

Ontological Text Mining of Software Documents

René Witte¹, Qiangqiang Li¹, Yonggang Zhang², and Juergen Rilling²

¹ Institut für Programmstrukturen und Datenorganisation (IPD)
Universität Karlsruhe (TH), Germany

² Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada

Abstract. Documents written in natural languages constitute a major part of the software engineering lifecycle artifacts. Especially during software maintenance or reverse engineering, semantic information conveyed in these documents can provide important knowledge for the software engineer. In this paper, we present a text mining system capable of populating a software ontology with information detected in documents.

1 Introduction

With the ever increasing number of computers and their support for business processes, an estimated 250 billion lines of source code were being maintained in 2000, with that number rapidly increasing [1]. The relative cost of maintaining and managing the evolution of this large software base now represents more than 90% of the total cost [2] associated with a software product. One of the major challenges for software engineers while performing a maintenance task is the need to comprehend a multitude of often disconnected artifacts created originally as part of the software development process [3]. These artifacts include, among others, source code and corresponding software documents, e.g., requirements specifications, design description, and user's guides. From a maintainer's perspective, it becomes essential to establish and maintain the semantic connections among all these artifacts. Automated source code analysis, implemented in integrated development environments like *Eclipse*, has improved software maintenance significantly. However, integrating the often large amount of corresponding documentation requires new approaches to the analysis of natural language documents that go beyond simple full-text search or information retrieval (IR) techniques [4].

In this paper, we propose a Text Mining (TM) approach to analyse software documents on a semantic level. A particular feature of our system is its use of formal ontologies (in OWL-DL format) during both, the analysis process and as an export format for the results. In combination with a source code analysis system for populating code-specific parts of the ontology, we can now represent knowledge concerning *both* code *and* documents in a single, unified representation. This common, formal representation supports further analysis of the knowledge base, like the automatic establishment of traceability links. A general overview of the proposed process is shown in Fig. 1.

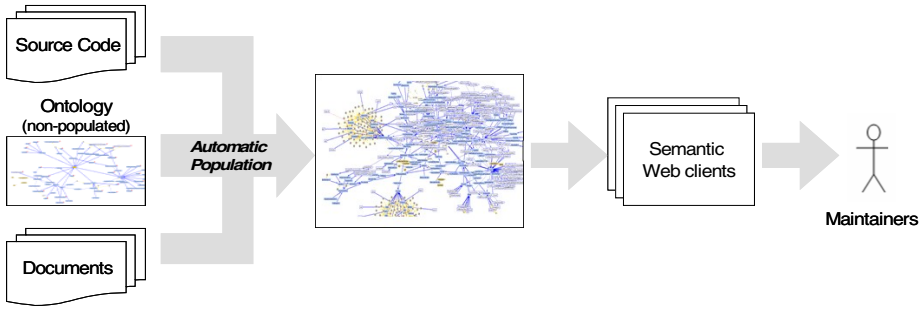


Fig. 1. Ontological Text Mining of software documents for software engineering

2 Ontological Text Mining for Software Documents

In this section, we present a brief motivation and overview of our ontology-based software environment and then discuss the text mining component in detail.

2.1 Software Engineering and NLP

As software ages, the task of maintaining it becomes more complex and more expensive. Software maintenance, often also referred to as software evolution, constitutes a majority of the total cost occurring during the life span of a software system [1, 2]. Software maintenance is a multi-dimensional problem space that creates an ongoing challenge for both the research community and tool developers [5, 6]. These maintenance challenges are caused by the different representations and interrelationships that exist among software artifacts and knowledge resources [7, 8]. From a maintainer’s perspective, exploring [9] and linking these artifacts and knowledge resources becomes a key challenge [4]. What is needed is a unified representation that allows maintainers to explore, query and reason about these artifacts, while performing their maintenance tasks [10].

