Reverse Engineering of Object-Oriented Code into Umple using an Incremental and Rule-Based Approach

Miguel A. Garzón[1], Timothy C. Lethbridge[1], Hamoud Aljamaan[1], Omar Badreddin[2]

School of Electrical Engineering and Computer Science
University of Ottawa, Canada[1]
School of Electrical Engineering and Computer Science
Northern Arizona University, Flagstaff, U.S.A[2]
mgarzon@uottawa.ca, tcl@eecs.uottawa.ca, hjamaan@uottawa.ca,
Omar.Badreddin@nau.edu

Abstract

In this paper we present a novel approach to reverse engineering, in which modeling information such as UML associations, state machines and attributes is incrementally added to code written in Java or C++, while maintaining the system in a textual format. Umple is a textual representation that blends modeling in UML with programming language code. The approach, called *umplification*, produces a program with behavior identical to the original one, but written in Umple and enhanced with model-level abstractions. As the resulting program is Umple code, our approach eliminates the distinction between code and model. In this paper we discuss the principles of Umple, the umplification approach and a rule-driven tool called the *Umplificator*, which implements and validates the depicted approach.

1 Introduction

Many software systems experience growth and change for an extended period of time. Maintaining consistency between documentation and the corresponding code becomes challenging. This situation has long been recognized by researchers, and significant effort has been made to tackle it. Reverse engineering is one of the fruits of this effort and has been defined as the process of creating a representation of the system at a higher level of abstraction [1].

Reverse engineering, in general, recovers documentation from code of software systems. When such documentation follows a well-defined syntax it is often now referred to as a model. Such models are often represented using UML (Unified Modeling Language), which visually represents the static and dynamic characteristics of a system.

There is a long and rich literature in reverse engineering [2]. Most existing techniques result in the generation of documentation that can be consulted separately from the code. Other techniques generate models in the form of UML diagrams that are intended to be used for code generation of a new version of the system. The technique discussed in this paper goes one step further: It modifies the source code to add model constructs that are represented textually, but can also be viewed and edited as diagrams. The target language of our reverse engineering process is Umple [3], which adds UML and other constructs textually to Java, C++ and PHP.

We call our approach to reverse engineering a software system umplification. This is a play on words with the concept of 'amplification' and also the notion of converting into Umple. In our previous work [4], we have found that umplifying code is reasonably straightforward for someone familiar with Umple, and with knowledge of UML

modeling pragmatics. Moreover, we have performed *manual* umplification of several systems, including Umple itself.

The present paper focuses on how the umplification process can be performed *automatically* by a reverse engineering technology and consists of four main parts. Section 2 introduces the Umple modeling language and its constructs. Section 3 describes the umplification technique and the steps required to transform a program into Umple. Section 4 describes the tool implemented to validate our approach. Finally, a case study is outlined in the last section.

1.1 Motivation

Developers often work with large volumes of legacy code. Reverse engineering tools allow them to extract models in a variety of ways [5], often with UML as the resulting formalism.

The extracted models can be temporary, just-in-time aids to understanding, to be discarded after being viewed. Such a mode of use can be useful, but is limited in several ways: Developers still need to know where to start exploring the system, and they need to remember how to use the reverse engineering tool every time they perform an exploration task.

Developers generally therefore would benefit from choosing reverse engineering tools that create a more permanent form of documentation that can be annotated or embedded in larger documents, and serve as the definitive description of the system.

However by making the latter choice, the developer then needs to maintain two different artifacts, the original code and the output model. The recovered models become obsolete quickly, unless they are continuously updated or are used for 'roundtrip engineering'. The complexity of this inhibits developers from using reverse engineering tools for permanent documentation.

The umplification technique we present in this paper overcomes the problems with either mode of reverse engineering described above. It results in a system with a model that can be explored as easily as with just-in-time tools. But there is also no issue with maintaining the model, because model and code become the same thing.

In other words, the key difference compared to existing reverse engineering techniques is that the end-product of umplification is not a separate model, but a single artifact seen as both the model and the code. In the Umple world, modeling is programming and vice versa. More specifically, for a programmer, Umple looks like a programming language and the Umple code can be viewed as a traditional UML diagram. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it into Umple [6].

In addition to solving the problem of having two different software artifacts to maintain, umplification can be used to simplify a system. The resulting Umple code base tends to be simpler to understand [7] as the abstraction level of the program has been 'amplified'.

2 Background: the Umple Language

Umple [3] is an open-source textual modeling and programming language that adds UML abstractions to base programming languages including Java, PHP, C++ and Ruby.

Umple has been designed to be general purpose and has UML class diagrams and UML state diagrams as its central abstractions. It has state-of-the art code generation and can be used incrementally, meaning that it is easy for developers to gradually switch over to modeling from pure programming. Umple was designed for modeling and developing large systems and for teaching modeling [8]. Umple is written in itself – the original java version was manually umplified many years ago. That experience was one of the motivations for the current work.

In addition to classes, interfaces and generalizations available in object oriented languages, Umple allows software developers to specify:

1. **Associations**: As in UML, these specify the links between objects that will exist at run time. Umple supports enforcement of multiplicity constraints and manages referential integrity – ensuring that bidirectional references are consistently maintained in both directions.
2. **Attributes**: These abstract the concept of instance variables. They can have properties such as immutability, and can be subject to constraints, tracing, and hooks that take actions before or after they are changed.
3. **State Machines**: These also follow UML semantics, and can be considered to be a special type of attributes, subject to events that cause transitions from one value to another.

States can have entry or exit actions, nested and possibly parallel substates, and activities that operate in concurrent threads.

