Revista Romana de Interactiune Om-Calculator 9 (2) 2016, 103- 118 © MatrixRom

Semantic Analysis of Source Code in Object Oriented Programming. A Case Study for C#

Claudiu Epure, Adrian Iftene

Universitatea Alexandru Ioan Cuza din Iasi
General Berthelot 16, Iasi
E-mail: {claudiu.epure, adiftene}@info.uaic.ro

Abstract. This paper describes the CSCRO ontology and the Sharp RDF system, used together to semantically analyze the C# source code. The CSCRO ontology formally describes the domain of C# programming language, in which the concepts are represented as classes, properties and individuals. The purpose of the project is to provide the ability to retrieve information about the source code in form of metadata. The first step to achieve this is to incrementally build a graph-oriented knowledge-base from plain source code, based on the ontology. This is done using a convenient format like the *Resource Description Framework* (RDF). Having the knowledge base in place, it is easy to query the system (e.g. SPARQL) about its interacting components and services, retrieving data in a machine readable format. Going further, an answering mechanism could be applied for enabling natural language questions on the knowledge base.

Keywords: C#, static analysis, syntax tree, ontology, RDF, OWL, graph-database, triple store, linked data, natural language, SPARQL.

1. Introduction and Related Work

Computer programs have become the most frequently used tools in our modern society. Nowadays, they are present at large scale in industry in the form of applications, platforms and services, covering multiple areas such as science and education, finance, commerce, etc.
Developing a software system is not an easy action. Instead, it is a complex process comprised of several phases, which are completed during a significant period of time. However, factors like customers high demand and market competition lead to acceleration of the process with negative impact on quality.

As complex software systems are built at a fast pace, they need to remain maintainable through time. For this reason, software quality must be at its highest level, yet in most cases, it decreases as the systems are getting bigger.

Testing the code is the way for assuring the required functionality from the perspective of the users. From the programmers point of view, the code needs to be clean and easy to extend or reuse. Design patterns, coding standards, static code analysis are software engineering methodologies serving such a purpose (Esposito, 2011). But still, there are old systems, hard to refactor and production source code that is not implementing any engineering technique, which is very easy to break at any small try to redesign.

Another key aspect of software development is the use of version control systems in order to keep track of changes and make possible for teams to collaborate. They also provide a general view on the projects and backup service as well. Although they help to keep track of the physical file changes over time, they do not provide a way of tracking the logical structure inside a project.

Some of the existing approaches that are based on similar ideas are mentioned below. They address singular or specific problems, so for the proposed system, the intent is to adapt, extend and combine some of the ideas, in order to achieve the goal.

None of the above mentioned techniques address the problem of retrieving meta information from the code, in a semantic manner. Large software projects involving thousands of source code files would be easier to understand, control and extend if they would be complemented by a solid information retrieval system.

1.1 Existing Systems for Extracting Structured Data from Source Files

Fuzzy Ontology Framework (FOF, 2016). The Fuzzy Ontology Framework is a library that helps to integrate a fuzzy ontology (Fuller, 2008) (Calegari and Sanchez, 2014) with object-oriented programming (OOP) classes written in .NET. It is a hybrid integration, i.e. some OWL concepts can be mapped directly to OOP classes, yet most OWL concepts are derived just from OOP instance properties, with no direct mapping to a .NET class. Hence the OOP instance-OWL concept(s) mapping can evolve dynamically in the course of time.

SCRO (SCR, 2016). SCRO is an ontology created to support major software understanding tasks by explicitly representing the conceptual knowledge structure found in source code (Alnusair, 2010).

SCRO captures major concepts of object-oriented programs and helps understand the relations and dependencies among source code artifacts. Supported features include, encapsulation, inheritance (sub-classing and sub-typing), method overloading, method overriding, and method signature information. It is designed for Java.

Similar, (Smeureanu and Iancu, 2013) used Protégé built to identify source code plagiarism and (Alboaie et al., 2004 and Buraga et al., 2005) used XML and RDF to exchange information in a multi-agent system.

1.2 Existing Systems for Information Retrieval based on Questions

Treo (Treo, 2016). The main ideas behind this system are: entity recognition and pivot determination through entity search, query syntactic analysis: partial ordered dependency structure (PODS) determination, and spreading activation using semantic relatedness.

The algorithm first determines the key entities present in the natural language query. The entity search engine receives the key entities and resolves pivot entities (URIs) in the Linked Data Web. The query is then analyzed and parsed to obtain partial ordered dependency structure (PODS).

