Promoting Traits into Model-Driven Development

by

Vahdat Abdelzad

Thesis submitted

in partial fulfillment of the requirements for the degree of

Ph.D. in

Electrical and Computer Engineering

University of Ottawa

Abstract

Traits are primitive units of code reuse that serve as building blocks of classes. In this research, we enhance reuse by extending the capabilities of traits; in particular, we add modeling abstractions to them.

Traits have a variety of benefits, including facilitating reuse and separation of concerns. They have appeared in several programming languages, particularly derivatives of Smalltalk. However, there is still no support for traits that contain modeling abstractions, and no straightforward support for them in general-purpose programming languages. The latter is due to structural concerns that exist for them at runtime, especially traits that contain modeling abstractions.

Model-driven technologies are making inroads into the development community, albeit slowly. Modeling abstractions such as state machines and associations provide new opportunities for reuse, and can be combined with inheritance for even greater reusability. However, issues with inheritance apply also when these new abstractions are inheritable units. This suggests that traits and models ought to be able to be synergistically combined. We perform a comprehensive analysis of using modeling elements in traits. We implement such traits in Umple, which is a model-oriented programming language that permits embedding of programming concepts into models.

The contributions of the thesis are: a) Adding new elements including state machines and associations into traits, hence bringing more reusability, modularity, and applications to traits; b) Developing an algorithm that allows reusing, extending, and composing state machines through traits; c) Extending traits with required interfaces so dependencies at the semantic level become part of their usage, rather than simple syntactic capture; d) Adding template parameters with associations in traits, offering new applications for traits in which it is possible to define design patterns and to have a library of most-used functionality; e) The implementation of all the above concepts, including generating code in multiple general-purpose programming languages through automatic model transformation.

Acknowledgments

This research by all means could not have been finished by me. It is a collaboration of working hard alone, being guided truly by knowledgeable people, and supported by people who are part of my life.

First and foremost, I would sincerely like to appreciate all technical and non-technical advocates I have received from my wonderful supervise professor Timothy Lethbridge. It has been an honor to be one of his Ph.D. students. He taught me many aspects of research, encouraged my research, and allowed me to grow as a research scientist. His enthusiasm to teach, supervise, and research in software engineering is an excellent pattern for my future career.

In my Ph.D. pursuit, my lovely family in particular my father and mother, has been with me and supported me, more specifically in those tough moments in which I have not had impetus to continue, and have felt completely hopeless. I believe there is no way to appreciate all the time they sacrificed for me, except to say, I love you so much.

Table of Contents

List of Figures

List of Listings

List of Tables

Chapter 1. Introduction

Reuse has long been an important objective in software engineering [58]. In this thesis, we demonstrate enhanced mechanisms for reuse, building on the concept of traits [93], which are reusable elements that can be composed to build classes. We show how traits can be extended to incorporate deeper semantics and modeling abstractions like UML associations and state machines. Furthermore, we demonstrate how this can be accomplished for popular programming languages.

These enhancements allow traits to be used in mainstream development by both modelers and programmers. Enhanced traits can be used to help build libraries of reusable models or code, and can reduce duplication and accidentally-duplicated defects. We demonstrate these capabilities by applying them to Umple [12,13,14,15,62], which is a model-oriented programming language that permits embedding of programming concepts into models.

We want to clarify that we are *not* talking about the semantically-distinct notion in C++ called traits classes [42,69]. They are small objects which carry information used by other objects to determine implementation details. For example, they can be used to indicate whether or not a type is "void" inside a C++ program. The use of the term "traits" in this paper follows the usage of the broader object-oriented literature.

This thesis has been written for readers who have knowledge of object orientation and programming languages, as taught at the undergraduate level to computer scientists and software engineers. However, we specified appropriate references for readers who may lack such background.

1.1. Problem Overview

Research in the area of software reuse has helped develop high-level languages, components, generative methods, new architectures, and domain engineering [17,41,70]. Abstractions,

such as those found in modeling languages, and inheritance play particularly important roles in facilitating reuse.

Inheritance shows itself in various forms such as single inheritance, multiple inheritance, and mixins. However, these variations suffer from conceptual and practical problems. For instance, there is a troublesome situation named the "diamond problem" or "fork-join inheritance" for multiple inheritance [28,36,92,97]. For mixins, there are also problems of linear composition, dispersal of glue code, and fragile inheritance [93]. Linear composition in mixins means that we may find a situation in which the total order of mixins does not exist. Dispersal of glue code indicates that for conflict resolution, sometimes developers need to modify the mixins, introduce new mixins, or use the same mixin twice. The process (or code) needed to resolve these issues adds complexity.

Model-driven technologies are making inroads into the development community, albeit slowly. Modeling abstractions such as state machines and associations provide new opportunities for reuse, and can be combined with inheritance for even greater reusability. However, the issues with inheritance described earlier apply also when these new abstractions are inheritable units.

Therefore, the problem we address is how such modeling elements can be reused in a more flexible manner than it is possible through inheritance. In other words, we investigate how to improve reusability at the modeling level for both low- and high-abstraction modeling elements including methods, associations, and state machines.

1.1.1 Problem Scenarios Focusing on State Machines

The following are several scenarios in which maximum reuse of modeling elements, in this case state machines, is not possible with inheritance. These scenarios are described symbolically to simplify understanding (Figure 1). The research presented in this thesis addresses each of these scenarios.

- **Scenario 1:** Class *A* has a state machine called *aSM1* and class *B* requires to reuse or extend state machine *aSM1*. The classes do not have superclasses (Figure 1.scenario 1).

- **Scenario 2:** Class *A* has a state machine called *aSM1* and class *B* requires to reuse or extend state machine *aSM1*. However, class *B* already has a superclass called *C* (Figure 1.scenario 2).
- **Scenario 3:** Class *A* has two state machines called *aSM1* and *aSM2*. Class *B* requires to reuse or extend just state machine *aSM1*. Class *B* might have a superclass or not (Figure 1.scenario 3).
- **Scenario 4:** Class *A* has two state machines called *aSM1* and *aSM2*. Class *B* has a state machine called *bSM1* with two states *s1* and *s2*. Class *B* wants to reuse state machine *aSM1* as the internal behavior of state *s2*. In fact, class *B* wants to make state *s2* a composite state (Figure 1.scenario 4).
- **Scenario 5:** Class *A* has a state machine called *aSM1* and class *B* wants to reuse state machine *aSM1* with different state and event names. This is mostly the case when domains of classes *A* and *B* are different. For example in Figure 1.scenario5, class *B* wants to change state names *S1* and *S2* to *P1* and *P2*; and also event *e1* to *t1*.

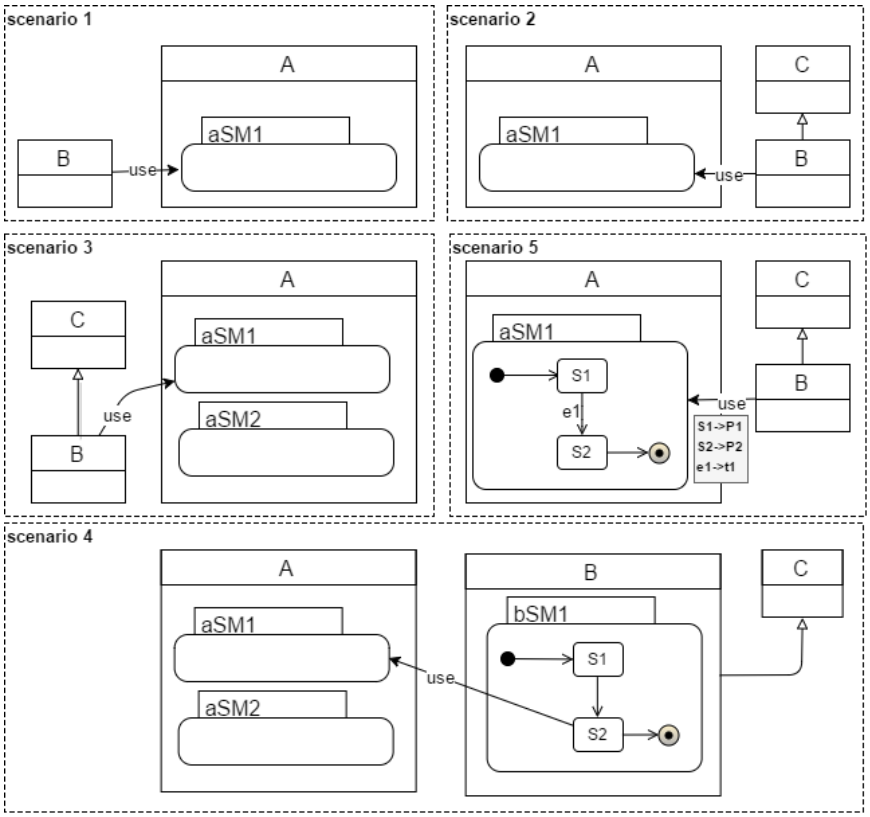


Figure 1. Different reuse scenarios for state machines

1.1.2 A Concrete Example

An instance of the scenarios defined above can be found in the alarm clock described in Objectcharts [30], an incremental way to extend state machines. Figure 2, reproduced from that paper, depicts two state machines *mainSM* and *timeSM* for the class *AlarmClock*. The state machine *mainSM* has two states *alarmOn* and *alarmOff* with two events *set* and *cancel*. The state *alarmOn* is a composite state with its own nested state machine, including states *quiet* and *ringing*. The state machine *timeSM* has one cyclic state called *timeUpdate.*

Scenarios 1 and 2 would occur in the situation where a designer wants to create an advanced alarm clock that has features like snoozing. Scenario 2 would apply if the advanced alarm clock already has a superclass. Scenario 3 could apply in the case of reusing state machine *timeSM* as it happens to be useful in other classes that have time functionality. Scenario 4 could apply in a situation where the nested state machine inside state *alamOn* comes from a state machine related to a watch.

Scenario 5 can be illustrated by considering the protocol between two states *alarmOn* and *alarmOff*. This protocol can easily be mapped to any electronic device if we just change the names of states and events (*alarmOn=on*; *alarmOff=off*; *set=turnOn*; *cancel=turnOff*). Taken together, these reuse scenarios enable state machines to be easily built from already-existing state machines.

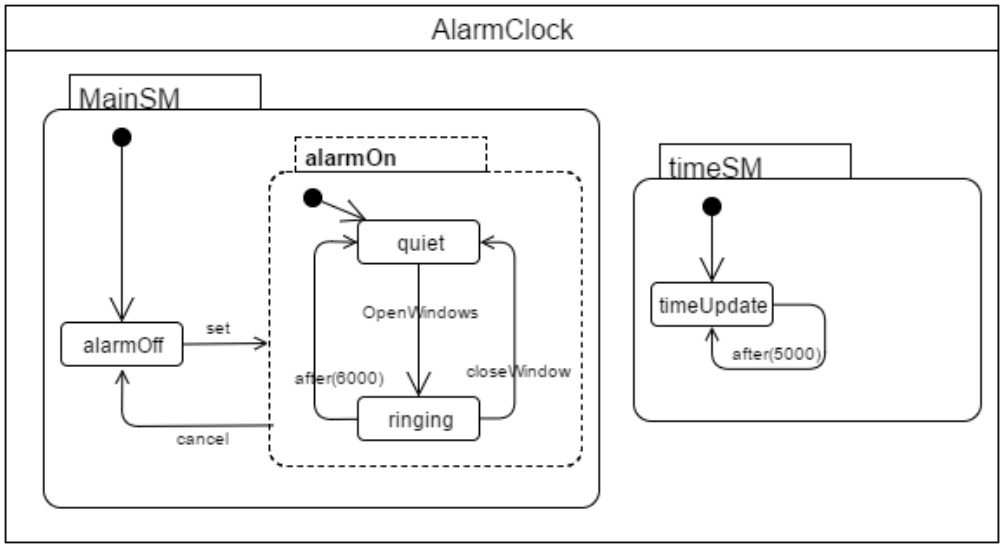


Figure 2. State machine related to the Alarm Clock, Coleman et al. [30]

1.1.3 Existing Partial Solutions

The current dominant solution for the scenarios above is based on inheritance. Using inheritance can easily satisfy scenario 1 in Figure 1 because it is a direct use of a state machine. Scenario 2 is not straightforward because class *B* already has a superclass. This can be resolved by having a root class called *R* containing the state machine *aSM1*. Then, class *A* and class *C* would extend *R*. However, this may not be possible because it might break the hierarchy of classes. Even if it is possible, this pollutes all other classes in the hierarchy that do not need the state machine *aSM1*, in this case, class *C*. Another option is to use associations. In this case, an association is created between class *A* and class *B*. Then, wrappers and delegation are implemented to achieve the behavior needed. This requires special implementation for every case, increases coupling and pollutes the implementation of class *A* with additional methods. Scenarios 3, 4, and 5 are impossible to achieve because inheritance just allows reusing an element as is.

Another approach is to use aspects [56], in which state machines are elements of the advice models. Aspects then need to specify to which points or elements these state machines must be applied. The main issue with such aspects is that classes that want to reuse state machines do not specify this need themselves; instead, such need is decided *for* them by the aspects. Therefore, the classes lose their authority to specify which elements they need to reuse. We will further discuss the use of aspects in related work, Chapter 4.

1.2. Overview of the Solution

In order to tackle aforementioned problems, we use the concept of traits. Traits were first introduced in dynamically-typed class-based languages by Schärli et al. [93]. A trait, in its original form, is a group of pure methods that serves as a building block for classes. It is a simple but powerful unit of code reuse that we will discuss in detail in Section 2.2. Traits can be used to structure object-oriented programs in a compositional manner.

Knowing the reuse capability of traits, we bring traits to the modeling level so as to be able to obtain such reuse flexibility for developing models. Model-based traits extend classic traits, so can contain methods, but also can have modeling elements such as state machines and associations. They are enriched with operators and a composition mechanism so

they can be integrated better with model-driven development. Our extensions to traits allow us to satisfy all reuse scenarios explained in Section 1.1.1.

Our solution also considers the fact that those scenarios should be implementable in major object-oriented programming languages. Therefore, it offers a mechanism that allows implementing model-based traits in such programming languages.

1.3. Contributions

In this thesis, we show that we can tackle limitations of reusability based on inheritance at the modeling level for object-oriented systems by adopting a new approach based on traits. We use this approach to define modeling elements in traits and reuse those elements with more flexibility than they could be reused by the concept of inheritance. We also show that the systems developed with our approach can be implemented with object-oriented programming languages.

The contributions of this thesis are as follows:

- Bringing the concept of traits to the modeling level and extending it with modeling concepts so traits can be used in model-driven development.
- A mechanism and operators to reuse, extend, and compose state machines.
- Extending traditional traits with required interfaces so dependencies at the semantics level become part of their usage, rather than simple syntactic capture.
- A model transformation to enable model-level traits to be implementable in object-oriented programming languages.
- A concrete implementation of all of the above in Umple [12,114].
- Two cases studies that show how modeling elements can be reused using traits.
- Reengineering two systems to use model-level traits, showing how our approach can be utilized to develop significant software systems.

1.4. Validation

We have validated our approach by implementing two systems and by reengineering two other systems. The results are provided in Chapter 5.

Our results confirm that traits can be used at the modeling level to encapsulate modeling elements such as methods, associations, and state machines. Moreover, traits can be applied through code generation automatically in multiple object-oriented programming languages.

Furthermore, our validation shows that modeling elements defined in traits have more opportunity for reuse in comparison with specifying them in classes and reusing by way of inheritance. This is explained in Section 1.1.3.

Our validation also shows that having traits at the modeling level enables modeling and implementation of real systems, demonstrating that our work is practical. We also show that a system modeled with traits has the same behavior as an equivalent system without traits.

Finally, we used test-driven development to further validate our implementation. The tests cases have been designed to make sure that what is defined as facts (rules) for traits at the modeling level are implemented correctly, and in particular that the resulting generated code behaves as expected.

1.5. Choice of the Tool (Umple)

Our work does not depend on any specific programming language, but we require a concrete platform to explore the notion of model-level traits and show the feasibility of the features and extensions. We have chosen Umple [12,114], a textual modeling language, as this platform. Umple is an open source modeling tool and programming technology that enables what we call Model-Oriented Programming. The main relevant features of Umple related to this work are as follows:

- It is a modeling language. This is important because our work extends traits to be used in model-driven development processes.
- It offers a textual syntax for modeling. This simplifies the way modeling concepts can be extended.
- It allows writing executable parts of models in any programming language supported by Umple. Therefore, such capabilities can be inherited by traits.

- It is enriched by code generators for several programming languages (currently Java, PHP, and C++ are the most supported ones), so traits and our extensions can be transformed to those languages.
- It is an open source project, so our work is immediately available to all developers and researchers.

In this work, we introduce traits into Umple, however, being able to graphically represent traits along with their elements and operators is also key for model-driven engineering. Although, we adopted a simple graphical representation for traits (described in Section 2.2.10) based on the previous research conducted by Schärli et al. [93] and extended it with our notation, we believe more studies are needed in this direction to represent traits as modeling elements in graphical modeling languages. Our suggested agile approach for graphical languages such as UML to obtain benefits from traits could be to implement the concepts of traits and their operators under a UML profile. We consider this as future work.

It should be point out that we focus on modeling languages (or frameworks) which are compatible with the concepts of object-orientation and UML. The philosophy of traits can be used in other modeling frameworks; however, new sets of studies are required to investigate which elements of each modeling language can be represented in traits, what kinds of conflicts might arise, and what operators would be required. These kinds of challenges are not explored in this work, and are left as future work.

1.6. Limitations of the Work

In model-driven software development, a system can be modeled from different perspectives. These perspectives are expressed through diagrams (or modeling elements) such as class diagrams, state machines, and activity diagrams. In this work, we extend traits to have associations, state machines, and attributes. Traits have the potential to contain other modelling elements too, but they are not covered in this work.

This work does not propose a formal graphical representation for concepts added to traits. We do present some of our examples using a graphical representation for basic features of traits, but we leave validation of the usability of this notation to future work.

The model transformations designed and implemented as part of this work are completely tested at the modeling level. This means they can be used to construct a valid system in any programming language generated by Umple. However, we only tested the generated code in the Java language. More work would have to be done to validate the generated code for other languages.

Finally, all our concepts, extensions, and algorithms have been implemented in Umple and have been subjected to many test cases. However, we have not applied formal methods to our work. We propose to do this as a future work because it would require formalization of all of Umple, and this is just currently being started by others.

1.7. Publications

The following scientific papers have been published or submitted by the author and are directly related to this thesis. During the time of writing this thesis, the author also published six additional papers that were not directly related to the thesis, and hence are not listed here. The first author is the main author.

