

XSDoc Wiki by Aguiar and David [16] weaves separate software artifacts such as source code or models into a wiki page. Traceability links can only be explicitly created using wiki tags for different types content

such as [<uml>], or [<javaSource>]. CodeLink infers traceability links in the documentation through the use of natural language analysis. In addition, CodeLink uses an ontology to formalise documentation and code concepts.

5 Future Work

Ontology Integration CodeLink currently uses an OWL ontology for specifying the concepts and relations, and a XML file for source code data. Removing the separate XML file and directly populating the OWL ontology with instance data is future work. The current implementation of CodeLink uses the EMF generated framework in order to serialise the source code data, however, EMF supports only the XML Metadata Interchange (XMI) and XML formats. The Eclipse EODM[9] project adds OWL compatibility to EMF. Using EODM, it would be possible to use the EMF framework to populate instance data into an OWL ontology.

In addition to using the OWL ontology to store instance data, a tighter integration of the CodeLink OWL ontology with SMW is planned. The current implementation of SMW uses database tables to store relations and attributes as subject-predicate-object triples. While a database implementation is efficient, it does not take advantage of the reasoning capabilities of an ontology.

Ontology Reasoning The SMW query language and SPARQL are insufficient for sophisticated reasoning on the knowledge model. One example where using ontology reasoning is advantageous over the current model was presented in Section 2.2. nRQL [17] is a description logic query language that can reason existentially on an ontology. The following is a nRQL query to retrieve all Documentation with a traceabilityLink to a deprecated JMethod:

```
(retrieve (?doc)
    (and (?doc Documentation)
         (?method JMethod)
         (and (?*depMethod isDeprecated) (?*depMethod true))
         (?doc ?method traceabilityLink)))
```

Future work will include identifying how the expressiveness of the description logic query languages can be used to retrieve useful traceability links from the knowledge base.

Using Synonyms The verb and noun mappings are manually populated in the current version of CodeLink. Annotation suggestions could be improved if synonyms were retrieved for each verb or noun. Using an approach described by Turney [18] to discover analogy pairs, synonyms could similarly be retrieved using a thesaurus.

[9] http://www.eclipse.org/modeling/mdt/?project=eodm

Improving Natural Language Recognition The current implementation of CodeLink applies shallow parsing to natural language text. Sentences much conform to a limited verb-object pattern in order to be recognised by CodeLink. Using deeper semantic analysis of the text could help to improve coverage. CodeLink currently uses the Brown corpus as its training data. Using a technical corpus, or even simply a bigger corpus could provide better word sense disambiguation, more accurate collocations and better part-of-speech tagging.

Searching Over Unannotated Text Extending the search capability of SMW is future work. The natural language analysis of CodeLink could be applied on unannotated text in order to derive suggested annotations about the documentation. The derived annotations could be useful in identifying related, but unannotated pages during a search. Search provides a lightweight approach to using the capabilities of CodeLink, allowing users to slowly transition their documentation to a fully annotated form.

Similar or Related Documentation Developing a similarity measure between documentation pages is future work. The annotations on a documentation page could be used as the basis of the similarity measure. Using a similarity measure, documentation could be clustered and related documentation could be identified.

Software Evolution CodeLink can help developers to identify outdated documentation. By establishing traceability links between documentation and code concepts, CodeLink is able to map changes in the code to affected documentation. As a result, maintenance efforts can concentrate on the affected documentation. Taking it a step further, CodeLink can attempt to automatically propagate code changes to documentation and vice versa. Automatically updating documentation requires the use of model comparison and merging techniques. In a paper by Brunet et al. [19], model merging as algebraic operations over models and relationships are described. Operators such as *match*, *diff* and *check-property* are described in the paper. The techniques describes in the paper could be applied in CodeLink to detect changes in the code and documentation models, however, further research is required in this area.

Code Comments Code comments provide developers with a means of adding natural language text to their code. Some of the comments, such as the JavaDoc tags `@see` and `@link` explicitly create links between code and documentation elements. Extracting these traceability links using the QDox JavaDoc parser is future work.

Occasionally, code comments are used to store documentation. For example, several methods in the Eclipse IDE contain guidelines and examples embedded as a comment header[10]. This in-line documentation is very similar to a cookbook recipe, since it provides a suggested framework solution to a problem. Further research is required to identify how this form of documentation can be incorporated into CodeLink.

[10]Refer to The Eclipse Foundation, Eclipse SDK—ViewPart JavaDoc

Generic Platform for Traceability An exciting prospect for CodeLink is the generalisation of the tool to enable traceability between domain concepts based on ontologies. CodeLink currently uses an ontology for source code as its knowledge model. The source code ontology could be replaced by an ontology in a completely separate domain. Annotations in the ontology could be used to provide a dictionary for natural language similar to the Predicate and Object annotations currently in use by CodeLink. For example, a banking ontology could be used in place of the current CodeLink ontology. References to transactions, such as "debit" and "credit" could be detecting in the wiki text using the approach described in this paper. Establishing CodeLink as a generic platform for traceability is potential long-term future work.