The spreading activation search takes the URIs of the pivots and the PODS structure, and thus, starting from the pivot node, the algorithm navigates through the neighbour nodes in the Linked Data Web computing the semantic relatedness between query terms and vocabulary terms in the node exploration process. The navigation process builds the answer to the query (Freitas, Currry, Oliveria, 2011; Freitas et al., 2011).

TBSL (TBSL, 2016). The ideas of this system are to use SPARQL Template from question and to map between NL expressions to the domain vocabulary.

The user’s input is a natural language question which is processed by a POS tagger. The result is the semantic representation of the natural language query, based on lexical entries that are created using a set of heuristics. In the next step, this is converted into a SPARQL query template which contains slots: missing elements of the query that have to be filled with URIs. The URIs are determined using sophisticated entity identification approaches, based on string similarity as well as on natural language patterns which are compiled from existing structured data in the Linked Data cloud and text documents.

2. C# Source Code Representation Ontology

For many years, the traditional way of storing data was by using relational databases. The entities in a table are restricted to follow this schema in order to provide consistency.

A different approach was the introduction of the document-oriented databases using a hierarchical model (e.g. XML, JSON files). This type of storage doesn’t use the concept of schema, but the drawback of a document-oriented database is that it uses a hierarchy of elements (nodes) in the form of a tree, so there are elements that have a bigger importance/priority against others (e.g., parent node vs. child node).

Although there are advantages and disadvantages of using each of the above database model, a new type of database is preferred when the absence of concepts like schema and element hierarchy is required: graph-oriented database.

2.1 RDF and Graph-Oriented Databases

The Resource Description Framework (RDF) is one important building block of the graph-oriented database (LinkedDataTools, 2009). A resource can have an infinite number of properties and there is no restriction that it should follow. The underlying mathematical model is a labelled directed multi-graph in which the nodes are the resources and the edges are the relations. As a result, all the nodes are equal in importance/priority.

For example, the following RDF graph expresses the relationships between a person identified “John_Doe” and some information about it: type (an object property) and age (a data property) (see Figure 1). By convention, the resource nodes are represented in ovals, the values are represented in rectangles and the properties are represented as arrows (Mostarda, 2010). The associated RDF code is:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ex="http://example/org#">
  <rdf:Description
rdf:about="http://example.org#John_Doe">
    <rdf:type rdf:resource="http://example.org#person"/>
    <ex:age>25</ex:age>
  </rdf:Description>
</rdf:RDF>
```

For querying, RDF has its own query language: SPARQL (SPARQL Protocol and RDF Query Language). Because RDF can be seen as a collection of relationships between resources, SPARQL queries are based on triple patterns, providing one or more patterns against such relationships, using variables in place of some resources. The result of the processed query is the set of resources for all triples that match these patterns.

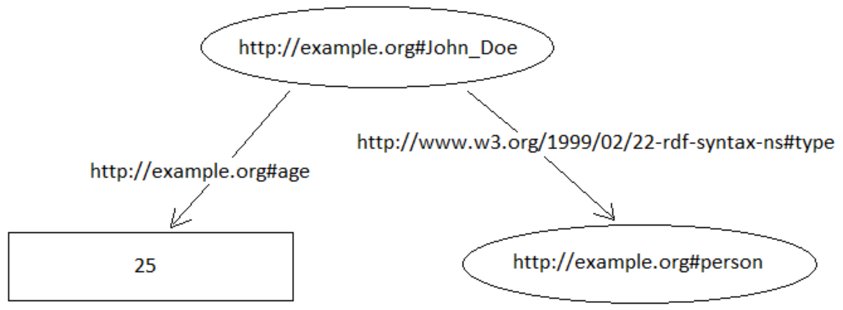
http://example.org#John_Doe
http://example.org#age
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
25
http://example.org#person

Figure 1: A visual representation of a RDF graph

For example, the query "*Select all the people being 25*" can be transformed in the following SPARQL query:

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex:  <http://example.org#>
select ?s where {?s rdf:type ex:person . ?s ex:age "25"}
```

2.2 RDF Schema and Web Ontology Language

There are two principal syntaxes for annotating RDF data with semantic metadata: RDF Schema (RDFS) and Web Ontology Language (OWL). Both of them are W3C specifications.

