Condensation of Reverse Engineered UML Diagrams by Using the Semantic Web Technologies

Meisam Booshehri, Peter Luksch
Institute of Computer Science, University of Rostock, Germany

Abstract - *Up-to-date software design documentation is valuable for maintenance engineers, testers and developers joining a project at a later stage; however, UML Models, e.g. class diagrams, are often poorly kept up-to-date during development and maintenance. Reverse engineering, therefore, has become a popular method to recover an up-to-date design from the underlying source code. However current techniques yet produce a detailed representation of the underlying source code that would reduce the understandability. In order for the understandability to enhance, condensation of the reverse engineered diagrams has been proposed as a solution. However, current state-of-the-art approaches in this area still demand a need for improving or alternative approaches. Consistently, in this paper we are putting forward a bridging idea of using the semantic web technologies for improving the condensation process. Two contributions are proposed that support each other. Firstly, the V-Ontmodel is suggested for updating software documentation over the software evolution. Secondly, a general architecture is proposed enabling us to reduce sophisticated analysis tasks for the condensation process of class diagrams to a few queries in SPARQL or its extensions like SPARQL-ML. To discuss the feasibility of the approach we focus on the condensation of reverse engineered class diagrams in the whole paper. Finally, an illustration example is presented in which helper classes are excluded from the class diagrams in the condensation process regarding the structural patterns extracted from the software metadata.*

Keywords: *reverse engineering; simplification of UML diagrams; semantic web technologies; up-to-date design; condensation of class diagrams*

1 Introduction

Booch et al. proposed the Unified Modeling Language (UML) as a standard for writing the software blueprints. “UML may be used to visualize, specify, construct and document the artifacts of a software-intensive system”[1]. Currently UML 2.0 is an ISO standard where it provides 13 different diagrams so that one can express all the important features of a system. One of the UML diagrams is the Class Diagram by which developers could model classes in an object oriented design including their attributes, operations and their relationships and associations with other classes.

In practice as software evolves, design documentations will become more and more outdated. UML models are often poorly kept up-to-date during development and maintenance. On the other hand, an up-to-date design is of special importance for maintenance engineers, testers, developers, and other engineers joining a software project at a later stage. As a solution, consequently, reverse engineering methods have been proposed in order to recover up-to-date design diagrams automatically[2].

However, current reverse engineering techniques do not yet solve the problem sufficiently. Particularly, the Computer-aided Software Engineering (CASE) tools that offer functionality for reverse engineering into UML class diagrams produce a detailed representation of the underlying source code that would reduce the understandability[3]. Therefore, software engineers would come across difficulties in recognizing the key elements in a software structure.

In order to the increase understandability, Osman et al. [3-5] propose the condensation of reverse engineered class diagrams where the elemental question is how to select the key elements in the software architecture to recover class diagrams of suitable abstraction level? Subsequently, Thung et al. [6] propose an extension to Osman et al.'s work which brings an improvement to the previous approach. However, according to Thung et al., there are threats to the validity of their approach, specifically for the generalization of their approach to other projects. Moreover, the quality of reverse engineered diagrams has yet to be improved in order to become more and more similar to their forward design counterparts in terms of understandability.

Consistently, it is our goal in this paper to explore ways in order to facilitate current methods of condensation of reverse engineered diagrams. Since software engineering is naturally a knowledge-intensive activity, we are motivated to put forward a general bridging idea of using the semantic web technologies for improving and facilitating the process of condensation of reverse engineered UML diagrams. We make use of the semantic web technologies such as RDF and OWL in order to bring the ability of knowledge management for the machines and facilitate the condensation process.

Overall, the hypothesis we are going to verify is the following: *“we can use the semantic web technologies in order to provide a semantic infrastructure for storing software metadata and reasoning about them enabling an easier recovery of UML diagrams from the underlying source code where sophisticated condensation analysis tasks will be reduced to some few SPARQL queries.”*

The rest of the paper is organized as follows. The second section is to review the background and some popular related work. Then, in the third section the proposed approach is described and a general architecture is presented in order to show the feasibility of the approach. The fourth section will then come up with an illustration example. Finally, the fifth section is to conclude and discuss the future work.

2 Related Work

The related work to our approach is divided into two categories. The first category is related to the applications of the semantic technologies in the software development life cycle and the second category includes the current methods for condensation of reverse engineered diagrams. Therefore, in the next subsections we review some popular work in these two areas.

