2012 19th Working Conference on Reverse Engineering

Exploring how to Develop Transformations and Tools for Automated Umplification

Miguel Garzón, Timothy Lethbridge
School of Electrical Engineering and Computer Science
University of Ottawa
Canada
mgarz042@uottawa.ca

***Abstract*—In this research we are exploring how to perform incremental reverse engineering from Java to Umple, a process we call Umplification. Umple is a textual representation that blends modeling in UML with programming language code. It is designed to allow anything from pure textual modeling to almost-pure traditional programming with some modeling concepts added. Umplification involves increasing the proportion of modeling concepts in the code. Novel features of this work are: a) the transformations required are intended to be applied incrementally by a programmer who has a body of legacy code and wants to gradually transform it into Umple, preserving much of the layout, comments and other aspects of the original code if possible; b) the transformations required are at the same time code-to-model, model-to-model and code-to-code. The main contributions will be developing the transformations, developing a usable tool, and demonstrating its effectiveness by means of case studies.**

***Index Terms*—Incremental Reverse Engineering, Textual Modeling, Umple.**

I. Introduction and Work So Far

In this ongoing doctoral research we are exploring how to incrementally add UML modeling constructs to a program written in Umple [1] [2]. We have developed a prototype tool to allow incremental reverse engineering – transformation of code by gradually adding UML constructs while preserving semantics, and many other aspects of the code. The research involves improving this tool and its transformations, and evaluating how effective this tool will be in practice.

A. Umple and Umplification

Umple is a modeling and programming language that adds UML abstractions to base programming languages such as Java, PHP, C++ and Ruby. These UML abstractions allow programmers to express common concepts more succinctly, developing at the code and model level simultaneously. Once a programmer starts working with Umple and generates code from an Umple model, he should never need to edit the generated code to accomplish any task. In other words, Round-tripping is not needed and only the Umple files need to be maintained. This because base language code for such elements as custom methods is incorporated directly in Umple files. Umple currently supports class and state diagrams, plus various other concepts such as aspect-orientation, certain design patterns and integrated tracing. It is being extended to support real-time concepts and component diagrams. The approach called umplification [3] explored in this research involves transforming step-by-step a base language program to an Umple program; each step is a refactoring. The starting point and the ending point of the transformations will be a model, which will primarily be rendered in a textual form. The key insight is that a pure Java program can be seen as a special case of an Umple program, as can a pure model. So Umplification involves repeatedly modifying the Umple to add abstractions, while maintaining the semantics of the program, and also such elements as layout, algorithmic methods and code comments.

An additional feature of Umple is that UML diagrams of the textual representation can be viewed and edited in parallel with the text. As Umplification proceeds, the user can see increasingly complete diagrams of the system if he or she so wishes. This is not central to Umplification, but is a useful by-product.

Figure 1 presents the umplification process.

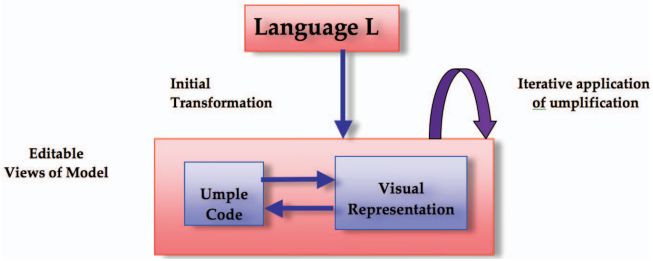

Figure 1: The Umplification Process

To start, source files with language L (e.g. Java) code are initially renamed as Umple files, with extension .ump. Then, iteratively, small transformations are performed that gradually add UML abstractions. Each iteration should involve testing to check that the program's semantics are preserved.

Umplification can be performed automatically by a reverse engineering technology that can parse a program in any native programming language and extract the Umple model from it. In the initial phase of this doctoral research, a prototype system has been implemented to handle the process of umplification from a Java program.

B. The Umplificator Prototype

Figure 2 shows the high-level architecture of the prototype system called *the Umplificator*.

1095-1350/91 $25.00 © 4891 IEEE
DOI 10.1109/WCRE.2012.58

The Umplificator is a reverse engineering tool for the incremental automatic detection of opportunities for refactoring in Java/Umple source code. The first version of the Umplificator targets only Java 1.4, 1.5 and 1.6 source code and is currently implemented in Java. The tool leverages two key technologies to implement its reverse engineering capabilities: JavaML and the Java Development Tools (JDT).

