Enhanced Semantic Access of Software Artefacts

Danica Damljanovic and Kalina Bontcheva

Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello Street
S1 4DP, Sheffield, UK
{D.Damljanovic,K.Bontcheva}@dcs.shef.ac.uk

Abstract. Large software frameworks and applications tend to have a significant learning curve both for new developers working on system extensions and for other software engineers who wish to integrate relevant parts into their own applications. Recent research has begun to demonstrate that Semantic technologies are a promising way to address some of these issues. In this paper, we present a semantic-based prototype that is made for an open-source software engineering project with the goal to explore the methods for assisting open-source developers and software users to learn and maintain the system without major effort. The paper describes how we learned the domain ontology from the software artefacts, which we then used in the process of semantic annotation to enrich all documentation, source code, user forum postings , and tutorial materials. Generated annotations are being transformed into OWL and natural language-based semantic queries are offered. Transparently to users, these are being translated to SeRQL in order to provide enhanced semantic-based access.

Key words: semantic annotation, ontology learning, semantic access, software artefacts

1 Introduction

Successful code reuse and bug avoidance in software engineering requires numerous qualities, both of the library code and of the development staff; two important qualities are ease of identification of relevant components and ease of understanding of their parameters and usage profiles. The attraction of using semantic technology to address this problem lies in its potential to transform existing software documentation into a conceptually organised and semantically interlinked knowledge space that incorporates unstructured data from multiple software artefacts: forum postings, manuals, structured data from source code and configuration files. The enriched information can then be used to add novel functionality to web-based documentation of the software concerned, providing the developer with new and powerful ways to locate and integrate components (either for reuse or for integration with new development).

In this context, one of the key requirements towards the semantic web technology is that it should be applied to existing software projects and require little human involvement in its creation and maintenance. Due to the fact that the majority of projects do not maintain an ontology already, an essential part of our methodological and technological approach is in developing automatic methods for ontology learning and content augmentation, specifically tailored to software artefacts.

Overall, there are three kinds of tasks to be completed in order to enable semantically an existing (legacy) software application:

- semi-automatic elicitation of an ontology to represent the application-specific concepts;
- alongside this, exposure of semantic-based access to software artefacts;
- finally, uniting the results of these two processes, to provide semantic-based access through an intuitive natural language-based interface.

This paper discusses each of these steps in terms and details their implementation in a prototype system, developed on the basis of GATE[1] – a very widely used open-source software project with hundreds and users and over twenty active developers, distributed over the internet. The goal of this prototype is to explore the methods for assisting distributed, dynamic groups of software developers and users to learn and maintain this system without major effort, through the application of semantic web technologies. As the core of any semantic-enabled system is in ontologies, we first acquired the domain ontology semi-automatically from the GATE source code, documentation, manuals and other software artefacts. Once a domain ontology is in place, semantic content augmentation methods are used to annotate automatically all software artefacts, with respect to the ontology. The results are stored in a knowledge store to enable users to carry out semantic searches and find easily all information relevant to a given GATE concept.

The paper is structured as follows. In Section 2 we discuss scenarios for the GATE case study and requirements derived from them. As the basic requirement for a semantic-based system is to build the domain ontology, in Section 4 we first describe how we learned the ontology from the available software artefacts, followed by a detailed description of the prototype for enhanced access of software artefacts. In Section 5 we describe preliminary evaluation of the prototype components, followed by a discussion on our next steps for evaluating the prototype as a whole. In Section 6 we position our work in the context of other semantic-enabled systems for software engineering. Section 7 draws conclusions and outlines directions for future work.

2 The Case Study

GATE [1] is an open-source architecture and infrastructure for the building and deployment of Human Language Technology applications, used by thousands of

[1] http://gate.ac.uk

users at hundreds of sites. The development team consists at present of over 15 people, but over the years more than 30 people have been involved in the project. As such, this software product exhibits all the specific problems that large long-running open-source projects encounter.

While GATE has increasingly facilitated the development of knowledge-based applications with semantic features (e.g. [2–4]), its own implementation has continued to be based on compositions of functionalities justified on the syntactic level, understood by informal human-readable documentation. By its very nature as a successful and accepted 'general architecture', a systematic understanding of its concepts and their relation is shared between its human users. It is simply that this understanding has not been formalised into a description that can be reasoned about by machines or made easier to access by new users. Indeed, GATE users who want to learn about the system are finding it difficult due to the large amount of heterogeneous information, which cannot be accessed via a unified interface.