RDF Schema is a semantic extension of RDF, which provides a vocabulary for data modelling of information written in RDF (Brickley and Guha, 2014). Unlike other kind of type system, it is designed to be property centric. This means the focus is on properties, which exists on their own.

Web Ontology Language is an ontology language for the Semantic Web (Group, 2012, 2014). It provides the syntax for defining an ontology, which is a formal description of a domain of interest. The syntax is divided into three categories: entities, expressions and axioms. A summarized view of *OWL* is available in Table 1:

Table 1: General view of OWL

Entities			
Classes	Properties		Individuals
	Object Properties	Data Properties	Datatypes
Expressions			
Axioms			
Annotations			

2.3 Building the C# Source Code Representation Ontology (CSCRO)

For creating an ontology that covers C# programming language, the SCRO ontology, described in (SCRO, 2016), was used as a model. For this reason, the new ontology is named CSCRO (C# SCRO). As a result, CSCRO can be considered a variation of SCRO that works for C#.

At first, the intention was to use SCRO concepts to annotate the resources extracted from C# source code. But the ontology uses entities like package, *java.lang.String*, final method, Java based access modifiers, so it was clear that it was designed to work with Java programming language. Therefore, a new ontology for C# was needed.

The new C# ontology, CSCRO, keeps the main ideas from SCRO ontology, but adds new concepts specific to the programming language itself, based on MSDN references. There are some significant differences between the two ontologies both in structure (taxonomy) and individuals. These differences are discussed in parallel in the following lines.

Different Access Modifiers. In *CSCRO* ontology, *AccessModifier* is a subclass of *Modifier* (instead of *AccessControl*). Also, C# has a slightly different set of access modifiers. It uses *internal* and *protected internal* as *C#* specific access modifiers (see Figure 2).

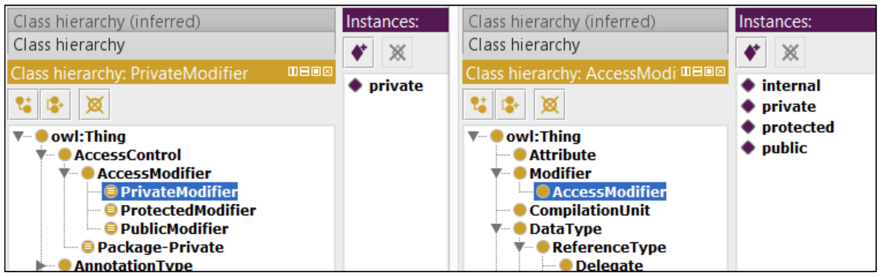

Figure 2: SCRO vs CSCRO (access modifiers)

Different Modifiers Taxonomy. In SCRO ontology, the modifiers are not represented by class of their own. Instead, they are spread under the taxonomy of data types.

For example, there is no static modifier itself, but there are StaticMemberClass and StaticMemberInterface classes. The same is valid for other modifiers too (e.g. final – FinalLocalClass, FinalMemberClass, abstract - AbstractLocalClass, AbstractMemberClass, etc.)

In CSCRO, there is the Modifier class under which there is an individual for each modifier (abstract, static, override, etc.). Besides, there are some C# specific modifiers: async, sealed, const, new, partial, etc. In the same manner, private, protected, public and internal are individuals of AccessModifier class.

The differences are marked with red in Figure 3.

Different Data Types Taxonomy. SCRO has a complicate taxonomy for data types. There is no single parent node in the hierarchy from which every type is derived. Instead, there is a parent class for each type (e.g. ClassType, EnumType, InterfaceType, etc.).

DataType class is used in another context than in the type hierarchy context (i.e., for specifying the "primitive" types: int, float, double, etc.). CSCRO rewrites completely the taxonomy of data types. Here, there is one parent class DataType from which every subclass defines a new hierarchy level of types.