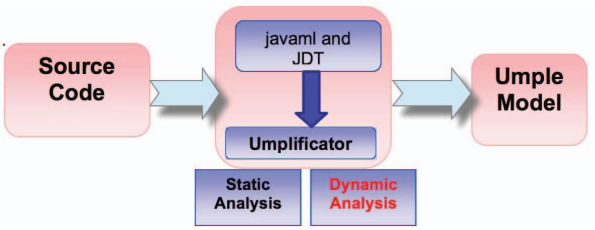

Figure 2: The Umplificator tool architecture. Source code is transformed into javaml, mapped into a Java model and output as an Umple model. This process iterates since an Umple Model is also source code.

Specifically, JavaML is used to produce an XML representation of the source code. The Umplificator uses the JavaML output to construct an abstract syntax tree (AST) and performs semantic analysis. Finally, a Java Model is constructed using the JDT technology and the Umple model is created by transforming in a series of steps the Java Model.

For the conversion of the Java Model into an Umple model, the Umplificator uses a set of mapping rules. The mappings defined are heuristics based on syntactic and semantic features of Java and Umple.

A Domain-Specific Language (DSL) is used to specify the set of mapping rules. A rule is an instruction indicating how a piece of the Java Metamodel is mapped to a piece of the Umple Metamodel. For instance, the following rules state that a JavaElement corresponds to an UmpleElement and a MethodDeclaration corresponds to an UmpleMethod:

```
[RULE]JavaElement::java -> UmpleElement::umple;

[RULE]MethodDeclaration::javaMethod -> UmpleMethod::uMethod;
```

In more detail, the Umplificator reverse engineering process as it stands is summarized below:

- The input is the source code of the system. Currently this must be Java. The intent is that it could be Umple (i.e. Java that already contains Umple constructs).
- The source code is transformed into an XML representation using JavaML. To allow importing of Umple we will have to enhance JavaML or develop an alternative for this step.
- A program model (in-memory representation) is extracted from the source code using JDT and complemented with the information derived from the previous step. Again, we will have to enhance JDT to allow importing of Umple.
- The model is incrementally transformed into an Umple model. This is the key step, and involves transformations that operate on the data structures in memory.
- The final result is the Umple model with the transformations applied, which is output as text to replace the original input. When compiled, it will produce essentially the same system as the original. It is an objective that correct umplification will result in a system that executes correctly and is indistinguishable from the end-user's perspective.

The initial version of the Umplificator supports configuration options for detection of constructs representing UML attributes, the interpretation of multiplicities, role names and the recovery of one-to-one associations.

The next steps include developing mapping rules and support for reverse engineering of constructs representing:

- State machines
- Other kinds of associations: unidirectional, bidirectional, reflexive, qualified and association classes.
- Exceptions (may possibly be mapped to OCL constraints)
- Software Design Patterns, particularly singleton and immutable, which are native to Umple.

Finally, we are planning to support multiple OO programming languages. For instance, to add support for C++ we need to plug into the architecture a new component (srcml) to parse C++ code (in Figure 2 the component would replace the javaml framework) and create a set of mapping rules that will allow us to convert a C++ model into an Umple Model.

C. Related Work

Numerous approaches have been described for reverse engineering associations and related information from Java, none of them are incremental or produces compilable artifacts.

Most of these techniques produce in one single step a UML model derived from the source code. In [4] Gogolla and Kollman infer UML associations, multiplicities and aggregation semantics by examining the source code. This approach defines one of the few technique for finding bi-directional associations.

In [5], Barowski and Cross propose a technique to extract dependency information from Java bytecode. This approach, however, is not able to express multiplicity of the recovered associations.

Kollman and Gogolla [6] use both static and dynamic analysis to recover UML constructs from C++ code. Tonella and Potrich [7] focus exclusively on recovering behavioral aspects of a C++ program.

Sutton and Maletic [8] [9] propose a set of mappings that are intended to recover design-level UML class models from source code. To validate their approach and accuracy of the mappings, they present a tool called Pilfer and use it to reverse engineer HypoDraw, an open source tool.

Commercial tools, including IBM Rational Software Architect [10], Together [11], Umbrello [12], Visual Paradigm [13] and ArgoUML [14] lack configurability options and incremental reverse engineering capabilities. Moreover, they detect simple attributes, as well as one-way

associations between classes but produce incomplete generated code (e.g. lacking methods and/or referential integrity) and fail to preserve semantics when the UML models derived from the tools are input in their own code generators [15].

Reclipse [16] another interesting tool that combines static analysis (graph matching) and dynamic analysis to detect pattern implementations in source code. Reclipse is an open source tool providing developers with support for model-based software engineering and re-engineering. This is so far the most popular and cited research tool found in the literature.