4. **Traits**: A trait is a partial description of a class that can be reused in several different classes, with optional renaming of elements. They can be used to describe re-usable custom patterns.
5. **Patterns**: Umple currently supports the singleton and immutable patterns, as well as keys that allow generation of consistent code for hashing and equality testing.
6. **Aspect Oriented Code Injection**: This allows injection of code that can be run before or after methods, including Umple-defined actions on attributes, associations and the elements of state machines. Such code can be used as pre-conditions and post-conditions or for various other purposes. Code can be injected into the API methods (those methods generated by Umple) as well as into user-defined methods.
7. **Tracing**: A sublanguage of Umple called MOTL (Model-oriented tracing language) allows developers to specify tracing at the model level, for example to enabling understanding of the behavior of a complex set of state machines operating in multiple threads and class instances [9].
8. **Constraints**: Invariants, preconditions and postconditions can be specified.
9. **Concurrency**: Umple provides several mechanisms to allow concurrency to be specified easily, including active objects, queuing in state machines, ports, and the aforementioned state activities.

The umplification method discussed in this paper currently focuses on associations and attributes, with some generation of Umple's patterns and code injections. As future work it is planned to extend it to encompass other Umple features.

The Umple compiler supports code generation for Java, PHP, Ruby, C++ as well as export to XMI and other UML formats. The compiler generates various types of methods including mutator, accessor, and event methods from the various Umple features. A mutator (e.g. set(), add()) method is a method used to control changes to a variable and an accessor (e.g. get()) method is the one used to return values of the variable. An event method triggers state change. An extended summary of the API generated by Umple from attributes, associations, state machines and other features can be found at [10]. Umple can also generate diagrams, metrics, and various other self-documentation artifacts. Umple models can be created or edited using the UmpleOnline Web tool [11], the command line compiler or an Eclipse plugin.

3 The Umplification Process

Umplification involves recursively modifying the Umple model/code to incorporate additional abstractions, while maintaining the semantics of the program, and also maintaining, to the greatest extent possible, such elements as layout. The end product of umplification is an Umple program/model that can be edited and viewed textually just like the original program, and also diagrammatically, using Umple's tools.

The umplification process has several properties. It is 1) incremental, 2) transformational, 3) interactive, 4) extensible, and 5) implicit-knowledge conserving.

The approach is incremental because it can be performed in multiple small steps that produce (quickly) a new version of the system with a small amount of additional modeling information, such as the presence of one new type of UML construct. At each step, the system remains compilable. The approach proceeds incrementally performing additional transformations until the desired level of abstraction is achieved. These incremental transformations allow for user interaction to provide needed information that may be missing or hard to automatically obtain because the input (the source code) does not follow any of the idioms the automatic umplification tool is yet able to recognize. This characteristic of umplification allows developers, if they wish, to repeatedly re-introspect the transformed program and manually validate each change with an understanding of the incremental purpose of the change.

The approach is transformational because it modifies the original source rather than generating something completely new. It first translates the original language (Java, C++ etc.) to an initial Umple version that looks very much like the original, and then translates step-by-step as more and more modeling constructs are added, replacing original code.

The approach is interactive because the user's feedback may be used to enhance the transformations.

The approach is extensible because it uses the set of transformation rules can be readily extended to refine the transformation mechanism.

Finally the approach is implicit-knowledge conserving because it preserves code comments, and, where possible, the layout of whatever code is not (yet) umplified. The latter includes as the bodies of algorithmic methods – known as action code in UML.

Taken together, the above properties allow developers to confidently umplify their systems without worrying about losing their mental model of the source code. Developers gain by having systems with a smaller body of source code that are intrinsically self-documented in UML.

The following gives a summary of the abstract transformations currently implemented.

Transformation 0: Initial transformation.

To start, source files with language L (e.g. Java, C++) code are initially renamed as Umple files, with extension .ump. File, package and data type's inclusions are translated into Umple dependencies by using the depend construct.

Transformation 1: Transformation of generalization/specialization, dependency, and namespace declarations.

The notation in the base language code for subclassing is transformed into the Umple 'isA' notation. Umple now recognizes the class hierarchy. Notations representing dependency are transformed into Umple 'depends' clauses, and notations for namespaces or packages are transformed into the Umple 'namespace' directives. At this stage, an Umple program, when compiled should generate essentially identical code to the original program.

Transformation 2: Analysis and conversion of many instance variables, along with the methods that use the variables.

This transformation step is further decomposed into sub-steps depending on the abstract use of the variables. The sub-steps are defined as follows.

Transformation 2a: Transformation of variables to UML/Umple **attributes**.

If variable a is declared in class A and the type of a is one of the primitive types in the base language, then a is transformed into an Umple attribute. Any accessor (e.g. getA()) and mutator (e.g. setA(…)) methods of variable a are transformed as needed to maintain a functioning system. In particular, any getter and setter methods in the original system must be adapted to conform to or call the Umple-generated equivalents.

Transformation 2b: Transformation of variables in one or more classes to UML/Umple **associations**.

If variable a is declared in Class A and the type of a is a reference type B, then a is transformed into an Umple Association with ends {a, b}. At the same time, if a variable b in class B is detected that represents the inverse relationship then the association becomes bidirectional. The accessor and mutator methods of variable a (and b) are adapted to conform to the Umple-generated methods. Multiplicities and role names are recovered by inspecting both types A and B; this is explained in Section 3.3.

Transformation 2c: Transformation of variables to UML/Umple **state machines**.

If a is declared in Class A, has not been classified previously as an attribute or association, has a fixed set of values, and changes in the values are triggered by events, and not by a set method, then a is transformed to a state machine. We will not cover this aspect of umplification further in this paper, and will leave the focus on attributes and associations.