1.7.1 Finalized Publications

- **Vahdat Abdelzad**, Timothy C. Lethbridge, Promoting Traits into Model-Driven Development, Software & Systems Modeling, pp. 1-21, 2015. [2]
- **Vahdat Abdelzad**, Extended Traits for Model-Driven Software Development, Doctoral Symposium at MoDELS, CEUR Workshop Proceedings 1531, 2015. [1]
- Timothy Lethbridge, **Vahdat Abdelzad**, Mahmoud Husseini Orabi, Ahmed Husseini Orabi, Opeyemi Adesina, Merging Modeling and Programming using Umple, International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Springer LNCS 9953, pp. 187-197, 2016. [63]

1.7.2 Submitted Publications

- **Vahdat Abdelzad**, Timothy C. Lethbridge, Increasing the Flexibility and Reusability of State Machines using Traits and Composition, submitted to IEEE Transaction on Software Engineering, 2017.

1.8. Thesis Outline

The rest of this thesis is organized as follows:

In Chapter 2, we introduce preliminary concepts and backgrounds related to the thesis including the required syntax and semantics of Umple. This chapter also encompasses the basic definition needed to understand traits. This is required to be able to understand better Chapter 3. People who are familiar with Umple, as well as traits in any programming languages, may skip this chapter.

In Chapter 3, we describe traits in model-driven software development. We first describe requirements of such traits and then explain how those traits should be processed in modeling languages. Then, we concentrate on semantics of our flattening algorithm that allow implementing model-based traits. This semantics is independent of actual implementation of such an algorithm in a specific modeling language. Furthermore, we present syntax and semantics of model-based traits implemented in Umple, including traits that contain state machines, associations, and required interfaces, as well as operators to deal with conflicts. We explain how model transformations are performed in Umple so as to enable traits to be accessible in major object-oriented programming languages. The chapter concludes by explaining the traits metamodel.

In Chapter 4, we present related work regarding traits and the way state machines are reused, extended, and composed through traits. The chapter also discusses in which areas traits could be useful.

In Chapter 5, three different scenarios are presented in order to demonstrate the usefulness and practicality of the work. The first one, a geometric system, represents how basic features of traits, implemented in Umple, can be used to build a system. The next two case studies show how traits in Umple can be useful if they are applied to already-developed systems in Umple: the Umple compiler and an Umple translation of JHotDraw. We demonstrate the usefulness of the work by way of an industrial case study in which we model a framework for service-oriented systems. Furthermore, we explain how we used test-driven approach at the modeling level to make sure our extensions are valid.

In Chapter 6, we discuss reusability aspects of our work, challenges we have faced during this research, and challenges might be faced with users of our work.

In Chapter 7, we conclude our work and present future work.

Chapter 2. Background

In this chapter, we describe the required concepts needed to understand the rest of the thesis. First we give a brief overview of Umple, in which our approach has been implemented. Then, we outline the pre-existing work on basic traits, the research we are extending.

We utilize domain-specific and symbolic examples whenever it is required to describe better the meaning of concepts. Symbolic examples are preferred to domain-specific examples because the meaning can be inferred without the need to domain knowledge.

2.1. Umple

Umple is an open-source software development technology designed to fully merge modeling and programming [115]. An Umple system looks and feels like code when a programming perspective is preferred but looks and feels like a model when it is explored in a modeling (e.g., graphical) environment [63]. Indeed, people who already know programming languages such as Java and C++ can add modeling concepts to their programs as they proceed, while people with a modeling perspective can model their systems with Umple and incrementally add execution elements (such as methods) into the model according to the languages they prefer. This key capability in Umple is called *model-code duality*. This capability is preserved in our work with traits as well.

Umple also supports *text-diagram* duality, which allows developing systems using both a textual and a graphical syntax. However, Umple concentrates more on textual syntaxes in favor of ease of editing, and better model management through version control systems such as Git [52]. It supports major programming languages and generates metrics for various forms of analysis, formal methods' code (e.g., Alloy [5] and SMV [4]), and SQL database code in addition to other modeling syntax such as XMI. Umple systems can be manipulated by the Eclipse IDE, command line technologies, and a web-based environment [116].

Most modeling concepts in Umple have been adapted from UML [117], however, it has its own concepts which are not available in UML. Umple's design philosophy emphasiz-

es simplicity so it does not incorporate everything available in UML. Key model types include classes, associations, state machines, constraints, patterns, aspect-orientation, and tracing [7]. An extensive set of examples of Umple can be found online, including in UmpleOnline [116], and in the Umple GitHub site [118]. In the following sections, we describe syntax and semantics of the main elements of Umple utilized in this thesis.

Listing 1. A simple Umple example

```
1  class Line {
2    isA GeometricObject;
3    Point pA;
4    Point pB;
5  }
6  class Cube{
7    isA Polyhedra;
8    Double edgeLength;
9    Double volume() { /*implementation */ }
10 }
11 class Canvas{
12   0..1 -> * GeometricObject objects;
13   internal Boolean selected = false;
14   status{
15     idle{
16       draw [selected]-> drawing;
17       close -> end;
18     }
19     drawing{
20       done -> /{ saveTemporary(); } idle;
21       entry /{ changeCursor(); }
22       do { draw(); }
23       exit /{ changeCursor(); }
24     }
25     final end{}
26   }
27 }
```

2.1.1 Classes

A class in Umple defines an object-oriented class available for use as a type. Classes are defined by the keyword *class*. The name of class comes after the keyword followed by a pair of curly brackets which encompasses all properties (content) related to the class (it is also called the body of class). A class can also be abstract and this is achieved through the keyword *abstract*. This keyword appears in the body of a class followed by a semicolon. The graphical syntax for a class is that of UML.

Listing 1 shows a simplified Umple model of a graphical system described in more detail as one of our case studies in Section 5.1. As seen, there are three classes called *Line*, *Cube*, and *Canvas* defined in lines 1, 6, and 11 respectively. Figure 3 depicts the UML class diagram related to the Umple model in Listing 1.

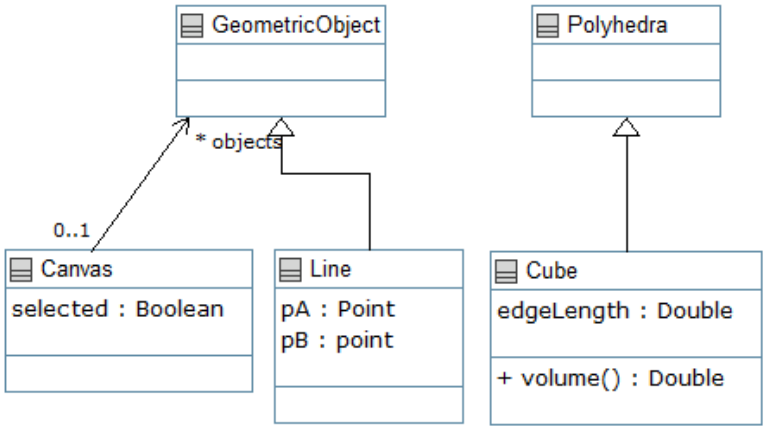


Figure 3. The visual representation of Umple code in Listing 1

2.1.1.1 Generalization

In order to have generalization (also called inheritance) among two classes, the keyword *isA* is used followed by the name of the superclass and ended by a semicolon. Umple supports only single inheritance, although the concept of traits described later in this thesis allows developers to overcome this limitation. In Listing 1, for example, class *Line* is a subclass of class *GeometricObject* defined in line 2. Class *Cube* is a subclass of class *Polyhedra* defined in Line 7. It is also worth pointing out that the keyword *isA* is also used for implementing interfaces and using traits, which are described as we progress.

2.1.1.2 Primitive types

Umple supports some built-in primitive types that can be used in models. It is recommended to use these primitive types instead of platform-specific types in an Umple model because the Umple compiler can recognize them and transform them into proper primitive types in target languages. These primitive types are Integer, Float, String, Double, Boolean, Date, and Time.

2.1.1.3 Attributes

Attributes are a representation of data held by classes. Each attribute has a type and can be defaulted to a certain value. If there is no type defined for an attribute, the type *String* is used by default.

A value can be given to an attribute upon initialization through the operator = followed by a value and a semicolon. Attributes can have stereotypes (or modifiers) such as *internal* and *autounique* that make them more specified. The comprehensive list of these stereotypes can be found in attribute section of the Umple user manual [114]. If an attribute does not have an initialization value, one parameter will be added to the constructor of the class in order to require developers to initialize the value.

By default, Umple generates accessor (get-) and mutator (set-) methods for each attribute; certain stereotypes override this. As seen in Listing 1, class *Line* has two attributes called *pA* and *pB* whose types are *Point* (lines 3 and 4). Class *Cube* has a *Double* attribute (line 8) called *edgeLength* and class *Canvas* has a *Boolean* attribute called *selected* (line 13). The latter attribute has a stereotype named *internal* and a default value which is *false*. When these classes are transformed to target language code, classes *Line*, *Cube*, and *Canvas* will have the accessors and mutators depicted in Table 1.

Table 1. Accessors and mutators related to the Umple model in Listing 1

(N/A: Not Applicable)

Class	Attribute	Accessor	Mutator
Line	pA	getPA	setPA
Line	pB	getPB	setPB
Cube	edgeLength	getEdgeLength	setEdgeLength
Canvas	selected	N/A	N/A

Umple allows modelers to define attributes (or other elements) based on the programming languages they are familiar with. This is possible with Umple because Umple supports incorporating programming languages' structures along with modeling elements. Our recommendation is to use the Umple syntax to make the model more reusable.

2.1.1.4 Methods

Umple allows modelers to extend the functionality of a class with user-defined methods. Such methods are defined similarly to the way methods are defined in Java. Methods' semantics and properties follow the UML conventions for methods.

A standard Umple user-defined method will specify the return type, the name, the argument list, and then the method body in curly brackets. The method body is not parsed by Umple, but is output just as the user has written it. It is possible for the user to specify different bodies, one for each language that can be generated, by preceding the curly brackets by the name of the language, such as 'Java', or 'Cpp'.

The default visibility for Umple user-defined methods is *public*, but it can be changed explicitly. The generated output for the method will use the correct format for specifying the return type, signature, and arguments as required by the language being generated.

Inside methods, developers may call other user-defined methods or any of the API methods generated by Umple that access the Umple attributes, associations, and state machines (described as we progress). To determine what API methods are available to be called by methods, refer to the API reference [48] or generate Javadoc from an Umple file.

As an example of a user-defined method, Listing 1 shows a method called *volume()* for the class *Line* in Line 9. The method *volume()* has a *Double* return type with no parameter. Its visibility is also public, by default. The method has a body (implementation) replaced by a comment in this example for brevity.

Abstract methods in Umple are defined by the keyword *abstract* at the beginning. The structure of an abstract method is similar to a method without a body. In fact, instead of curly brackets and its content, a semicolon is used. For example, the abstract form of the method *volume()* in Listing 1 would be as follows:

```
abstract Double volume();
```

2.1.1.5 Code blocks

Umple allows different programming languages to be used as its action language. By action language, we mean the blocks of code that are present in user-defined methods as well as in other constructs such as computed attributes, state machine actions, and state machine activities. We will discuss the latter items later. Such code blocks are always indicated by target-

language code in curly brackets, optionally preceded by the target language name (e.g., Java or Cpp). For example, if a class has a method, its implementation can be written by any programming language supported by Umple. The target language to be generated by the compiler is specified directly using a 'generate' keyword in the Umple code, or as an argument when invoking the Umple compiler. This language will be used to select which code blocks are output. By default, Umple generates Java as its target.

All code blocks that are not tagged with a target language are assumed to be written in the target language that is currently being generated. This simplifies the design of systems that are always intended to be generated in one target language (e.g., an all-Java system), as the 'Java' keyword then does not need to be specified before every set of curly brackets. Details of all supported target languages can be found in the grammar part of the Umple user manual [51]. As an example, the C++ implementation of the method *volume()* in Listing 1 would be as follows:

```
Double volume() Cpp{/*implementation */}
```

2.1.2 Interfaces

An interface in Umple defines a list of methods that are implemented in one or more classes. An interface is defined by the keyword *interface* followed by a unique name and a pair of curly brackets. The signatures of all methods of interfaces must be defined inside curly brackets. A class implements an interface through the keyword *isA*, just like a class extends another class. A class can implement more than one interface. Moreover, an interface can extend another interface as well. This is also achieved through the keyword *isA*.

Listing 2. A simple interface example in Umple

```
1  interface I1 {
2    void method1();
3  }
4  interface I2{
5    isA I1;
6    void method2();
7  }
8  class C{
9    isA I2;
10   void method1() { /*implementation */ }
11 }
```

Listing 2 shows an example in which two interfaces *I1* and *I2* are defined in lines 1 and 4. Interface *I1* has an abstract method *method1()* (line 2). Interface *I2* extends interfaces *I1* (line 5) and has its own abstract method *method2()* (line 6). Class *C* declares that it implements interface *I2* in line 9 and also implements one of its abstract method *method1()*. However, class *C* does not have an implementation for the abstract method *method2()*. If such a case happens, the Umple compiler detects and automatically implements the method. The implementation has an empty code body and it is there to make sure the final system will be able to be built. Umple developers use this technique to develop a prototype faster.

2.1.3 Umple Mixins

Mixins in Umple are designed to reduce the complexity of large systems and add flexibility to design. Umple constructs such as classes can be split into several parts, each typically in a different file. The multiple definitions are combined to create a complete definition. For example, through mixins, we can define a class with one set of attributes in one file, and a different set of attributes in a second file. When the Umple model is built, these classes are mixed to gather and compose the final class. This allows separate features to be separately developed. Indeed, in some cases, only one of the features may be needed; in that case one of the files can be omitted. Another use of mixins is to keep the user-defined methods of a class in a separate file from the 'pure' modeling elements of that same class, such as the attributes and associations.

An example of mixins is in Listing 3. This is a symbolic example in which there are two classes called *C* with their own unique attributes and methods. These classes are inside two files called *model-file1.ump* and *model-file2.ump* (part a). These can be combined using the Umple expression, described below, that would be present in a third Umple file.

```
use model-file1.ump; use model-file2.ump;
```

Once the final model is built, the result will be equal to what would have been achieved if the model in Listing 3.b had instead been present. It is worth noting that the semantics of Umple mixins differs somewhat from the general semantics of mixins defined in the literature [28]. Specifically, while in Umple multiple definitions of a class with the same name are merged and any elements of the class can participate in this merging (attributes,

methods, and so on), in other languages (such as Ruby), mixins are classes with different names whose methods are merged into multiple client classes.

Listing 3. A symbolic mixin example in Umple

a

```
1  // model-file1.ump
2  class C {
3    attribute1;
4    attribute2;
5    void method1 ();
6    void method2()
7  }
8
9  // model-file2.ump
10 class C {
11   attribute3;
12   attribute4;
13   void method3(){/*impl… */}
14   void method4(){/*impl… */}
15 }
```

b

```
// model-file.ump
class C {
  attribute1;
  attribute2;
  attribute3;
  attribute4;
  void method1(){/*impl… */}
  void method2(){/*impl… */}
  void method3(){/*impl… */}
  void method4(){/*impl… */}
}
```

2.1.4 Associations

Umple supports directional and bidirectional associations, as well as compositions, which are associations with additional constraints. It uses the symbols -- for bidirectional, -> for directional, and <@>- for composition associations. For example, line 12 of Listing 1 shows a direct zero or one to many association between the class Canvas and GeometricObject.

Multiplicities appear before the name of classes and as in UML can be zero or one (0..1), one (1), zero or many (*), one or many (1..*), or have specific upper and lower bounds. Role names are optional and appear after the name of a class. If there is no specified role name, Umple automatically generates suitable names based on the multiplicity and names of involved classes. The association defined in line 12 has a defined right role name called *objects*, while the left side role name is automatically determined to be *canvas*.

In the same manner that Umple creates accessors and mutators for attributes, it creates useful APIs for associations. The number of API methods and their signatures depend on the type of each association, and particularly the values for the multiplicities at both ends of the association. Furthermore, based on the type of association, Umple might add specific pa-

rameters to the constructor of classes. For example, Umple generates the Java APIs described in Table 2 for the association defined for the class *Canvas*. The signatures of APIs are self-explanatory.

Table 2. Generated APIs for directional associations

`public GeometricObject getObject(int index)`
`public List<GeometricObject> getObjects()`
`public int numberOfObjects()`
`public boolean hasObjects()`
`public int indexOfObject(GeometricObject aObject)`
`public static int minimumNumberOfObjects()`
`public boolean addObject(GeometricObject aObject)`
`public boolean removeObject(GeometricObject aObject)`
`public boolean addObjectAt(GeometricObject aObject, int index)`
`public boolean addOrMoveObjectAt(GeometricObject aObject, int index)`

2.1.5 State Machines

State machines are widely used to model discrete event-driven behaviors of systems, or parts of systems [47,95,115]. For example, they can model the behavior of object-oriented classes and have been used to develop real-time, desktop, mobile, and web-based systems. They have also been supported with a range of verification and testing techniques [39,73].

Umple supports UML state machine semantics. However, it does not support non-deterministic state machines. State machines control the behavior of instances of classes; it is a key way that UML envisions using state machines [117]. There is considerable literature focusing on such state machines and also on code generation from such machines [30,34,79,95,96] to avoid the need for manual coding of their logic.

A state machine in Umple is defined by a unique name inside a class followed by a pair of curly brackets. All elements related to the state machine must be defined inside the brackets. For example, there is a state machine called *status* in Listing 1 for class *Canvas,* defined in lines 14-26.

In Umple, a class can have several state machines. Transitions of a state machine cannot have a destination state in another state machine. However, since events assigned to

these state machines belongs to a class, multiple state machines can communicate with each other indirectly through calling those events in actions. Umple supports code generation for state machines in Java and C++. At the current time state machine code is not generated in other target languages such as PHP and Ruby. In the following sections, we describe state machines' elements and their syntax.

2.1.5.1 Simple states

A simple state is defined by a unique name followed by a pair of curly brackets inside state machines. Simple states can have *do* activities, *exit* actions, and *entry* actions; these are all described in the coming sections. A simple state can be the source or destination of transitions. However, they cannot encompass other states. For example, there are two simple states belonging to state machine *status* in Listing 1 named *idle* and *drawing*. They are defined in lines 15 and 19 and have their own content expressed in their curly brackets.

2.1.5.2 Transitions

A transition is shown as a directed connection between a source and a destination state. A transition may include an event specification, a guard, and actions. An event specification is a method name (possibly followed by arguments). An event occurs when there is a call to a method whose signature matches the event specification. If the state machine is in the source state and the event occurs, then the state machines changes to be in the destination state; the transition is said to be *triggered* or *fired*.

The triggering of a transition can be prevented by the presence of a *guard*. A guard is a Boolean expression in square brackets; it is evaluated when the event occurs. If the guard evaluates to false, then the transition is not triggered.