Information contained in software documents is important for a multitude of software engineering tasks, but within this paper, we focus on a particular use case: the concept location and traceability across different software artifacts. From a maintainer’s perspective, software documentation contains valuable information of both functional and non-functional requirements, as well as information related to the application domain. This knowledge often is difficult or impossible to extract only from source code [11]. 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 [4]. As a result, maintainers have to spend a large amount of time on synthesizing and integrating information from various information sources in order to re-establish the traceability links among these artifacts.

Our approach is based on a common formal representation of both source code and software documentation using an ontology in OWL-DL format [12]. Instances are populated automatically through automatic code analysis (described in [13]) and text mining (described in this paper).

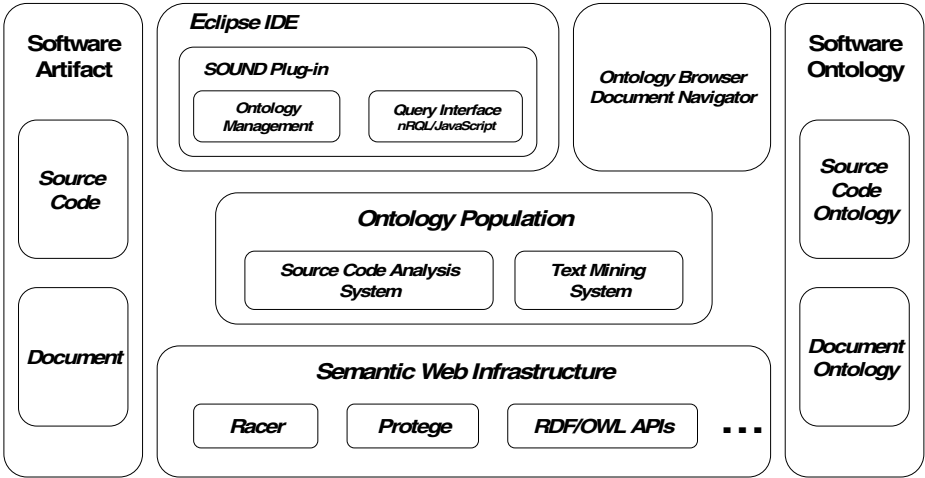


Fig. 2. Ontology-based program comprehension environment overview

2.2 System Architecture and Implementation Overview

In order to utilize the structural and semantic information in various software artifacts, we have developed an ontology-based program comprehension environment, which can automatically extract concept instances and their relations from source code and documents (Fig. 2).

An important part of our architecture is a software ontology that captures major concepts and relations in the software maintenance domain. This ontology consists of two sub-ontologies: a *source code* and *document* ontology, which represent information extracted from source code and documents, respectively. The ontologies are modeled in OWL-DL and were created using the OWL extension of Protégé,¹ a free ontology editor. Racer [14], an ontology inference engine, is integrated to provide reasoning services. Racer is a highly optimized DL system that supports reasoning about instances, which is particularly useful for the software maintenance domain, where a large amount of instances needs to be handled efficiently. Automatic ontology population is handled by two sub-systems: The source code analysis, which is based on the JDT Java parser² provided by Eclipse [13]; and the document analysis using the text mining system discussed in this paper. The query interface of our system is a plug-in that provides OWL integration for Eclipse, a widely used software development platform. The expressive query language nRQL provided by Racer can be used to query and reason over the populated ontology. Additionally, we integrated a scripting language, which provides a set of built-in functions and classes using

¹ Protégé ontology editor, <http://protege.stanford.edu/>

² Eclipse Java Development Tools (JDT), <http://www.eclipse.org/jdt/>

the JavaScript interpreter Rhino.³ This language simplifies querying the ontology for software engineers not familiar with DL-based formalisms.