As mentioned before, as part of each transformation step, the accessor, mutator, iterator and event methods are adapted (refactored) to conform to the Umple generated methods. Table 1 summarizes these additional required refactorings.

Table 1. Refactorings to methods required for each transformation.

Transformation case	Method Transformations
(0) Classes	None
(1) Inheritance	None
(2a) Attributes	Accessor (getter) and mutator (setter) methods are removed from the original code if they are simple since Umple-generated code replaces them. Custom accessors and mutators are refactored so Umple generates code that maintains the original semantics.
(2b) Associations	Accessor and mutator methods are removed or correctly injected into the umple code.
(2c) State Machines	Methods triggering state change are removed if they are simple (just change state) or modified to call Umple-generated event methods. Not covered further in this paper

In the following sub-sections, we provide a more detailed view of the transformation cases. To help distinguish between Umple and Java code presented in this paper, the Umple examples appear in dashed borders with grey shading, pure Java examples have solid borders with no shading. Mapping rules (in the Drool language, that we will describe shortly) appear using double line borders with no shading.

3.1 Initial Refactoring of Classes

The first step in umplification (Transformation 0) is to rename the Java/C++ files as .ump files. After this, various syntactic changes are made (Transformation 1) to adapt the code to Umple's notations for various features that are expressed differently in Java and C++. Umple maintains its own syntax for these features so as to be language-independent.

First the base language notation for inheritance (e.g. 'extends' in Java) or interface implementation (e.g. 'implements') is changed into the Umple notation 'isA'. This Umple keyword is used uniformly to represent the generalization relationship for classes, interfaces and traits. The same notation is used for all three for flexibility – so that, for example, an interface can be converted to a class with no change to its specializations, or a trait can be generated as a superclass in languages such as C++ where multiple inheritance is allowed.

After this, the dependency notation in the native language (e.g. 'import' in Java) is changed to the 'depend' notation in Umple. Finally 'package' declarations are transformed into Umple namespace declarations.

Transformations made as part of these first refactoring steps, are one-to-one direct and simple mappings between constructs in the base language and Umple. No methods need changing. The final output after execution of the above transformations, is an Umple model/program that can be compiled in the same manner as the original base language code. At this point, any available test cases may be run to ensure that the program's semantics are preserved. For instance, the Java code (in file Student.java) shown below:

```
package university;

import java.util.*;
public class Student extends Person
{ }
```

would result in the following Umple implementation (in file Student.ump).

```
namespace university;
class Student {
   depend java.util.*;
   isA Person;
}
```

3.2 Refactoring to Create Attributes

In this sub-section, we present how member variables possessing certain characteristics are transformed into Umple attributes (Transformation 2a). An Umple attribute is a simple property of an object, but following UML semantics, it is more than just a plain private variable: It is designed to be operated on by mutator methods, and accessed by accessor methods. These methods, in turn can have semantics such as preconditions and tracing injected into them.

We start by analyzing all instance variables for their presence in constructor and get/set methods and decide whether the member variable is a good candidate to become an Umple attribute [12]. In Table 2, we present the developed (programmable) heuristics used for the partial analysis of member variables. The instance variables with a low or very low probability of being attributes are ignored for now. Those with high and medium probability are further analyzed.

Table 2. Analyzing instance variables for presence in the constructor and getter/setters.

Constructor	Setter	Getter	Attribute (probability)
Yes	Yes	Yes	High
Yes	Yes	No	Low
Yes	No	Yes	High
Yes	No	No	Low
No	Yes	Yes	High
No	Yes	No	Low
No	No	Yes	Medium
No	No	No	Very Low

Furthermore, we check the type of the *candidate attributes* (those with a High or Medium probability) and draw a conclusion regarding whether or not the member variable corresponds to an Umple Attribute, because some will be left to be later transformed into associations. If the candidate attribute has as its type either: a) a simple data

type, as in Table 3 or b) a class that only itself contains instance variables meeting conditions in a and b (for attributes with 'many' multiplicity), then the member variable is transformed into an Umple Attribute.

Table 3. Umple Primitive Data Types.

Type	Description
Integer	Includes signed and unsigned integers.
String	All string and string builder types
Boolean	true/false types
Double	All decimal object types
Date/Time	All date, time and calendar object types.

We culminate this refactoring step by removing or refactoring getters and setters of the previously identified attributes. More specifically, the getters and setters need to be refactored if they are not simple, but are custom. Simple getters/setters are those that *only* return/update the attribute value. Custom getters/setters are those that provide behavior apart from setting the variable such as validating constraints, managing a cache or filtering the input.

Let us now illustrate this refactoring through an example. Assume that we have already transformed the Java class into an Umple class, so the input at this point is an Umple file containing Java.

```
class Student {
 public static final int
            MAX_PER_GROUP = 10;
 private int id;
 private String name;
 public Student(int id,String name){
  id = id; name = name;
 }
 public String getName(){
  String aName = name;
  if (name == null) {
   throw new
        RuntimeException("Error");}
   return aName;
  }
 public Integer getId() {
   return id; }
 public void setId(Integer id) {
   this.id = id; }
 public boolean getIsActive() {
   return isActive;}
 public void setIsActive(boolean
                      aIsActive)
  {isActive = aIsActive;} }
```

In this example code we first analyze the member variables to determine the following:

1. Is the field present in the parameters of the constructor?
2. Does the field possess a getter?
3. Does the field possess a setter?
4. Is the field's type, a primitive type?

Table 4 shows the analysis results for each of the member variables of the Student class.

Table 4. Analysis of Member variables of class Student.