In the context of the TAO project[2], GATE is used as a case study where the main stress is on transitioning this application to ontologies in order to enable enhanced knowledge access to support distributed teams of software developers and users.

The first challenge to be addressed is learning domain ontologies from *multiple* sources associated with the software project, i.e., software code, user guide, and discussion forums. What is needed is a technique which does not simply deal with these sources independently, but can also exploit their redundancy and diverse formats in order to improve the resulting ontology.

Another important aspect that needs to be dealt with is *dynamics of software artefacts*, i.e., software artefacts grow and are updated continuously (e.g., forum postings, APIs between software versions). Therefore, one needs to implement a semantic annotation process which indexes them regularly with respect to the latest domain ontology and updates the knowledge base/semantic annotation repository as artefacts evolve over time.

In the context of semantic-based access, several scenarios have been identified as relevant:

Automatic generation of reference pages from the ontology – provides users with a single point of access to all knowledge, continuously kept up to date. The current GATE web site provides links to various kinds of documentation about the system: user/developer manuals, source code, javadoc, papers, tutorials, movies, etc. In addition, there is a link to the discussion forum, which is hosted on sourceforge.net. A new user who wishes to find out information about a GATE component, e.g., POS (part-of-speech) Tagger, needs to do these searches in two separate places, i.e., even for important GATE components there are no single comprehensive reference pages. In addition, the user does not have control over the results, i.e., see results ordered by recency (only possible in the forum search) or type of source (request results only from papers or the user guide). Another problem is that

[2] http://tao-project.eu

there are typically more than one terms referring to the same concept (POS Tagger, Hepple Tagger, part-of-speech tagger). Currently searching for these gives different results, whereas a concept-based search would return the same hits. Instead of being shown triples about the POSTagger instance in a formal way (e.g., as does Protege or the KIM KnowledgeBase Explorer[5]), the prototype generates automatically a web page, which can be shown on its own or alongside the ontology tree, where POSTagger is selected.

Natural language-based queries for semantic search – this is an intuitive way to formulate semantic queries without need for knowledge of the underlying query language of the semantic repository. When compared to the traditional keyword search, semantic search is capable of providing better results, across the multiple software artefacts. However, there is a problem with helping non-expert users to formulate their queries in an intuitive fashion, without any knowledge of formal languages (e.g. SeRQL). One promising solution explored here is to offer natural language queries, so that software developers and users can easily access the knowledge without the need to learn about semantic technologies and formal languages like SeRQL[3] or SPARQL[4].

Semantic-based filtering of forum postings – allow users to filter which postings they see, by defining a set of relevant concepts from the domain ontology. GATE's discussion forum has on average about 120 posts per month, with up to 15 on some days. Due to the volume of this information, it would be helpful if developers could choose to read only postings related to their areas of interest. Therefore, what is required is automatic classification of postings with respect to concepts in the ontology and a suitable interface for semantic-based search and browse.

Expertise location – enable a (new) team member to identify which are the most appropriate team members to consult on a given problem. Developer expertise will be discovered automatically from forum postings. As part of the process of designing the semantic-based search and browse facilities, we carried out a small experiment with using a set of domain concepts to discover who is the most suitable GATE developer to address for a given problem. A subset of the GATE forum postings were analysed to identify all responses by GATE developers, whose names were supplied as a list. The result was an association of domain concepts, developer names (as initials), and frequency of answers. Some examples are: POS tagger (DM (43 postings), IR (12)), Jena ontologies (VT (45), KB (6), IR (2)). The goal is to enable developers to identify easily who they should consult when working on a new topic. Conversely, as already discussed, the assignment of GATE concepts to forum postings will enable our system to provide developers with the facility to be notified only of postings related to their area of expertise.

In order to meet the requirements described in the context of these scenarios, we have developed a semantic-based prototype system for enhanced access

[3] http://www.openrdf.org/doc/sesame/users/ch06.html
[4] http://www.w3.org/TR/rdf-sparql-query

to software artefacts. The next section describes how we acquired the domain ontology, followed by details on the prototype architecture and evaluation.

3 Ontology Acquisition

The acquisition of the domain ontology is the enabling block that enables semantic-based access to software artrefacts. Whereas traditionally such an ontology would be defined by hand, in our case we first apply ontology learning tools to the GATE source code and javadocs to obtain a draft ontology (model bootstrapping), which is then post-edited by a domain expert (i.e. a GATE expert) to obtain the gold-standard.