2.1 The semantic technologies and software engineering

The semantic web technologies such as RDF and OWL bring the ability of knowledge management for the machines, and consequently, they can help software engineers with automating different activities involved in software development life cycle. There are a significant number of research activities in this field which utilize the semantic web technologies including requirement engineering [7, 8], software testing[9], automatic component selection [10], automatic model creation from textual specifications of the softwares[11-13], and automatic detection and selection of design patterns[14-16]. The most related approach to our paper is the Tappolet et al.'s approach where they propose an approach for reasoning about software evolution[17]. They introduce EvoOnt Ontologies which provide the basis for representing source code and meta-data in OWL. This representation reduces analysis tasks to simple queries in SPARQL (or its extensions). There is also a limitation to their approach which is the loss of some information due to the use of the FAMIX meta model. The FAMIX describes the core structure of object oriented software without being fixed to one programming language such as C++, Java, etc; however, it does not model language constructs like switch-statements. As a result, by using the FAMIX model measurements cannot be conducted at the level of statements.

2.2 Condensation of Reverse Engineered UML Diagrams

As for the condensation of the reverse engineered class diagrams, according to Osman et al. [4], *GUI-related Information*, *Private and Protected Operations of a class* and *Helper classes* can be excluded from a class diagram in order to increase the understandability. There are also another measures that can be considered in the condensation process. For instance, it is a common belief that the classes which frequently changes over software evolution are considered as candidates to key classes in a class diagram. Besides, according to Osman et al.[5] the number of public operations of a class is one the most important metrics indicating the importance of the class.

3 Proposed Approach

Our main idea is to make use of the semantic web technologies in order to prepare a suitable infrastructure for reasoning about the evolution of a software project enabling a more accurate recovery of UML diagrams from the underlying source code. As a result we expect to become capable of performing sophisticated condensation analysis with only some few SPARQL queries. By providing a framework for representing software source code and meta-data in an OWL ontology, we will be able to reason over software meta-data and thus facilitating software understandability and maintenance

In this section therefore, we introduce a general model for updating software documentation over the software evolution by using the ontology-based modeling. Next, we deepen our approach by focusing on the reverse engineered class diagrams and a semantic model is proposed specifically for the condensation of the reverse engineered class diagrams.

3.1 The V-OntModel for Updating Software Documentation

According to the history of utilizing the semantic technologies in different steps of software development life cycle as discussed in the related work section, we propose a general model called the *V-OntModel* for updating software documentation at different stages of software development by using ontology-based modeling of the software. We can argue that the main benefit of using the V-OntModel is to help overcome lack of information about the elements of the artifacts produced in the software evolution. For instance, consider a class diagram and a metric called the Number of Revisions (NOR) of a class which enables us to recognize the key classes in an application. The more revisions made to a class, the more important the class is. By exploiting the V-OntModel from the beginning of the development, we will be able to record the number of revisions of a class while this is not possible when we do not store metadata about a software over its evolution.

The main idea behind the V-OntModel is that the ontologies can define any kind of relationships between different items in the micro real world at different levels of abstraction, thus various types of UML diagrams can be modeled by using ontologies at different levels of abstraction.

As represented in Figure 1, the V-OntModel depicts the relationship of ontology-based modeling actions to the actions associated with requirements engineering, software design and construction activities. As a software team moves down the left side of the V, basic problem requirements in the micro real world are refined into more and more technical representation of the problem while also ontology-based models are being created (semi)automatically. Once the software code has been generated, we can move up the right side of the V, making use of ontology-based models, essentially performing a series of reverse engineering tasks to recover up-to-date design documentations. The V-OntModel provides a way of visualizing how reverse engineering tasks are applied to different software development steps by creating the ontology-based models.

In order to illustrate how the V-OntModel works, we draw your attentions to two examples regarding *Component Diagrams* and *Use-Case Diagrams*. In a semantic infrastructure, changes to object oriented code or requirements analysis documents can be controlled by a *software ontology* Such as EvoOnt [17] and directly applied to UML diagrams.

- Example 1: As for a Component Diagram, in the case of any change in the relationships between components in the source code, the changes can be applied simply to the Component Diagram with respect to the software ontology.
- Example 2: Regarding the Use-Case Diagrams, any considerable change in user stories prepared in the requirement analysis can be applied to the software ontology, thus changing the use-case diagram.