Member variable	1	2	3	4
id	Yes	Yes	Yes	Yes
isActive	No	Yes	Yes	Yes
name	Yes	Yes	No	Yes
MAX_PER_GROUP	No	No	No	Yes

The results of this analysis allow us to generate Umple code with the required types and stereotypes. For example the stereotype 'lazy' is added of 'isActive' because it should not appear in the constructor, and the stereotype 'immutable' is added to 'name' since there is no setter. The transformed Umple code after completion of this refactoring step (transformation 2a) is shown below. Note that this continues to generate a program that is semantically identical to pre-transformation version.

```
1  class Student {
2   Integer id;
3   lazy Boolean isActive;
4   immutable name;
5   const Integer
6              MAX_PER_GROUP = 10;
7   after getName {
8    if (name == null) {
9     throw new
10      RuntimeException("Error");}
11  }
12 }
```

The following gives details of the above:

Line 2. Field **id** becomes an Umple attribute. Getter *getId*() and setter *setId*() are removed.

Line 3. Field **isActive** becomes an Umple attribute of Boolean type. As the field is not required in the constructor we marked as 'lazy' so the umple compiler does not generate a constructor argument for this attribute.

Line 4. Field **name** becomes an Umple attribute and is marked as 'immutable'. Immutable attrib-

utes must be specified in the constructor, and no setter is provided.

Line 5. Field **MAX_PER_GROUP** becomes a constant (special type of Umple attribute). We have drawn this conclusion because of the field modifiers (e.g. static final) and because of the *ALL_CAPS* convention.

Line 7-10. As the getter for field **name** was custom, we have adapted it so it conforms to the one that can be generated by the umple compiler. A code injection, code that is injected before and/or after statements, have been used for this purpose.

3.3 Refactoring to Create Associations

In this sub-section, we discuss how the umplification technique infers associations from source code (Transformation 2b). More specifically, we discuss how our technique infers all the fields that represent associations including the role name, association ends, multiplicities and directionality.

As discussed earlier, in the various cases of the refactoring steps, analyses are applied to the input variables to determine whether each variable can be transformed into an Umple association. An association specifies a semantic relationship that occurs between typed instances. A variable represents an association if all of the following conditions apply:

- Its declared type is a Reference type (generally a class in the current system).
- The variable field is simple, or the variable field is a container (also known as a collection).
- The class in which the variable is declared, stores, access and/or manipulates instances of the variable type.

In the Umplificator, the tool we will describe in the next section, these conditions are expressed as rules. The transformation of variables into associations involves a considerable number of transformations and code manipulations. In order to guarantee the correct extraction of an association and to avoid false-negative cases, we consider not only the getter and setter of the fields but also the iteration call sequences (iterators). Table 5 and Table 6 present the list of methods considered (parsed and analyzed) in order to infer associations. These methods can be categorized as mutator and accessor methods. In the tables, W is the name of the class at the other end of the association and '…' refers to a collection of elements. We have considered those collections of elements defined using Map, Set, List and Hash classes (from the Java collections framework or the Standard Template Library in C++).

Table 5. Accessor Methods parsed and analyzed.

Accessor Methods	
Method Signature	**Description**
W getW()	Returns the W
W getW(index)	Picks a specific linked W
List<W> getWs()	Returns immutable list of links

Table 6. Mutator methods parsed and analyzed.

Mutator Methods	
Method Signature	**Description**
boolean setW(W)	Adds a link to existing W.
W addW(args)	Constructs a new W and adds link.
boolean addW(W)	Adds a link to existing W.
boolean setWs(W…)	Adds a set of links.
boolean removeW(W)	Removes link to W if possible.

A simple example is presented now to summarize the main idea behind this transformation step. Assume that Umple code shown below has already passed through the two first refactoring steps. As a result, classes, dependencies, and attributes (if any) have been properly extracted.

```
class Mentor {
 depend java.util.Set;

 public Set<Student> students;
 public Set<Student> getStudents()
  { return students; }

 public void  setStudents
      (Set<Student>students)
  { this.students = students; }

 public void addStudent( Student
                         aStudent)
  { students.add(aStudent); }

 public void removeStudent(Student
                           aStudent)
  { students.remove(aStudent);}
}
class Student {
```

```
public Mentor mentor;
public Mentor getMentor()
 { return mentor; }
public void setMentor(Mentor mentor)
 { this.mentor = mentor; }
}
```

The resulting Umple code after completion of this refactoring step (transformation 2b) is shown below.

```
1  class Mentor {
2   0..1 -- 0..* Student;
3  }
4  class Student {}
```

Line 2 contains the association derived from the Java code that can be read as: a mentor can have many students associated but a student can only be associated to at most one mentor.

The following particularities have been taken into consideration during the extraction of the association:

In class Mentor:

- The *students* variable in class Mentor is of a reference type and possesses a getter and a setter.
- We inferred the multiplicity of the association end "0..*" by a) inspecting the cardinality of the member, and b) by analyzing the getter/setter of the member variable.
- We inferred the navigability of the association "--" by inspecting the two classes involved. In this case, each class can access the linked objects of the other class. The notation "->" would otherwise have been used to represent a unidirectional association.
- The association end is optional-many because the member is not present as a parameter in the constructor (not required upon construction) of an instance of the class Mentor and because the member represents a collection of elements.

In class Student:

- The *mentor* in class Student is of a Reference type and it possesses a getter and a setter.
- We inferred the multiplicity of the association end "0..1" by inspecting the constructor of the class Student. It is optional-one because it is not required upon construction of class Student.