The goal of OL methods is to derive automatically (parts of) an ontology from existing data (for state of the art overview see [6]). In earlier work it has been demonstrated that textual sources attached to software artifacts (e.g., software documentation) contain a wealth of domain knowledge that can be extracted automatically to build a domain ontology [4]. It was observed however that existing generic ontology learning approaches need to be adapted to handle the particularities of software specific textual sources (i.e., low grammatical quality, using a sublanguage). The approach presented in [4] explores the particularities of the sublanguage specific to software documentation to manually derive knowledge extraction rules. While the most generic rules can be used across domains they only extract a limited amount of knowledge. More specific rules need to be manually identified for new domains. Also, this technique has been applied on a single type of software specific textual sources, namely short functionality descriptions. This is a drawback because the work of Ankolekar et al. [7] showed that knowledge about a software project is often spread in several different information sources such as source code, discussion messages, bug descriptions, documentation and manuals.

Taking into consideration the lessons learnt from previous approaches, we experimented with acquiring a domain ontology from the *multiple* artefacts associated with any given software project, i.e., software code, user guide, and discussion forums [8]. Our technique does not simply deal with these different types of sources, but it goes one step further and exploits the redundancy of information to obtain better results. The strength of our technique is that it relies on learning methods that are portable across domains.

After further post-editing by a domain expert, the final version of the ontology contains 42 classes and 23 object properties. It is available online at `http://gate.ac.uk/ns/gate-ontology`. OWLIM[9] is used to reason over the created ontology and reveal any inconsistencies, as well as for semantic queries.

In addition to the generic ontology learning experiment, we also implemented a mapping between the application-specific entries in the software's XML configuration files to instances in the domain ontology. The resulting populated ontology, including all 594 automatically acquired instances, is available at `http://gate.ac.uk/ns/gate-kb`. It is this populated ontology which is used in the semantic annotation and language-based semantic search experiments discussed next.

4 The Semantic Annotation and Knowledge Access Prototype

Once the domain ontology is in place, the next step is to design a system so that it uses this ontology both for semantic annotation *of* the software artefacts, and for advanced semantic access *to* them.

However, in order to reach our goal which is to provide users with a single point of access to all domain knowledge and artefacts, and continuously keep it up to date, we first need to gather these software artefacts as they are often dispersed across different locations on the Web. As shown in Figure 1), after we have collected the data, the second step is semantic annotation with regards to the domain ontology, which is performed by the *CA Service*. The automatically produced annotations need to be exported to the *Content Augmentation (CA) Index* and stored in the knowledge store. Finally, these annotations are made accessible via a tool which accepts natural language queries as an input (Document Finder in Figure 1). The following sections describe each of these steps in further detail.