2.3 Software Document 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. Additionally, the software documentation sub-ontology has been specifically designed for automatic population through a text mining system. In particular, we included: (1) A *Text Model* to represent the structure of documents, e.g., classes for sentences, paragraphs, and text positions, as well as NLP-related concepts that are discovered during the analysis process, like noun phrases (NPs) and coreference chains. These are required for anchoring detected entities (populated instances) in their originating documents. (2) *Lexical Information* facilitating the detection of entities in documents, like the names of common design patterns, programming language-specific keywords, or architectural styles. (3) *Lexical normalization rules* for entity normalization. (4) *Relations* between the classes, which extend the ones modeled in the source code ontology. These allow us to automatically restrict NLP-detected relations to semantically valid ones. For example, a relation like `<variable> implements <interface>`, which can result from parsing a grammatically ambiguous sentence, can be filtered out since it is not supported by the ontology. Finally, (5) *Source Code Entities* that have been automatically populated through source code analysis can also be utilized for detecting corresponding entities in documents, as we describe below.

2.4 Ontology Population Through Text Mining

We developed our text mining system for populating the software documentation ontology based on the GATE (*General Architecture for Text Engineering*) framework [15]. The system is component-based, utilizing both standard tools shipped with GATE and custom components developed specifically for software text mining. An overview of the workflow is shown in Fig. 3. In the following discussion, we omit several standard NLP analysis steps, like part-of-speech (POS) tagging, noun phrase (NP) chunking, or stemming. For readers unfamiliar with these tasks, we refer to the GATE user's guide.⁴

Ontology Initialization. When analysing documents specific to a source code base, our text mining system can take instances detected by the automatic code analysis into account. This is achieved in two steps: first, the source code ontology is populated with information detected through static and dynamic code analysis [13]. This step adds instances like method names, class names, or detected design patterns to the software ontology. In a second step, we use this information as additional input to the OntoGazetteer component for named entity recognition.

³ Rhino JavaScript interpreter, <http://www.mozilla.org/rhino/>

⁴ GATE user's guide, <http://gate.ac.uk/sale/tao/index.html>

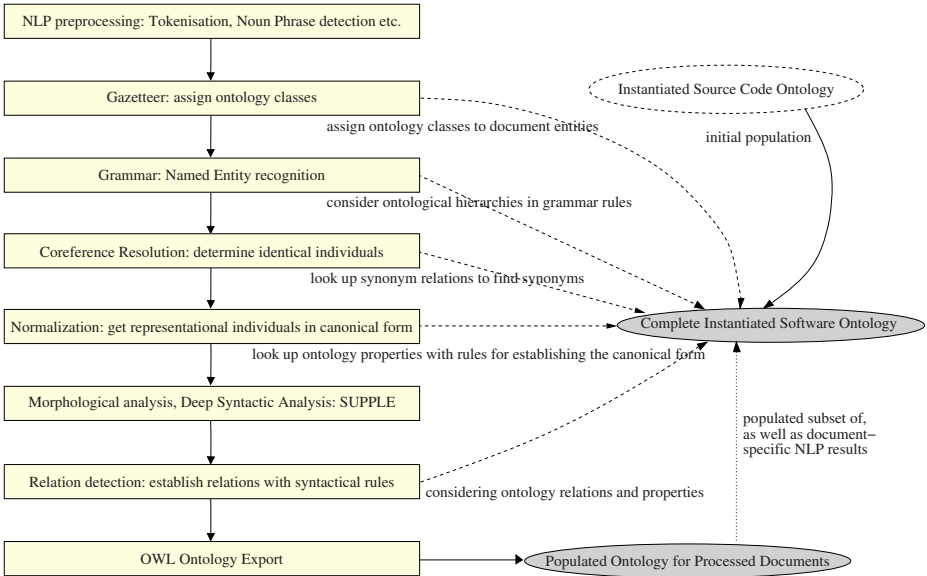


Fig. 3. Workflow of the software text mining subsystem

Named Entity Detection. The basic process in GATE for recognizing entities of a particular domain starts with the gazetteer component. It matches given lists of terms against the tokens of an analysed text and, in case of a match, adds an annotation named `Lookup` whose features depend on the list where the match was found. Its ontology-aware counterpart is the *OntoGazetteer*, which incorporates mappings between its term lists and ontology classes and assigns the proper class in case of a term match. For example, using the instantiated software ontology, the gazetteer will annotate the text segment *method* with a `Lookup` annotation that has its `class` feature set to “Method.” Here, incorporating the results from automatic code analysis can significantly boost recall (cf. Section 3), since entity names in the software domain typically do not follow naming rules, like in the biological domain.