Consider again the previous example. If we inject now the following constructor into the Student class, the multiplicity for the association end would become "1" instead of "0..1".

```
public Student(Mentor aMentor){
  mentor = aMentor;
}
```

Note that in the examples, the Java input made use of generics (templates using <> syntax) for the specification of a collection of elements. For those cases in which the type of the member variable cannot be directly inferred (in older Java code), we analyze the add/remove methods to determine the type of the element that is added to the collection.

Ultimately, each refactoring step should involve testing (running the test suites) to check that the program's semantics are preserved.

4 The Umplificator: A Tool for Semi-Automated Umplification

In this section, we provide an overview of the tool we have developed to support umplification; as well as discuss some of its technical details.

Our tool is called the Umplificator. It assumes that the input is a set of classes written in base language code (Java, C++ etc.), Umple files, source code directories or software projects (source code containers as represented in many popular IDEs such as Eclipse). The output is an Umple textual model containing base language code with modeling abstractions. The Umple model is fully compatible with many UML and XMI formats and can be viewed or edited diagrammatically.

At its core, the Umplificator is a language interpreter and static analyzer that parses base language and Umple code/models, populates a concrete syntax graph of the code/model in memory (JavaModel, CPPModel), performs model transformation on the base language representation in memory and then outputs Umple textual models.

The Umplificator relies on initial parsing by tools such as the Java Development Tool (JDT) for Java, CDT for C++, and PDT for PHP. These extract the input model from base language code. The use of JDT and its siblings reduces the need to write an intermediate parser for the base language.

The base language model is then transformed in a series of steps into an Umple model. To do this, the Umplificator uses a pre-defined set of refactoring rules written in the Drools rule language [13]. Drools is a rule management system with a forward and backward chaining inference based rules engine. The rule engine is explored in more detail in Section 4.2.

The Umplificator includes other subsidiary and internal tools such as:

- **Language validators** – A set of base language validators allowing validation of the base language code that is generated after compilation of the recovered Umple models.
- **Umplificator statistics –** A metrics-gathering tool to analyze certain aspects of a software system such as the number of classes and interfaces, the number of variables present in the code, the cyclomatic complexity, the number of lines of code [14].
- **Umplificator Workflow** – A tool that guides the umplification process within Eclipse.

The Umplificator is available as an IDE and works within Eclipse; it also operates as a command-line tool to allow rapid bulk umplification and easier automated testing. Both tools are built and deployed using the Ant scripting language; resulting in several executable jars as well as for the Eclipse plugins.

The development of the Umplificator follows a test-driven approach to provide confidence that future enhancements will not regress previously functioning and tested aspect of the system.

4.1 Architecture

The Umplificator has a layered and pipelined architecture. The pipelines (components) in this architectural style are arranged so that the output of each element is the input of the next. Figure 1 presents the architecture.

The process of umplifying a system in this architecture is described below (Figure 2).

1. The input is a set of source code files in the base language and/or Umple.
2. The source code is transformed into base-a model of the base language and Umple constructs.
3. The model previously obtained is entered into the next stage of the pipeline. The input model is transformed a model with additional Umple features using pre-defined mapping rules.
4. The target Umple model, is then validated.

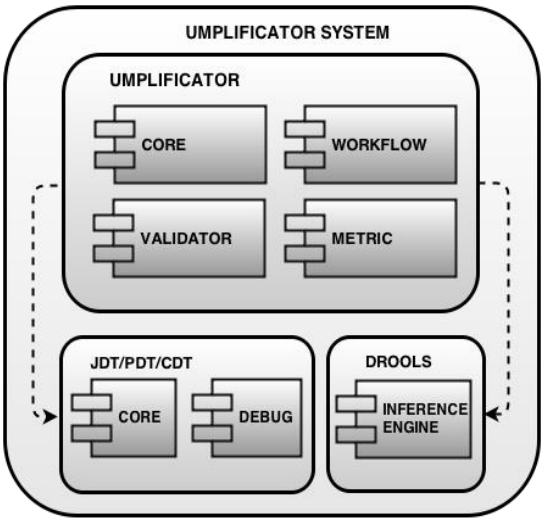

Figure 1. The Umplificator components

The mapping rules and rule engine are introduced in the following sub-section.

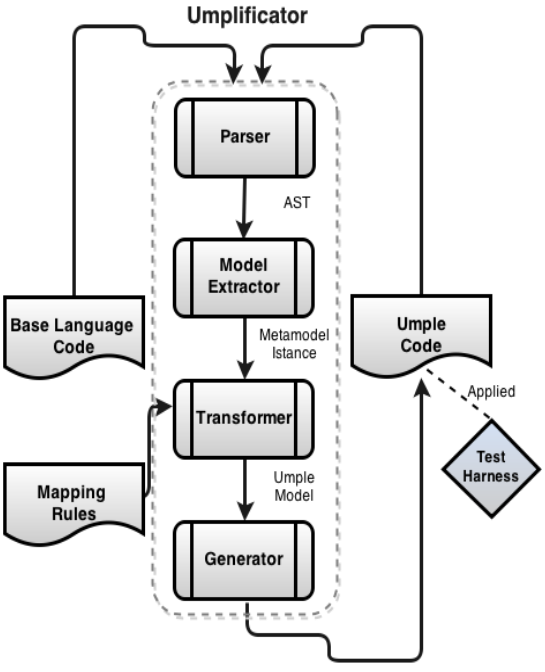

Figure 2. The umplification process flow

4.2 Rule-Based Language

The rule engine interprets and executes the mapping rules on the source model and target model to produce the umplified version of the target model.