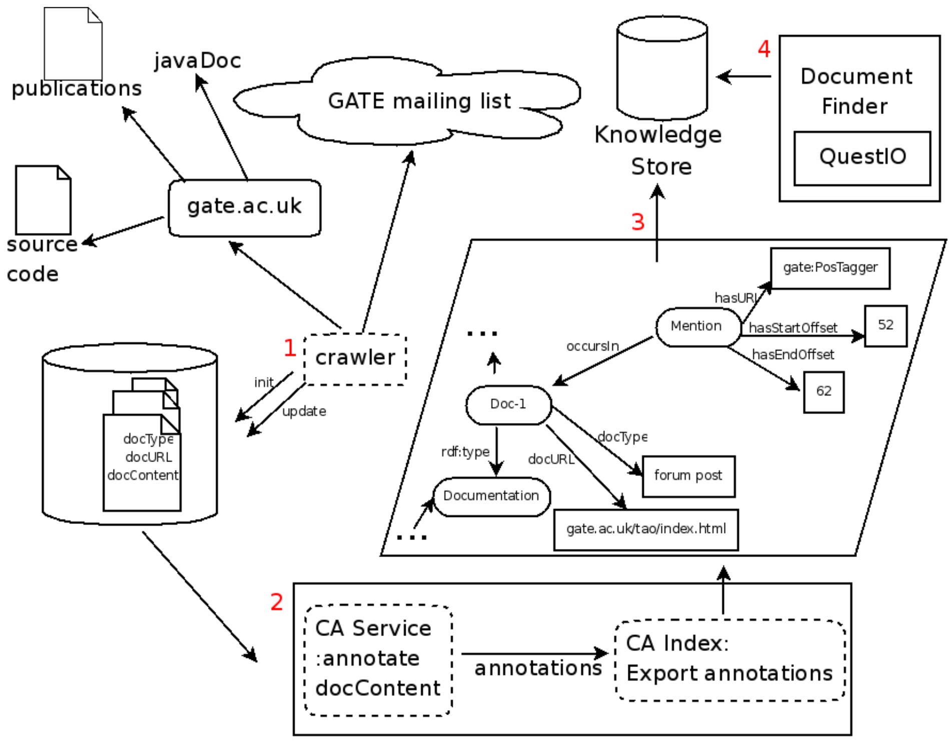


Fig. 1. System architecture

4.1 Data Collection

The first step is to gather all data about the software application which could be of interest to developers and users (especially novice ones). In our case study, we implemented a crawler that visits first gate.ac.uk and downloads the relevant content (manuals, papers, JavaDoc, source code). This content includes all data related to the GATE software linked from the GATE home page, including external links if there are any (for example, PDF files of publications about the system). Additionally, the crawler is visiting the GATE mailing list which is hosted on sourceforge.net, and downloading all new posts, which have not been indexed already in previous iterations. This process is run on a daily basis to capture and index new software artefacts, as soon as they become available. At the time of writing we have collected around 2GB of content, the majority of which is text-based.

When storing documents, we not only store their content but also associated metadata: the URL from which the document was downloaded and the type of the document (e.g. publication, forum post, manual, source code). We use several simple heuristic rules in order to predict what is the type of the document based on the URL. For example, if the URL contains javadoc, it is easy to conclude that the document is a JavaDoc file.

Once all software artefacts are downloaded and stored, they need to be enriched with semantic information. For annotation purposes we use a Content Augmentation (CA) web service (see Figure 1) which wraps our Key Concept Identification Tool (KCIT). KCIT is an Information Extraction application, based on several general-purpose GATE [10] components plus an ontology-based gazetteer which is capable of producing semantic annotations given an ontology. Next we discuss this process in more detail.

4.2 Automatic Content Augmentation

Given an ontology and a document, KCIT attaches ontology-aware annotations to the document, i.e., annotations refering to classes, instances and properties in the ontology. During the KCIT initialisation phase, all ontology resources are processed automatically in order to extract any human-understandable lexicalisations which could be used at runtime. To achieve this we first extract a list of the following:

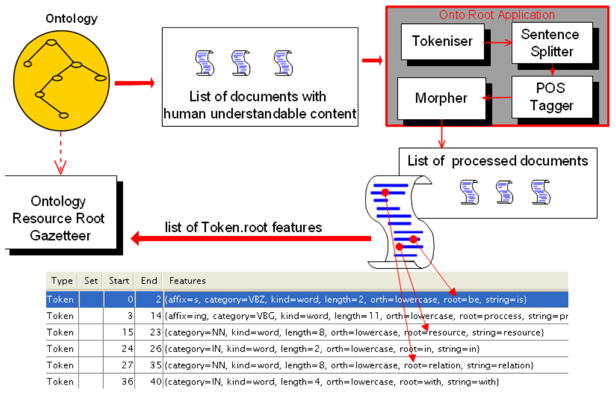
- names of all ontology resources i.e. fragment identifiers [5] and
- assigned property values for all ontology resources (e.g., label and datatype property values)

Each item from the list is processed further so that:

[5] An ontology resource is usually identified by an URI concatenated with a set of characters starting with '#'. This set of characters is called a *fragment identifier*. For example, if the URI of a class representing *GATE POS Tagger* is: 'http://gate.ac.uk/ns/gate-ontology#POSTagger', the fragment identifier will be 'POSTagger'.

– any name containing dash (`"-"`) or underscore (`"_"`) characters is processed so that each of these characters is replaced by a blank space. For example, `Project_Name` or `Project-Name` would become a `Project Name`.
– any name that is written in *camelCase* style is actually split into its constituent words, so that `ProjectName` becomes a `Project Name`.

Next, each item from this list is analysed separately by our Onto Root Application (see Figure 2), which carries out shallow linguistic analysis of the ontology resources and derives their base (root) forms, in order to enable successful matching of the different syntactic forms. In a nutshell, the Onto Root Application is a pipeline of several linguistic analysis modules. It first tokenises each list item (i.e., splits the text into its constituent words) and then assigns part-of-speech and lemma (i.e. root) information to each token. It is this lemma or a set of lemmas which are then added to a dynamic gazetteer list (the Ontology Resource Root Gazetteer).