In a second step, grammar rules written in the JAPE⁵ language are used to detect and annotate complex named entities. Those rules can refer to the `Lookup` annotation generated by the *OntoGazetteer*, and also evaluate the ontology directly. For example, when performing a comparison like `class=="Keyword"` in a grammar rule, the ontological hierarchy is taken into account so that also a `Java_keyword` matches, since a Java keyword *is-a* keyword in the ontology. This feature significantly reduces the overhead for grammar development and testing.

The developed JAPE rules combine ontology-based lookup information with noun phrase (NP) chunks to detect semantic units. NP chunking is performed

⁵ JAPE is a regular-expression based language for writing grammars over annotations, from which finite-state transducers are generated by a GATE component.

using the MuNPEX chunker,⁶ which relies mostly on part-of-speech (POS) tags, but can also take the lookup information into account. This way, it can prevent bad NP chunks caused by mis-tagged software entities (e.g., method names or program keywords tagged as verbs). Essentially, we combine two complementary approaches for entity detection: A *keyword*-based approach, relying on lexical information stored in the documentation ontology (see above). For example, the text segment “*the storeAttribute() method...*” will be annotated with a lookup information indicating that the word *method* belongs to the ontology class “Method.” Likewise, the same segment will be annotated as a single noun phrase, showing determiner (“*the*”), modifier (“*storeAttribute()*”), and head noun (“*method*”). Using an ontology-based grammar rule implemented in JAPE, we can now combine these two information and semantically mark the NP as a method. Similar rules are used to detect variables, class names, design patterns, or architectural descriptions. Note that this approach does not need to know about “*storeAttribute()*” being a method name; this fact is induced from a combination of grammatical (NP chunks) and lexical (ontology) information.

The second approach relies on source code analysis results stored in the initialized software ontology (see above). Every method, class, package, etc. name will be automatically represented by an instance in the source code sub-ontology and can thus be used by the OntoGazetteer for entity detection. This applies also in case when these instances appear outside a grammatical construct recognized by our hand-crafted rules. This is especially useful for analysing software documents in conjunction with their source code, the primary scenario our system was designed for.

Coreference Resolution. We use a fuzzy set theory-based coreference resolution system [16] for grouping detected entities into *coreference chains*. Each chain represents an equivalence class of textual descriptors occurring within or across documents. Not surprisingly, our fuzzy heuristics developed originally for the news domain (e.g., using *WordNet*) were particularly ineffective for detecting coreference in the software domain. Hence, we developed an extended set of heuristics dealing with both pronominal and nominal coreferences.

For *nominal coreferences*, we rely on three main heuristics. The first is based on simple string equality (ignoring case). The second heuristic establishes coreference between two entities if they become identical when their NPs’ HEAD and MOD slots are inverted, as in “*the selectState() method*” and “*method selectState()*”. The third heuristic deals with a number of grammatical constructs often used in software documents that indicate synonymous entities. For example, in the text fragment “... we have an action class called *ViewContentAction*, which is invoked.” we can identify the NPs “*an action class*” and “*ViewContentAction*” as being part of the same coreference chain. This heuristic only considers entities of the same ontology class, connected by a number of pre-defined relation words (e.g., “named”, “called”), which are also stored in the ontology.