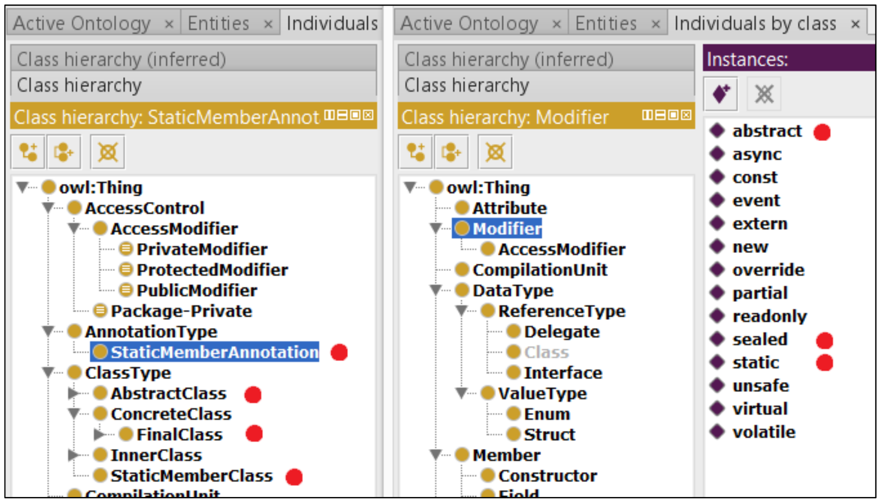

Figure 3: SCRO vs CSCRO (modifiers taxonomy)

Hence, on the second level there are ReferenceType and ValueType and on the third level there are Delegate, Class and Interface – as subclasses of ReferenceType and Enum and Struct – as subclasses of ValueType. Eventually, the "primitive" types are in fact individuals of class Struct. The differences are explained in Figure 4, by assigning numbers to similar concepts.

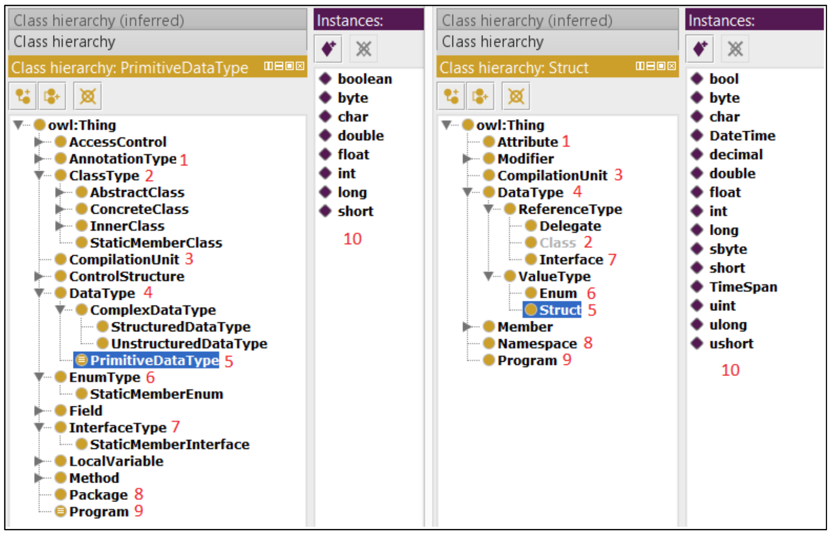

Figure 4: SCRO vs CSCRO (data types taxonomy)

Different Object Properties and Data Properties. There are also difference between the properties of the two ontologies, both object properties and data properties. In the case of object properties, the differences appear mainly because of the changes in the class taxonomy.

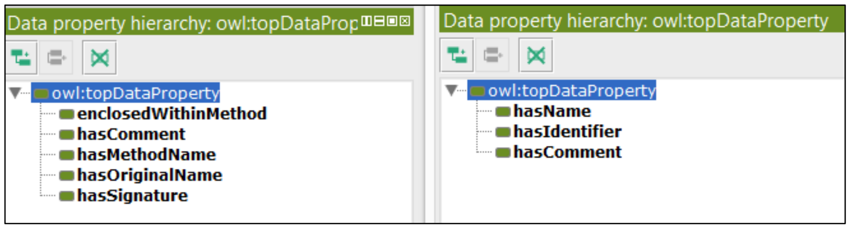

Figure 5: SCRO vs CSCRO (data properties)

Moreover, there are differences because of the terminology used in C# versus Java (e.g. package vs namespace: hasPackageMember vs hasNamespace; import vs using: imports vs uses, etc.).