Finally, MoDisco [17], provides a set of generic tools to understand and transform complex models created out of existing systems. MoDisco uses JDT [18] to discover Java elements in Java projects and allow the user to gather metrics and visualize results in a tree representation. Transformations can be performed to the discovered elements using another Eclipse technology, the ATL project [19].

What distinguishes the work described in this from other work, is the incremental detection of the refactorings, the configurability of the mappings rules, and the preservation of semantics when code is generated from the recovered models.

II. Problem Statement and Research Questions

The problem to be addressed is this:

Developers currently often work with large volumes of legacy code. Tools exist to allow them to extract models or transform their code in a variety of ways. However doing so tends to result in a system that is quite different in syntax and structure. They are thus inhibited from using reverse engineering tools except to generate documentation. The Umple technology partly solves this problem by allowing incremental addition of modeling constructs into familiar programming language code. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it. Converting to Umple (Umplification) has been done manually – indeed it was applied to the Umple compiler itself [1] – , but ought to have tool support so it can be done in a more automatic, systematic and error-free manner on large systems.

The research questions are as follows:

- What transformation technology, transformations and refactoring patterns will work best for umplification?
- What percentage of code reduction and complexity reduction can we achieve by umplification, and how can we measure the complexity reduction?
- Overall, what are the benefits of automated or semi-automated umplification as compared to manual umplification or the use of other reverse-engineering or transformation approaches.
- What should be the architecture, implementation and user interface of an umplification tool?

III. Methodology and Proposed Solution

The major steps in the methodology are the following:

1. Manually perform umplification to gain an understanding of what will be needed
2. Iteratively develop The Umplificator tool, exploring the effectiveness of various reusable components and transformation approaches. This includes selection or creation of an easy-to-use tool to express transformations from the base language to Umple. We want to avoid complex XML-based solutions since usability will be key.
3. Start with a major case study (JHotDraw), iteratively umplifying it and improving the Umplificator until the Umple version of the case study compiles and a significant number of constructs have been umplified successfully
4. Iteratively develop more and more transformations to convert additional Java code into Umple. Introduce additional case studies until the Umplificator works well on 10-15 reasonably large open-source systems.
5. Compare the work to alternative approaches.

IV. Evaluation Method and Results So Far

So far we have build the Umplificator prototype and successfully partially umplified JHotDraw [20], converting variables to Umple attributes and modestly reducing code size.

A. JHotDraw Case Study

JHotDraw 7 (version 7.5.1), an open source graphic editor that supports operations on many graphic file formats. JHotDraw makes extensive use of software design patterns, and contains fairly extensive documentation of the patterns used and of which classes embody them. Additionally, it is highly cited in the Software engineering literature. Finally, JHotDraw is written in Java. The analyzed version consists of 138 packages, 689 classes, 8720 methods, 1503 constructors and 81632 lines of code (LOC).

In the first step of umplification applied to the JHotDraw, we refactored the Java classes, Java interfaces and their dependencies.

In the second step, we refactored the primitive data types. The primitive data types become Umple attributes. This step required the cleaning of the getters and setters of these attributes and the insertion of aspect-oriented code injections in some getters and setters to maintain the semantics of the original code.

The resulting Umple code was 74,972 lines. This represents an 8.15% improvement (fewer lines of code).

B. Planned Evaluation

When complete, the work will be evaluated in the following manner:

One part of the evaluation approach will follow the model often taken in machine learning approach. We will use a set of open-source systems as the 'training set', developing transformations that allow for successful umplification of these. Then, as a last set, we will umplify a set of previously-untried systems, the 'testing set' and assess the extent to which our transformations still work. We will compare the results of the Umplificator with manual umplification of at least one of these 'unknown' systems.

In addition we will subject the Umplificator to evaluation with real developers We will arrange for a group

of users to umplify some small systems. They will umplify some systems by hand, and others with the Umplificator. We will be able to measure the speed and correctness of the result, and also administer a questionnaire to the participants. In addition to working with small systems, we will video developers (both students and those in industry) as they umplify larger programs, while being encouraged to think aloud. We will then analyze the videos looking for places where the participants make mistakes, express frustration, spent a lot of time thinking, have to refer to help, etc. We will also evaluate in a similar manner the language used to express the mapping rules

V. Expected Contributions

Key contributions of this work are expected to be the following:

- The overall concept of umplification
- An understanding of how umplification compares with other reverse engineering techniques (incrementality, minimal adjustment of code to prevent disruption)
- The Umplificator tool itself
- Case studies of Umplification, demonstrating strengths, weaknesses and opportunities, as well as hopefully demonstrating that the resulting system is easier to understand and has less code.
- Transformation patterns for (mapping rules) Umplification and the language for expressing these. These should be general-purpose and easily modifiable to allow future researchers, and even end users, to add to them.
- Detection of associations (of all different types) and state machines in a body of code. There is little successful work in this area in the literature.

