Towards Knowledge-Intensive Software Engineering

Samuel R. Cauvin, Derek Sleeman, and Wamberto W. Vasconcelos

Department of Computing Science, University of Aberdeen, King's College, Aberdeen, AB24 3UE, United Kingdom
{s.cauvin, d.sleeman, w.w.vasconcelos}@abdn.ac.uk

Abstract

This research explores relations between software artefacts and explicitly represented (domain) knowledge. More specifically, we investigate ways in which domain knowledge (represented as ontologies) can support software engineering activities and, conversely, how software artefacts (e.g., programs, methods, and UML diagrams) can support the creation of ontologies. In our approach, class names, and class properties are the principal entities which are extracted from both sources. We implemented a tool, called Facilitator, to support programmers and knowledge engineers when they develop ontologies or programs. This tool provides a list of connections between the ontology and Java project, and provides reasons why these connections have been identified. These connections are created by matching names, types, and superclass-subclass relationships. Facilitator provides a range of semantic web enabled functionalities.

1 INTRODUCTION

This research seeks to provide computational links between software artefacts and explicitly represented (domain) knowledge. More specifically, this project investigates ways in which domain knowledge (represented as OWL ontologies) can support software engineering activities and, conversely, how software artefacts (programs, methods, and UML diagrams) can support knowledge engineering activities.

Currently, there is a substantial gap between (domain) knowledge and software engineering. This gap creates extra burdens on programmers who must re-engineer domain knowledge (introducing possible misconceptions) when they could exploit existing domain knowledge (e.g. ontologies). Consider a situation where a programmer is developing a program for calculating council tax; it is likely that the programmer would look through a list of source materials to find out how council tax is calculated. The programmer would then encode this calculation as a program, possibly creating inaccuracies due to misunderstanding the domain knowledge. Whereas for many common domains, an ontology exists which formally represents this knowledge, from which the programmer could extract concepts and relations that have already been represented in a machine-processable format by an expert. This is the important issue that this research addresses, namely providing methods to relate knowledge represented in an ontology to software engineering activities. We have also addressed the reverse process, i.e. matching the knowledge available in software artefacts to ontologies.

We implemented various functionalities in a tool called Facilitator, which takes an ontology and a Java project and infers links between them. This tool exhaustively analyses the knowledge contained in an ontology and in a Java project and infers potential connections among concepts, as well as providing the reasons why these connections have been formed. Facilitator can also create a "skeleton" project from a source project so that a Java program can be created from an ontology, and vice versa.

Section 2 provides an overview of research in this field. Section 3 describes the goals, requirements and architecture of Facilitator, and discusses technologies used and technical details of the system. Section 4 provides a set of illustrative scenarios to demonstrate matches that Facilitator can detect. Section 5 presents the performance of Facilitator on a variety of tasks. Section 6 discusses some of the problems that were encountered, provides an overall conclusion of this research, and outlines future work.

2 BACKGROUND & RELATED WORK

Knowledge-based software engineering (Havlice et al., 2009; Kravets et al., 2014) aims to support activities and stages of the software life-cycle, such as development, testing, integration, and so on. The research reported here fits within this broad remit, and we show how explicitly represented domain knowledge can be used to support software development.

There have been many attempts to integrate ontologies into the software development process, with most of them focussing on the design phase. Some deal with the use of meta-modelling (Pan et al., 2012). Some deal with translating ontologies into UML models to be used in the design phase of software development (Parreiras et al., 2007). Equally, work has been done to convert UML models into OWL ontologies (Gasevic et al., 2004). However, little work has been done to integrate ontologies into the implementation phase of Software Development, which is the activity which Facilitator supports.

Happel and Seedorf (2006) conducted a high level review of the potential benefits of using domain ontologies in different stages of software development. They suggest that in the implementation phase, ontologies could be used in various ways including: as a domain object model, in coding support, in code documentation, and to integrate with software modelling languages. They also note that ontologies could be used in the analysis and design, deployment and runtime, and maintenance stages in a variety of ways. They do not discuss plans to implement these suggestions.