6 Conclusions

In this paper, CodeLink, a semantic wiki for code documentation is presented. CodeLink enables developers to harness the inherent connection between software documentation and the code it describes. An ontology is used to formalise documentation and code concepts, and a semantic wiki is used to manage and present the knowledge model to the user. CodeLink discovers traceability links between software documentation and code concepts using natural language analysis. Traceability links are presented to the user as suggested annotations.

CodeLink consists of three components: knowledge model, semantic wiki, and natural language processing. The knowledge model provides a formalisation of the documentation and code concepts and relations. The semantic wiki component serves as a frontend for interacting with the knowledge model. The natural language processing component performs shallow parsing on wiki text to discover potential traceability links.

The natural language processing (NLP) component of CodeLink was of particular focus in this paper. The NLP component begins by extracting natural language paragraphs from the wiki text. The natural language paragraphs are then chunked into sentences, then each sentence is chunked into tagged phrases. Five tag categories are used to separate the tagged phrases: *common nouns*, *proper nouns*, *modal auxiliaries*, *verbs* and *others*. A custom proper noun chunker was developed in order to detect references to code concepts. CodeLink identifies traceability links through the use of annotated traceability relations in the ontology. Predicate and object annotations are placed on traceability relations as conditions that a natural language sentence must be fulfilled in order for the relations to be valid. The semantic wiki knowledge model is used as a dictionary for matching object names.

CodeLink is still in its early stages, however, it provides an extensible platform for research in establishing traceability between natural language and a formal knowledge model. Through the use of natural language analysis, traceability links from natural language phrases to formal concepts in an ontology can be established. Much work remains to be done before CodeLink achieves its full potential. In spite of this, the early prototype is promising and further work will bring CodeLink a step closer to its goal of connecting software documentation with the very code it describes.

References

[1] G. E. Krasner and S. T. Pope, "A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[2] A. Schappert, P. Sommerlad, and W. Pree, "Automated support for software development with frameworks," in *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, (New York, NY, USA), pp. 123–127, ACM Press, 1995.

[3] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquis.*, vol. 5, no. 2, pp. 199–220, 1993.

[4] M. Krötzsch, D. Vrandecic, and M. Völkel, "Semantic mediawiki.," in *International Semantic Web Conference* (I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, eds.), vol. 4273 of *Lecture Notes in Computer Science*, pp. 935–942, Springer, 2006.

[5] D. L. McGuinness and F. van Harmelen, "OWL web ontology language overview," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[6] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: combining logic programs with description logic," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*, (New York, NY, USA), pp. 48–57, ACM Press, 2003.

[7] F. Manola and E. Miller, "RDF primer," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[8] D. Brickley and R. V. Guha, "RDF vocabulary description language 1.0: RDF schema," W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[9] E. Maler, T. Bray, C. M. Sperberg-McQueen, F. Yergeau, and J. Paoli, "Extensible markup language (XML) 1.0 (fourth edition)," W3C recommendation, W3C, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[10] P. Sawyer, P. Rayson, and R. Garside, "Revere: Support for requirements synthesis from documents," *Information Systems Frontiers*, vol. 4, no. 3, pp. 343–353, 2002.

[11] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad, "Active guidance of framework development," *Software - Concepts and Tools*, vol. 16, no. 3, pp. 136–, 1995.

[12] A. Seaborne and E. Prud'hommeaux, "SPARQL query language for RDF," candidate recommendation, W3C, June 2007. http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/.

[13] D. E. Knuth, "Literate programming," *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.

[14] L. Goldin and D. M. Berry, "Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation," *Automated Software Engg.*, vol. 4, no. 4, pp. 375–412, 1997.

[15] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements: A case study," in *ICRE '00: Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, (Washington, DC, USA), p. 140, IEEE Computer Society, 2000.

[16] A. Aguiar and G. David, "Wikiwiki weaving heterogeneous software artifacts," in *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, (New York, NY, USA), pp. 67–74, ACM Press, 2005.

[17] V. Haarslev, R. Möller, and M. Wessel, "Querying the semantic web with racer + nrql," in *Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04), Ulm, Germany, September 24*, 2004.

[18] P. D. Turney, "Similarity of semantic relations," *Comput. Linguist.*, vol. 32, no. 3, pp. 379–416, 2006.

[19] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, (New York, NY, USA), pp. 5–12, ACM Press, 2006.