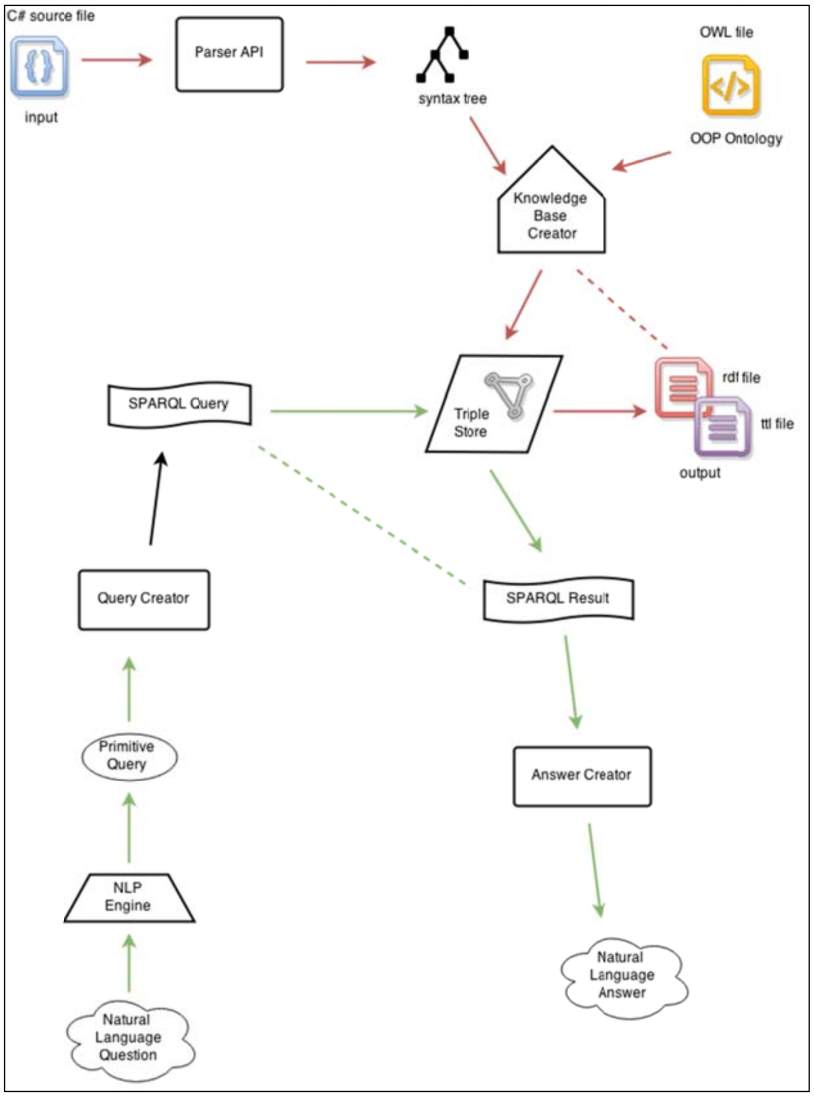

Figure 6: System Architecture

In the case of data properties (see Figure 5), CSCRO contains fewer properties that SCRO (e.g. hasName, hasIdentifier, hasComment). In CSCRO, the properties are only for basic functionality of the system. Complex properties can be added in case of complex requirements.

3. System Architecture

The proposed system (see Figure 6) is designed to follow two main ideas.

First, there is the process of extracting structured data from C# source files, based on an existing specific ontology (e.g. C# object-oriented ontology) and store that data in a way such that it can be easily retrieved later. In this case, the data will be saved in a triple store.

Second, there is the process of retrieving data from the store, having hierarchical levels of querying. While SPARQL queries are used for this purpose at the lowest level, at the highest level, the goal is the use of natural language questions. In the middle, annotated questions are used, based on NLP techniques.