One major example of a large scale project attempting to integrate ontologies and software is the Marrying Ontologies and Software Technologies (MOST) project[1]. The project produced several papers describing techniques for integrating Ontologies into the software design process. One paper presents a detailed method for mapping class relationships to description logics (Parreiras et al., 2008). Another discusses the potential for combining UML and Ontologies using a framework called TwoUse (Parreiras et al., 2007). A third presents an approach to integrating ontology based meta-models in software modelling, again using the TwoUse framework (Staab et al., 2010).

A tool called RDFReactor[2] provides methods for generating Java classes from an RDF ontology, which is similar to the skeleton creation functionality of Facilitator (Quasthoff and Meinel, 2008).

An outline for a tool that maps software applications to domain ontologies

[1] http://west.uni-koblenz.de/Projects/MOST

[2] http://semanticweb.org/wiki/RDFReactor

is presented in (Hruby, 2005). The mapping is achieved by creating a model of the software addressing two orthogonal dimensions: Categories from the domain ontology, and functional concerns from user requirements. Their approach begins by determining the domain, specifically the scope of the application within a given domain. The tool then locates an ontology to cover this domain, and also to contain the "minimal set of concepts that completely covers the domain". The system then takes into account specific user requirements, specifically those that could not be captured within the ontology. The last step is to construct the application model using the information gathered above. This final step essentially encompasses the matching task performed by Facilitator.

Aspects of the research presented here have been addressed previously in (Cauvin, 2014), but have subsequently been substantially expanded and revised. Specifically, Facilitator can now display matches not just as a list, but the information can be overlaid on the Java source, displayed as a textual critique, or displayed as a statistical overview. Facilitator has also been extended to detect more types of relationships and properties in the sources.

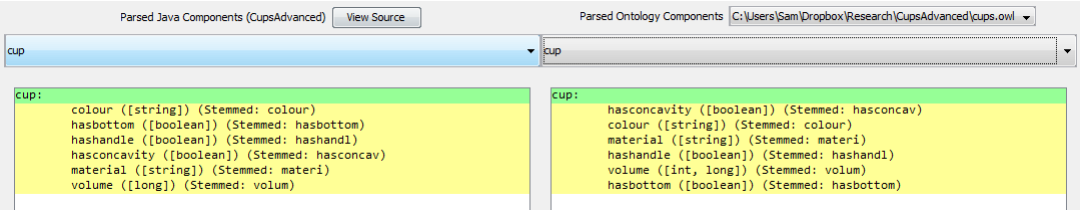

Figure 1: Screenshot of Parsed Components (Showing Java and Ontology Components)

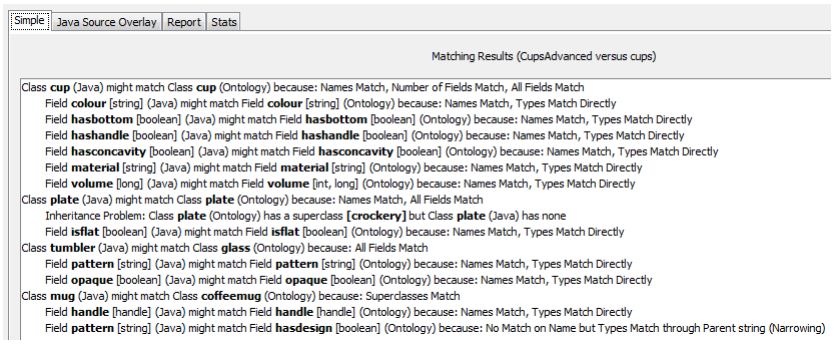

Figure 2: Screenshot of Matching Results

3 SYSTEM DESCRIPTION

Here we discuss the goals, methodology, requirements, and architecture of the system before discussing the technologies and methods used by Facilitator. Source code from this research is available at `https://github.com/Glenugie/Facilitator/`.

3.1 Goals

Facilitator's overall hypothesis is that "It is possible to connect a software artefact with an ontology", which can be decomposed into the following goals:

- To explore ways in which explicitly represented knowledge (e.g., domain ontologies) can support software engineering activities.
- To investigate how to computationally relate domain knowledge with mainstream software technology.
- To connect software artefacts with existing domain knowledge.
- To provide a means of relating an ontology to existing software.