The Drools engine used by the Umplificator is composed of an inference engine that is able to scale to a large number of rules and facts. The inference component matches facts and data (base language models) against rules to infer conclusions, which result in actions (model transformations). A rule is a two-part structure (LHS and RHS) using first order logic for reasoning over knowledge representation. Pattern matching is performed to match facts against rules and is implemented using the Rete algorithm [15]. The rule engine is initialized with the rules. A Drools rule has the basic form:

```
rule "name"
  when LHS then RHS
end
```

where LHS is the conditional part of the rule and RHS is a block that allows dialect-specific semantic code to be executed.

The rules are grouped in files for each of the cases (levels of refactoring) discussed earlier. In other words, there is a rule file containing rules, functions and queries to transform variables into attributes, another file containing those to transform variables into associations and so on.

The rules as explained in this paper are instructions indicating how a piece of the Base language model (Java Model, C++ model, etc.) is mapped to a piece of an Umple model. Additionally, in Drools, one can specify:

- **Functions**: These are used for invoking actions on the consequence (then) part of the rule, especially if that particular action is used over and over again. In the Umplificator, functions are used instead of helper classes so the logic is kept in one place.
- **Queries**: These provide a means to search working memory and store the results under a named value. In the Umplificator, they are used to gather metric information about the models analyzed. For instance, a query *numberOfPublicMethods*(..) returns the number of methods having 'public' as modifier. Queries do not have side effects, meaning that their evaluation cannot alter the state of the corresponding executing unit.

In the Umplificator, the logic used for model transformations resides in the rules. Moreover, by using rules, we have a single point of truth, a centralized repository of knowledge. Rules can be also read and understood easily, so they can also serve as documentation.

Traditionally, rule engines have two methods of execution [16]: forward chaining and backward chaining. In forward chaining, the facts are asserted into working memory resulting in one or more rules being concurrently true and scheduled for execution. In backward chaining (goal driven), one starts with a conclusion, which the engine tries to satisfy. Drools is a Hybrid Chaining System because it implements both forward and back-ward mechanism. Our Umplificator uses the forward chaining method of operation in which the inference engine starts with facts, propagates through the rules, and produces a conclusion (e.g. a refactoring).

As an example, consider the rules in Listing 1. The rule named "transform_Import" (Lines 1-10) matches and converts any Import Declaration (Java Language) into an Umple depend construct. The dependency (Line 8) is then added to a matched Umple Class. The Umple Class is then put into the working memory (Line 9) so subsequent transformations can be made on the object (forward chaining). The rule named "Java-Field_IsUmpleAttribute" converts Java fields into basic Umple attributes. The attribute is then added to a matched Umple Class (Line 24). The attribute is put into the working memory (Line 25) so subsequent transformations can be made such as determining if the attribute is lazy or not. The rule named "isLazy_Attribute", not shown here, is used for this purpose. This rule matches and converts any basic attribute (in memory) that conforms to the required conditions into a lazy attribute (e.g. attribute.setIsLazy(true)). The complete set of mapping rules for the umplificator can be found at the umple code repository [17].

Listing 1. Umplificator Rules

```
1  rule "transform_Import"
2   when
3    import: ImportDeclaration();
4    uClass: UmpleClass() ;
5   then
6    Depend depend = new
7     Depend(getImportName(import));
8    uClass.addDepend(depend);
9    insert(uClass);
10 end
```

```
11  rule "JavaField_IsUmpleAttribute"
12   when
13   field:FieldDeclaration
14   (field.getType().isUmpleAttr())
15   uClass: UmpleClass()
16   then
17   Attribute uAttr =
18   new Attribute(null, null, null,
19             null, false, uClass);
20   uAttr.setName
21  (getFieldName(fieldDeclaration));
22  uAttr.setType(getAttributeType
23               (fieldDeclaration));
24  uClass.addAttribute(uAttr);
25  insert(uAttr);
26  end
```

5 Case Studies

We tested the Umplificator on several systems. Here we will discuss our experiences with JHotDraw and Weka.

JHotDraw7 [18] is an open source graphic editor that supports operations on many graphics file formats. It makes extensive use of software design patterns and has detailed documentation about its design. We selected JHotDraw for umplification to be able to apply our transformations on documented frameworks and to compare results with the documentation of these frameworks and the analyses performed by other tools [19]. For this research we worked with JHotDraw 7.5.1.

Table 7 shows the results of detection of attributes and associations for the JHotDraw framework. It details the number of classes, the number of attributes and the different types of associations. We also performed a manual analysis to check the accuracy of our algorithms and mapping rules. After improving and refining our rules, we have obtained a precision of 100%. The refinement consisted of adding the Java idioms that our detection algorithms were not able to catch on the first attempt. For instance, not all the setters in the framework return always a void, some of them return a boolean.

In previous work [20], nine different types of associations were identified. In this paper, we present results for three of them, corresponding to the top multiplicity patterns in industry: optional-one-to-many, optional-one-to-one and many-to-many associations.

During the initial transformation of JHotDraw from Java into Umple, we encountered code blocks in Java code that could not be readily transformed into Umple; we enhanced Umple to support the missing features. Java annotations, multiline comments and abstract classes are examples of missing features that were implemented as part of this reverse-engineering exercise.

Table 7. Results of analysis for JHotDraw.

JHotDraw 7.5.1 (138 classes and 22 interfaces)			
UML abstraction	**Found**	**Expected (Manual)**	**False Hits**
Attributes	363	363	0
Associations **optional-one-to-many**	49	49	0
Associations **optional-one-to-one**	185	185	0
Associations **many-to-many**	32	32	0

The Umplificator was hence tuned to be able to umplify JHotDraw. With each new system we umplify, we increase the accuracy of the mapping rules as well as the overall effectiveness of the umplificator. In general, tuning the Umplificator to increase the accuracy includes one or more of the following manual steps:

1. If there is an Umple construct that was missed from the extraction (false negative), we may add a new mapping rule to cover this case.
2. If there is an Umple construct that was incorrectly identified (false positive), we may edit the corresponding mapping rule.
3. If one of the methods requiring additional transformations (as described in Table 1) was incorrectly refactored, we may review and correct the refactoring action (a function in Drools language) that led to the incorrect piece of code.