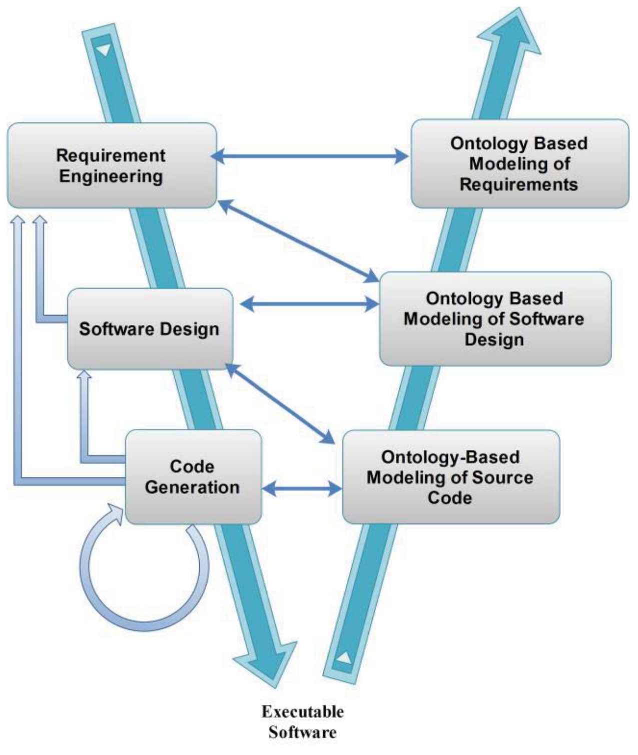
Requirement Engineering
Ontology Based Modeling of Requirements
Software Design
Ontology Based Modeling of Software Design
Code Generation
Ontology-Based Modeling of Source Code
Executable Software

Figure 1 The V-OntModel

3.2 Proposed Architecture

One of the suitable architectures that can show the feasibility of our approach has been depicted in Figure 2. It consists of seven components: *Source code Parser*, *RDF/OWL Converter*, *Reference Ontology*, *Decision List*, *SPARQL Query Generator* and *Condensation Unit*. We discuss these components in more details in the sections below.

3.2.1 Source Code Parser

To parse the source code and generate a first model of the software project, a Source Code Parser has been embedded in the architecture in order to recognize different components like Class, Method or Attribute in an Object Oriented source code. Besides, the model is expected to contain different relations between the components such as association relations.

3.2.2 RDF/OWL Converter

The model created by the Source Code Parser has to be converted into RDF/OWL format in order to populate the reference ontology. Reference Ontology is a manually designed OWL ontology enabling us to store meta-data about a software system and reason over them. Naturally, modeling system metadata in OWL brings us Interchangeability, Non-ambiguity, Machine-readability and Extensibility[17].

By embedding a RDF/OWL converter in the proposed architecture, a database of RDF triples is produced as the software metadata repository.

3.2.3 Condensation Unit

The Condensation unit consists of a decision list and a SPARQL Query Generator. In this unit, the RDF data produced by the RDF/OWL converter is analyzed so that we can rank the classes and then decide on the candidate classes to be excluded or included in the reverse engineered class diagram according to a decision list as will be discussed. Obviously, there should be a SPARQL Query Generator embedded in the architecture in order to extract the insight from the provided database.

In the following section we elaborate on the decision list and the SPARQL Query Generator.

3.2.3.1 Decision List

There are a number of measures that can be considered for condensing a reverse engineered class diagram. For instance, it is believed that the classes which frequently changes over software evolution are considered as candidates to key classes. In addition, Osman et al. [4] provide a research which concludes the information that should be excluded from a class diagram includes *GUI-related Information*, *Private and Protected Operations of a class* and *Helper classes.*

Besides, according to Osman et al.[5] the number of public operations of a class is one of the most important metrics indicating the importance of the class. Accordingly, we believe that different kinds of *cohesion* and *coupling* are important in order to recognize the key classes in a component-level design. As for the coupling, software engineers try to keep it as low as possible in order to decrease the complexity of the system; however, sometimes high coupling cannot be avoided and its ramifications have to be understood[1]. Consequently, when a class is inevitably highly coupled with other classes through a large number of public operations, it has to be a sign of importance for that class.