3.2 Methodology

The project methodology comprised the following activities:

1. Literature review to discover existing work in this field.
2. Study of stereotypical activities carried out by developers to construct a list of functional requirements (Section 3.3).
3. Propose a reference architecture to cater for functional requirements (presented in Section 3.4).
4. Incremental development and integration of functionalities.
5. Testing and evaluation of the tool to cater for different scenarios that fulfil the defined functional requirements.

3.3 Functional Requirements

A stereotypical user of the tool would need the following functional requirements (finer-grained requirements have been omitted from this list due to space constraints, they can be found in full in (Cauvin, 2014)):

1. To formally connect an ontology with a software artefact
2. To reason with/about an ontology and the software artefact with a view to understand more about the software and the ontology

Requirement-1 is needed to achieve Requirement-2. Requirement-2 is important to achieve various stages of software development – specifically critique of current design choices as well as revising explicit domain knowledge.

3.4 Architecture

The diagram in Figure 3 describes the components of the system and how they interact. In this diagram, processes are represented by a square, data structures by a barrel, and the user by a stickman. Arrows represent interactions between components, and each interaction is numbered.

The components are:

- UI (GUI) – The interface through which the user interacts with the system
- Control – The main logic behind the system. Responsible for loading files, storing information and querying the ontology/software artefact. This component also utilises the stemmer.
- Matcher – Determines matches between components of the ontology and the software artefact
- Reasoner – The ontology reasoner is used to extract more detailed information from the ontology

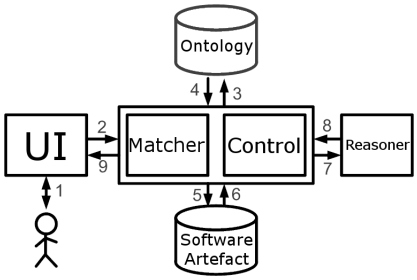

Figure 3: System Architecture

- Ontology – The ontology which the system is analysing
- Software Artefact – The Java project which the system is analysing

Each of the components helps to fulfil at least one of the requirements. Requirement 1 is fulfilled by the Control, Matcher, Ontology, and Software Artefact components. Requirement 2 is fulfilled by the Control, Reasoner, Ontology, and Software Artefact components.

A typical session is outlined in Table 1 which also summarises the system interactions.

Table 1: Architecture Interactions

Interaction	Explanation
1	User interaction with UI
2	Commands from the User
3	Querying the Ontology
4	Results of Query
5	Gathering information from the Software Artefact
6	Receiving information from the Software Artefact
7	Querying the Reasoner
8	Results from Reasoner
9	Updating the UI

3.5 Technologies

We now discuss technologies used in this project.

3.5.1 Java Parser

It is important for the system to parse Java files for content. When searching for an effective way to do this, it quickly became apparent that there were a number of choices in the form of pre-existing APIs. Habelitz JSourceObjectizer[3] is the API that was eventually chosen, as it works with Java 1.7 and can parse source files (without compiling first). JSourceObjectizer produces an Abstract Syntax Tree (AST) which the system can then search for specific components. This package parses Java files accurately and efficiently. The parsing process uses the JSourceObjectizer library to traverse the Java file. While traversing a file the system detects certain Java declarations – Classes, Types, Variables and Methods.

3.5.2 Ontology Access

We chose to use OWL as the format for our ontologies, due primarily to its wide spread availability, so we needed a means to parse OWL ontologies. The obvious

[3] http://www.habelitz.com/

choice for this parser was the OWL API[4], as it fully supports reading and creating OWL ontologies and has extensive documentation.

3.5.3 Stemming

The matching process includes the ability to loosely match names as part of the stemming process. Stemming provides a means to detect names that share similar roots, but are not an exact match. As stemming is a fairly common operation, a stemmer tool was sought to avoid reimplementation. The Snowball Stemmer[5], specifically the Modified Snowball Stemmer[6] which adds compatibly for Java 1.6 and up, was chosen as it returns a single stem. This allows matching to remain a one-click process. The stemming operation simply takes a word and produces a stemmed version of that word, in theory. In practice the word produced is not always real, due to inaccuracies in the algorithm – however it does provide a stemmed match in many circumstances.