Briefly, the complexity of the tuning depends on the number of false positives and false negatives that the tool generates.

The next system we focused on was the machine learning tool Weka [21]. As with our the first attempt at umplifying JHotDraw, our first attempt at automatically umplifying Weka resulted in a precision of less than 100% – some idioms it uses were not yet detected by our tool.

For example, some classes in the classifiers package implement add() and remove() methods with different argument types. Also, the Confusion class declares add(RuleSet) and re-

move(Antecedent) to add and remove a set of rules from the evaluation algorithm. In addition, we detected, after execution of the test suite, that two classes were not compiling due to an unexpected constructor signature.

Initial Umplification results for Weka nonetheless have a precision of 85% when it comes to attributes and 38% for 1-to-many associations. Table 8 summarizes the results. Note that a precision of 38% doesn't mean that the Umplificator has missed 62% associations of this type. It means that some of them were not correctly transformed into Umple (e.g. incorrect navigability, role names or transformation of accessor/mutator methods). The extensibility and flexibility of our tool allows us to add and refine rules without having to recompile the system.

It is our objective to successively umplify more and more systems, with the hope that eventually our rule base will cover the vast majority of cases needed to successfully umplify new systems the Umplificator is presented with. However, even with a precision in the high 80% range, our tool serves as a useful tool for umplification. Users can leave some variables un-umplified, or can manually umplify the rest.

Table 8. Results of analysis for the Weka tool.

WEKA (1349 classes)			
UML abstraction	**Found (1)**	**Expected (2)**	**Precision (1/2)**
Attributes	1281	1510	85%
Associations **optional-one-to-many**	168	442	38%

6 Related Work

A number of approaches have been presented in the literature for reverse engineering to discover associations and other related information from source code; none of them are incremental or produce compilable artefacts.

The majority of these produce in one single step a UML model derived from the source code [22].

In [23] Gogolla and Kollman present a technique using both static and dynamic analysis to recover UML class models from C++ source code. This approach defines one of the few techniques for finding bidirectional associations.

In [24], Barowski and Cross propose a technique to extract dependency information from Java bytecode. This approach, however, is not able to express correctly the multiplicity of the recovered associations.

Sutton and Maletic [25] propose a set of mappings that are intended to recover design-level UML class models from source code. They present their prototype tool, pilfer, and use it to reverse engineer HypoDraw [26], an open source tool.

Gueheneuc [27] proposes a technique that infers associations, aggregation and composition relationships from Java programs using graph theory. However, their proposed approach requires the availability and analysis of both static and dynamic models to build the class diagrams.

Commercial tools and widely used open source tools, including IBM Rational Software Architect [28], Visual Paradigm [29] and ArgoUML [30] 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 [31].

Reclipse [32] is another interesting tool that combines static analysis (graph matching) and dynamic analysis to detect pattern implementations in source code. This is so far the most popular and cited research tool found in the literature.

MoDisco [33], provides a set of generic tools to understand and transform complex models created out of existing systems. MoDisco uses JDT [34] 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 on the discovered elements using another Eclipse technology, the ATL project [35], a model-to-model (M2M) transformation toolkit.

Finally, TXL [36] is a rule-based language designed for a variety of source-to-source transformations tasks. We experimented with TXL as well as with ATL, to implement the umplification technique, and conclude that the two technologies were not suitable for incremental transformation of multi-language input models (Umple, Java, Umple+Java, etc.). A core concept behind umplification is that we want tool support for incremen-

tality. We need to be able to load and transform a target model, already (partially) transformed by our tool so that users can convert Umple to Umple in steps of their choosing, always passing test cases, and maintaining confidence of the results. Existing model transformation tools proved not readily suitable for the multi-language incremental mode of use.

What differentiates our work from other work is this incremental refactorings, the configurability of the mappings rules, the form of the output (model and code as a single entity), and the preservation of semantics when code is generated from the recovered models.

7 Conclusion and Future Work

In this paper we have presented our reverse engineering approach called umplification and the corresponding tool, the Umplificator. Umplification is the process of transforming step-by-step a base language program to an Umple program that merges textual modeling constructs directly into source code.

An important contribution is the set of transformation patterns that allows developers to easily extend and refine the umplification transformation rules. Another important contribution is the comprehensiveness of our detection of associations and the additional refactoring required to comply with all the different types of associations. Major advantages of our work, as compared to other reverse engineering approaches, are the concept of incrementally, the ease of addition of mapping rules, and the preservation of the system in a textual format.

For the future, we plan to apply the approach to other open source systems, gradually increasing the ability of the Umplificator to obtain a higher and higher first-pass precision on new systems it encounters. We also will integrate the mapping rules for state machines and refine some of the existing rules to make them more maintainable.

About the Authors

Miguel A. Garzón is a PhD candidate and a part-time professor at University of Ottawa, School of Electrical Engineering and Computer Science. He is an Engineer-In-Training (EIT).

Timothy Lethbridge has been a professor of Software Engineering and Computer Science at the University of Ottawa since 1994, where he leads the Complexity Reduction in Software Engineering Lab. He is a Professional Engineer and a member of the IEEE and ACM.

Hamoud Aljamaan is a PhD candidate at University of Ottawa, School of Electrical Engineering and Computer Science.

Omar Badreddin is an assistant professor at Northern Arizona State University. He is a PhD graduate of the University of Ottawa.