Transitions without event specifications are called *auto* transitions. In the normal case, when a state machine enters the source state of an auto transition, the state will be immediately changed to the destination state. In other words, the transition is triggered as soon as the state machine enters the source state. This behaviour can be blocked by a guard as described above. In other words, checking whether a guard is true or false happens before triggering an auto transition. Moreover, a slightly different behaviour occurs if there is a do activity, as discussed later.

Once a transition is triggered (normal or auto transition), before changing the current state of the state machine to the destination state of the transition, any action of the transition is executed.

Transitions are defined through the symbol -> in states. The source of a transition is defined implicitly because a transition is defined in a state. Therefore, the source state of a transition is the state in which the transition is defined. The destination state, which is mandatory, should appear after the symbol -> followed by a semicolon. The event specification of a transition is defined on the left side of the symbol; this is followed optionally by a pair of square brackets that encompass a guard expression. The action of a transition is defined before the name of the destination state through the symbol slash (/) followed by a pair of curly brackets enclosing a code block. There may be more than one code block, each for a different target language, as described in Section 2.1.1.5.

For instance, line 16 of Listing 1 shows a transition leaving the state *idle* and transitioning to the state *drawing* with the event *draw* and a guard which has a condition on a *Boolean* attribute called *selected*; while line 20 shows a transition with an action, which calls the method *saveTemporary()*;

2.1.5.3 Events

Events specifications have the structure used for defining methods in classes, without the body. They can have name and parameters, but their return type is always *Boolean*. The *Boolean* return value can be used in target language code to determine whether or not a transition was successfully triggered. If an event method is called, and there is no matching event specification in any current state, or transition guards block such transitions, then the event method returns false.

If an event does not have parameters, the parentheses can be removed to make the syntax simpler. For example, the event used for the transition defined in line 20 in Listing 1 has the name *done* without any parameter. If the event had an integer parameter, the transition would be defined as follows:

```
done(Integer i) -> /{ saveTemporary(); } idle;
```

2.1.5.4 Guards

As outlined earlier, guards are Boolean expressions used to apply restrictions on the way transitions are triggered. They are surrounded by a pair of square brackets; this is consistent with other uses of Boolean expressions in Umple, such as class preconditions or invariants.

Within guards, modelers can refer to attributes, mutators, accessors, and other API methods such as those generated for associations. Umple supports major arithmetic and Boolean operators such as “and”, “or”, and “<” in guards. Furthermore, modelers can write their Boolean expression in any target language they prefer, however, in that case they will not be able to benefit from model analysis. If this case is chosen (i.e., Umple’s attempt to parse the guard is not successful), Umple will transfer such guard code without modification to the target language when the code is generated. For example, line 16 in Listing 1 shows a guard whose Boolean expression is just a Boolean attribute called *selected*. Umple is able to parse this, so it can be analysed.

2.1.5.5 Transition actions

Transition actions are executable logic executed when a transition is triggered. They are executed before a state machine changes its current state. As pointed out in Section 2.1.5.2, such actions are defined by the symbol slash (/) followed by a pair of curly brackets. The execution part of actions is written inside curly brackets and includes setting attributes or calling methods. The execution part must have negligible logical execution time. It is the responsibility of a developer to write code in actions that executes essentially instantaneously – in other words code that does not loop or have the potential to be blocked in some way.

An example of a transition action is given in line 20 of Listing 1. It shows an action calling method *saveTemporary()*. Modelers can use target-language specific execution code in actions. In that case, as we discussed in Section 2.1.1.5 on Code Blocks earlier, they need to specify the name of target language before the curly bracket.

2.1.5.6 Entry actions

Entry actions of a state are executed immediately when the state becomes the current state of its state machine. In other words, when a state is entered by a transition. When such a transition is triggered, the entry action of the destination state is executed immediately upon entry

into the new state (after any transition action). Entry actions have a similar appearance to transition actions, except that they are preceded by the keyword *entry*. As with transition actions, they are followed by the symbol slash (/) and a pair of curly brackets containing code that must run in negligible time.

For instance, line 21 in Listing 1 depicts an entry action for the state *drawing*. It calls method *changeCursor*() as it execution command. Umple allows having more than one entry action in states. Each entry action is defined independently and they are executed in the sequence they appear in the Umple model.

2.1.5.7 Exit actions

Exit actions of a state are executed immediately when the state is no longer the current state of its state machine. In other words, when a state exits due to triggering of a transition. When a transition is triggered, first the exit action is executed and then any transition action is executed. Exit actions are defined just like entry actions, except using *exit*.

For example, line 23 in Listing 1 depicts an exit action for the state *drawing*. It calls method *changeCursor*(). Umple also allows having more than one exit action for states. Each exit action is defined independently and they are executed in the sequence they appear in the Umple model.

Later on we will discuss nested states; it is worth noting that when exiting several levels of nesting, the exit actions are called sequentially starting with the innermost (most nested) state being exited. Any transition action then occurs. After this any entry actions occur starting with the outermost (least nested) state being entered.

2.1.5.8 Do activities

When longer-running or potentially blockable code is to be executed while in a state, a do activity is used. Such code cannot be executed in actions because they do not have negligible logical execution time. The activity defined in the do activity of a state is run in a separate thread that can continue to execute as long as the state machine remains in that state. Any do activity is initiated after entry actions are completed. A do activity might run to completion of its own accord, but if it is still running when an exit transition is triggered, then its thread will be terminated; this occurs before any exit action is run.

In the case of a state with an auto-transition and a do-activity, the auto transition is only triggered when the do-activity runs to completion. Handling do activities in threads allows the state machine to 'stay live' and be able to respond to other events, even while the do activity is running.

Do activities are defined by the keyword *do* followed a pair of curly brackets containing the code to execute. For instance, line 22 in Listing 1 shows a do activity that runs the method *draw()*. Like entry and exit actions, Umple allows having more than one do activity. All of them are defined separately and are started in their own threads in the sequence they appear in the Umple model.

2.1.5.9 Initial state

Every state machine needs to have an initial state to make sure execution is started from the right state. Initial states can be used to set the value of some attributes or trigger some actions required for the state machines. In Umple, initial states are defined implicitly. In fact, the first state that is defined inside a state machine is considered as the initial state. For example, the initial state of the state machine *status* in Listing 1 is *idle* (line 15).

2.1.5.10 Final states

Final states are states upon whose entry the state machine must be terminated. More specifically it means the instance of the class including the state machine must be deleted. Final states are defined using the keyword final within the definition of such states. Final states cannot have do activities or exit actions. However, they can have entry actions. They are allowed to be destinations of any transition, but they cannot be the source of any transition. A state machine can have more than one final state. For example, the final state for the state machine *status* in Listing 1 is *end* (line 25).

2.1.5.11 Composite states

Composite states are different from simple states because they can have other states inside. In fact, they allow an unlimited hierarchy of states. A composite state must have at least one simple or composite state inside it. Composite states can have, like simple states, their own entry and exit actions, transitions, and do activities.

Listing 4. A state machine for a television with a composition state

```
1  class Television {
2    mode{
3      on{
4        turnOff -> off;
5        stop{
6          play -> playing;
7        }
8        playing{
9          stop -> stop;
10         pause -> paused;
11       }
12       paused{
13         stop -> stop;
14         play -> playing;
15       }
16     }
17     off{
18       turnOn -> on;
19       shutDown -> end;
20     }
21     final end{}
22   }
23 }
```

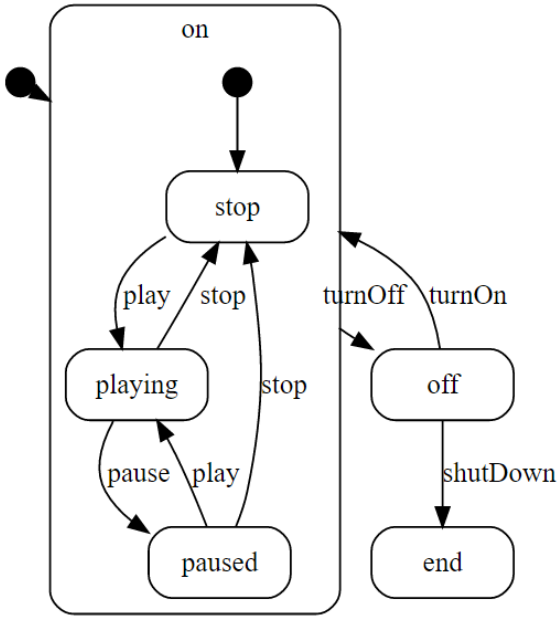


Figure 4. The state machine diagram for the Umple model in Listing 4

States nested within composite states are defined in the same manner as they are defined within a state machine. Every composite state has an initial state that is the first state defined in it, and it can have its own final states. Every time a composite state is entered, it will by default be in its initial state. Transitions can also have source and destination states that are inside the composite state or outside it.

For example, Listing 4 shows a state machine named *mode* for the class *Television*. It has a composite state called *on*, a simple state called off, and a final state called *end*. The initial state of this state machine is the composite state *on.* Composite state *on* has three simple states *stop*, *playing*, and *paused*. The initial state of the composite state *on* is the simple state *stop*. The composite state *on* also has a transition with the even name *turnOff*. When this event happens, the current state will be changed to the simple state *off* without any consideration of which inner state the composite state is in. Figure 4 depicts how a composite state defined in Listing 4 is drawn graphically in Umple.

2.1.5.12 Regions

Regions are a concept allowing parallelism in state machines. The presence of more than one region specifies more than one independent state machine nested inside an outer state machine. In Umple, regions are defined inside states and have unique names. Their names are determined automatically from the name of the first state, which is their initial state. A composite state has at least one region that allows it to have nested states and transitions. Most composite states have a single region.

When a state machine has more than one region, they are said to be orthogonal, or concurrent. In such a case, a state machine has more than one active state at a time (one for each region). These orthogonal regions can be executed in collaboration with each other or independently. It is recommended to keep regions independent so as to reduce the complexity that comes from collaboration.

Orthogonal regions are defined by the symbol || within a composite state. Composite states have by default a single region so the second and other regions are separated by the symbol ||, appearing at the end of the last state related to the previous region. For instance, Listing 5 shows the extended version of the composite state *on* in Listing 4 in which the second region is added to model the behavior of a menu. As seen, the symbol || is added (line

14) to the end of the last state related to the region *stop*, which is state *paused*. The second region is automatically detected after the symbol || and it is named *invisible*. Umple automatically detects regions and names them; they are not named manually. The second region named *invisible* has two simple states called *invisible* and *visible*. Figure 5 depicts the graphical representation for orthogonal states in Listing 5.

Listing 5. The composite state on related to the class Television with parallel regions

```
1     on{
2       turnOff -> off;
3       // the default and first region: stop
4       stop{  play -> playing;   }
5       playing{
6         stop -> stop;
7         pause -> paused;
8       }
9       paused{
10        stop -> stop;
11        play -> playing;
12      }
13      // the end of the region stop
14      ||
15      // the second parallel region: invisible
16      Invisible { showMenu -> visible; }
17      Visible { cancelMenu -> invisible; }
18    }
```

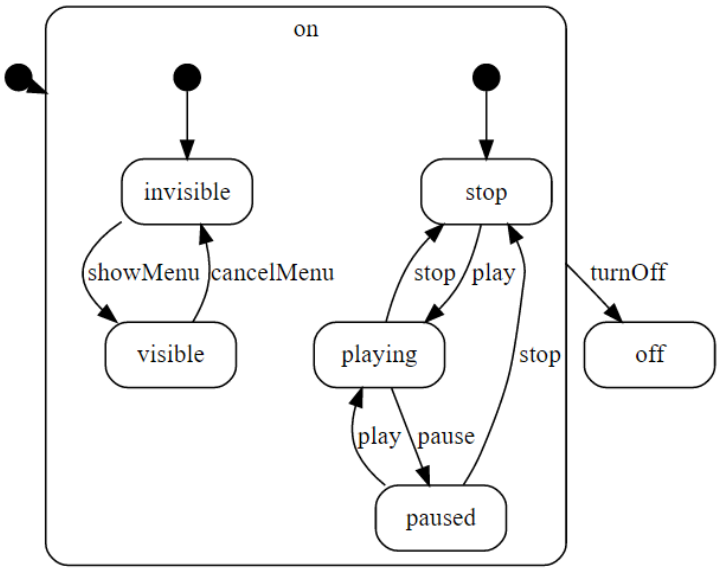


Figure 5. State machine diagram for the Umple model in Listing 5

Although Umple detects the names of regions automatically in order to reduce the amount of effort required by modelers to create an orthogonal state machine and also make models simpler, it forces the initial states of orthogonal regions in a composite state to be unique. Most of the time modelers never need to know the names of regions.

2.1.6 Umple Grammar

Umple uses its own extended Backus–Naur form (EBNF) syntax for the grammar. This was created to allow Umple to deal with code blocks coming from different programming languages. It also has its own parser tool. In fact, the Umple syntax offers a very simple mechanism to define a new language, as well as extend an existing one. We explain here some of the required notations for an understanding of the rest of this thesis. Full details about the Umple grammar can be found in the Umple user manual [49].

2.1.6.1 Terminals

Terminals in the Umple grammar are any characters other than whitespace, +, *, (,) or the contents of matching sets of square brackets. The parser expects to match the characters exactly.

2.1.6.2 Non-terminals

Umple has two types of non-terminals: simple and rule-based, both found within matching sets of square brackets. Simple non-termals can be considered as references to built-in parser rules.

A basic simple non-terminal appears as a name (a sequence of non-whitespace characters) surrounded by *single* square brackets. It indicates that the parser should recognize an identifier, defined as a sequence of alphanumeric characters, with underscores allowed. Several special symbols can precede the name to modify this behaviour; these are discussed in a later sections. Full details can be found in the Umple user manual [49]. A simple non-terminal that can match one of a set of strings is marked with an equals sign, and the symbol | to separate the strings. The following is an example.

```
[=visibility:public|private|protected]
```

A rule-based non-terminal is a name within *double* square brackets, and references a grammar rule. A rule is a sequence of terminals, simple non-terminals, and rule-based non-terminals. A rule can be reused many times in the grammar. For additional clarity, rule-based non-terminals are shown using blue in this document.

Listing 6. A snippet of Umple grammar

```
1  classDefinition: class [name] { [[classContent]]* }
2  classContent- : [[comment]]
```

Listing 6 shows a snippet of Umple grammar for classes. There are two rules, *classDefinition* and *classContent*. There is a simple non-terminal *name* encompassed by square brackets (*[name]*), and two rule-based non-terminals named, *classContent*, and *comment*. The word 'class', as well as the curly brackets { and } are terminals. The * is a special symbol defined below.

The sequence of tokens generated by the parser normally just includes the patterns matched by the simple non-terminals. Umple uses the symbol minus "-" after a rule name to indicate that the rule name should be added to the tokenization string. This appears in line 2 of Listing 6. It allows the analyser to interpret the token stream correctly. Terminals are not added to the tokenization string. For instance, in Listing 6, the terminal *class* is used to direct parsing, but will not appear in the tokenization string. If it is required to tokenize such symbols, a constant can be defined using the notation [=name].

2.1.6.3 Optionality and repetition

The Umple grammar allows repeating or making optional some elements in the grammar. Parentheses can be used to group several elements in the grammar into a single element for the purposes of applying operations such as the following. An asterisk * means that zero or more of the preceding elements may occur while a plus sign + means that one or more of the preceding elements must occur. The optional elements can be defined by the symbol ?, which means that the preceding element may occur. For example, the rule *classContent* in Listing 6 should be repeated zero or more times because of asterisk * at the end (line 1).

Since Umple uses parentheses to surround several elements, they cannot be used as terminals. In order to have them as terminals, special keywords OPEN_ROUND_BRACKET and CLOSE_ROUND_BRACKET must be used.

2.1.6.4 Special matching cases

By default, simple non-terminals match identifiers that can include underscore and certain other symbols. In order to match alphanumeric identifiers only, the symbol tilde ~ is used before the name (e.g., [~name]). The Umple grammar also allows matching based on regular expression. For example, [!bound:\d+] matches a sequence of one of more digits in this case.

2.2. Basic Traits

In this section, we introduce the history, basic terms, concepts, and semantics related to traits. We focus on the concepts that pre-existed our work. The new syntax and semantics of traits designed and implemented in our work are described in Chapter 3.

2.2.1 Background

The term ‘trait’ was first introduced by Ungar et al. [99] in a dynamically-typed prototype-based language called Self [60]. Traits objects were considered as shared parent objects that provide common behavior to be shared among their instances and refinements. These kinds of traits might contain state (attributes) as well. The foremost goal for such traits was to achieve flexibility while supporting all organizational functions carried out by classes. The organizational functions are as follows [99]:

- Sharing implementation and state among the instances of a data type and among related data types.
- Defining strict interfaces for data types that protect and hide implementation.
- Using global names to refer to data types.
- Categorizing large name spaces into structured parts for easier browsing.

A slightly revised version of traits was introduced in dynamically-typed class-based languages by Schärli et al. [93]. The main motivations for having traits in these languages are reusability problems related to multiple inheritance and mixins. A summary of problems that Schärli et al. try to address through traits is as follows:

- Multiple inheritance
  - Dealing with conflicting features coming from multiple parents: the issue is about the ambiguity that arises when conflicting features are inherited along different paths. A particular version of this issue is known as “diamond problem” or “fork-join inheritance”.
  - Accessing overridden features: this issue happens when identically named features are inherited from different base classes and a single keyword such as *super* is not enough to access inherited methods unambiguously.
  - Factoring out generic wrappers: multiple inheritance allows a class to reuse features from multiple base classes, but it does not allow to write a reusable entity that wraps methods implemented in as-yet unknown classes. Languages such as C++ and Eiffel use templates to compensate this limitation.
- Mixin inheritance
  - Total ordering: since mixins composition is linear, all mixins used by a class need to be inherited once at a time. Mixins used later in the order will override all the identical named features coming from previous earlier mixins. If a conflict happens it might not possible to find a suitable total order in order to resolve the conflict.
  - Dispersal of glue code: When a composite entity uses mixins, it does not have full control of the way used mixins are composed. If a conflict happens, it should be resolve through intermediate classes created when mixins are used (one at a time). This might even result in the need to modify the involved mixins.
  - Fragile hierarchies: The linear nature of mixins limits possibilities for resolving conflicts. Therefore, using multiple mixins results in inheritance chains that are fragile with respect to changes. If changes are required, they might happen in mixins and so it can break the chain in such a way that consistent behavior cannot be established.

Schärli et al. [93] presented definitions, interpretation rules, structures, and specified conflict situations for traits. This formulation of traits was implemented in Squeak [54], an open-source dialect of Smalltalk-80. A trait, in these kinds of languages, is a group of pure methods that serves as building blocks for classes. Schärli emphasized the idea that such traits include only a set of *provided* methods and *required* methods. In comparison to the traits in Self [60], these traits cannot have state (attributes). State had to be obtained through 'glue' code, which must be provided by classes as implementations of required methods. As a result, such traits are called *stateless* traits. Such traits can be composed of other traits but, they cannot include any classes. Classes use such traits in the same way they use generalizations. When classes (clients) declare they use a trait, the contents of the trait's provided methods become logically part of that class (client).