3.6 Taxonomy of Types

Facilitator, rather than trying to match field types directly, makes use of a taxonomy of types which attempts to find the match between two types as low in a tree-like structure as possible. This type hierarchy is shown in Figure 4. The top of the tree has the type "Generic" such that anything can match it. When reporting a field match based on type, the system distinguishes between direct matches and matches which occur further up the taxonomy of types, which are then reported to the user.

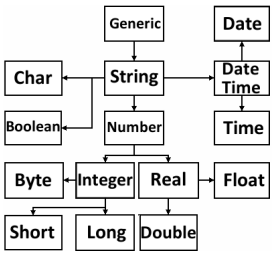

Figure 4: Taxonomy of Types

This taxonomy of types was created by mapping basic OWL data types to the equivalent Java classes. These were then re-ordered slightly to produce the hierarchy shown in Figure 4.

3.7 Matching Process

The matcher compares each Java class/field to each ontology class/field, and if it detects a potential match logs this together with a reason. This reason ensures transparency to the user, allowing them to appreciate why a match has been reported and thus letting them decide whether to accept it. If, for example, the user has written a program with a class Car and it is being compared to an ontology which describes all sorts of vehicles, using classes Vehicle and its subclass Car. In this situation the matcher could potentially match the two Car classes, but highlight

[4] http://owlapi.sourceforge.net/
[5] http://snowball.tartarus.org/
[6] http://trimc-nlp.blogspot.co.uk/2013/08/snowball-stemmer-for-java.html

that the superclass Vehicle is missing in the Java project. Another role for displaying reasons is to give the user an approximate idea of the strength of the match (a match with five reasons is much stronger than a match with just one reason).

The algorithm effectively has three stages:

1. Detecting class matches between class and field names
2. Detecting class matches using superclass relationships
3. Detecting field[7] matches

The last stage, detecting field matches, can be further divided into three sub-stages:

1. Detecting field matches by name and type
2. Detecting field matches using inferred fields
3. Detecting field matches by type but not name, exclusively using previously unmatched fields

These sub-stages are repeated for every class match, as this is effectively looping for each "set" of fields which should match. Two key concepts are introduced by these sub-stages: Being able to detect unmatched fields, and inferred fields. The latter is a field which has been "inferred" from a superclass or subclass, meaning that it is not necessarily accurate. Matches using these fields are labelled as such to alert the user to the potential errors that can occur. Only Java classes have inferred fields, if the ontology were also to have inferred fields the system would be overwhelmed with (incorrect) matches. The choice was made to associate the inferred fields with Java, as we assumed the ontology is the more accurate source. While this is not always true, in most cases it would be reasonable to assume that the ontology has undergone more verification than the Java project/program.

Detecting unmatched fields is handled by a distinct process which simply goes through the list of all fields and compares them against the list of field matches. It then returns the list of fields which have not appeared in the field matches. Using this information is important, as some specific matches are treated as a final attempt at generating matches. An example of this is attempting to detect matches by type when names do not match. Due to the limited number of types, if this was performed on all fields there would be a huge number of incorrect matches generated. Whereas if this process is only run on unmatched fields, there is a lower chance of generating a large number of incorrect matches.

4 ILLUSTRATIVE SCENARIOS

Illustrative scenarios were explored to enable us to demonstrate Facilitator's (reasoning) capabilities. These provide situations (with worked examples) of how reasoning can be used to perform more sophisticated matching. These scenarios are based on two different example domains: Cars and Cups. These two domains are detailed below, along with associated scenarios. The Cars example is from (Cauvin, 2014) and the Cups example is from (Carbonara and Sleeman, 1999). Toy examples are presented here so as to enable a detailed discussion, but we explore larger and more sophisticated examples in Section 5.

4.1 Cars Example

In the Java program we have the following classes:

- Car with fields: String colour, int wheels, and Engine engine

[7] Fields have both a name and a type.