⁶ MuNPEX, <http://www.ipd.uka.de/~durm/tm/munpex/>

Table 1. Lexical normalization rules for various ontology classes

Ontology Class	H	DH	MH(cM)	MH(cH)	DMH(cM)	DMH(cH)
Class	H	H	H	lastM	H	lastM
Method	H	H	H	lastM	H	lastM
LayeredArchitecture	H	H	MH	MH	MH	MH
AbstractFactory	H	H	MH	MH	MH	MH
OO_Interface	H	H	H	lastM	H	lastM

For *pronominal resolution*, we implemented a number of simple sub-heuristics dealing only with 3rd person singular and plural pronouns: *it*, *they*, *this*, *them*, and *that*. The last three can also appear in qualified form (*this method*, *that constructor*). We employ a simple resolution algorithm, searching for the closest anaphorical referent that matches the case and, if applicable, the semantic class.

Normalization. Normalization needs to decide on a canonical name for each entity, like a class or method name. This is important for ontology population, as an instance, like of the ontology class **Method**, should reflect only the method name, omitting any additional grammatical constructs like determiners or possessives. Thus, a named entity like “*the static TestClass() constructor*” has to be normalized to “**TestClass**” before it can become an instance (ABox) of the class **Method** (TBox) in the populated ontology.

This step is performed through a set of lexical normalization rules, which are stored with their corresponding classes in the software document sub-ontology, allowing us to inherit rules through subsumption. Table 1 shows a number of these rules for various ontology classes: D, M, H refer to determiner, modifier, and head, respectively, and $c(x)$ denotes the ontology class of a particular slot; the table entry determines what part of a noun phrase is selected as the normalized form, which is then stored as a feature in the entity’s annotation.

Relation Detection. The next major step is the detection of *relations* between entities, e.g., to find out which interface a class is implementing, or which method belongs to which class. Relation detection in our system is again done with two complementary approaches: a set of hand-crafted grammar rules implemented in JAPE, and a deep syntactic analysis using the SUPPLE parser. Afterwards, detected relations are filtered through the software ontology to erase semantically invalid results. We now describe these steps in detail.

Rule-Based Relation Detection. Similarly to entity recognition, rule-based relation detection is performed in a two-step process: first, a JAPE-based transducer is run to detect verb groups (VGs) based on POS tags. Then, tokens that are candidates for relation predicates (e.g., “implements,” “extends”) are marked by the OntoGazetteer. Combining these two information, we can create custom JAPE rules to detect relations between entities detected previously, for example, to find the classes creating a certain design pattern or to find relations between described classes and methods. Using the voice information (active/passive)

provided by the VG chunker, we can then assign subject/object slots for the entities participating in a relation.

Deep Syntactic Analysis. For a deep syntactic analysis, we currently employ the SUPPLE parser [17], which is integrated into GATE through a wrapper component. SUPPLE is a general-purpose bottom-up chart parser for feature-based context-free phrase structure grammars, implemented in Prolog. It produces syntactic as well as semantic annotations to a given sentence. Grammars are applied in series allowing to choose the best parse for each step and continue to the next layer of grammatical analysis with only the selected best parse. The identification of verbal arguments and attachment of nominal and verbal post-modifiers, such as prepositional phrases and relative clauses, is done conservatively. Instead of producing all possible analyses or using probabilities to generate the most likely analysis, SUPPLE only offers a single analysis that spans the input sentence only if it can be relied on to be correct, so that in many cases only partial analyses are produced. SUPPLE outputs a logical form, which is then matched with the entities detected previously to obtain predicate-argument structures.

Result Integration. The results from both rule- and parser-based relation detection form the candidate set for ontology relation instances created for a text. As both approaches may result in false positives, e.g., through ambiguous syntactical structures or rule mismatches, we prune the set by checking each candidate relation for semantic correctness using our software ontology. As each entity participating in a relation has a corresponding ontology class, we can query the ontology to check whether the detected relation (or one of its supertypes) exists between these classes. This way, we can filter out semantically incorrect relations like a *variable* “implementing” an *interface* or a *design pattern* being “part-of” a *class*, thereby significantly improving precision (cf. Section 3).

Note that relation detection and filtering is one particular example where an ontology delivers additional benefit when compared with classical NLP techniques like plain gazetteering lists or statistical/rule-based systems [18].