The key properties of traits, in Schärli's formulation, are the following; except where noted, these remain true in later formulations of traits, including those we have developed in this thesis:

- A trait provides a set of methods that implement behavior (the provided methods).
- A trait requires a set of methods (the required methods) that serve as parameters for the provided behavior. These must be provided by the client in some way, either directly or from some other traits.
- Traits do not specify any state variables, and the methods provided by traits never access state variables directly (this limitation is lifted in stateful traits, such as those we present in this thesis).
- Classes and traits can be composed of other traits, but the composition order is irrelevant. Conflicting methods must be explicitly resolved.
- Trait composition does not affect the semantics of a class. The meaning of the class is the same as it would be if all the methods obtained from the trait(s) were defined directly in the class.
- Similarly, trait composition does not affect the semantics of a trait. A composite trait is equivalent to a flattened trait containing the same methods.

A formal definition of traits and their basic properties was provided by Ducasse et al. [35]. This established how traits can be composed of other traits or used by classes. In this thesis, we focus on the style of traits introduced by Schärli et al. [93]. All extension and algorithms we develop are based on them. In the following sections, we explore in detail these traits.

2.2.2 Clients

Clients of traits can be either classes or traits. If the type of a client is a class, it is called a *final* client. If the type of a client is a trait, the client is called a *composed* trait. Traits use other traits to increase levels of granularity. A trait can have any number of clients, but it cannot be its own client. The client relation among traits forms a partial order (no cycles).

2.2.3 Provided Methods

Provided methods are defined in traits and supply functionality to clients. They are similar to standard object-oriented methods. They have their own parameters and implementations. Generally, they use (call) *required* methods to achieve their goals.

2.2.4 Required Methods

Required methods are abstract functionality required by traits in order to provide promised functionality. They have a signature like methods in object-orientation, but they do not have a body (implementation). *Required* methods must be satisfied by clients of traits. If the client of a trait is a final client (a class), its required methods can be satisfied by the client itself, another trait, or its superclass. If a client is another trait, the *required* methods may be satisfied by the trait or the clients of composed trait.

2.2.5 Attributes

Basic traits do not have attributes and, as was mentioned earlier, they receive the state through their required methods. However, not having attributes results in incompleteness in stateless traits. Incompleteness causes classes to require a significant amount of boilerplate

glue code when they use basic traits. In order to deal with this issue, stateful traits [18] were introduced. In our work, we accept this extension and allow traits to have attributes.

Attributes in traits play the same role they have in classes. They can have their own default values, accessors, and mutators. However, there are limitations in their definition when they are used in combination with other traits. We describe these cases in detail when we describe our syntax in Umple.

2.2.6 Use of Traits

Traits can be used by any class that satisfies their required methods. A trait can be used by more than one class. Traits can also be used by other traits, but they are not required to satisfy those required methods. If the required methods are not satisfied by the composed traits, then they will be part of the required methods of the composed traits. Furthermore, a trait cannot be used by itself.

2.2.7 Template Parameters

Traits can also be more general, which is achieved through template parameters. Template parameters are a technique to increase the genericity and hence flexibility and reusability of various elements. In C++ and Java, one can specify generic classes, interfaces, and operations. One can also model UML elements with unbound formal parameters that can be used to define families of classifiers, packages, and operations. This feature is also applied to traits to substantially increase reusability of traits and broaden their appeal. Currently, template parameters are supported by Scala [104].

Template parameters of traits are bound to real types when traits are used by clients. The parameters are treated as types so they can be used in the manner types are used. For example, they can define the type of attributes or parameters of both provided and required methods.

2.2.8 Flattening

Flattening is a concept that says the semantics of a class defined using traits is exactly the same if the class is defined without them. In other words, if a class uses a trait that has provided methods, it is exactly the same as if those methods are defined directly in the class. In

a technical manner, making provided methods (or other elements defined in this thesis for traits) as local properties of clients is called *flattening*.

When inheritance is used among classes, some methods in a class override methods with the same signature that come from its superclass. The same semantics is applied to traits as well. If a class has a concrete method and its used trait has a method with the same signature, the method that comes from the trait is disregarded. The same rule is applied when a trait uses other traits. The process of flattening can be manipulated at runtime or directly applied during compilation (like copying provided methods into clients).

2.2.9 Conflicts in Traits

When traits are used in the design of systems, there are some situations in which the result will be either incorrect or ambiguous. These situations are considered as conflicts and need to be resolved directly by designers. For instance, if a class uses two different traits, and each trait has a method with the same signature but different implementations, this causes a conflict.

The conflicts are detected automatically by Umple and we have defined operators for resolving them – i.e., providing a way to specify which alternative should have precedence. Details about these conflicts and their resolution operators will be explained for each element defined in traits.

2.2.10 Graphical Representations

Traits were first introduced as a programming concept and there has been no standard graphical representation for them suitable to be used for graphical modeling. However, Schärli et al. [93] used a graphical representation for describing traits in their work (as an extension to UML[117]). Figure 6 depicts that representation. As seen, the name of the trait comes at the top and then lists of *provided* and *required* methods appear in the left and right columns, respectively. Furthermore, there are bubble-headed lines attached to the left column for *provided* methods and arrows attached to the right column for *required* methods. We utilize a simplified version of this representation in this thesis, depicted in Figure 7. Our focus is on the textual representation because of its simplicity and scalability, but we accompany our textual representation with a graphical one when it is beneficial.

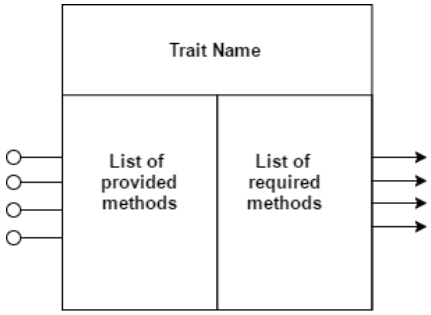


Figure 6. A graphical representation for traits [93]

Figure 7. A simplified graphical representation for traits

2.2.11 Traits and UML Components

UML Components [117] are a type of modular and reusable element in modeling, which has a common terminology with traits. Components and traits have provided and required interfaces and can have a fine and coarse level of control on them. For example, they can have one or several methods as required or provided interfaces/methods. There is no restriction on the granularity of either. They can be substituted by another one of their own types if provided and required interfaces/methods are identical. However, these do not mean that they function in a semantically consistent way.

We consider the following differences between them, which make traits unique at the modeling level; a) components can be instantiated, but traits cannot because they are inherently abstract; b) a UML component may encompass classes, but this is not allowed for traits; c) we cannot remove/rename/change visibility of provided interfaces in UML components while we can do these operations for provided methods of traits; d) we can compile a component and use its binary version, but we cannot perform this for traits; e) internals of

components (except provided interfaces) are hidden for the elements that use them, but for traits, those internals are flattened in the elements; f) there is no concept of port for traits; h) it is possible to use some provided methods of traits, but when we want to use components, we should load all provided interfaces.

2.2.12 Features Lacking in Traits Prior to This Research

In above sections, we discussed the main characteristics of traits. We explained that if a method is defined in a trait, it can be reused in several ways that is not possible if the same method is defined in class. The main reason for this is that traits do not have limitations of inheritance and they are not dependent on the inheritance hierarchy (see Section 1.1). However, there are some features that they do not have. Not having these features might be considered as a reason why traits are not used much in current software engineering practice. These features can be summarized as follows:

1. Traits cannot be used for modeling software systems. In model-oriented software design practices, an architecture of the system is created using a modeling language and then the architecture is improved into a detailed design. Therefore, if traits cannot be used for modeling software systems, they will not be adopted by modern software development methods.
2. Traits cannot contain modeling elements as a way to describe their functionality. Therefore, they cannot offer their functionality in an abstract way. For example, a state machine can be reused better that several methods because it has high abstraction, but traits cannot support this prior to our work.
3. Traits are supported by only a few programming languages, and not in the most widely used languages such as Java and C++. Therefore, it is logical that traits are not yet known and well-accepted by the software development community.

These limitations are the key motivation for this research. We will describe in Chapter 3 how our approach overcomes these limitations.

2.3. Summary

In this chapter, we reviewed basic concepts related to Umple and its modeling elements that will be used along with traits at the modeling level. We also reviewed the semantics and relevant concepts related to classis traits. Knowing these concepts will help the reader to understand our model-based traits and their implementation in Umple, covered in the next chapter.

In the next chapter, we introduce requirements, syntax, and semantics of model-based traits. We also demonstrate mechanisms that allow composing modeling elements defined in this chapter. We also demonstrate how traits can be implemented in object-oriented programming languages based on our approach.

Chapter 3. Traits in Model-Driven Software Development

In this chapter, we provide an overview of requirements that must be fulfilled in order to have traits at the modeling level. We then explain activities that need to be performed to process traits in modeling languages. Next, we describe our flattening algorithm including semantics of each step of the algorithm. The algorithm also covers how state machines can be composed. The described semantics is independent of the actual implementation, so other modeling languages can adopt traits.

After describing the semantics of our algorithm, we explain the core syntax of traits that we implemented in Umple. We demonstrate enhanced features such as state machines, associations, and required interfaces. We present various automatic code generation mechanisms for traits and describe our implementation. The syntax and features of traits are described by simple examples in Umple; these are followed by a graphical representation to improve understanding. Finally, we demonstrate the metamodel of traits and describe the application of each class in the metamodel.

We want to indicate that we are not purporting to add traits to UML. We are introducing them as a textual modeling concept in the Umple language that strongly aligns with UML but is not UML itself. Our graphical representation is designed to help visualize traits and understand our work, but is not proposed as a UML extension or even a formal contribution of our work.

In our approach, diagrams are generated rather than being drawn using a drawing tool. We do not make any claims about whether it would be usable or useful to edit the diagrams or use them to create traits in models.

3.1. Requirements and Design Goals

In order to have traits at the modeling level, we need to clearly specify what the requirements are, why those requirements are important or needed, and whether those requirements are

based on already developed concepts or are new. Furthermore, it is required to specify the design goals. In this section, we cover these subjects.

Requirements for model-based traits were extracted from different sources. We started by studying different variations of inheritance [28,36,92,97] in order to conceptualize the problems we would need to resolve. Then, we concentrated on classical traits in programming languages [8,54,104,105,108]. This was important because we wanted to have the classical traits' flexibility at the modeling level as well. We extended our requirements extraction by studying other research conducted to extend features of classical traits [18,20,29,33,65,74,83,84,100]. The most inspiring source for the requirements was model-based engineering. This was clear because we wanted to have traits at the modeling level so requirements of being a modeling element must be added to traits.

The following shows a summary of main requirements, however, the full list of requirements extracted along with their justification, source, and satisfaction (in our modeling languages) is listed in Appendix II.

- Traits should define their functionality through provided methods
- Traits should define their required functionality through required methods
- Traits should have template parameters
- Traits should be able to define constraints on their template parameters
- Traits should be able to define functionality through associations
- Traits should be able to define functionality through state machines
- Traits should be able to have attributes
- Traits should be reusable by classes and other traits
- Traits should be able to be used along with inheritance
- Traits should be able to define required interfaces for their class clients
- Traits should be able to be reused by more than one client
- Traits should offer mechanisms to resolve conflicts
- Traits should offer mechanism to control their elements' granularity when they are reused by client.
- Traits should provide a mechanism to extend functionality of other trait elements (e.g., state machines)

- Traits should be implementable by object-oriented programming languages

The design goals of having traits at the modeling level are as follows:

- Improve reusability mechanisms of the modeling elements
- Increase the level of abstraction in which functionality of traits can be expressed
- Improve reuse, extension, and composition of state machines
- Improve the way traits can be used by final clients to control (specify) their content
- Support implementation of traits in object-oriented programming languages

3.2. Workflow of Processing Traits

When a system is modeled based on traits, the modeled system needs to be processed in order to be transformed into an executable system in an object-oriented programming language. Figure 8 depicts the necessary phases of processing a trait-based model. Figure 8 also shows what kinds of artifacts are produced and required in each processing phase. The different phases are described in the following sections.

3.2.1 Parsing Phase

As seen in Figure 8, the inputs of this phase are a trait-based model and grammar files. The grammar files include the grammar for traits as well as for other modeling elements such as classes and interfaces. The model and grammar files are dependent on the language in which traits are implemented. In our case, we use the Umple model and grammar files. The structure of the Umple grammar is discussed in Section 2.1.6 and the grammar of traits is explained in Section 3.4. The complete Umple grammar file related to traits can be found in our online Github repository [119].

In the parsing phase, the parser (Umple parser) parses a given model and builds an abstract syntax tree for the model based on the grammar files. The parser guarantees that the model is valid syntactically. The output artifact of this phase is an abstract syntax tree.

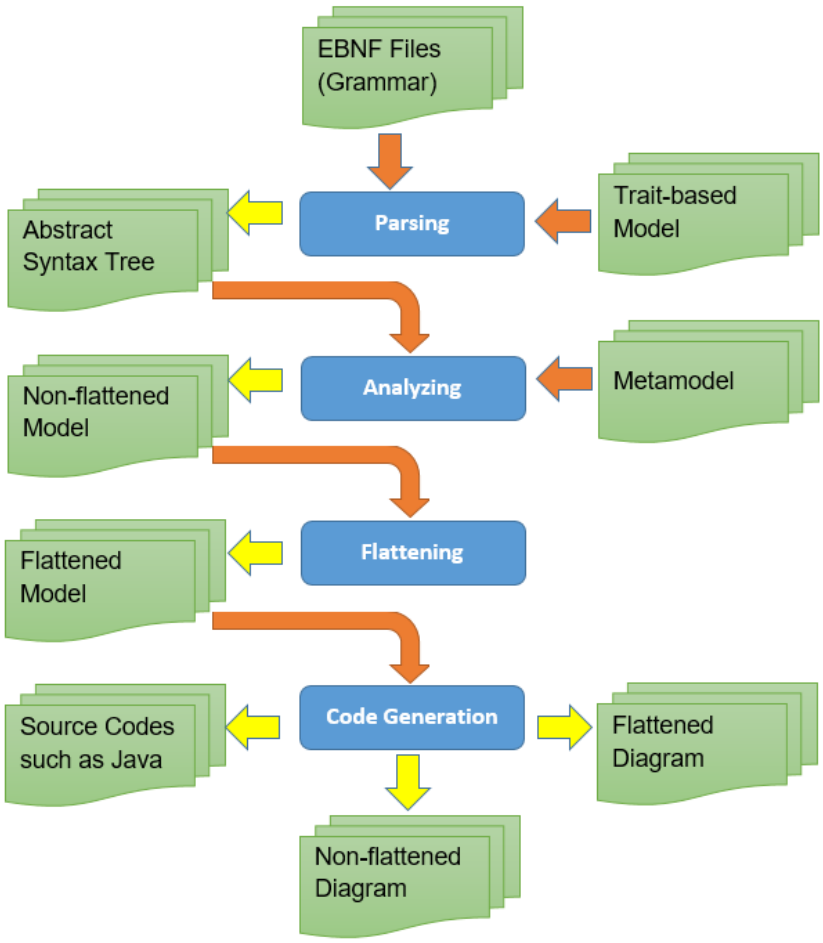


Figure 8. The general workflow of processing traits

If the modeling language is not textual, then it is required to make sure the modeled system is validated syntactically through other mechanisms available for the language. Graphical modeling languages, for example, make sure their models are valid syntactically through their user interfaces.

3.2.2 Analyzing Phase

As seen in Figure 8, the inputs of this phase are an abstract syntax tree obtained from the parsing phase and the metamodel of the modeling language (including the metamodel of traits and other modeling elements). The metamodel of traits is explained in Section 3.9 and the complete Umple metamodel can be found in our online Github repository [50].

In the analyzing phase, the analyzer builds an instance of the metamodel based on its inputs. In our case, it builds an instance of the Umple metamodel. While the analyzer builds

an instance of the metamodel, it also validates that defined traits and their elements are valid alone. For example, the names of traits are unique in the system or there are not more than one provided or required method with the same signature.

Moreover, other modeling elements such as classes and interfaces are also validated alone in this phase. Umple elements and their semantics are explained in Section 2.1. The output of this phase is a valid instance of the metamodel, which we call the non-flattened model. This phase of processing (analyzing) exists in both textual and graphical modeling languages. Therefore, the languages that want to adopt traits need to extend their metamodel and their analyzing phase.

3.2.3 Flattening Phase

As seen in Figure 8, the input of this phase is a non-flattened model from the previous phase (the analyzing phase). The flattener receives the non-flattened model and then applies specified operators (explained in Sections 3.4.11 and 3.5.4) to the model's elements, composes state machines (explained in Section 3.5.5), and finally flattens elements (composed and not composed) into the final clients (Section 3.3.2). The output artifact of this phase is an instance of the metamodel called the flattened model. The flattening algorithm is explained in Section 3.3.1.

Modeling languages that want to adopt traits need to add this phase to their processing phases. The language-specific implementation of the flattening algorithm (see Section 3.3.1) can be different based on the languages' limitations and strengths. In other words, modeling languages must implement the semantics of flattening elements defined in Section 3.3.2. Note that composition of state machines is covered as part of the flattening algorithm.

3.2.4 Code Generation Phase

As seen in Figure 8, the input of this phase is a flattened model from the flattening phase. A code generator receives a flattened model and generates a programming language defined by the developers (described in Section 3.8). There can be several different code generators; for example Umple can generate Java, C++, and other languages. Other code generator can also produce artifacts such as diagrams of the model, including both flattened and non-flattened diagrams (described in Section 3.4.7). In our work, we reuse already-developed code genera-

tors for different programming languages supported by Umple. However, we developed diagram generators for flattened and non-flattened models as part of this work.

Modeling languages that adopt traits might have their own code generators. Therefore, their code generators must be able to generate the functional system without any modification. If a modeling language does not have a code generator and they run the model directly, it must also be possible without any modification. The main reason is that traits are flattened to modeling elements that are already recognized by code generators or model runners. However, if modeling languages want to have non-flattened and flattened diagrams, they need to achieve this through the technologies available for them.

3.3. Flattening of Modeling Elements

Flattening is one of the fundamental concepts behind traits. Therefore, in this section, we describe our flattening algorithm as part of processing activities for traits explained in section 3.2. We also explain the semantics of flattening for each modeling element defined in our flattening algorithm. As pointed out in Section 3.2.3, the real implementation (language-specific) of the flattening algorithm can be different based on the languages' limitations and strengths. Each modeling language must implement the semantics of flattening elements defined in Section 3.3.2.