- Engine with fields: int horsepower, and boolean turbo

In the corresponding ontology we have the following classes:

- Car with fields: String colour, int numberOfWheels, int horsepower, boolean turbo, int doors

4.2 Cups Example

In the Java program we have the following classes:

- Cup with fields: String colour, boolean hasBottom, boolean hasHandle, boolean hasConcavity, String material, int volume
- Mug (Subclass of Cup) with fields: String pattern
- Tumbler (Subclass of Cup) with fields: String pattern, boolean opaque
- Plate with fields: String colour, boolean isFlat, String material

In the corresponding ontology we have the following classes:

- Crockery with fields: boolean isCrockery
- Cup with fields: String colour, boolean hasBottom, boolean hasHandle, boolean hasConcavity, String material, int volume
- CoffeeMug (Subclass of Cup) with fields: boolean hasDesign
- Glass (Subclass of Cup) with fields: String pattern, boolean opaque
- Plate (Subclass of Crockery) with fields: String colour, boolean isFlat, String material

4.3 Functionalities

The following is a list of scenarios which relate to the examples above.

- Loosely match fields with different names if a) they have the same type and have not already been matched, and b) if the Java and ontology class names match. In the Cars example, the Java field wheels would be loosely matched to $numberOfWheels$, $horsepower$ and $doors$ in the ontology as the latter are unmatched and have the same type as $wheels$. This example shows that Java fields with no match on name can be matched by type.
- In a Java class if a field is a non-primitive type (most likely a user-defined class), and the fields do not match the ontology completely then the system could inspect the user defined class and check its fields. In the Cars example this would allow the algorithm to deduce that class *Car* effectively has the fields: String $colour$, int $wheels$, int $horsepower$, and boolean $turbo$. This combined list of fields matches the ontology better as the process increases the number of field matches by two. This example states that Java fields with a non-primitive type can have *their* fields combined with those of the top-level class.
- If two superclasses match, their subclasses are likely to match. In the Cups example the classes (Java) Mug and (ontology) $CoffeeMug$ do not match on name or type, however they both have superclasses Cup which do match. The system would suggest this as a match, and in this situation it would be correct. This example states that if two classes share a superclass, then they potentially match.
- Missing inheritance – Super and subclass exist in Java, only the subclass exists in Ontology. The system should point out the (potentially) missing superclass to the user. Conversely, the same is reported when the ontology

has a superclass that the Java does not have. In the Cups example the class *Plate* is present in both sources and matches by name and fields, however the Java program does not have the superclass *Crockery* that exists in the ontology. This is flagged as a potential Inheritance Problem. This example indicates that if two classes match but only one of them has a superclass, then the other is assumed to be missing.

- If a class is "misnamed" in Java, the system could look for any class in the ontology with similar fields and infer a possible match if more than a predetermined number of the fields match. In the Cups example the class *Tumbler* in Java and the class *Glass* in the Ontology have the same fields, and would be marked as a possible match. This example indicates that if two classes have similar fields, then they might match.

5 EVALUATION

To determine if Facilitator can process a wide range of Java projects/ontologies a set of tests were conducted. There were three Java projects, and three ontologies involved in the test. For each combination, the time to parse each component was recorded. The matching times were not recorded, as they run in less than a second when there are no matches. A study of the matcher performance appears in Section 5.1, where two sets of related Java projects and ontologies are compared.

The three Java projects were:

- Facilitator – The code for this project (19 classes, 3643 lines – parsed in 0.001 seconds).
- TDWB – Code from the TDWB system (Sleeman et al., 2014), used for discovering patterns in temporal data (62 classes, 20007 lines – parsed in 2 seconds).
- Xerces – Apache Xerces project[8] which is an XML parser written in Java (707 classes, 216744 lines – parsed in 20 seconds).