Ontology Export. Finally, the instances found in the document and the relations between them are exported to an OWL-DL ontology. Note that entities provided by source code analysis are only exported in the document ontology if they have also been detected in a text (cf. Fig. 3).

In our implementation, ontology population is done by a custom GATE component, the *OwlExporter*, which is application domain-independent. It collects two special annotations, `OwlExportClass` and `OwlExportRelation`, which specify instances of classes and relations (i.e., object properties), respectively. These must in turn be created by application-specific components, since the decisions as to which annotations have to be exported, and what their OWL property values are, depend on the domain.

The class annotation carries the name of the class, a name for the instance (the normalized name created previously), and the GATE internal ID of an annotation representing the instance in the document. If there are several occurrences of the same entity in the document, the final representation annotation

Table 2. Evaluation results: Entity recognition and normalization performance

Corpus	Text Mining Only				With Source Ontology			
	<i>P</i>	<i>R</i>	<i>F</i>	<i>A</i>	<i>P</i>	<i>R</i>	<i>F</i>	<i>A</i>
Java Collections	0.89	0.67	0.69	75%	0.76	0.87	0.79	88%
uDig	0.91	0.57	0.59	82%	0.58	0.87	0.60	84%
Total	0.90	0.62	0.64	77%	0.67	0.87	0.70	87%

is chosen from the ones in the coreference chain by the component creating the `OwlExportClass` annotation. In case of the software text mining system, a single representative has to be chosen from each coreference chain. Remember that one chain corresponds to a single semantic unit, so the final, exported ontology must only contain one entry for, e.g., a method, not one instance for every occurrence of that method in a document set. We select the representative using a number of heuristics, basically assuming that the longest NP that has more slots (DET, MOD, HEAD) filled is also the most salient one.

From this representative annotation, all further information is gathered. After reading the class name, the `OwlExporter` queries the ontology via the `Jena`⁷ framework for the class properties and then searches for equally named features in the representation annotation, using their values to set the OWL properties.

3 Evaluation

So far, we evaluated our text mining subsystem on two collections of texts: a set of 5 documents (7743 words) taken from the Java documentation for the *Collections* framework⁸ and a set of 7 documents (3656 words) from the documentation of the *uDig*⁹ geographic information system (GIS). The document sets were chosen because of the availability of the corresponding source code.

Both sets were manually annotated for named entities, including their ontology classes and normalized form, as well as relations between the entities. In what follows, we present results for the named entity recognition, entity normalization, and relation detection tasks.

Named Entity Recognition Evaluation. We computed the standard precision, recall, and F-measure results for NE detection. A named entity was only counted as correct if it matched both the textual description and ontology class. Table 2 shows the results for two experiments: first running only the text mining system over the corpora (left side) and second, performing the same evaluation after running the code analysis, using the populated source code ontology as an additional resource for NE detection as described above. As can be seen, the text mining system achieves a very high precision (90%) in the NE detection task,

⁷ Jena Semantic Web Framework for Java, <http://jena.sourceforge.net/>

⁸ Java Collections Framework Documentation, <http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>

⁹ uDig GIS Documentation, <http://udig.refractor.net/>

Table 3. Evaluation results: Relation detection performance

Corpus	Before Filtering			After Filtering			
	<i>P</i>	<i>R</i>	<i>F</i>	<i>P</i>	<i>R</i>	<i>F</i>	ΔP
Text Mining Only							
Java Collections	0.35	0.24	0.29	0.50	0.24	0.32	30%
uDig	0.46	0.34	0.39	0.55	0.34	0.42	16%
Total	0.41	0.29	0.34	0.53	0.29	0.37	23%
With Source Ontology							
Java Collections	0.14	0.36	0.20	0.20	0.36	0.25	30%
uDig	0.11	0.41	0.17	0.24	0.41	0.30	54%
Total	0.13	0.39	0.19	0.22	0.39	0.23	41%

with a recall of 62%. With the imported source code instances, these numbers become reversed: the system can now correctly detect 87% of all entities, but with a lower precision of 67%.