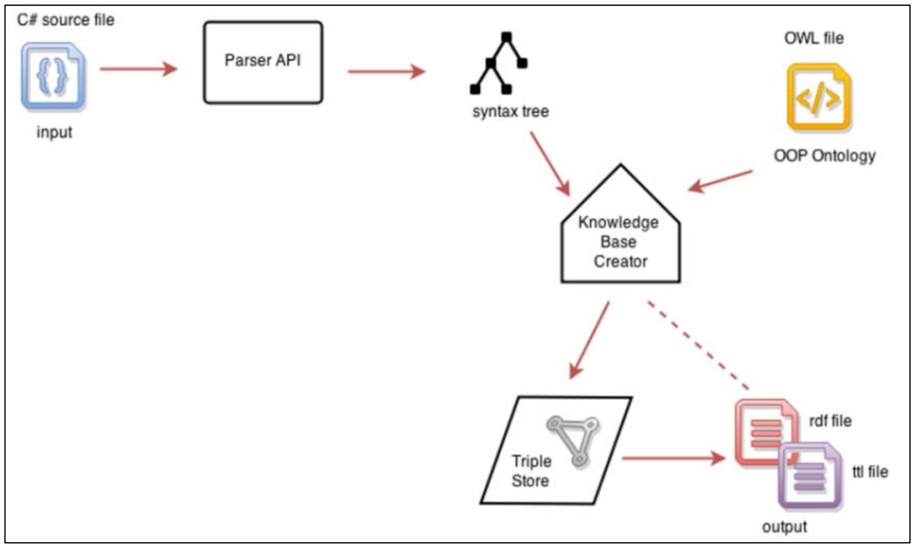

Figure 7: Knowledge base builder module

3.1 The Knowledge Base Builder Module

This module is designed to build a knowledge-base of the components found in a C# project or in C# source files. The result is represented as a collection of RDF graphs, known as a triple store. The graphs are built by following the CSCRO ontology defined in (2.3) (see Figure 7).

The Ontology class encapsulates the ontology built in OWL. Its main responsibility is to load the ontology file and create an instance of Graph class from it.

The ISyntaxTreeBuilder interface is responsible for creating the syntax tree of a source file content. The class that implements this interface is SyntaxTreeBuilder. Internally, it uses the .NET Compiler API to do this.

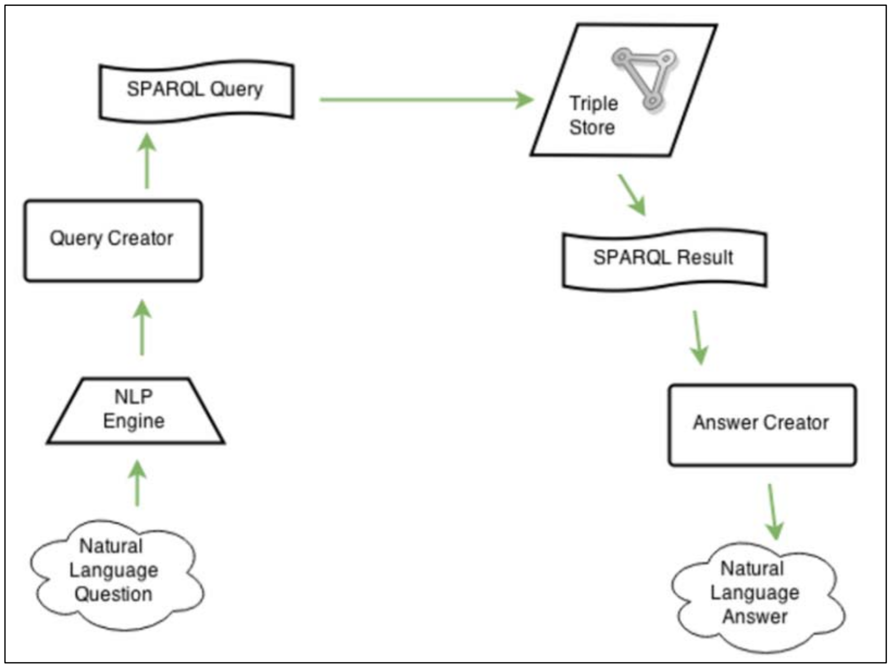

Figure 8: Information retrieval module

The IGraphBuilder interface is dealing with creating and populating a graph with data. The data is extracted from a syntax tree that is given as input. The class that implements this interface is KnowledgeBase.

The KnowledgeBase class has a dependency on Ontology. Its main purpose is to create graphs and populate them with data, according to the ontology. The graph data has the form of a syntax tree. A syntax tree is an abstraction that is used to model the internal structure of a source code file. The root of the tree is a compilation unit, the child nodes are namespaces, type declarations and so on.

The ontology uses a similar approach in the classification of the entities. Therefore, having the syntax tree and the ontology, the RDF graph is created by traversing the syntax tree and mapping the nodes to ontology entities, in order to create nodes in the new graph.

The Engine class is the entry point of this module from the external environment. It has dependencies on the KnowledgeBase and Ontology classes and it contains methods that create RDF graphs by receiving source code as input.

3.2 The Information Retrieval Module

In every information store/database/knowledge base there is the need to have a good mechanism of retrieving data. Query languages have evolved in order to fulfil this need (e.g. SQL, SPARQL). In recent years, because of the new technologies of mobile devices, the trend is to create and use query languages that are more human friendly, much closer to the human language. Natural Language Processing techniques are used to answer questions about things stored in a knowledge base.