Ontology
List of documents with human understandable content
Onto Root Application
Tokeniser
Sentence Splitter
POS Tagger
Morpher
List of processed documents
Ontology Resource Root Gazetteer
list of Token.root features

Type	Set	Start	End	Features
Token		0	2	{affix=s, category=VBZ, kind=word, length=2, orth=lowercase, root=be, string=is}
Token		3	14	{affix=ing, category=VBG, kind=word, length=11, orth=lowercase, root=proccess, string=pr
Token		15	23	{category=NN, kind=word, length=8, orth=lowercase, root=resource, string=resource}
Token		24	26	{category=IN, kind=word, length=2, orth=lowercase, root=in, string=in}
Token		27	35	{category=NN, kind=word, length=8, orth=lowercase, root=relation, string=relation}
Token		36	40	{category=IN, kind=word, length=4, orth=lowercase, root=with, string=with}

Fig. 2. Building a Dynamic Gazetteer from the Ontology

For instance, if there is a resource with a fragment identifier *ProjectName*, with assigned property *rdfs:label* with value *project names*, the created list before executing the Onto Root application will contain the following strings:

– *ProjectName* as a fragment identifier,
– *Project Name* as split fragment identifier,
– *project names* as the value of *rdfs:label*.

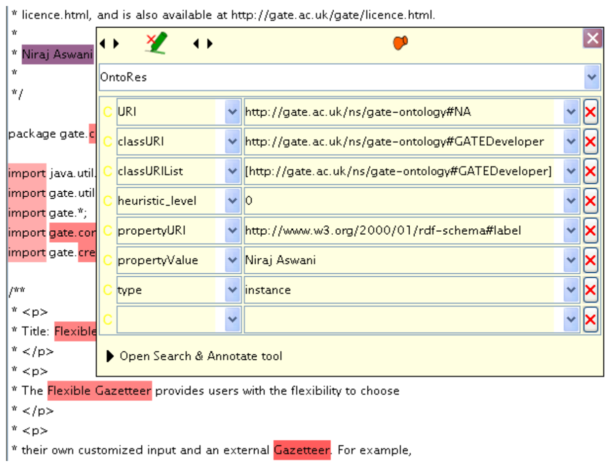
Each of the items from the list is then analysed separately and the results would be:

– For *ProjectName* and *Project Name* the output will be the same as the input, as the lemmas are the same as the input tokens.

– For *project names* the output will be the set of lemmas from the input, resulting in *project name*.

The overall performance of KCIT is directly proportional to the quality of the formal descriptions residing inside the ontology: the more human understandable descriptions there are in the ontology – the better the quality of the annotations.

The output of the semantic annotation process is a set of annotations in XML format which contains different features about the term from the text that is being annotated. For example, the *URI* of the ontology resource to which the term refers to, the *type* of that ontology resource (e.g., an instance, a class or a property), and other features that could be used later in the process of annotation extraction. An example of semantically annotated document (a Java class from the GATE source code) is shown in Figure 3.



```
* licence.html, and is also available at http://gate.ac.uk/gate/licence.html.
*
* Niraj Aswani
*
*/

package gate.c

import java.util.
import gate.util
import gate.*;
import gate.cor
import gate.cre

/**
* <p>
* Title: Flexible
* </p>
* <p>
* The Flexible Gazetteer provides users with the flexibility to choose
* </p>
* <p>
* their own customized input and an external Gazetteer. For example,
```

OntoRes

URI	http://gate.ac.uk/ns/gate-ontology#NA
classURI	http://gate.ac.uk/ns/gate-ontology#GATEDeveloper
classURIList	[http://gate.ac.uk/ns/gate-ontology#GATEDeveloper]
heuristic_level	0
propertyURI	http://www.w3.org/2000/01/rdf-schema#label
propertyValue	Niraj Aswani
type	instance

Fig. 3. An example of annotating a Java class with KCIT

The pop-up table in Figure 3 shows an annotation created by KCIT and all its features. In particular, the annotated term ('Niraj Aswani' in this case) shows the features of the annotation such as the *URI* of the corresponding ontology resource and the *class URI* from which we can see that this name refers to a GATE developer. Additionally, from the list of annotation features in Figure 3 we can see that Niraj Aswani is a value of the property *rdfs:label* for an instance that is of type *GATE developer*. *Heuristic level* feature which is 0 in this case indicates that no heuristic rules were used during the semantic annotation process.

Once the semantic content augmentation stage is completed, these annotations needs to be saved with the documents, or stored separately in a way that makes them accessible through semantic search. Namely, there are two types of annotations that could be generated [12]:

– *explicit* annotations: add new tags to the content so that these tags refer to resources in the ontology
– *implicit* annotations: the software artefact remains unchanged, but instead the positional and semantic information are stored in an externally specified knowledge base.