Overall, there should be a list of measures for the proposed architecture in order to recognize the key classes. We propose a classification of the most important factors into three categories:

- **Software Design Patterns:** Software design patterns might include classes which do not play a top priority role in the application. The proposed procedure is therefore to mine the software metadata repositories for finding different design patterns by posing few SPARQL Queries. Next, we will be able to recognize the low priority classes in a design pattern in order to exclude them from reverse engineered class diagrams. To make this subject clear, an illustration example is proposed in the next section.

- **Metrics:** Tappolet et al. [17] compute different metrics based on their reference ontology that can be useful for our objectives as well. Among these metrics, NOPA (Number of Public Attributes) and NOR (Number of Revisions) can be of valuable help for deciding on the key classes.

- **Algorithms:** while deciding on the key classes by using the machine learning techniques, we have to decide on different methods such as Relational Probability Trees and Relational Bayesian Classifier. The end user can make a final decision on this task.

3.2.3.2 SPARQL Query Generator:

In the proposed architecture, the SPARQL Query Generator will pose SPARQL queries based on the decision list where it might include machine learning tasks. Therefore, we can make use of SPARQL extensions such as SPARQL-ML or any other customized version of SPARQL.

4 Illustration Example

A **helper class** is used in object oriented programming in order to assist some functionality, which is not the primary objective of the application or class in which it is invoked. An instance of a helper class is called a **helper object**. As correctly argued by Osman et al. [4] helper classes should be excluded to simplify a reverse engineered class diagram. In order for that to happen, our hypothesis is to mine the metadata repositories for finding structural patterns such as delegation patterns in which a helper object can be occurred. A **delegation pattern** in software engineering is a design pattern that delegates a responsibility to an associated helper object. Fortunately, finding each type of a structural pattern, such as delegation pattern can be done by posing few SPARQL queries over the software metadata repository[17]. Finally, after finding delegation patterns and helper classes inside them, we can decide on each of helper classes to be excluded or included based upon, for instance, user-defined conditions in a (semi)automatic mode.

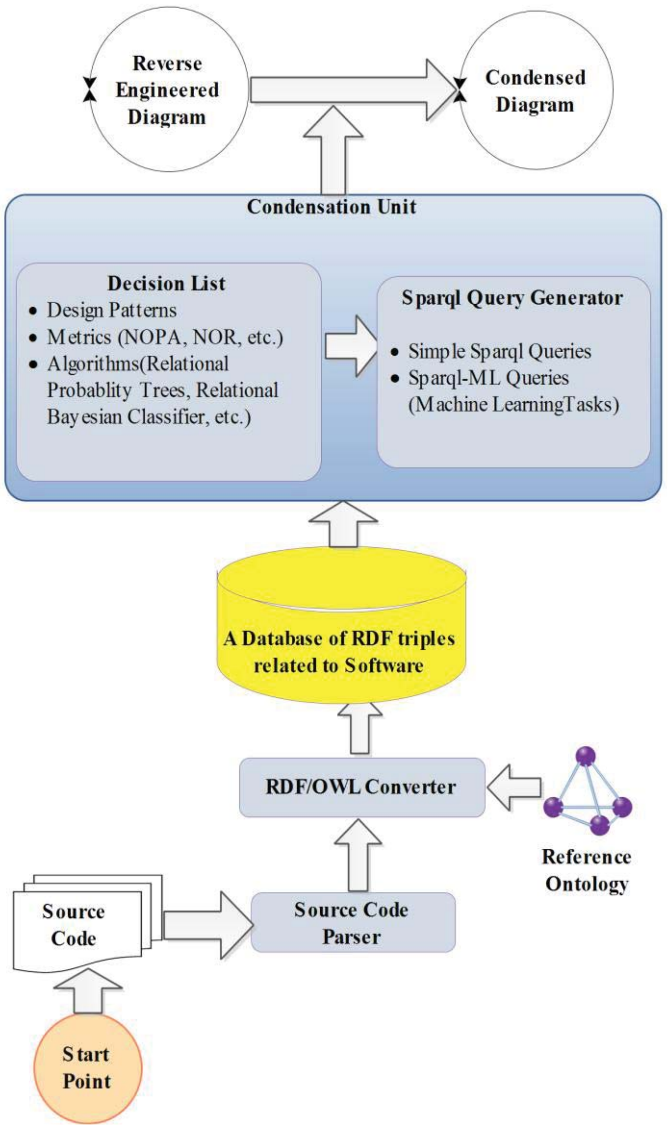
Reverse Engineered Diagram
Condensed Diagram
Condensation Unit
Decision List
• Design Patterns
• Metrics (NOPA, NOR, etc.)
• Algorithms(Relational Probablity Trees, Relational Bayesian Classifier, etc.)
Sparql Query Generator
• Simple Sparql Queries
• Sparql-ML Queries (Machine LearningTasks)
A Database of RDF triples related to Software
RDF/OWL Converter
Reference Ontology
Source Code
Source Code Parser
Start Point