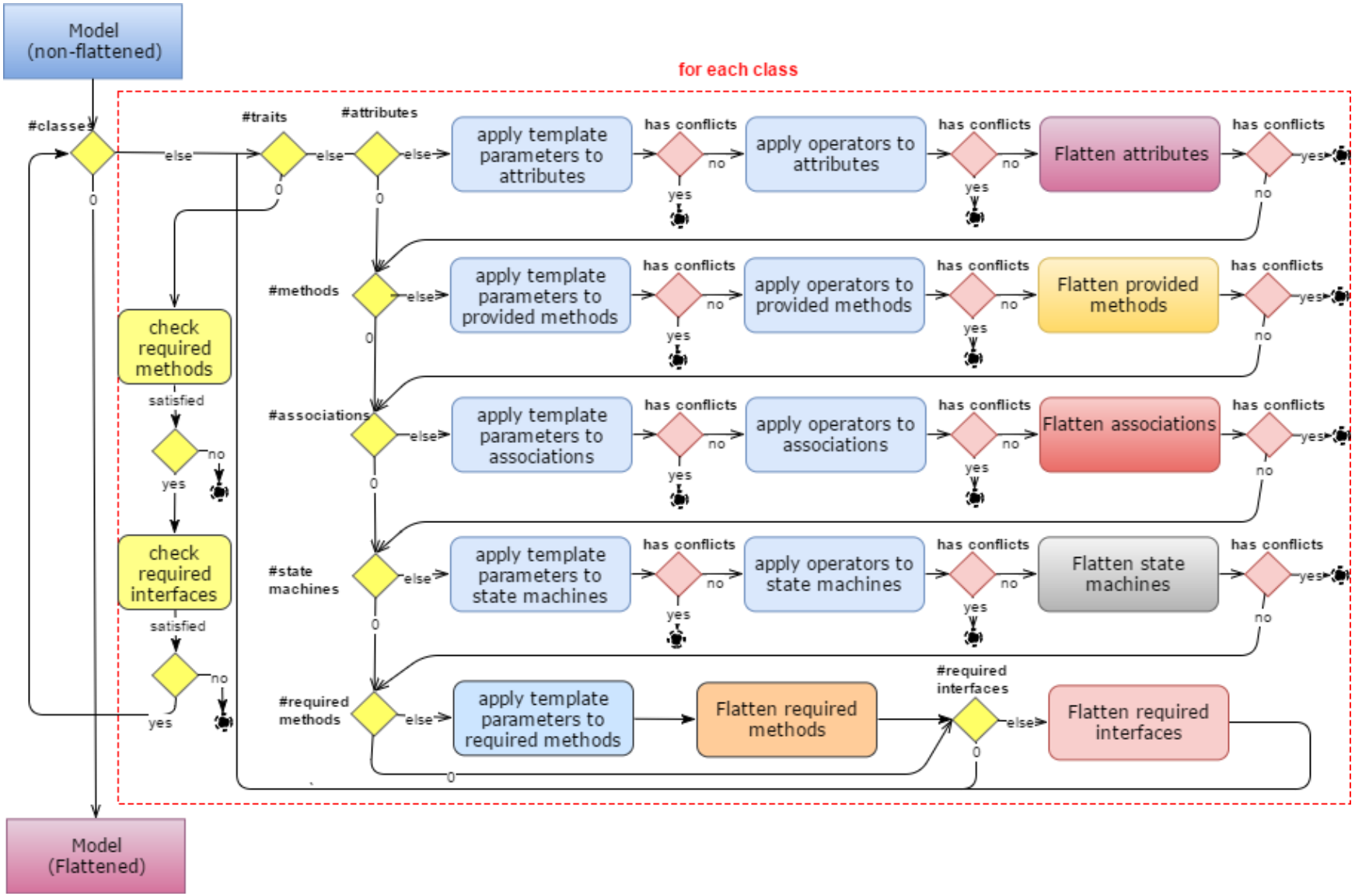


Figure 9. The general flattening algorithm for traits

3.3.1 Flattening Algorithm

As described in Section 3.2.3, the flattening phase receives a non-flattened model and then builds a flattened model used for the code or diagram generation. The general algorithm used to obtain a flattened model from a non-flattened model is depicted in Figure 9. The Umple-specific implementation of the algorithm can be found in our online Github repository [31]. Flattening of traits is initiated by calling a method named "*void applyTraits()*".

As seen in Figure 9, the flattening algorithm is applied to each class in the non-flattened model (input) and if there is no class left, the flattened model (output) is delivered to the code generation phase.

For every class in the non-flattened model, the algorithm goes through each trait used by the class and flatten attributes, provided methods, associations, state machines, required methods, and required interfaces. In the following, we describe the different steps of the algorithm:

Step 1: For each class in the non-flattened model execute steps 2 through 4. If there is no class left, return the flattened model.

Step 2: For each trait used by the class (specified in step 1) execute steps 2.1 through 2.6. If there is no used trait left, execute step 3.

Step 2.1 (for attributes):

2.1.a: Apply template parameters to attributes and if there is no conflict, proceed to the next sub-step (2.1.b). Otherwise, terminate with an error.

2.1.b: Apply operators to attributes and if there is no conflict, proceeds to the next sub-step (2.1.c). Otherwise, terminate with an error. *Note that in the current implementation of the algorithm in Umple, operators related to attributes are not implemented yet.*

2.1.c: Flatten the attributes into the class, and if there is no conflict, proceed to flatten provided methods (step 2.2). Otherwise, terminate with an error. The semantics of flattening attributes is described in Section 3.3.2.1.

Step 2.2 (for provided methods):

2.2.a: Apply template parameters to provided methods and if there is no conflict, proceed to the next sub-step (2.2.b). Otherwise, terminate with an error.

2.2.b: Apply operators to provided methods and if there is no conflict, proceed to the next sub-step (2.2.c). Otherwise, terminate with an error.

2.2.c: Flatten the provide methods into the class and if there is no conflict, proceed to flatten associations (step 2.3). Otherwise, terminate with an error. The semantics of flattening provided methods is described in Section 3.3.2.2.

Step 2.3 (for associations):

2.3.a: Apply template parameters to associations and if there is no conflict, proceed to the next sub-step (2.3.b). Otherwise, terminate with an error.

2.3.b: Apply operators to associations and if there is no conflict, proceed to the next sub-step (2.3.c). Otherwise, terminate with an error. *Note that in the current implementation of the algorithm in Umple, operators related to associations have not implemented yet.*

2.3.c: Flatten the associations into the class and if there is no conflict, proceed to flatten state machines (step 2.4). Otherwise, terminate with an error. The semantics of flattening associations is described in Section 3.3.2.6.

Step 2.4 (for state machines):

2.4.a: Apply template parameters to state machines and if there is no conflict, proceed to the next sub-step (2.4.b). Otherwise, terminate with an error.

2.4.b: Apply operators to state machines and if there is no conflict, proceed to the next sub-step (2.4.c). Otherwise, terminate with an error.

2.4.c: Flatten the state machines into the class and if there is no conflict, proceed to flatten required methods (step 2.5). Otherwise, terminate with an error. The semantics of flattening state machines is described in Section 3.3.2.5.

Step 2.5 (for required methods):

2.5.a: Apply template parameters to required methods and proceed to the next sub-step (2.5.b).

2.5.b: Flatten the required methods into the class and then proceed to flatten required interfaces (step 2.6). The semantics of flattening required methods is described in Section 3.3.2.3.

Step 2.6 (for required interfaces):

2.6.a: Flatten the required interfaces into the class and then proceed. The semantics of flattening required interfaces is described in Section 3.3.2.4.

Step 3: (handling required methods)

Check whether the class is abstract or not.

If the class is abstract, consider unsatisfied flattened required methods as abstract methods of the class. Then, proceed to step 4.

If the class is not abstract, check whether or not it satisfies all flattened required methods. If the class satisfies them, proceed to step 4; otherwise, terminate with an error.

Step 4: (handling required interfaces)

Check whether the class is abstract or not.

If the class is abstract, consider unimplemented flattened required interfaces as interfaces that must be implemented by the class. Then, proceed.

If the class is not abstract, check whether or not it implemented all flattened required interfaces. If the class implements all of them, proceed; otherwise, terminate with an error.

3.3.2 Semantics of Flattening Elements

The general concept of flattening was explained in Section 2.2.8. However, we describe here the semantics of flattening in detail when it comes to all elements that can be defined in traits (e.g., state machines and methods). In our approach, flattening is implemented at the modeling level. In fact, when an element from a trait can be flattened into a client, we simply make a copy of that element and add it to the client. Working at the modeling level enables traits' implementation to be supported automatically by all code generators available for Umple. We suggest the same perspective when other modeling languages or frameworks want to adopt traits.

Please consider that not always all elements of traits are flattened into clients because they might be under the effect of removing operators. In the same vein, some elements might be changed before flattening because of renaming operators or the composition mechanism (which is valid for state machines). In the following sections, the semantics of flattening is described with respect to the fact that template parameters and operations have already been applied on elements and so elements are ready to be flattened.

3.3.2.1 Flattening of attributes

When a client uses a trait and the trait has one or many attributes, those attributes are added (technically are copied) to the client under the following conditions:

- *cond1*: the client does not have local attributes with the same name and type.
- *cond2*: the client does not obtain attributes with the same name and type from other used traits, except those attributes come from the same source (trait) by another path.

Based on the defined conditions above, other cases might happen:

- If condition *cond1* is false and condition *cond2* is true or false: those attributes coming from the trait are disregarded.
- If condition *cond1* is true and condition *cond2* is false: this is a conflict, and flattening should not proceed until the modeler resolves the conflict.

3.3.2.2 Flattening of provided methods

When a client uses a trait and the trait has one or many provided methods, those provided methods are added to the client under the following condition:

- *cond1*: the client does not have local methods with the same signature.
- *cond2*: the client does not obtain provided methods with the same signature from other used traits, except those provided methods come from the same source (trait) by another path.

Based on the defined conditions above, other cases might happen:

- If condition *cond1* is false and condition *cond2* is true or false: those provided methods coming from the trait are disregarded.
- If condition *cond1* is true and condition *cond2* is false: this is a conflict and flattening should not proceed until the modeler resolves the conflict.

3.3.2.3 Flattening of required methods

When a trait (host) uses another trait and the used trait has one or many required methods, those required methods are flattened to the host trait under the following conditions:

- *cond1*: the host trait does not implement the required methods.
- *cond2*: the host does not implement the required methods by using another trait.
- *cond3*: the host trait does not have local required methods with the same signature. Note that a trait cannot have a provided and required method with the same signature. Therefore, *cond1* and *cond3* cannot be false at the same time.

- *cond4*: the host trait does not obtain required methods with the same signature from other used traits.

Based on the defined conditions above, other cases might happen too:

- If condition *cond1* is false and other conditions are true or false: those required methods coming from the trait are disregarded.
- If condition *cond2* is false and other conditions are true or false: those required methods coming from the trait are disregarded.
- If condition *cond3* is false and other conditions are true or false: those required methods coming from the trait are disregarded.
- If condition *cond4* is false and other conditions are true: this results in two scenarios as follows.
    - Other used traits have not been flattened yet: in this case, those required methods coming from the trait are flattened into the host trait.
    - Other used traits have been flattened: in this case, those required methods coming from the trait are disregarded.

When a class uses a trait and the trait has one or many required methods, the same process explained above is performed to obtain all required methods needed to be implemented by the class. In fact, in this case, flattening does not happen because the class must implement those required methods. However, if the class is abstract, flattening happens and those required methods are considered as abstract methods for the abstract class and must be implemented by concrete subclasses of the abstract class (note: this is supported by Step 3 of the flattening algorithm described in Section 3.3.1).

3.3.2.4 Flattening of required interfaces

When a trait (host) uses another trait and the used trait has one or many required interfaces, those required interfaces are added to the host trait under the following conditions:

- *cond1*: the host trait does not have local required interfaces with the same name.

- *cond2*: the host trait does not obtain required interfaces with the same name from other used traits.

Based on the defined conditions above, other cases might happen:

- If condition *cond1* is false and condition *cond2* is true or false: those required interfaces coming from the trait are disregarded.
- If condition *cond1* is true and condition *cond2* is false: this results in two scenarios as follows:
    - Other used traits have not been flattened yet: in this case, those required interfaces coming from the trait are flattened into the host trait.
    - Other used traits have been flattened: in this case, those required interfaces coming from the trait are disregarded.

When a class uses a trait and the trait has one or many required interfaces, the same process is performed to obtain all required interfaces that must be implemented in advance by the class. In fact, in this case, flattening does not happen because classes must implement those required interfaces. However, if the class is abstract, flattening happens and those required interfaces are considered as interfaces implemented by the abstract class. The concrete subclasses of the abstract class will be forced to implement those interfaces (note: this is supported by Step 4 of the flattening algorithm describe in Section 3.3.1).

3.3.2.5 Flattening of state machines

When a client uses a trait and the trait has one or many state machines, those state machines are added to the client under the following conditions:

- *cond1*: the client does not have local state machines with the same name.
- *cond2*: the client does not obtain state machines with the same name from other used traits.

Based on the defined conditions above, other cases might happen:

- If condition *cond1* is false and condition *cond2* is true: those state machines coming from the trait are composed with the same machine in the client. The composition algorithm is described in Section 3.3.3.
- If condition *cond1* is true and condition *cond2* is false: those state machines coming from the trait are composed with the same machines coming from the used traits. The final composed state machines are added to the client.
- If condition *cond1* is false and condition *cond2* is false: in this case the following steps are performed.
    - Step 1: Those state machines coming from the trait are composed with the same machines coming from the used traits.
    - Step 2: The composed state machines (from Step 1) are composed with the same machines in the client.

3.3.2.6 Flattening of associations

When a client uses a trait and the trait has one or many associations, those associations are added to the client under the following conditions:

- *cond1*: the client does not have local associations with the same role name.
- *cond2*: the client does not obtain associations with the same role name from other used traits, except those associations coming from the same source (trait) that the client has gotten its associations from too.

Based on the defined conditions above, the following cases might happen:

- If condition *cond1* is false and condition *cond2* is true or false: those associations coming from the trait are disregarded.
- If condition *cond1* is true and condition *cond2* is false: this is a conflict and flattening should not proceed until the modeler resolves the conflict.

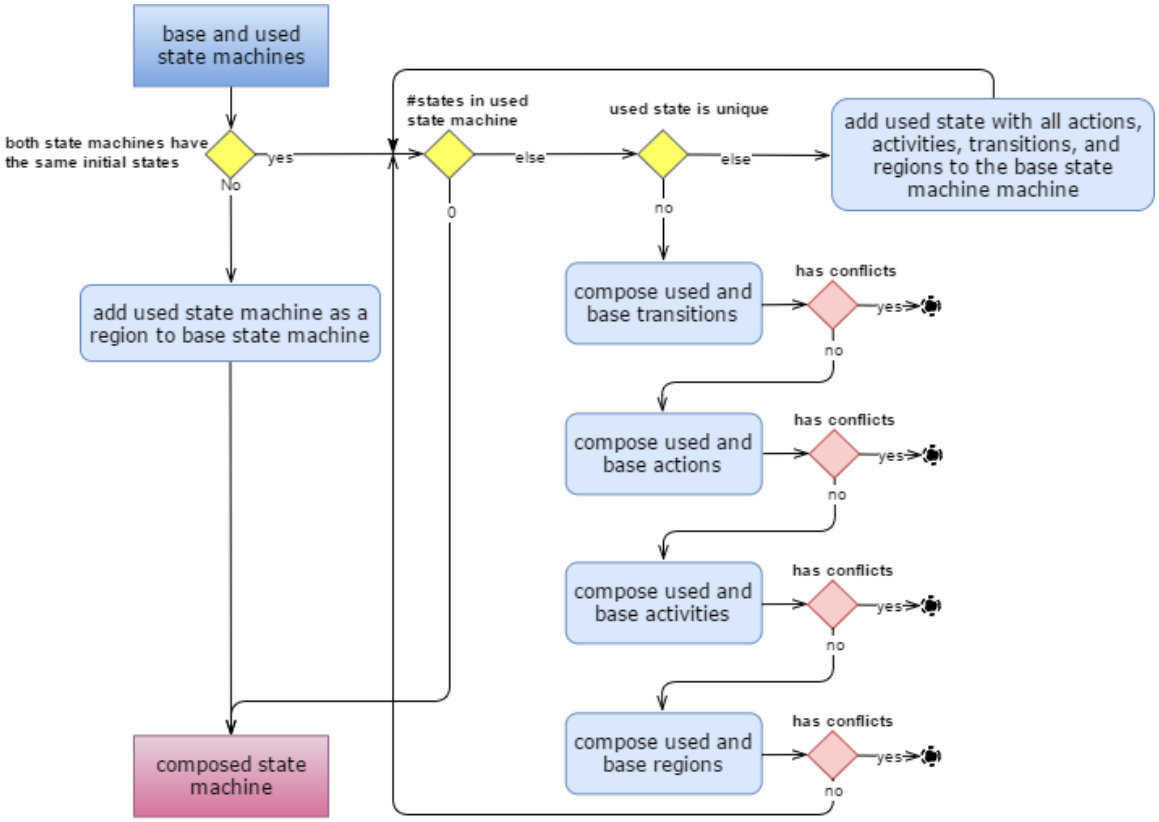


Figure 10. The general activities of the composition algorithm for state machines

3.3.3 Composition Algorithm for State Machines

As part of the semantics for flattening state machines define in Section 3.3.2.5, state machines with the same name should be composed. In this section, we describe the general activities of our composition algorithm. The semantics of each activity is described in Section 3.3.4. The exact implementation of our algorithm in the Umple language can be found in our online repository [31]. In order to describe the algorithm and also semantics of activities (in Section 3.3.4), we consider the following assumptions:

- The state machine defined in the client is called *base* state machine. The same concept is considered for other internal elements of the state machine such as base state, base transition, and so on.
- The state machine defined in the used trait is called *used* state machine. The same concept is considered for other internal elements of the state machine such as used state, used transition, and so on.

- All the time, the base state machine accepts new elements and modification. In fact, the composed state machine is the base state machine with all new and composed elements.
- When actions are indicated, we mean three groups of actions: entry actions of states, exit actions of states, and actions of transitions.

As seen in Figure 10, the algorithm receives two state machines named base and used state machines and then it goes through the following steps:

Step 1: Checks whether or not both state machines have the same initial states. If they have, then proceed to step 2, otherwise, add the used state machine as a region to the base state machine. The base state machine is returned as the composed state machine. The semantics of this phase is explained again in Section 3.3.4.1.

Step 2: Execute step 3 for each state in the used state machine. If there is no used state left, return the composed state machine.

Step 3: Check whether or not the used state exists in the base state machine. If it does not, add the used state to the base state machine. Otherwise, execute steps 3.1 through 3.4. The semantics of this step is described in Section 3.3.4.2.