In our case, we store both explicit and implicit annotations, in order to make the output suitable for processing different tools. However, for the following document retrieval step, we use only the latter annotations, and therefore we only focus on them in this paper.

4.3 Storing Implicit Annotations

The annotation extraction phase (performed via the *CA Index* shown in Figure 1) comprises of reading the produced annotations and their features, in order to generate the OWL output which merges document-level metadata and the semantic annotations in a format which is then easily queried via a formal language such as SPARQL. More specifically, this extracted information needs to 'connect' a *document* with different *mentions* of the ontology resources inside that document. For example, if the document contains mentions of the class *Sentence Splitter*, the output should be modeled in a way that preserves this information during query time (i.e. the URLs of all documents mentioning this class should be found easily). For this purpose, we use the PROTON KM ontology[6], and more specifically, the *Mention* and *InformationResource* classes and the *occursIn* property, which links the two classes. The *Mention* class also contains properties which enable storing the position of the semantic annotation within the document content, namely *hasStartOffset* and *hasEndOffset*.

An example of an extracted OWL output, where the *Document* class is instantiated, is as follows:

```
<rdf:Description rdf:about=
 "gate:id_ee7ba66b-cd71-4993-9635-777b24f46372">
<rdf:type rdf:resource=
  "http://proton.semanticweb.org/2005/04/protont#Document"/>
<rdfs:label>
  Flexible Gazetteer
</rdfs:label>
<protont:informationResourceIdentifier>
  http://gate.ac.uk/gate/doc/java2html/gate/creole/gazetteer/
  FlexibleGazetteer.java.html
</protont:informationResourceIdentifier>
<protonkm:resourceType>
  Java Class
</protonkm:resourceType>
</rdf:Description>
```

[6] http://proton.semanticweb.org/2005/04/protonkm

An example of instance of *Mention* and the relation between the forementioned instance of *Document* is encoded as follows:

```
<rdf:Description rdf:about=
"gate:mention_0c45b1dc-efab-48a2-8242-bb78c1ddd3b5">
<rdf:type rdf:resource=
"http://proton.semanticweb.org/2005/04/protonkm#Mention"/>
<rdfs:label>
  Niraj
</rdfs:label>
<protonkm:occursIn rdf:resource=
"gate:id_ee7ba66b-cd71-4993-9635-777b24f46372"/>
<protonkm:hasStartOffset>
  404
</protonkm:hasStartOffset>
<protonkm:hasEndOffset>
  409
</protonkm:hasEndOffset>
<protonkm:hasString>
  http://gate.ac.uk/ns/gate-ontology#NA
</protonkm:hasString>
</rdf:Description>
```

Note that *gate:* is used in the above examples instead of the full namespace of the ontology which is `http://gate.ac.uk/ns/gate-ontology#` simply for the sake of brevity. The long names for the new instances of both *Document* and *Mention* classes are created automatically.

The extracted annotations are then stored in an OWL-compatible knowledge repository (OWLIM [9] in our case) and can be accessed using SeRQL or SPARQL. However, in order to query this semantic information successfully, one must be familiar with at least one of the two formal query languages. Such languages – while having a strong expressive power – require detailed knowledge of their formal syntax and understanding of ontologies and the way in which they are encoded in OWL. One of the ways to lower the learning overhead and make semantic-based queries more straightforward and easily accessible to software developers and users, is through a Natural Language Interface (NLI).

4.4 Semantic-based Access through Text-based Queries

In order to enable advanced semantic-based access through natural language, we developed a Question-based Interface to Ontologies (QuestIO). QuestIO [13, 14] is a domain-independent system which translates natural language queries into the relevant SeRQL queries, executes them and presents the results to the user. QuestIO works so that it first recognises key concepts inside the query, detects any potential relations between them, and creates the required semantic query. For example, if the query consisted of two concepts (e.g. 'plugins in GATE') the matching triples from the ontology will be extracted and shown (in this case – a list of all instances of GATE plugins).

In order to access the data stored in the implicit annotations and retrieve all relevant information (i.e. the URLs of Documents with particular Mentions) we had to make QuestIO more intuitive, by customising it so that the user can omit some obvious concepts when posting the query. For example, if the user needs more information about the *countries in Europe* (i.e. doc URLs which mention these concepts), the query for QuestIO would need to be formed as *information resource about countries in europe* or *documents about countries in europe.* We customised QuestIO so that *document* or *information resource* is added to the query by default, so that users do not have to specify this explicitly each time.