Figure 2 The Proposed Architecture

5 Conclusion and Future Work

In this paper, we are investigating effective methods to (semi)automatically recover an up-to-date UML design documentation which is helpful for the software engineers joining a project at a later stage. Reverse engineering is a popular method for this goal; however, current techniques provide detailed reverse engineered UML diagrams which are difficult to understand. Therefore, we propose two contributions while the focus of this research remains on the class diagrams. First, the V-OntModel is proposed to help overcome lack of information about the elements of the artifacts produced in the software evolution, such as classes in a class diagram. Next, in order to reach a higher level of abstraction and increase the comprehensibility of reverse engineered class diagrams, a condensation architecture is proposed in which the classes are ranked according to a decision list. Then, a final decision can be made on the candidate classes to be excluded from a reverse engineered class diagram. Finally, we can make an obvious argument as follows. When the “V-OntModel” is used over the software evolution, the proposed condensation architecture can achieve better performance in reverse engineering.

As for the future work, we will implement an open-source core system to which everybody can add his own packages for handling different kinds of condensation while there will be also the possibility of adding different algorithms to the core system for a specific goal.

6 References

[1] R. Pressman, *Software Engineering: A Practitioner's Approach*: McGraw-Hill, Inc., 2010.

[2] H. Osman and M. R. Chaudron, "Correctness and Completeness of CASE Tools in Reverse Engineering Source Code into UML Mode," *GSTF Journal on Computing,* pp. 193-201, 2012.

[3] H. Osman*, et al.*, "An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams," unpublished.

[4] H. Osman*, et al.*, "UML class diagram simplification: what is in the developer's mind?," in *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, 2012, p. 5.

[5] H. Osman*, et al.*, "UML Class Diagram Simplification - A Survey for Improving Reverse Engineered Class Diagram Comprehension," in *1st International Conference on Model-Driven Engineering and Software Development*, 2013, pp. 291-296.

[6] F. Thung*, et al.*, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 110-121.

[7] S. J. Körner and T. Brumm, "Improving Natural Language Specifications with Ontologies," in *SEKE*, 2009, pp. 552-557.

[8] M. Ilieva and H. Boley, "Representing Textual Requirements as Graphical Natural Language for UML Diagram Generation," in *SEKE*, 2008, pp. 478-483.

[9] S. Paydar and M. Kahani, "Ontology-based web application testing," in *Novel Algorithms and Techniques in Telecommunications and Networking*: Springer, 2010, pp. 23-27.

[10] O. Hartig*, et al.*, "Automatic component selection with semantic technologies," in *Proc. s of the 4th Int. Workshop on Semantic Web Enabled Software Engineering (SWESE) at ISWC*, 2008.

[11] A. Fatolahi*, et al.*, " A Model-Driven Approach for the Semi-Automated Generation of Web-based Applications from Requirements," in *2008 International Conference on Software Engineering and Knowledge Engineering*, 2010, pp. 619-624.

[12] Y. Tian*, et al.*, "An Ontology-based Model Driven Approach for a Music Learning System," in *SEKE*, 2009, pp. 739-744.

[13] S. J. Körner and T. Gelhausen, "Improving Automatic Model Creation Using Ontologies," in *SEKE*, 2008, pp. 691-696.

[14] J. Dietrich and C. Elgar, "A formal description of design patterns using OWL," in *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, 2005, pp. 243-250.

[15] J. Dietrich and C. Elgar, "An ontology based representation of software design patterns," *Design Patterns Formalization Techniques,* p. 258, 2007.

[16] H. Kampffmeyer and S. Zschaler, "Finding the pattern you need: The design pattern intent ontology," in *Model Driven Engineering Languages and Systems*: Springer, 2007, pp. 211-225.

[17] J. Tappolet*, et al.*, "Semantic web enabled software analysis," *Web Semantics: Science, Services and Agents on the World Wide Web,* vol. 8, pp. 225-240, 2010.