Step 3.1: Compose transitions of base and used states. If there is a conflict then terminate with an error. Otherwise, it goes to step 3.2. The semantics of this step is described in Section 3.3.4.5.

Step 3.2: Compose actions of base and used states. If there is a conflict then terminate with an error. Otherwise, proceed to step 3.3. The semantics of this step is described in Section 3.3.4.3.

Step 3.3: Compose activities of base and used states. If there is a conflict then terminate with an error. Otherwise, proceed to step 3.4. The semantics of this step is described in Section 3.3.4.4.

Step 3.4: Compose regions of base and used states. If there is a conflict then terminate with an error. Otherwise, proceed. The semantics of this step is described in Section 3.3.4.6.

3.3.4 Semantics of Composition Activities

In this section, we describe the semantics of activities defined as part of our composition algorithm in Section 3.3.3.

3.3.4.1 Composing state machines

A base state machine and a used state machine can be composed based on the following conditions:

- If both state machines have the same initial state. In this case, internal elements of both state machines are composed as well (see Sections 3.3.4.2, 3.3.4.3, 3.3.4.4, 3.3.4.5, 3.3.4.6).
- If base and used state machines have different initial states. In this case, internal elements of both state machines are kept separate and the used state machines is added as a region to the base state machine.

3.3.4.2 Composing states

States (simple or composite) of a base state machine and a used state machines are composed based on the following conditions:

- The used state is not available in the base state machine. In this case, the used state is added to the base state machine. When a state is added to the base state machine, its actions, activities, transitions, and regions (valid for composite states as well) are also added to the base state machine.
- The used state is also available in the base state machine. In this case, the used state's actions, activities, transitions, and regions (valid for composite states as well) are composed with those of the base state based on semantics defined for them in Sections 3.3.4.3,3.3.4.4, 3.3.4.5, and 3.3.4.6, respectively.

3.3.4.3 Composing actions

Actions of a base element (state or transition) and a used element are composed under the following conditions:

- If the base element has an action without the keyword *superCall* and the following conditions happen as well:

- The base element does not receive an action from other used traits. In this case, the actions of the used element are disregarded.
- The base element receives an action from other used traits. In this case, the actions of the used element are disregarded.

- If the base element has an action with the keyword *superCall* and the following conditions happen as well:
  - The base element does not receive an action from other used traits. In this case, the keyword *superCall* is replaced with the actions of the used element.
  - The base element receives an action from other used traits. In this case, a conflict happens.
- If the base element does not have an action and the following conditions happen as well:
  - The base element does not receive an action from other used traits. In this case, the actions of the used element are added to the base state.
  - The base element receives an action from other used traits. In this case, a conflict happens.

3.3.4.4 Composing activities

Activities of a base state and a used state are composed under the following conditions:

- If the base state has an activity without the keyword *superCall* and the following conditions happen as well:
  - The base state does not receive an activity from other used traits. In this case, the activities of the used states are disregarded.
  - The base state receives an activity from other used traits. In this case, the activities of the used state are disregarded.
- If the base state has an activity with the keyword *superCall* and the following conditions happen as well:
  - The base state does not receive an activity from other used traits. In this case, the keyword *superCall* is replaced with the activities of the used state.
  - The base state receives an activity from other used traits. In this case, a conflict happens.

- If the base state does not have an activity and the following conditions can happen as well:
    - The base state does not receive an activity from other used traits. In this case, the activities of the used state are added to the base state.
    - The base state receives an activity from other used traits. In this case, a conflict happens.

3.3.4.5 Composing transitions

When transitions of states are composed, we need to have a signature for transitions to be able to differentiate them from each other. The signature of a transition is the combination of the signature of the event and its guard. Therefore, transitions of a base state and a used state are composed under the following conditions:

- If the base state has the used transition then only actions of two transition are composed based on the semantics defined in Section 3.3.4.3.
- If the base state does not have the used transition then the used transition is added to the base state. However, adding the used transition must not cause the base state machine to be non-deterministic. If it would do so, then a conflict occurs.

3.3.4.6 Composing regions

Regions of a base composite state and a used composite state are composed under the following conditions:

- The base composite state has a region with the name of the used region: then the used region is composed with the base region. The semantics of composition for regions is like that described for two state machines in Section 3.3.4.1.
- The base composite state does not have the used region: then the used regions is added to the base composite state.

3.4. Traits in Umple

Umple traits are defined through the keyword *trait* followed by a unique name and a pair of curly brackets. The name must be alphanumeric and start with an alpha character, or the symbol (underscore), otherwise, the Umple compiler raises error code 200. We also recom-

mend capitalizing the first letter of traits names, as is the case for classes and interfaces in Umple. If it is not capitalized, the warning code 201 is raised. Furthermore, if the name is not unique in the system under development, error code 203 is raised by the compiler. All errors and warnings generated by the Umple compiler and related to the correctness and validity of traits are summarized in Table 5 in Appendix I.

All elements of traits are defined inside the curly brackets except template parameters defined between the trait name and the curly brackets (described as we progress). Listing 7 shows the top-level grammar rule for traits. Other rules are defined as they are introduced. If a rule is defined in more than one place, the actual Umple rule is a union of all of them, in the same manner as if they are logically separated by the vertical bar symbol |. In the definitions related to traits, we have reused some already-defined rules from other parts of the Umple grammar. This ensures consistent syntax for common elements between traits and classes.

Listing 7. The top-level grammar rule for the definition of traits

```
1  traitDefinition: trait [name] [[traitParameters]]? { [[traitContent]]* }
```

Listing 8 shows a symbolic example through which we will describe the basics of traits. As seen, there are two traits called *T1* and *T2* defined in lines 1 and 6. Furthermore, there are two classes called Class *C1* and *C2* defined in 12 and 15. Class *C2* is a subclass of class *C1*. Other elements of the model are described as we progress.

Figure 11 shows the graphical representation of the model in Listing 8. It is generated automatically by Umple and can be accessed in UmpleOnline [116]. Each trait is depicted based on the notation introduced in Section 2.2.10. We will explain the detail of the graphical representation when we describe the corresponding textual syntax. The graphical representation used for expressing how a trait is used by clients is the same as the arrow used for inheritance. We have chosen the same notation because Umple uses *isA* for inheritance and also for using traits. This keeps textual and graphical syntax compatible. More detail regarding reusing traits is provided in Section 3.4.3.

Listing 8. A symbolic example describing basic syntax of traits

```
1  trait T1{
2    abstract void method1();
3    abstract void method2();
4    void method4(){/*implementation*/ }
5  }
6  trait T2{
7    isA T1;
8    void method3();
9    void method1(){/*implementation*/ }
10   void method2(){/*implementation*/ }
11 }
12 class C1{
13   void method3(){/*implementation*/ }
14 }
15 class C2{
16   isA C1;
17   isA T2;
18   void method2(){/*implementation*/ }
19 }
```

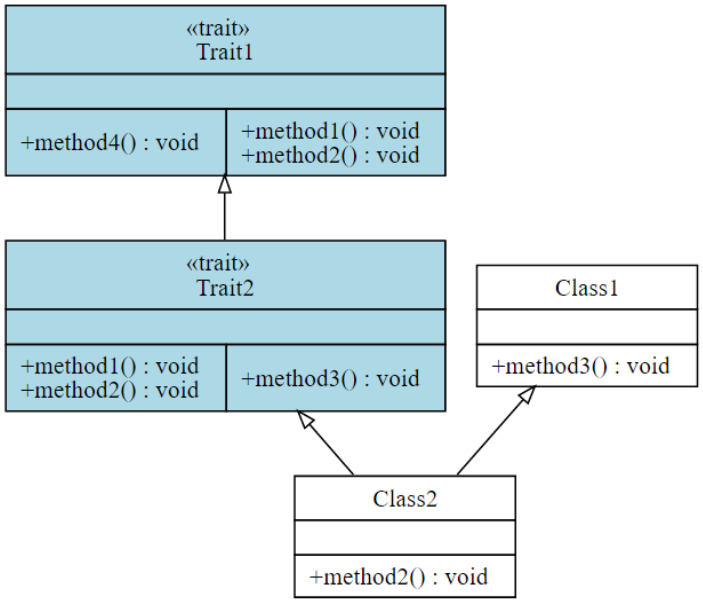


Figure 11. The graphical representation of the Umple model in Listing 8

3.4.1 Definition of Required Methods

Required methods are defined similarly to the way abstract methods are defined in classes. They have exactly the same syntax, but it is also possible in traits to define required methods without the keyword *abstract*. If a method is defined like a normal method without a body

(or implementation), the Umple compiler will consider that as a required method. Listing 9 shows the grammar related to required methods and provided methods (described in the next section). Some of referenced rules are omitted because of brevity.

For example, trait *T1* in Listing 8 has two required methods named *method1()* and *method2()* in lines 2 and 3. These required methods have been defined by the keyword *abstract* while trait *T2* has a required method named *method3()* defined in line 8 without the keyword abstract. Figure 11 depicts those required methods in the right column on each trait.

Listing 9. The grammar related to the definition of required and provided methods

```
1  traitContent-:  [[abstractMethodDeclaration]]
                 | [[concreteMethodDeclaration]]
2  concreteMethodDeclaration : [=modifier:public|protected|private]?
       [=static]? [type]? [[methodDeclarator]] [[methodThrowsExceptions]]?
       [[methodBody]]
     | [=modifier:public|protected]? [=abstract] [type]?
       [[methodDeclarator]] [[methodThrowsExceptions]]? ;
3  abstractMethodDeclaration:  [type] [[methodDeclarator]] ;
4  methodDeclarator: [methodName] [[parameterList]]
5  parameterList: OPEN_ROUND_BRACKET ([[parameter]] ( , [[parameter]] )* )?
      CLOSE_ROUND_BRACKET
6  parameter: [[typedName]]
7  typedName-: [type]? [[list]]? [~name]
8  list-: [!list:\[\s*\]]
```

3.4.2 Definition of Provided Methods

Provided methods are defined in the same way concrete methods are defined in classes. Indeed, they have exactly the same syntax and semantics. Provided methods also support multiple code blocks, for generation of systems in different languages. Listing 9 shows the grammar related to the definition of provided methods.

For example, trait *T1* in Listing 8 has a provided method named *method4()* defined in line 4 while trait *T2* has two provided methods *method1()* and *method2()* defined in lines 9 and 10. Figure 11 depicts those provided methods on the left column of corresponding traits.

3.4.3 Reusing Traits inside Clients

Traits are used by clients by specifying the keyword *isA* followed by their names and a semi-colon. When clients specify names of traits for use, those traits must exist in the system, otherwise, the Umple compiler detects non-existent traits and raises error code 202. If a client uses more than one trait, it can separate them by comma or use the keyword isA for each one.

As pointed out in Section 2.2.2, traits cannot use themselves. For example, in Listing 10.a, trait T1 uses itself. Therefore, the Umple compiler detects this case and raises error code 204. The compiler also ensures that this case does not happen through several uses in a cyclic manner, otherwise, raises error code 205. For example, in Listing 10.b, trait T1 uses trait T2 and trait T2 uses trait T3, Trait T3 again uses trait T1. This cyclic use makes trait T1 again uses itself.

Listing 10. Errors related to using traits in a cyclic manner

a

```
1  trait T1{
2    isA T1;
3  }
```

b

```
trait T1{  isA T2; }
trait T2{  isA T3; }
trait T3{  isA T1; }
```

Listing 11. The grammar related to the use of traits

```
1   traitContent-: [[softwarePattern]]
2   softwarePattern- : [[isA]]
3   isA- : [[singleIsA]] | [[multipleIsA]]
4   singleIsA- : isA [[isAName]] ( , isA [[isAName]] )*  ;
5   multipleIsA- : isA [[isAName]] ( , [[isAName]] )* ;
6   isAName- : " [**extendsNames] " [[gTemplateParameter]]?
      | [extendsName] [[gTemplateParameter]]?
7   gTemplateParameter : < [[AllInclusionExclusionAlias]] >
8   AllInclusionExclusionAlias- : [[InclusionExclusionAlias]]
      ( , [[InclusionExclusionAlias]]  )*
9   InclusionExclusionAlias- : [[functionIncludeExcludeAlias]]
10  functionIncludeExcludeAlias- : [[functionInExAlias]]
      ( , [[functionInExAlias]]  )*
11  functionInExAlias- : [[traitAppliedParameters]]
12  traitAppliedParameters : [~pName] = [~rName]
```

Listing 11 shows the grammar related to using traits. It also includes the grammar related to binding values to the template parameters described in Section 3.4.9. Note that the rule *softwarePattern* is also referenced in the grammar for classes, therefore classes use the same structure to use traits.

For example, trait *T2* in Listing 8 uses trait *T1* defined in line 7 and class *C2* uses trait *T2* defined in line 17. Clients can use other traits if they satisfy their required methods. Satisfaction of required methods is performed by having exactly the same methods implemented in clients. For example, trait *T2* in Listing 8 uses trait *T1* and so it is required to have the required methods of trait *T1* implemented in trait *T2*. Trait *T2* achieves this through implementing two methods named *method1()* and *method2()* defined in lines 9 and 10. If a class uses a trait without satisfying its required methods, the Umple compiler detects unsatisfied required methods and raises error code 208.

Trait T2 in Listing 8 is not a final client, so, it could use trait *T1* without implementing those required methods. Therefore, the required methods of trait *T2* are *method1()*, *method2()*, and *method3()*. Class *C2*, which is a final client, uses trait *T2* and therefore needs to implement its required method, which is *method3()*. However, there is no direct implementation for it. Instead, class *C2* obtains such an implementation indirectly from its superclass, which is *C1*. Therefore, it satisfies the required method of trait T2. Figure 11 depicts the use of associations among traits and their clients.

When clients use traits, they obtain all provided methods defined in the traits. This includes all other provided methods those traits might obtain from their own used traits. For example, trait *T2* gets the provided method *method4()* from trait *T1*. This provided method can be called by all other provided methods defined in trait *T2*. Therefore trait *T2* provides three provided methods named *method1()*, *method2()*, and *method4()*. Class C2 uses trait T2 and so it obtains all provided methods.

3.4.4 Traits and Umple Mixins

In the same way Umple supports mixins to compose classes, traits can also be composed in this way. This means that a trait can be defined in several places or files and when they are used by clients, all elements defined in those separate places will be applied to clients. The benefits achieved is like the one described for classes in Section 2.1.3.

Listing 12. Using traits with mixings

a

```
1  trait T1{
2    void method1();
3    void method2(){/*impl… */ }
4  }
5  trait T1{
6    void method3(){/*impl… */ }
7  }
8  class C1{
9    isA T1;
10   void method1(){/*impl… */ }
11 }
```

b

```
trait T1{
  void method1();
  void method2(){/*impl… */ }
  void method3(){/*impl… */ }
}
class C1{
  isA T1;
  void method1(){/*impl… */ }
}
```

For example, Listing 12.a depicts two definitions for trait *T1* (lines 1 and 5). Class *C1* uses trait *T1* and implements the required method *method1()* and also obtains two provided methods *method2()* and *method3()*. Listing 12.b shows the same model as defined in Listing 12.a without any mixin.

3.4.5 Traits and Abstract Classes

Classes in Umple can be abstract, therefore, they can have completely abstract methods like interfaces can (although abstract classes can also have concrete methods). In order to leverage this case, it is allowed for abstract classes to use traits without satisfying their required methods. If this is the case, the required methods of traits will be considered as abstract methods for the abstract class. Then, all concrete subclasses of the abstract class are required to implement those abstract methods. This process ensures that required methods will finally be implemented. It is worth noting that interfaces cannot use traits because they cannot have concrete methods in their definitions.

Listing 13 shows an example in which class *C1* is abstract (line 6) and uses trait *T1*. It does not have an implementation for required method *method1()* of trait *T1*. Therefore, that required method becomes an abstract method for class *C1*. Class *C1* now has two abstract methods *method1()* and *method3()*. It also has concrete method *method2()* coming from trait *T1*. Class *C2* is a subclass of class *C1* and has to provide an implementation for abstract methods of class *C1* (lines 12 and 13).

Listing 13. Using traits through abstract classes

```
1  trait T1{
2    void method1();
3    void method2(){/*implementation*/}
4  }
5  class C1{
6    abstract;
7    isA T1;
8    abstract void method3();
9  }
10 class C2{
11   isA C1;
12   void method1(){/*implementation*/}
13   void method3(){/*implementation*/}
14 }
```

3.4.6 Flattening of Traits

A system modeled with traits can be transformed to a compatible model without traits. This model is useful when it is required to understand what provided methods are available in each final client. This is accomplished through flattening. There are two ways to represent a flat model of the system: graphical and textual. These flat models are generated automatically. In fact, modelers can toggle between the trait and flat model for better understanding of their models.

Listing 14 shows the flattened textual model of the model in Listing 8. As seen, there is no definition for traits in the model and classes have all functionality they needed from traits as their local methods. Figure 12 depicts the corresponding flattened graphical model.

The model in Listing 14 can motivate the question why a modeler should not be able to just design this model. The main reason is that, for example, method *method3()* might be used in several places. In each place, developers need to implement the method again. If there is a bug in the implementation of the method, developers need to apply a patch to all those places. Having the method defined a trait helps avoid the pitfall and reach the required reusability.

Listing 14. The flattened model related to the model in Listing 8

```
1  class C1 {
2    void method3() { /*implementation*/ }
3  }
4  class C2 {
5    isA C1;
6    void method2() { /*implementation*/ }
7    void method4() { /*implementation*/ }
8    void method1() { /*implementation*/ }
9  }
```

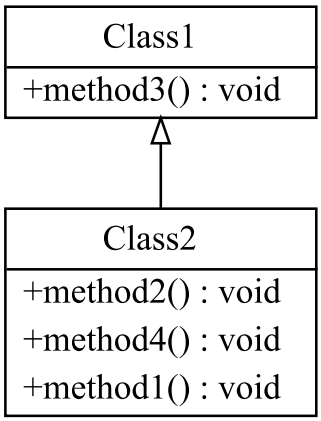


Figure 12. The graphical representation of the Umple model in Listing 14

3.4.7 Exploring Traits and their Flattened models

Traits can be explored textually in Umple; others have already proposed a graphical representation for them [93]. We wanted to generate such a graphical representation automatically for Umple. In order to achieve this, we developed a diagram generator with Graphviz [103] which represents traits, classes, and interfaces all together under one diagram.

Traits are flattened to clients when they are used by clients, so it would also be beneficial to be able to instantly switch between a view with traits and a flattened view. This could help modelers to understand how traits are represented in any object-oriented target languages, as well as the implications of the traits to the generated system. In order to enable such a feature, we needed to represent flattened models graphically. Umple already had a class diagram generator based on Graphviz so we simply reused this.