VI. Conclusions

In this paper we presented our reverse engineering approach called Umplification and the corresponding prototype tool the Umplificator. Umplification is the process of transforming step-by-step a base language program to an Umple program.

The major advantages compared to other approaches are the concept of incrementality and the mapping rules, which we intend to make general-purpose and easy to modify and add.

We presented some evaluation results showing that our approach and its current implementation are effective and efficient enough to be applied in the future to real systems.

As this doctoral work progresses, we plan to apply the approach to other open source systems, improve the mapping rules, improve the technology to allow Umple code to be input and add support for dynamic analysis for cases in which static analysis and parsing of the code is not enough to capture all the possible umplifiable concepts that can be derived from a program. We also want to integrate mapping rules for state machines and some popular software design patterns.

References

[1] A. Forward, T.C. Lethbridge, and D. Brestovansky, (2009), “Improving Program Comprehension by Enhancing Program Constructs: An Analysis of the Umple language”, International Conference on Program Comprehension (ICPC) 2009, Vancouver, IEEE Computer Society, pp. 311-312.

[2] A. Forward, O. Badreddin, T.C. Lethbridge, and J. Solano, (2011) “Model-Driven Rapid Prototyping with Umple”, Software Practice and Experience, 42: pp. 781-707 DOI: 10.1002/spe.1155

[3] A. Forward, T.C. Lethbridge and Omar Badreddin, “Umplification: Refactoring to Incrementally Add Abstraction to a Program”. 2010 17th Working Conference on Reverse Engineering.

[4] M. Gogolla, and R. Kollman, (2000), "Re-Documentation of Java with UML Class Diagrams", in Proceedings of 7th Reengineering Forum, Reengineering Week 2000, Zurich, Switzerland, Feb 29 - Mar 3, pp. 41-48.

[5] I.A. Barowski, and J.H. Cross, (2002), "Extraction and Use of Class Dependency Information in Java", in Proceedings of Ninth Working Conference on Reverse Engineering (WCRE'02), Richmond, Virginia, Oct 29-Nov 1, pp. 309-318.

[6] R. Kollman, and M. Gogolla, (2001), "Application of UML Assciations and Their Adornments in Design Recovery", in Proceedings of Eigth Working Conference on Reverse Engineering (WCRE'01), Stutgart, Germany, Oct 2-5, pp. 81-92.

[7] P. Tonella, P. and A. Potrich, (2001), "Reverse Engineering of the UML Class Diagram from C++ Code in the Presence of Weakly Typed Containers", in Proceedings of International Conference on Software Maintenance (ICSM'01), Florence, Italy, Nov 6-10, pp. 376-385.

[8] A. Sutton, and J.L. Maletic, (2005), "Mappings for Accurately Reverse Engineering UML Class Models from C++", in Proceedings of 12th Working Conference on Reverse Engineering (WCRE ' 05), Pittsburgh, PA, Nov 7-11, pp. 175-184.

[9] A. Sutto, (2005), Accurately Reverse Engineering UML Class Models from C++, Kent State University, Kent, Ohio, Masters Thesis.

[10] IBM Corp., Rational Software Architect, http://www.ibm.com/developerworks/rational/products/rsa/, visited 2012

[11] MicroFocus Corp, Together, http://www.microfocus.com/products/micro-focus-developer/together/index.aspx, Visited 2012.

[12] SourceForge, Umbrello UML Modeller, http://uml.sourceforge.net/, visited 2012

[13] Visual Paradigm, http://www.visual-paradigm.com/, visited 2012

[14] Tigris, ArgoUML, http://argouml.tigris.org/, visited 2012

[15] I.D. Baxter, C. Pidgeon, and M. Mehlich, (2004), "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 -28, pp. 625-634.

[16] Fujaba, Reclipse – Reverse Engineering for Eclipse, accessed 2012, http://www.fujaba.de/no_cache/projects/reengineering/reclipse.html

[17] MoDisco, Discovering Source code Elements, accessed 2012, http://www.eclipse.org/MoDisco/

[18] Eclipse Java development tools (JDT), accessed 2012, http://www.eclipse.org/jdt/

[19] ATL (ATL Transformation Language) , accessed 2012, http://www.eclipse.org/atl/

[20] JHotDraw, Visited 2012, http://sourceforge.net/projects/jhotdraw/?source=directory