The drop in precision after code analysis is mainly due to two reasons. Since names in the software domain do not have to follow any naming conventions, simple nouns or verbs often used in a text will be mis-tagged after being identified as an entity appearing in a source code. For example, the Java method *sort* from the collections interface will cause all instances of the word “sort” in a text to be marked as a method name. Another precision hit is due to the current handling of class constructor methods, which are typically identical to the class name. Currently, the system cannot distinguish the class name from the constructor name, assigning both ontology classes (i.e., **Constructor** and **OO_Class**) for a text segment, where one will always be counted as a false positive.

Both cases require additional strategies when importing entities from source code analysis, which are currently under development. However, the current results already underline the feasibility of our approach of integrating code analysis and NLP.

Entity Normalization Evaluation. We also evaluated the performance of our lexical normalization rules for entity normalization, since correctly normalized names are a prerequisite for the correct population of the result ontology. For each entity, we manually annotated the normalized form and computed the accuracy *A* as the percentage of correctly normalized entities over all correctly identified entities. Table 2 shows the results for both the system running in text mining mode alone and with additional source code analysis. As can be seen from the table, the normalization component performs rather well.

Relation Detection Evaluation. Not surprisingly, relation detection was the hardest subtask within the system. Like for entity detection, we performed two different experiments, with and without source code analysis results. Additionally, we evaluated the influence of the semantic relation filtering step using our ontology as described above. The results are summarized in Table 3. As can be seen, the current combination of rules with the SUPPLE parser does not achieve

a high performance. However, the increase in precision (ΔP) when applying the filtering step using our ontology is significant: upto 54% better than without semantic filtering.

The overall low precision and recall values are mainly due to the unchanged SUPPLE parser rules, which have not yet been adapted to the software domain. Also, the conservative PP-attachement strategy of SUPPLE misses many predicate-argument structures. We currently experiment with different parsers (RASP and MiniPar) and are also adapting the SUPPLE grammar rules in order to improve the detection of predicate-argument structures.

4 Related Work and Discussion

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 [19, 20] or detect inconsistent requirements [21]. 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 combine source code analysis with the text mining of software documents, which is an important contribution of our work.

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 [4], as well as latent semantic indexing (LSI) [22]. In contrast with these IR approaches, our work also takes advantage of structural and semantic information in both the documentation and source code by means of text mining and source code parsing.

5 Conclusions and Future Work

We presented a text mining system for the software domain that is capable of extracting entities from software documents. The system's output is a populated OWL-DL ontology containing normalized instances and their relations. The system is novel in two important aspects: First, it employs a formal ontology, based on description logics, both as a processing resource for the various NLP components and the result export format. Second, as the system is part of a larger ontology-based program comprehension environment, it can incorporate results from automated source code analysis subsystems in its NLP processing pipeline.

The ontological foundation allows for important improvements in software engineering, as it supports *queries* and *reasoning* services on semantic knowledge automatically derived from large amounts of documentation in natural language form. We previously showed how automated reasoning can support a software maintainer when performing knowledge-intensive tasks, like architectural recovery or source code security analysis. We are also currently experimenting with ontology alignment strategies to automatically establish links between code and

its corresponding documentation [13]. This will allow, for the first time, the automatic establishment and analysis of traceability links, which is of high importance for the software industry.

Besides improving the individual components as discussed in the evaluation section, we plan to extend our system to explicitly deal with documents associated with the different steps in the software life-cycle, from white papers and requirements over design and implementation documents to user's guides and source code comments. This will allow us to trace concepts and entities across the different states of software development and different levels of abstraction.