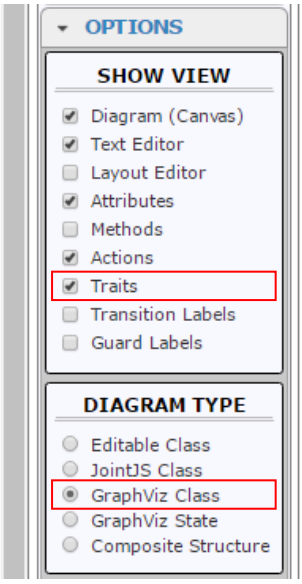


Figure 13. How to switch between different views for traits

Figure 13 shows the middle menu of UmpleOnline. In order to have traits represented graphically, first the menu OPTIONS must be selected. Then, sub menus Graphviz Class and Traits must be selected for DIAGRAM TYPE and SHOW VIEW respectively (as highlighted in the figure). In order to switch to the flattened model, modelers simply need to deselect Traits in SHOW VIEW. These two views can also be generated through using the Umple command line interface and the Umple Eclipse Plugin.

3.4.8 Definition of Attributes in Traits

Attributes in traits are defined in the same way they are defined for classes. Traits also support all modifiers that can be applied to attributes. Listing 15 gives an example of the way attributes can be defined and used in traits. As discussed earlier, these kinds of traits are called stateful.

As seen in the listing, there is a trait named Identifiable that has five attributes: *firstName*, *lastName*, *address*, *phoneNumber*, and *fullName* (lines 2-6). It also has a provided method named *isLongName*() (line 7). There are no required methods because the trait offers pure functionality to its clients. Class *Person* uses trait *Identifiable* and obtains the provided method and defined attributes from the trait. Class *Company* also uses the trait *Identifiable* and extends class *Organization*. It obtains both attributes coming from its superclass and

used trait. Figure 14 shows the graphical representation for the model in Listing 15. Attributes are presented in the same manner they are presented for classes. Figure 15 depicts the flattened model for the model in Listing 15. All attributes are flattened similar to the way provided methods are flattened.

Listing 15. An example representing how attributes are defined and used in traits

```
1  trait Identifiable {
2    firstName;
3    lastName;
4    address;
5    phoneNumber;
6    fullName = {firstName + " " + lastName}
7    Boolean isLongName() {return lastName.length() > 1;}
8  }
9  class Person {
10   isA Identifiable;
11 }
12 class Organization {
13   Integer registrationNumber;
14 }
15 class Company {
16   isA Organization, Identifiable;
17 }
```

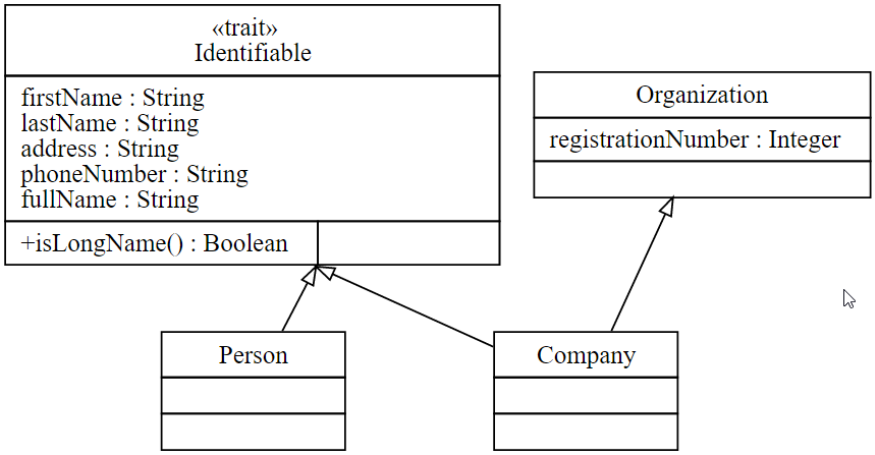


Figure 14. The graphical representation of the Umple model in Listing 15

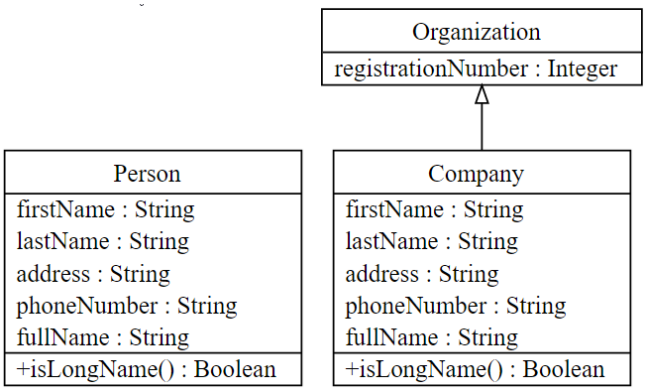


Figure 15. The flattened model for the Umple model in Listing 15

When clients use traits, a name conflict might happen because a client might have an attribute and obtain a new attribute with the same name from a trait. Modelers are responsible to resolve the conflict. The conflict is detected automatically and is presented to the user with warning code 218.

Unlike conflicts with other elements in traits that we will describe later, in the current implementation of our work, there is no operator to resolve an attribute name clash conflict. The main reason for not having this operator is that attributes can be used in the body of provided methods and so any operator would need to be applied in those places too. The current version of Umple does not analyse the semantics of provided methods, therefore, there is no information at the modeling level to show in which provided methods those attributes have been used. If Umple is later extended in this respect, we would then be able to add suitable operators to deal with conflicts among attributes in traits. Our current recommendation is to change the name of attributes in the clients of traits and avoid changing names in traits because this could break other clients of those traits.

Another way to avoid conflicts is to use stateless traits. In that case, traits use required methods to have access to states (attributes). Listing 16 shows an example in which trait *T1* uses two required methods *getData()* and *setData(Integer)* to have access and write to the attribute *data* of type Integer (lines 2 and 3). It also has a provided method that uses the method *getData()* to obtain the value of the attribute *data*. Then, it performs some operation on it (in this case adding 2 to the value, line 6) and finally uses the method *setData(Integer)* to save the new value into the attribute *data* (line 7). The class *C1* uses trait *T1* and has the attribute *data*. Since Umple automatically generates accessor and mutators for

attributes (more information described in Section 2.1.1.3), they satisfy the required methods defined in trait *T1* and so there is no need to implement them manually in class *C1*.

Listing 16. Using required methods to obtain states

```
1  trait T1{
2    Integer getData();
3    void setData(Integer);
4    Integer processData(){
5      int data = getData();
6      data=data+2;
7      setData(data);
8    }
9  }
10 class C1{
11   Integer data;
12   isA T1;
13 }
```

It is worth mentioning that although Umple supports stateful traits, we recommend using stateless traits and obtain access to states through required methods. However, Umple allows stateful traits so they can be used when it would result in simpler designs.

3.4.9 Template Parameters

Template parameters at the modeling level are combined with other modeling elements like associations (described as we progress). This combination increases modularity and reusability to an extent that is not achievable at the implementation level.

Template parameters can be referred to in required and provided methods and attributes. Traits can have template parameters with generic or primitive data types. As mentioned in Section 2.1.1.2, primitive types include Integer, Float, String, and so forth. Generic types include classes and interfaces. The difference in their use is that it is possible to put restrictions on bound types of generically-typed parameters. Such restrictions might include a declaration that the interfaces or classes must be extended or implemented by other specific classes. These restrictions are only available for generic-type parameters because primitive types cannot implement or extend any other type and so there is no way of imposing such constraints on them.

Listing 17. An example of traits without template parameters

```
1  trait T1{
2    abstract C1 method2(C1 data);
3    String method3(C1 data) {/*implementation is unique for T1*/ }
4  }
5  trait T2{
6    abstract C2 method2(C2 data);
7    String method3(C2 data) {/*implementation is unique for T2*/ }
8  }
9  interface I1{
10   void method1();
11 }
12 class C1{
13   isA I1;
14   isA T1;
15   void method1(){/*implementation is unique for C1*/}
16   C1 method2(C1 data){ /*implementation is unique for C1*/ }
17 }
18 class C2{
19   isA I1;
20   isA T2;
21   void method1(){/*implementation is unique for C2*/}
22   C2 method2(C2 data){ /*implementation is unique for C2*/ }
23 }
```

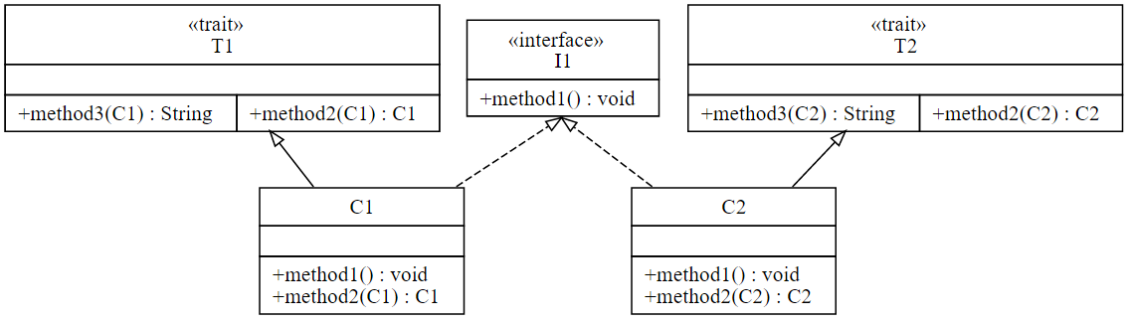


Figure 16. The graphical representation of the Umple model in Listing 17

Listing 17 and Figure 16 show an example in which template parameters have not been used. In Listing 17, there are two traits *T1* and *T2* each having a required method called *method2()* but with the different return and parameter types. Their provided methods have the same logic but operating on different implementations of interface *I1* (*C1* and *C2*) through the parameter "*data*". In the absence of template parameters, the design is required to have two traits in this case. In the following, we describe how template parameters are defined and can tackle this redundancy.

Template parameters are defined after the name of a trait inside a pair of angle brackets. Each parameter has a name and they are separated by a comma. Restrictions on the templates applied in the same manner Umple allows extending and implementing interfaces and classes respectively. In other words, the keyword *isA* is used after the name of a template parameter followed by the name of interfaces or a class. If there are more than one interface or one interface and one class, they are separated by the symbol &. Listing 18 shows the grammar related to the definition of template parameters in traits.

Listing 18. The grammar related to the definition of template parameters

```
1  traitParameters: < [[traitFullParameters]] (,[[traitFullParameters]] )* >
2  traitFullParameters : [~parameter] ([[traitParametersInterface]])?
3  traitParametersInterface- : isA [~tInterface]( & [~tInterface])*
```

When clients use traits, they must bind types to their parameters and also types must satisfy their restrictions. Values are bound through the symbol =, in the manner values are generally assigned to attributes. The bindings are performed inside a pair of angle brackets, each one separated by a comma. Therefore, when a client uses a trait with template parameters, they need to extend the normal way of using traits by having angle brackets appearing after the name of traits. Listing 11 showed the grammar related to binding values to template parameters when they are used by clients.

Listing 19 shows how the model in Listing 17 can be remodeled based on template parameters. Listing 19 depicts a trait called *T1* (line 1) with one template parameter named *TP*. The template parameter is restricted to implement the interface *I1*, defined in line 5. The restriction is applied to make sure a correct type will be bound to the template parameter. Trait *T1* has a required method named *method2()* with a return value and a parameter of type *TP* (which is a template parameter). Furthermore, it has a provided method named *method3()* with a parameter of type *TP*.

Class *C1* defined in line 8 uses trait *T1* and assigns class *C1* as a binding type to *TP*. Since class *C1* has already implemented interface *I1* (line 9), the type is acceptable. Class *C1* needs to implement the required method of trait *T1*, but it must be performed based on the type assigned to the template parameter *TP*. Therefore, class *C1* implements *method2()* with

the correct data type (which is *C1*) for the parameter and return types. Line 12 shows the exact signature of the implemented method.

Listing 19. An example shows how template parameters are defined and used

```
1  trait T1< TP isA I1 > {
2    abstract TP method2(TP data);
3    String method3(TP data){ /*implementation*/ }
4  }
5  interface I1{
6    void method1();
7  }
8  class C1{
9    isA I1;
10   isA T1<TP = C1>;
11   void method1(){/*implementation*/}
12   C1 method2(C1 data){ /*implementation*/ }
13 }
14 class C2{
15   isA I1;
16   isA T1< TP = C2 >;
17   void method1(){/*implementation*/}
18   C2 method2(C2 data){ /*implementation*/ }
19 }
```

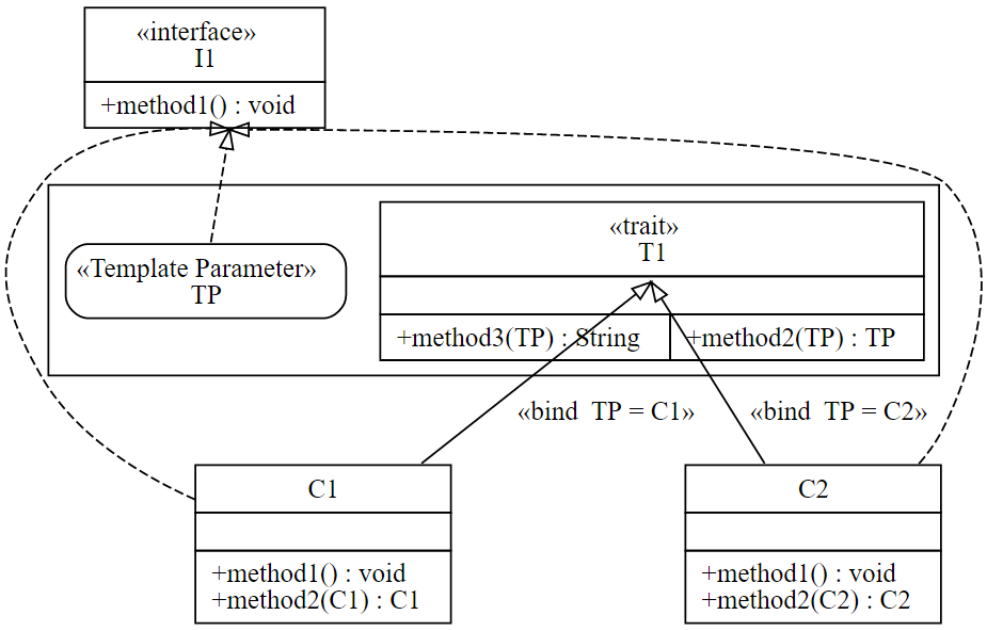


Figure 17. The graphical representation of the Umple model in Listing 19

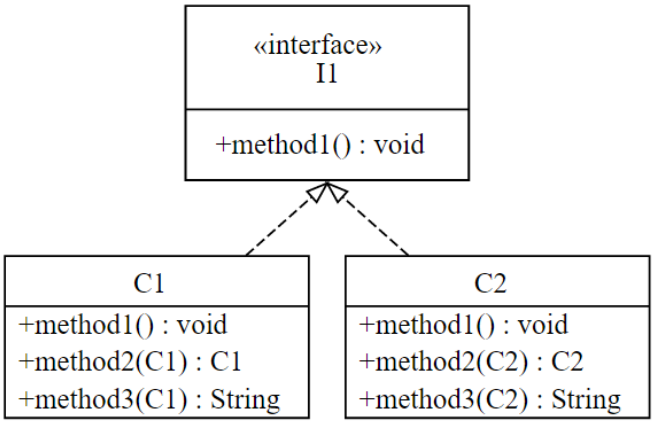


Figure 18. The flattened model for the Umple model in Listing 19

Figure 17 presents how template parameters and their bindings are shown graphically. When a trait has template parameters, they are annotated with «Template Parameter». The same kind of annotation is used to specify what kinds of restriction each template parameter has. When a client binds a value to a template parameter, the corresponding arrow is annotated with «bind» along with the name of the template parameter and its bound type.

Figure 18 depicts the flattened model related to the model in Listing 19. As seen, classes *Cl* and *C2* have obtained the same provided methods but with different data types based on the binding types. Using template parameters reduces the number of traits required to be implemented and therefore allow to have more general traits.

3.4.9.1 Nested template parameters

Since a trait can use other traits, a trait with template parameters should also be able to use other traits with template parameters. Therefore, it should also be possible to bind a template parameter to another template parameter to achieve better flexibility. This is performed in the same manner a type is bound to a template parameter. Furthermore, template parameters are modulated for each trait so it is possible for a trait with template parameters to use other traits with the same names for their template parameters.

Listing 20 gives an example in which one template parameter is bound to another one. Trait *T1* has the template parameter *TP* used to define the type of parameter for the provided method *method2()*. Trait *T2* has the template parameter *TP* used to define the type of

parameter for the provided method *method3()*. It also uses trait *T1* and binds its own template parameter to traits T1's template parameter (line 6). Note that trait *T2* can also bind any other type to the template parameter *TP*, if it is required. Finally, class *C1* uses trait *T2* and binds String to the template parameter TP (line 10).

Listing 20. An example of nested template parameters

```
1  trait T1<TP>{
2    void method1();
3    void method2(TP data) {/*implementation*/}
4  }
5  trait T2<TP>{
6    isA T1< TP = TP >;
7    void method3(TP Data) {/*implementation*/}
8  }
9  class C1{
10   isA T2< TP = String >;
11   void method1(){/*implementation*/}
12 }
```

3.4.9.2 Validity of template parameters and their constraints

There are several cases that can stop clients from using traits with template parameters or traits from defining template parameters. These cases are checked automatically by the Umple compiler and a proper error is raised for each case. These case are as follows:

a) If traits define template parameters, clients of those traits need to bind types to them. Otherwise, error code 219 is raised. For instance, in Listing 21.a class *C2* uses trait *T1* but it fails to bind a type to the template parameter *TP*.

b) If clients bind types to the template parameters of used traits and the types are not available in the system under design, error code 221 is raised. For example, in Listing 21.b class *C2* uses trait *T1* and binds class *C1* to the template parameter *TP*, but class C1 does not exist.

c) If traits define template parameters and put constraints on them, the classes and interfaces involved in the constraints must exist in the system under design, otherwise, error code 223 is raised. For instance, in Listing 21.c trait *T1* has the template parameter *TP* with the restriction of implementing the interface *I*, but interface *I* does not exist.

d) If clients bind types to the template parameters of used traits and the types exist but they do not satisfy the constraints of the template parameters, error codes 206 and 225 are raised for the class and interfaces respectively. For example, in Listing 21.d, class *C* uses trait *T1* and binds the class *C1* to the template parameter *TP*, but the class *C1* does not implement the interface *I*.

e) If a multiple inheritance is applied as a constraint to a template parameter, it is detected and error code 224 is raised. Umple does not support multiple inheritance. For example, in Listing 21.e, trait *T1* tries to constrain the type of template parameter *TP* to be a subclass of *C1* and *C2*.

f) When template parameters used for attributes and several traits are used by a client, the composition of traits should not cause a conflict in types and attribute names. For example, in Listing 21.f, class *C1* uses two traits *T1* and *T2*, which bring two attributes named *name* with two different types, Integer and String. The compiler raises error code 217 for this case.

g) Two different types cannot be given to a template parameter and the detection of this is reflected by error code 216. For example, in Listing 21.g, class *C* uses trait *T*, but it cannot bind two types Integer and String to template parameter TP.

h) The name of template parameters must be unique in the definition of each trait. For example, in Listing 21.h, trait *T* cannot have two template parameters with the name *TP*. Breaking this rule results in error code 214.

i) Clients must always bind types to the correct template parameter of a trait, otherwise, error code 215 is raised. For example, in Listing 21.i, class *C* uses trait *T* but it cannot bind String to template parameter *TP1* that is not defined for trait *T*.