The information retrieval module of the system presented in this paper is based on the source code knowledge base. The information is kept in a form of a triple store server that can be queried via a public endpoint.

The purpose of creating this module is to enable users to query the knowledge base in SPARQL and eventually to ask for information it in natural language. At this point, the natural language part is a work in progress, only SPARQL query being available.

Regarding natural language questions, these can vary in difficulty from simple to complex, depending on the number of compounding sentences (number of predicates). Simple questions (with only one predicate) are mapped directly to a RDF triple based on keywords, whilst complex ones needed a pre-processing step to divide them in sentences before mapping to triples. The module contains the following entities (see Figure 8):

- The NaturalLanguageProcessor class is a text annotator component. It receives plain text (representing the question) and returns the list of annotated compounding words. For each word the following properties are provided: text, part of speech, and lemma, begin offset in the sentence, end offset in the sentence. It uses Stanford Core NLP for text processing.
- The TripleBuilder is the component that, given an input list of keywords and URIs, it builds a list of triples of the form: TriplePattern {subject keyword, subject pattern (regex), predicate keyword, predicate URI, object pattern (regex)}.
- The QueryBuilder class is the component that, given the list of annotated words (from 1) and the list of triples (from 2) it builds the query in SPARQL as follows: For each triple, checks if there are matching subject key or predicate key in the list of words lemmas. If predicate key is

matching, the URI for that predicate provide us the subject type and the object type. If the subject keyword is not matching, then the object pattern will be found and the query is built based on that.

The future work will be focused on the Triple Builder module, trying to automate the process. The goal is to get lists of keywords/URIs from linked data sources over the Internet instead of using manual lists. Also, the system will try to answer not only simple questions (one sentence) but complex ones too (Ungher et al., 2014).

4. Project Details and Technologies

The building process of the project was divided in two parts: (1) Creating the C# Source Code Representation Ontology (CSCRO); (2) Building the system (Sharp RDF).

The ontology was built using Web Ontology Language (OWL) based on Protégé tool. The system is developed in Visual Studio using .NET technologies: C# programming language, .NET Compiler API (Roslyn), ASP.NET MVC framework. It is organized in a solution named *sharprdf*, that contains the following projects:

- Sharprdf.Core - this is the main project.
- Sharprdf.Cmd - this is a command line application designed to be used as a tool in an automation process like continuous integration. It exposes the functionality to create RDF graphs in a configurable manner, using command line arguments.
- Sharprdf.WebApp - this is an MVC web application that exposes the functionality online, as a service.
- Sharprdf.Nlp - this is the module that deals with text processing, used to transform natural language questions in SPARQL queries. It depends on Stanford Core NLP library and it is work in progress.
- Sharprdf.Nlp.Cmd - this is a command line application that is used to expose the functionality of Sharprdf.Nlp project. This is also work in progress.

The project uses Git as version control system so the log history is available on the Github at: https://github.com/ claudiuepure/sharprdf. In the future, a NuGet package will be also available.

5. Conclusions and Future Work

This paper presents a system that was created to offer a different perspective on the applications written in the C# programming language, in terms of structure and content. Regarding large projects, it is easier to retrieve information about them having a helper data structure in a form of a graph.

Before creating the system, a detailed analysis was made on the existing projects that follow similar ideas, learning from them. However, none of them combines the ideas introduced in this paper.

One important contribution to the project is the C# Source Code Representation Ontology (CSCRO). Although it is created based on the existing SCRO ontology (2.3), in the end the new ontology differs almost completely from the original one, in terms of taxonomy and properties.

Another contribution is the idea and the method of creating a RDF graph from C# source code based on a given ontology. There is a similar idea in Fuzzy Ontology Framework (FOF, 2016), but the final purpose of this work is different.

The new .NET Compiler API, which was released in the first quarter of this year is a fresh technology that helps programmers to develop new tools for source code analysis (Parson, 2015). The project was built using this state of the art API.

The idea of combining the Stanford Core NLP library with a model of patterns for recognizing question format based on the compounding words is another contribution to this project which will be extended in future work.

For the future, we intend to build a system with three main components: (1) Knowledge Base Builder module (that will support detecting advanced object oriented characteristics like inheritance, polymorphism, etc.); (2) a module for the detection of design patterns in source code, like in (Kirasic & Basch, 2008); and (3) an Information Retrieval module (to support questions in natural language). Also, we want to provide a web application or a public API for this functionality.