References

1. Sommerville, I.: *Software Engineering*, 6th edn. Addison-Wesley, Reading (2000)
2. Seacord, R., Plakosh, D., Lewis, G.: *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI Series in SE. Addison-Wesley, Reading (2003)
3. Jin, D., Cordy, J.: *Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology*. In: 21st IEEE International Conference on Software Maintenance (ICSM) (2005)
4. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D.: *Information retrieval models for recovering traceability links between code and documentation*. In: Proc. of IEEE Intl. Conf. on Software Maintenance, San Jose, CA, USA (2000)
5. IEEE: *IEEE Standard for Software Maintenance*. IEEE 1219 (1998)
6. Riva, C.: *Reverse Architecting: An Industrial Experience Report*. In: 7th IEEE Working Conference on Reverse Engineering (WCRE), pp. 42–52 (2000)
7. Storey, M.A., Sim, S.E., Wong, K.: *A Collaborative Demonstration of Reverse Engineering tools*. ACM SIGAPP Applied Computing Review 10(1), 18–25 (2002)
8. Welty, C.: *Augmenting Abstract Syntax Trees for Program Understanding*. In: Proc. of Int. Conf. on Automated Software Engineering, pp. 126–133. IEEE Computer Society Press, Los Alamitos (1997)
9. Lethbridge, T.C., Nicholas, A.: *Architecture of a Source Code Exploration Tool: A Software Engineering Case Study*. Technical Report TR-97-07, Department of Computer Science, University of Ottawa (1997)
10. Meng, W., Rilling, J., Zhang, Y., Witte, R., Charland, P.: *An Ontological Software Comprehension Process Model*. In: 3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM, Genoa, Italy (October 1st 2006)), pp. 28–35 (2006)
11. Lindvall, M., Sandahl, K.: *How well do experienced software developers predict software change?* Journal of Systems and Software 43(1), 19–27 (1998)
12. Johnson-Laird, P.N.: *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness*. Harvard University, Cambridge, MA (1983)
13. Rilling, J., Witte, R., Zhang, Y.: *Automatic Traceability Recovery: An Ontological Approach*. In: International Symposium on Grand Challenges in Traceability (GCT'07), Lexington, Kentucky, USA (March 22–23, 2007)
14. Haarslev, V., Möller, R., RACER,: *System Description*. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 701–705. Springer, Heidelberg (2001)

15. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: Proc. of the 40th Anniversary Meeting of the ACL (2002)
16. Witte, R., Bergler, S.: Fuzzy Coreference Resolution for Summarization. In: Proceedings of 2003 International Symposium on Reference Resolution and Its Applications to Question Answering and Summarization (ARQAS), Venice, Italy, Università Ca' Foscari (June 23–24 2003), pp. 43–50 <http://rene-witte.net>
17. Gaizauskas, R., Hepple, M., Saggion, H., Greenwood, M.A., Humphreys, K.: SUPPLE: A practical parser for natural language engineering applications. In: Proc. of the 9th Intl. Workshop on Parsing Technologies (IWPT2005), Vancouver (2005)
18. Witte, R., Kappler, T., Baker, C.J.O.: Ontology Design for Biomedical Text Mining. In: Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences, pp. 281–313. Springer, Heidelberg (2006)
19. Mencl, V.: Deriving behavior specifications from textual use cases. In: Proceedings of Workshop on Intelligent Technologies for Software Engineering, Linz, Austria, Oesterreichische Computer Gesellschaft, pp. 331–341 (2004)
20. Ilieva, M., Ormandjieva, O.: Automatic transition of natural language software requirements specification into formal presentation. In: Montoyo, A., Muñoz, R., Métais, E. (eds.) NLDB 2005. LNCS, vol. 3513, pp. 392–397. Springer, Heidelberg (2005)
21. Kof, L.: Natural language processing: Mature enough for requirements documents analysis? In: Montoyo, A., Muñoz, R., Métais, E. (eds.) NLDB 2005. LNCS, vol. 3513, pp. 91–102. Springer, Heidelberg (2005)
22. Marcus, A., Maletic, J.I.: Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In: Proc. of 25th Intl. Conf. on Software Engineering (2002)