The three ontologies were:

- Eng UoL quONTOm – quONTOm[9] project based in the University of Lodz, Poland containing details about quantum physics (58 concepts – parsed in 0.001 seconds).
- Ling GOLD 2010 – GOLD (General Ontology for Linguistic Description)[10] is an ontology for descriptive linguistics (last updated in 2010). Source appears to be the E-MELD project (503 concepts – parsed in 0.001 seconds).
- Chem RSoC CMO – Chemical Methods Ontology from the Royal Society of Chemistry[11] (2358 concepts – parsed in 0.001 seconds).

5.1 Matcher Performance

To test the performance of the matcher algorithm, it is important to use two related sources so that at least some matches exist. To this end, there are two sets of Java projects and ontologies that are comparable:

- Cups Advanced – This consists of an ontology (6 concepts) which was extended from a cup theory (Winston et al., 1983). The corresponding Java

[8] http://xerces.apache.org/
[9] http://merlin.phys.uni.lodz.pl/quONTOm/
[10] http://datahub.io/dataset/gold
[11] http://www.rsc.org/ontologies/CMO/

source (4 classes, 41 lines) was constructed to match the ontology, with a few differences introduced. Matching was run in 0.001 seconds, with 1 out of 6 classes being reported as matching exactly.

- Xerces – Discussed above, and based on the Apache Xerces project which is an XML parser written in Java (707 classes, 216744 lines). Skeleton creation was used to make an ontology (743 concepts) which matched the Java source. Matching was run in 26 seconds, with 515 out of 743 classes matching exactly.

5.2 User Experiments

We are currently carrying out user experiments to ascertain if/how Facilitator can support the various stages of software and knowledge engineering. Our initial studies use as subjects computing science students in their final years and present them with a fragment of a Java program (printed on paper) and an ontology (represented as a UML diagram and printed on paper) and ask them to perform a series of tasks with and without the aid of Facilitator. We designed a questionnaire to get the subject's opinion on whether they agree with the suggestions made by Facilitator. In this initial study, we present participants with the results from Facilitator and only have them use Facilitator as an informal task at the end of the study. In a future study we intend to have participants use Facilitator directly to help them locate and correct a complex bug within a piece of software.

The Java/Ontology source used in the first experiment is the Cups example from Section 4.2. The questionnaire that participants were asked to fill out consisted of a list of all the matches produced by Facilitator, and it provides space to explain whether they agree or disagree with each match. We have not yet completed enough of this study to produce meaningful quantitative data, but informal feedback provided by the participants on the Facilitator software has so far been positive, and the features they have requested were already planned to be implemented in the near future. For example, one of them asked if there was a way to view the ontology as a graph, which is discussed in Section 6.

5.3 Comparison with Existing Work

We compare Facilitator with related work surveyed in Section 2). RDFReactor[12] performs a subset of Facilitator's features – it provides a similar functionality to skeleton creation, but with no matching facility. TwoUse (Parreiras et al., 2007) discusses combining UML and Ontologies, which would suggest it applies to the design phase – whereas Facilitator works with source code in the implementation phase. The tool proposed by Hruby (2005) is the most similar to Facilitator in that it also proposes mapping software to an ontology. However, rather than mapping two specific sources together it deals with situations in which you have an existing piece of software and then the tool will determine the appropriate domain and locate an ontology automatically. Reviewing the paper it is not clear where they get these ontologies from (e.g. searching the web or a centralised repository). Additionally, the method seems to just consider simple concept relationships without performing any reasoning.

[12] `http://semanticweb.org/wiki/RDFReactor`

6 DISCUSSION, CONCLUSIONS & FUTURE WORK

Facilitator encountered a problem (caused by the differences between ontology and Java structures) when dealing with classes with the same name. Within a Java project, there can be multiple classes with the same name occurring in different packages. In contrast, an ontology can only contain one class with a given name. This creates an inconsistency when converting between the formats (through the skeleton creation features) as the multiple Java classes are automatically merged into a single class conglomeration (union operation) in the ontology. The same problem occurs with fields. Facilitator has been designed in such a way that this problem does not affect the regular running of the system, it is only when creating a skeleton (specifically creating an Ontology skeleton from Java) that the issue arises. After analysing possible solutions to the problem, it was decided to pre-process Java classes to add numbers to disambiguate class names.