Acknowledgements

The research presented in this paper was funded by the project MUCKE (Multimedia and User Credibility Knowledge Extraction), number 2 CHIST-ERA/01.10.2012.

References

Alboaie, S., Buraga, S., Alboaie, L. (2004). An XML-based Serialization of Information Exchanged by Software Agents, *International Informatica Journal* 28(1), 13-22.

Alnusair, A. (2010). *SCRO (Source Code Representation Ontology)*. Retrieved from http://www.cs.uwm.edu/~alnusair/ontologies/scro.html

Brickley, D., Guha, R. (2014). *RDF Schema 1.1*. Retrieved from http://www.w3.org/TR/rdf-schema/

Buraga, S., Alboaie, S., Alboaie, S. (2005). An XML/RDF-based Proposal to Exchange Information within a Multi-Agent System, *Concurrent Information Processing and Computing*, (Eds. D. Grigoras, A. Nicolau), 336, IOS Press.

Calegari, S., Sanchez, E. (2014). *A Fuzzy Ontology-Approach to improve Semantic Information Retrieval*. Retrieved from http://ceur-ws.org/Vol-327/pos_paper3.pdf

Esposito, D. (2011). *Static Code Analysis and Code Contracts*. Retrieved from https://msdn.microsoft.com/en-us/magazine/hh335064.aspx

FOF, (2016). Fuzzy Ontology Framework - http://www.codeproject.com /Articles/348918/Fuzzy-Ontology-Framework (accesed last time in April, 2016)

Freitas, A., Currry, E., Oliveria, G. (2011). *A distributional structured semantic space for quering RDF graph data*. Retrieved from http://andrefreitas.org/papers/ preprint_distributional_structured_space.pdf

Freitas, A., Oliveira, G., O'Riain, S., Curry, E., & Pereira da Silva, J. C. (2011). *Querying Linked Data using Semantic Relatedness: A Vocabulary Independent Approach*. Retrieved from http://andrefreitas.org/papers/nldb2011_preprint.pdf

Fuller, R. (2008). *What is fuzzy logic and fuzzy ontology?* Retrieved from http://uni-obuda.hu/users/fuller.robert/otaniemi-2.pdf

Group, R. W. (2014). *Resource Description Framework*. Retrieved from http://www.w3.org/RDF/

Group, W. O. (2012). *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Retrieved from http://www.w3.org/TR/owl2-overview/

Kirasic, D., Basch, D. (2008). *Ontology-Based Design Pattern Recognition*. Retrieved from http://www.fer.unizg.hr/_download/repository/kes2008[1].pdf

LinkedDataTools. (2009). *Introducing Linked Data And The Semantic Web*. Retrieved from http://www.linkeddatatools.com/semantic-web-basics

Mostarda, M. (2010). *RDF Coder*. Retrieved from https://code.google.com/p/rdfcoder/

Parsons, J. (2015). *Getting Started: Semantic Analysis*. Retrieved from https://github.com/dotnet/roslyn/blob/master/docs/samples/csharp-semantic.pdf

Pidcock, W. (2009). What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model? Retrieved from http://infogrid.org/trac/wiki/Reference/PidcockArticle

SCRO, (2016). http://www.cs.uwm.edu/~alnusair/ontologies/scro.html (accesed last time in April, 2016)

Smeureanu, I., Iancu, B. (2013). *Source Code Plagiarism Detection Method Using Protégé*

Built Ontologies. Retrieved from http://revistaie.ase.ro/content/67/07%20-%20Smeureanu,%20Iancu.pdf

Sudarsun, S. (2007). *Introduction to Ontology*. Retrieved from http://www.slideshare.net/sudarsun/ontology

TBSL, (2016). http://svn.aksw.org/papers/2013/KESW_AutoSparql Tbsl_Demo/public.pdf (accesed last time in April, 2016)

Treo, (2016). http://treo.deri.ie/ (accesed last time in April, 2016)

Ungher, C., Forascu, C., Lopez, V., Ngonga Ngomo, A.-C., Cabrio, E., Cimiano, P., & Walter, S. (2014). *Question Answering over Linked Data (QALD-4)*. Retrieved from http://ceur-ws.org/Vol-1180/CLEF2014wn-QA-UngerEt2014.pdf