References

[1] E. J. Chikofsky and J. H. C. II, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Softw., vol. 7, no. 1, pp. 13–17, Jan. 1990.

[2] G. CanforaHarman and M. Di Penta, “New Frontiers of Reverse Engineering,” in Future of Software Engineering (FOSE ’07), 2007, pp. 326–341.

[3] O. Badreddin, “Umple: a model-oriented programming language,” 2010 ACM/IEEE 32nd Int. Conf. Softw. Eng., vol. 2, pp. 337–338, 2010.

[4] T. C. Lethbridge, A. Forward, and O. Badreddin, “Umplification: Refactoring to Incrementally Add Abstraction to a Program,” Reverse Eng. (WCRE), 2010 17th Work. Conf., 2010.

[5] H. Osman and M. R. V Chaudron, “An Assessment of Reverse Engineering Capabilities of UML CASE Tools,” in 2nd Annual International Conference Proceedings on Software Engineering Application, 2011, pp. 7–12.

[6] A. Forward and T. C. Lethbridge, “Problems and opportunities for model-centric versus code-centric software development,” in Proceedings of the 2008 international workshop on Models in software engineering - MiSE ’08, 2008, p. 27.

[7] A. Forward, T. C. Lethbridge, and D. Brestovansky, “Improving program comprehension by enhancing program constructs: An analysis of the Umple language,” 2009 IEEE 17th Int. Conf. Progr. Compr., pp. 311–312, 2009.

[8] T. C. Lethbridge, G. Mussbacher, A. Forward, and O. Badreddin, "Teaching UML using umple: Applying model-oriented programming in the classroom," 2011 24th IEEECS Conf. Softw. Eng. Educ. Train. CSEET, pp. 421–428, 2011.

[9] H. Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying Trace Directives for UML Attributes and State Machines," in International Conference on Model-Driven Engineering and Software Development, 2014, pp. 79–86.

[10] CRuiSE, "Umple API summary." [Online]. Available: http://api.umple.org.

[11] CRuiSE, "Umple online." [Online]. Available: http://try.umple.org.

[12] O. Badreddin, A. Forward, and T. Lethbridge, "Exploring a Model-Oriented and Executable Syntax for UML Attributes," Softw. Eng. Res. Manag. Appl. SE - 3, vol. 496, pp. 33–53, 2014.

[13] P. Browne, JBoss Drools Business Rules. Packt Publishing, 2009.

[14] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08, 2008, p. 121.

[15] Ding Xiao and Xiaoan Zhong, "Improving Rete algorithm to enhance performance of rule engine systems," in 2010 International Conference On Computer Design and Applications, 2010, vol. 3, pp. V3–572–V3–575.

[16] H. Huang, K. Sugihara, and I. Miyamoto, "A rule-based tool for reverse engineering from source code to graphical models," in Software Engineering and Knowledge Engineering, 1992. Proceedings., Fourth International Conference on, 1992, pp. 178–185.

[17] CRuiSE, "Mapping Rules in Umple code repository." [Online]. Available: http://goo.gl/DFGkZB.

[18] E. Gamma and T. Eggenschwiler, "JHotDraw." [Online]. Available: http://www.jhotdraw.org/.

[19] Y. Guéhéneuc, "A systematic study of UML class diagram constituents for their abstract and precise recovery," in Software Engineering Conference, 2004. 11th Asia-Pacific, 2004, pp. 265–274.

[20] O. Badreddin, A. Forward, and T. C. Lethbridge, "Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity," in SERA 2013, 2013, pp. 129–149.

[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," ACM SIGKDD Explor. Newsl., vol. 11, no. 1, p. 10, Nov. 2009.

[22] M. L. Nelson, "A survey of reverse engineering and program comprehension," arXiv Prepr. cs/0503068, 2005.

[23] R. Kollmann and M. Gogolla, "Application of UML associations and their adornments in design recovery," 2001, pp. 81–90.

[24] L. A. Barowski and J. H. C. II, "Extraction and use of class dependency information for Java," in Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, 2002, pp. 309–315.

[25] R. Kollmann and M. Gogolla, "Application of UML associations and their adornments in design recovery," in Proceedings Eighth Working Conference on Reverse Engineering, 2001, pp. 81–90.

[26] P. Kunz, "HypoDraw." [Online]. Available: http://www.slac.stanford.edu/grp/ek/hippodraw/.

[27] Y. Guéhéneuc, "Abstract and precise recovery of UML class diagram constituents," in Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, 2004, p. 523.

[28] IBM, "Rational Software Architect." [Online]. Available: http://www.ibm.com/developerworks/rational/products/rsa/.

[29] Visual Paradigm, "Visual Paradigm for UML." [Online]. Available: http://www.visual-paradigm.com/.

[30] ArgoUML, "ArgoUML Modeling Tool." [Online]. Available: http://argouml.tigris.org/.

[31] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf, "A study on the current state of the art in tool-supported UML-based static reverse engineering," in Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, 2002, pp. 22–32.

[32] Fujaba, "Fujaba CASE tool." [Online]. Available: http://www.fujaba.de/.

[33] Eclipse, "MoDisco." [Online]. Available: http://www.eclipse.org/gmt/modisco/.

[34] Eclipse, "Eclipse Java development tools (JDT)." [Online]. Available: http://projects.eclipse.org/projects/eclipse.jdt.

[35] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," Sci.Comput.Program., vol. 72, no. 1–2, pp. 31–39, Jun. 2008.

[36] J. R. Cordy, "The TXL source transformation language," Sci. Comput. Program., vol. 61, no. 3, pp. 190–210, Aug. 2006.