This solution allowed the system to retain all classes, without requiring user intervention. However, a further problem remains. If one of the "duplicate" classes was a parent of another class, that child class would now be pointing to an ambiguous class name. This problem was overcome by using Java import statements of a class to resolve the package of the parent class. Since a package can only contain one class of a given name, knowing the package allowed unambiguous identification of the parent class.

We report on work aiming at bridging the gap between software and knowledge engineers. We developed a tool, Facilitator, as a proof-of-concept prototype to implement various functionalities to support knowledge-based software engineering. We present an overview of the techniques used by Facilitator to make use of ontologies in the implementation phase of the software development process. This has included a review of other tools with a similar purpose, a detailed overview of how Facilitator performs the matching process, discussion of some of the important functionalities of Facilitator, and a list of planned features of Facilitator.

Connecting software design and domain knowledge has the potential to increase the productivity of programmers by automatically spotting misconceptions at an earlier stage. Similarly, a mismatch between software and domain knowledge could result in the latter being revised. There are also advantages of explicitly modelling knowledge in software as well articulated components.

We have reported some preliminary results from evaluation studies earlier in Section 5.2; more extensive evaluations are planned.

Two major additional system features are planned: Harmonisation and Ontology Graphing. Harmonisation is a proposed feature of Facilitator, where the system can make specific, user-specified changes/corrections to an *existing* project based on another source. Ontology Graphing will use a graphing API to display ontology source as a UML-like graph, with the further possibility of overlaying matches onto this graph.

ACKNOWLEDGEMENTS

The first author would like to acknowledge the support of the University of Aberdeen, Development Trust Intelligent System Fund.

We would also like to thank Dr. Honghan Wu and Dr. Yuan Ren from the University of Aberdeen for their insight into the current state of Knowledge-Based Software Engineering.

REFERENCES

Carbonara, L. and Sleeman, D. (1999). Effective and efficient knowledge base refinement. *Machine Learning*, 37(2):143–181.

Cauvin, S. R. (2014). Towards knowledge-intensive software engineering. Honours B.Sc. Dissertation, Dept. of Comp Sci, University of Aberdeen.

Gasevic, D., Djuric, D., Devedzic, V., and Damjanovi, V. (2004). Converting uml to owl ontologies. In *Procs of the 13th International WWW Conference*.

Happel, H.-J. and Seedorf, S. (2006). Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC*.

Havlice, Z., Adamuščínová, I., Pločica, O., Révés, M., and Železník, O. (2009). Knowledge based software engineering. *Computer Science and Technology Research Survey, elfa, Kosice*, pages 1–10.

Hruby, P. (2005). Ontology-based domain-driven design. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development, San Diego, CA, USA*.

Kravets, A., Shcherbakov, M., Kultsova, M., and Iijima, T. (2014). *Knowledge-Based Software Engineering: 11th Joint Conference, JCKBSE*, volume 466. Springer.

Pan, J. Z., Staab, S., Aßmann, U., Ebert, J., and Zhao, Y. (2012). *Ontology-Driven Software Development*. Springer.

Parreiras, F. S., Staab, S., and Winter, A. (2007). Twouse: Integrating uml models and owl ontologies. Technical Report 16/2007, Institut für Informatik, Universität Koblenz-Landau.

Parreiras, F. S., Staab, S., and Winter, A. (2008). Improving design patterns by description logics: A use case with abstract factory and strategy. In *Modellierung 2008, 12.-14. Mrz 2008, Berlin*.

Quasthoff, M. and Meinel, C. (2008). Semantic web admission free–obtaining rdf and owl data from application source code. In *4th International Workshop on Semantic Web Enabled Software Engineering*.

Sleeman, D., Moss, L., and Kinsella, J. (2014). Temporal discovery workbench: a case study with icu patient datasets. In *BCS Health Informatics Scotland Conference, Glasgow*.

Staab, S., Walter, T., Grner, G., and Parreiras, F. S. (2010). Model driven engineering with ontology technologies. In *Reasoning Web. Semantic Technologies for Software Engineering*, volume 6325.

Winston, P. H., Binford, T. O., Katz, B., and Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. In *National Conf on A.I.*