Also, as the output of QuestIO is a set of triple-like rows, we have customised it to produce a list of results (list of document URLs), rather than a table with all relations between concepts. At the moment, our prototype is showing a list of all relevant documents, without any ranking. Our next step is to investigate methods for result summarisation and clustering.

5 Evaluation

In order to evaluate the prototype described in this paper, we started with evaluating its components, prior to user experiments with the complete prototype. As QuestIO and its performance directly influences the performance of the prototype, we have evaluated it using the GATE knowledge base and a set of questions collected from the GATE mailing list. Namely, 36 questions were collected from the GATE user mailing list where users are enquiring about various GATE modules and plugins. These questions were run against the GATE ontology. This ontology (as described in Section 3) encodes the component model of GATE, the available plug-ins, the types of components included in each plug-in, the parameters for the different modules, etc. In the terms of precision and recall, the score for QuestIO was 82.14% and 71.87% respectively [7]. We also evaluated QuestIO's performance against other similar systems and further details are available in [13, 14].

For the evaluation of the prototype as a whole, in the forthcoming months we will carry out a user-centric evaluation and report the results at the workshop. The evaluation will be along the following dimensions:

– ease of finding information on a given topic, with and without semantic-based access
– benefits and usability of our language-based knowledge access approach
– scalability of the knowledge stores and ability to store all software artefacts

6 Related work

6.1 Semantic Technologies for Software Engineering

Perhaps the closest work to our approach is the Dhruv system [7], which relies on a semantic model obtained by instantiating hand-built domain ontologies with

[7] The definition of precision and recall we use here is that adapted from information retrieval (see [15, 16]).

data from different information sources associated with an open source project to support bug resolution.

The main differences of our work from that of Dhruv [7] are:

- The application-specific ontology is not represented explicitly in Dhruv, whereas learning and using such ontologies is our focus. The ontologies in Dhruv are general purpose software ontologies;
- Dhruv only populates the ABox (i.e., instances) and uses a fixed TBox (4 generic software engineering ontologies), whereas we address both ;
- The focus of Dhruv is different in that it aims at helping the developers at a very low level (e.g., method names), aka JavaDoc on steroids. Our focus is at higher, service/component level, although visualisations similar to those in Dhruv can be used.

Another similar system is the Ontology-driven Software Engineering Environment (OSEE) [17]. The goal of OSEE is to support semantic-based software development. In contrast to the approach used in OSEE, we do not alter the software development practices at all, but rather layer some semantic technology on top, to enable new usage of already existing software artefacts.

The Hipikat [18] Eclipse plugin for supporting software development is relevant to our forum postings and expertise location scenarios. The difference again is that Hipikat is integrated with the software development environment, whereas our scenarios address the problem of helping any user/reader with finding relevant information from the forum postings, which are not tightly controlled artefacts.

6.2 Semantic-based search and browsing

There is a lot of research on semantic-based search and browsing in areas other than software engineering. We discuss here only those that are in a way similar to our approach, even if they have not been trialed in the domain of sotware engineering. More specifically, we mention systems which, similar to us, perform the process of semantic annotation automatically in order to enable semantic-based information retrieval or browsing in a user-friendly manner.

Magpie [19] is a suite of tools which supports the interpretation of webpages and "collaborative sense-making". Similar to KCIT, it annotates webpages with metadata and needs no manual intervention by matching the text against instances in the ontology. In effect, Magpie adapts the web browsing experience for different users depending on their goals, knowledge, and context, which is one approach to deriving user profiles and customised interfaces for our prototype.

KIM [5] is an extendable platform for knowledge management, which includes semantic-based search. It also includes a set of front-ends for online use, that offer semantically enhanced browsing. The KIMExplorer allows web-based browsing of the ontology, which can also be quieried by constructing SERQL queries via web forms.

Similar to KIM, the prototype presented in [12], automatically annotates Web pages, which results in generating both explicit and implicit semantic annotations with regards to the domain ontology. Implicit annotations are stored in a separate file with RDF data, where in addition to the specific features of an annotation, they also store the name of the document - Web page where this annotation belongs to, and also the instance position: the offset of the instance in the cached web page. Querying annotated semantic Web pages is performed by querying implicit annotations from RDF files, by generating appropriate SPARQL queries. Their system also supports querying these annotations by using free-form text which is transformed to the relevant SPARQL queries.

Considering the application of their approach to different domains, the prototype described in [12] is good for data-rich web pages in a narrow domain. However, although there is no restriction in using their prototype for different domains, as the domain of pages broadens their approach becomes less accurate, because of the instance recognition semantics overlap more and become harder to segment [12, p.13]. Also, the importance of the domain ontology is essential here, and there is no information of how much effort is needed for creating an information-extraction ontology for the specific domain. Considering resiliency, it is reported in [12] that while the system is "resilent to web page layout changes", there is a trade-off and that is execution speed and accuracy.

7 Conclusion and Future Work

This paper described a prototype for enhanced semantic access to software artefacts using the GATE open-source project as an example. The core of our system is in the domain ontology which was learnt from the various software artefacts. Once the ontology is acquired, it is used for semantic annotation which is performed automatically. The generated annotations together with document metadata are stored in a repository in OWL format and are made accessible by natural language-based queries. The preliminary evaluation results are encouraging.

The next steps include the improvement of the current interface and the employment of result clustering and summarisation. Scalability evaluation also needs to be performed, after which user-centric evaluation will follow.

Acknowledgements. This research was partially supported by the EU Sixth Framework Program project TAO (FP6-026460).

References

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

2. Bontcheva, K., Tablan, V., Maynard, D., Cunningham, H.: Evolving GATE to Meet New Challenges in Language Engineering. Natural Language Engineering **10**(3/4) (2004) 349—373
3. Kiryakov, A., Popov, B., Ognyanoff, D., Manov, D., Kirilov, A., Goranov, M.: Semantic annotation, indexing and retrieval. Journal of Web Semantics, ISWC 2003 Special Issue **1**(2) (2004) 671–680
4. Sabou, M.: Building Web Service Ontologies. PhD thesis, Vrije Universiteit (2006)
5. Popov, B., Kiryakov, A., Ognyanoff, D., Manov, D., Kirilov, A.: KIM – A semantic platform for information extraction and retrieval. Natural Language Engineering **10** (2004) 375–392
6. Buitelaar, P., Cimiano, P., Magnini, B.: Ontology learning from text: Methods, applications and evaluation. IOS Press (2005)
7. Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R.: Supporting Online Problem Solving Communities with the Semantic Web. In: Proc. of WWW. (2006)
8. Bontcheva, K., Sabou, M.: Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources. In: Workshop on Semantic Web Enabled Software Engineering (SWESE), Athens, G.A., USA (November 2006)
9. Kiryakov, A.: OWLIM: balancing between scalable repository and light-weight reasoner. In: Proc. of WWW2006, Edinburgh, Scotland (2006)
10. Cunningham, H.: GATE, a General Architecture for Text Engineering. Computers and the Humanities **36** (2002) 223–254
11. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: Proceedings of the 40^{th} Anniversary Meeting of the Association for Computational Linguistics (ACL'02), Philadelphia, USA (Jul 2002) `http://gate.ac.uk/sale/acl02/acl-main.pdf`.
12. Ding, Y., Embley, D., Liddle, S.: Automatic creation and simplified querying of semantic web content: An approach based on information-extraction ontologies. In: Proceedings of the 1st Asian Semantic Web Conference, Springer Berlin / Heidelberg (September 2006) 400–414
13. Damljanovic, D., Tablan, V., Bontcheva, K.: A text-based query interface to owl ontologies. In: 6th Language Resources and Evaluation Conference (LREC), Marrakech, Morocco, ELRA (May 2008)
14. Tablan, V., Damljanovic, D., Bontcheva, K.: A natural language query interface to structured information. In: Proceedings of the 5h European Semantic Web Conference (ESWC 2008), Tenerife, Spain (June 2008)
15. Popescu, A.M., Etzioni, O., Kautz, H.: Towards a theory of natural language interfaces to databases. In: IUI '03: Proceedings of the 8th international conference on Intelligent user interfaces, New York, NY, USA, ACM (2003) 149—157
16. Cimiano, P., Haase, P., Heizmann, J.: Porting natural language interfaces between domains: an experimental user study with the orakel system. In: IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces, New York, NY, USA, ACM (2007) 180–189
17. Thaddeus, S., Raja, S.K.: A Semantic Web Tool for Knowledge-based Software Engineering. In: Workshop on Semantic Web Enabled Software Engineering (SWESE), Athens, G.A., USA (2006)
18. Cubranic, D., Murphy, G., Booth, K.: Hipikat: A Developer's Recommender. In: OOPSLA. (2002)
19. Domingue, J., Dzbor, M., Motta, E.: Magpie: Supporting Browsing and Navigation on the Semantic Web. In Nunes, N., Rich, C., eds.: Proceedings ACM Conference on Intelligent User Interfaces (IUI). (2004) 191–197