Listing 21. The case in which template parameters can not be used or defined

a

```
trait T1<TP>{/*implementation*/}
class C2{
  isA T1;
  /*implementation*/
}
```

b

```
trait T1<TP>{/*implementation*/}
class C2{
  isA T1<TP=C1>;
  /*implementation*/
}
```

c

```
trait T1<TP isA I>{
  /*implementation*/
}
class C1{/*implementation*/}
class C{
  isA T1< TP = C1 >;
  /*implementation*/
}
```

d

```
trait T1<TP isA I>{
  /*implementation*/
}
interface I{/*implementation*/}
class C1{/*implementation*/}
class C{
  isA T1< TP = C1 >;
  /*implementation*/
}
```

e

```
trait T1<TP isA C1&C2>{
  /*implementation*/
}
class C1{ /*implementation*/ }
class C2{ /*implementation*/ }
```

f

```
trait T<TP>{  TP name;  }
trait T1{ isA T< TP = Integer >;}
trait T2{ isA T< TP = String >;}
class C1{ isA T1,T2;}
```

g

```
trait T<TP>{ TP name;  }
class C{
  isA T< TP = Integer , TP= String>;
}
```

h

```
trait T<TP,TP>{ TP name;  }
class C{
  isA T< TP =Integer >;
}
```

i

```
trait T<TP>{ TP name;  }
class C{
  isA T< TP = Integer , TP1 = String >;
}
```

3.4.9.3 Template parameters in code blocks

Since Umple supports using programming languages to implement the body of methods, it is beneficial to allow template parameters to be used in collaboration with them. The current implementation of Umple does not involve parsing the code blocks, since Umple supports

many languages in code blocks and those languages constantly evolve. As a result, detecting template parameters and replacing them with the binding types in places that are syntactically and semantically correct poses challenges. In order to tackle this issue, Umple introduces a special syntax to allow having template parameters that can be substituted in code blocks.

Listing 22. An example model of using template parameters in code blocks.

```
1  trait T1 <TP>{
2    String method1();
3    String method2(){
4      #TP# instance = new #TP#();
5      return method1() +":"+instance.process();
6    }
7  }
8  class C1{
9    String process(){/*implementation*/}
10 }
11 class C2{
12   isA T1< TP = C1 >;
13   String method1(){/*implementation*/ }
14 }
```

Listing 23. The generated Java method for the template parameter described in Listing 22

```
1  public String method2(){
2      C1 instance = new C1();
3      return method1() + ":" +instance.process();
4  }
```

For template parameters to operate on code in code blocks, they need to be encompassed within a pair of the symbol #. This ensures that the dedicated Umple scanner (i.e., not a full code parser) will be able to detect strings matching the template parameters correctly and replace them with binding values.

Listing 22 shows an example in which a template parameter is used in the body of a method. As seen, trait *T1* has a template parameter named *TP* (line 1). The provided method *method2()* needs to return a string which is a combination of calling the required method *method1()* and a method named *process()* from the template parameter *TP*. In the body of the provided method *method2()*, an instance of the template parameter needs to be created so as to call the method *process()*. This is achieved by having the name of template parameter en-

compassed by # in places that require types (line 4). The rest needs to follow the syntax of the language used to implement the code blocks, in this case, Java.

Class *C2* uses trait *T1* and binds class *C1* to the template parameter *TP* (line 12). It also implements the required method *method1()* in order to use the trait. Listing 23 depicts the method *method2()* in Java generated for the class *C2*.

3.4.10 Recognized Conflicts

As mentioned earlier, using traits is not always a straightforward mechanism and sometimes there are conflicts. Provided methods are the main reasons for conflicts when they appear in clients, derived from different traits and hierarchy levels. If a method with the same signature comes to a client from two different traits, it is considered as a conflict that must be resolved. However, if the method comes from two different traits but with a common source (i.e., both different traits use a common third trait), it is not considered as a conflict. Conflicts are detected in our implementation automatically.

Listing 24. The conflict arising from provided methods but different implementations

```
1  trait T1{
2    void method1(){/*implementation related to T1*/ }
3  }
4  trait T2{
5    void method1(){/*implementation related to T2*/ }
6  }
7  class C1{
8    isA T1;
9    isA T2;
10 }
```

Listing 24 describes the first case of conflict. As seen, trait *T1* has a provided method named *method1()* with its unique implementation (line 2) and trait *T2* also has a provided method with the same signature but different implementation (line 5). Class *C1* uses traits *T1* and *T2* and therefore there is a conflict regarding which provided method should be accepted. The Umple compiler automatically detects this situation under the error code 210.

Listing 25 illustrates an example of two cases, in which one of them results in a conflict and another does not. Class *C1* uses traits *T2* and *T3* and it hence will get two provided methods – *method1()* and *method2()*– twice, coming from two different traits. There is no

conflict for *method1()* because *T2* and *T1* get the same method from a common source (trait *T1*). However, this is not the case for the *method2()* because this method has been overridden by trait *T3* (line 10). In other words, the sources are now not the same. Therefore, in class *C1* there is a conflict. It is important to note that there is no graphical representation for this example because Umple detects the conflict (under the error code 210) and does not allow generation of an inconsistent diagram.

Listing 25. The conflict arising from provided methods in different levels

```
1  trait T1{
2    void method1(){/*implementation related to T1*/ }
3    void method2(){/*implementation related to T1*/ }
4  }
5  trait T2{
6    isA T1;
7  }
8  trait T3{
9    isA T1;
10   void method2(){/*implementation related to T3*/ }
11 }
12 class C1{
13   isA T2;
14   isA T3;
15 }
```

3.4.11 Operators

Operators are capabilities that allow resolving the conflicts and also managing the granularity of traits. Operators are applied to traits when traits are used by clients. Clients can apply more than one operator to a specific trait, but those operators need to be compatible with each other. There is no sequence in the way operators are applied. Operators are defined inside angle brackets after the name of traits and can be mixed with binding types to template parameters. The general structure by which a client uses operators is as follows.

```
isA TName <Operator₁, Operator₂, …, Operatorₙ>
```

TName specifies the name of a trait to which each $Operator_{i,i=1..n}$ is applied. There is no limitation on the number of operations that can be applied to a trait. Two operators with the same semantics (such as removing and renaming) can only be applied to different elements. Errors resulting from violations such rules are detected and reported to the modeler.

Note that all processes related to the flattening and composition algorithm (described later) are performed after applying the operators.

In the discussion that follows here and in Section 3.5.4 (operators on state machines), the syntax and semantics of the operators are described in detail. There may appear to be a lot of details for the user to comprehend. However, all the operators follow a pattern, and the semantics has been designed to be consistent with operators on other modeling elements.

If traits have template parameters and those template parameters have been used as types in provided methods (or event of state machines), their types do not affect the signature of provided methods referred to by operators. In other words, types of template parameters are applied to traits *after* operators are applied.

The identification factor for selecting a provided method is its signature. However, the return type of provided methods is not used for identification because the name and list of types of parameters can uniquely differentiate each provided method from others. When parameters of provided methods are specified, there is no need to define the name of parameters.

3.4.11.1 Removing/keeping provided methods

This operator allows removing or keeping provided methods of a trait. The syntax for this operator is as follows:

```
(+|-) methodName(argumentTypes)
```

The symbol – indicates removing while the symbol + indicates keeping. The symbol must precede the signature of the provided method. When the symbol – is applied to a provided method of a trait, it removes the method from the set of provided methods. However, when the symbol + is applied to a provided method of a trait, the provided method is kept and the rest are removed. If a method signature is specified in this operator and it is not available in the used trait, the Umple compiler raises error code 212. The Umple compiler also detects if this operator was applied more than once on a specific method and raises error code 211 in that case. Listing 26 shows the Umple grammar related to this operator.

Listing 26. The grammar related to operator removing/keeping

```
1  iEFunction: [=modifier:+|-] [~methodName] [[iEParameterList]]
2  iEParameterList: OPEN_ROUND_BRACKET
     ( [parameter] ( , [parameter] )* )? CLOSE_ROUND_BRACKET
```

For instance, Listing 27 depicts an example in which class *C1* removes the provided methods *method2()* and *method3()* (line 9) while class *C2* keeps only the provided method *method5()*, coming from trait *T1* (line 13). Figure 19 shows the flattened class diagram for Listing 27. As seen, class *C1* obtains two provided methods *method4()* and *method5()* in addition to the *method1()* which satisfies the required methods of trait *T1*. Class *C2* just has the provided method *method5()* in addition to *method1()*;

Listing 27. An example that shows how to control granularity of traits

```
1  trait T1{
2    abstract method1();
3    void method2(){/*implementation*/}
4    void method3(){/*implementation*/}
5    void method4(){/*implementation*/}
6    void method5(){/*implementation*/}
7  }
8  class C1{
9    isA T1< -method2() , -method3()>;
10   void method1() {/*implementation related to C1*/}
11 }
12 class C2{
13   isA T1< +method5() >;
14   void method1() {/*implementation related to C2*/}
15 }
```

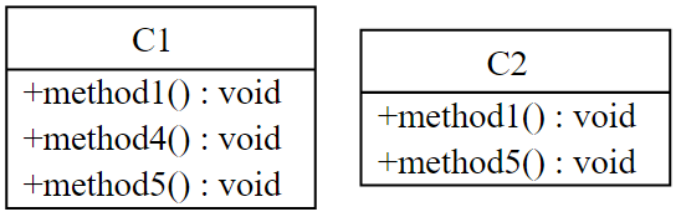


Figure 19. The flattened model for the Umple model in Listing 27

In Section 3.4.10, two cases that could cause conflicts were described. Listing 28 shows how the removing operator " - " can be applied to resolve the conflict in Listing 24. There are two options to apply the operator. The first one is to remove provided method

method1() from trait *T1*, modeled in Listing 28.a line 12. The second option is to remove provided method *method1()* from trait *T2*, modeled in Listing 28.b line 13. The selection of options depends on the specific needs of the particular design. The developer needs to investigate which method satisfies the requirements of class *C1*. In most cases, it is clear from the functionality the provided method offers.

Listing 28. Resolving the conflict in Listing 24 by the removing operator

a

```
1  trait T1{
2    void method1(){
3      /*impl.. related to T1*/
4    }
5  }
6  trait T2{
7    void method1(){
8      /*impl… related to T2*/
9    }
10 }
11 class C1{
12   isA T1< -method1() >;
13   isA T2;
14 }
```

b

```
trait T1{
  void method1(){
    /*impl.. related to T1*/
  }
}
trait T2{
  void method1(){
    /*impl… related to T2*/
  }
}
class C1{
  isA T1;
  isA T2< -method1() >;
}
```

Listing 29. Resolving the conflict in Listing 25 by removing/keeping operator

a

```
1  trait T1{
2    void method1(){
3      /*impl.. related to T1*/
4    }
5    void method2(){
6      /*impl.. related to T1*/
7    }
8  }
9  trait T2{
10   isA T1;
11 }
12 trait T3{
13   isA T1;
14   void method2(){
15     /*impl.. related to T3*/
16   }
17 }
18 class C1{
19   isA T2;
20   isA T3< -method2() >;
21 }
```

b

```
trait T1{
  void method1(){
    /*impl.. related to T1*/
  }
  void method2(){
    /*impl.. related to T1*/
  }
}
trait T2{
  isA T1< -method2() >;
}
trait T3{
  isA T1;
  void method2(){
    /*impl.. related to T3*/
  }
}
class C1{
  isA T2;
  isA T3;
}
```

Listing 29 shows how the operator should be applied to the model in Listing 25 to resolve the conflict. In this case, again, Umple developers are responsible for deciding which method needs to be removed. Listing 29.a models the case in which the provided method *method2()* is removed from trait *T3* (line 20) while Listing 29.b removes the provided method *method2()* from trait *T1* (line 10).

Please consider that in the case of having many conflicts, the keeping operator (+) can also be used for the same result. Multiple keeping operators can be used together to keep several methods and throw away the rest.

3.4.11.2 Renaming (Aliasing)

The renaming operator allows changing the name of provided methods and also their visibilities. This operator can also be mixed partially with the keeping operator to provide better flexibility. The operator does not allow changing the types of parameters or number of parameters. The reason is that the provided methods might be used by other provided methods and so any change regarding types and numbers can break traits.

The current implementation of the operator does not support renaming recursive methods. The reason for this is that the current version of Umple does not support parsing the body of methods. Therefore, if a method is used in the body of other methods, we cannot renamed it at the modeling level. Being able to parse the body of methods will allow this operator to be applied to recursive methods as well.

When a provided method is referred in the operator, it must exist in the trait, otherwise, the Umple compiler raises error code 212. Furthermore, the new name of the provided method must be unique in the list of provided methods, otherwise, error code 220 is raised. The syntax for this operator is as follows:

```
(+) methodName(argumentTypes) as newName
```

Listing 30 shows the Umple grammar related to this operator. This operator is a good candidate for resolving conflicts because it allows clients to have access to both conflicting provided methods.

Listing 30. The grammar related to operator renaming

```
1  functionAliasName: [=modifier:+]? [~methodName]
      [[iEParameterList]] as [[IEVisibilityAlias]]
2  IEVisibilityAlias-: ([[IEVisibility]] [~aliasName]?) | ([~aliasName])
3  IEVisibility-: [=iEVisibility:public|private|protected]
```

Listing 31. An example that shows how to customize vocabulary of traits

```
1  trait T1{
2    abstract method1();
3    void method2(){/*implementation*/}
4    void method3(){/*implementation*/}
5    void method4(){/*implementation*/}
6    void method5(Integer data){/* implementation*/}
7  }
8  class C1{
9    isA T1< method2() as function2 >;
10   void method1() {/*implementation related to C1*/}
11 }
12 class C2{
13   isA T1< method3() as private function3 >;
14   void method1() {/*implementation related to C2*/}
15 }
16 class C3{
17   isA T1< +method5(Integer) as function5 >;
18   void method1() {/*implementation related to C3*/}
19 }
```

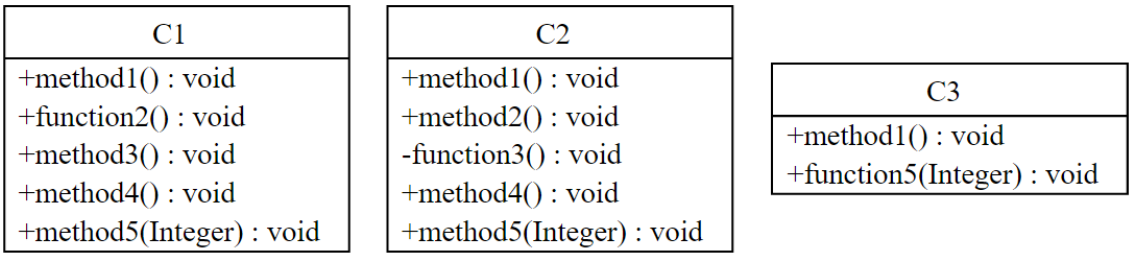


Figure 20. The flattened model for the Umple model in Listing 31

For example, Listing 31 shows this operator in action. Class *C1* uses trait *T1* (line 9) and renames its provided method *method1()* to *function2*. There is no need to specify paren-

theses for the new name. Class *C2* uses trait *T1* (line 13) and while it renames provided *method3()* to *function3*, it also changes the visibility of the provided method to be private. Finally, class *C3* uses trait *T1* and renames the provided method *method5(Integer)* to *function5*. However, it also forces other provided methods of trait *T1* to be removed. This feature can be really useful if there is a utility trait and we are just interested in a provided method with the name that suits our domain. Figure 20 presents a class diagram in which the final result of applying operators to traits can be seen.

Now, we want to see how the operator can be used to resolve the conflicts described in Section 3.4.10. Listing 32 shows two different ways to resolve conflicts. Listing 32.a shows how the provided method *method1()* of trait *T1* is renamed to *func1* (line 12) while Listing 32.b presents how the provided method *method1()* of trait *T2* is renamed to func2 (line 13). As seen, the operator allows having access to both conflicting methods in clients. This is not possible with the removing operator.

Listing 32. Resolving the conflict in Listing 24 by renaming operator

a

```
1  trait T1{
2    void method1(){
3      /*impl.. related to T1*/
4    }
5  }
6  trait T2{
7    void method1(){
8      /*impl… related to T2*/
9    }
10 }
11 class C1{
12   isA T1<method1() as func1>;
13   isA T2;
14 }
```

b

```
trait T1{
  void method1(){
    /*impl.. related to T1*/
  }
}
trait T2{
  void method1(){
    /*impl… related to T2*/
  }
}
class C1{
  isA T1;
  isA T2<method1() as func2>;
}
```

Listing 33 shows how the operator is used to deal with the conflict in Listing 25. In Listing 25.a, class *C1* changes the name of provided method *method2()* of trait *T3* to *func3* (line 20). In Listing 25.b the provided method *method2()* of trait *T1* is changed to *func2* inside trait *T2* (line 10). Although the final method names of class *C1* in Listing 33.a and b are different, they are exactly the same regarding the functionality offered.

Listing 33. Resolving the conflict in Listing 25 by renaming operator

a

```
1  trait T1{
2    void method1(){
3      /*impl.. related to T1*/
4    }
5    void method2(){
6      /*impl.. related to T1*/
7    }
8  }
9  trait T2{
10   isA T1;
11 }
12 trait T3{
13   isA T1;
14   void method2(){
15     /*impl.. related to T3*/
16   }
17 }
18 class C1{
19   isA T2;
20   isA T3<method2()as func3>;
21 }
```

b

```
trait T1{
  void method1(){
    /*impl.. related to T1*/
  }
  void method2(){
    /*impl.. related to T1*/
  }
}
trait T2{
  isA T1<method2() as func2>;
}
trait T3{
  isA T1;
  void method2(){
    /*impl.. related to T3*/
  }
}
class C1{
  isA T2;
  isA T3;
}
```

Note that, it is not allowed to rename required methods because they have been used in the body of methods. Therefore, if we were allowed changing them, the implementation would be broken. This feature is, however, available in programming languages such as FRTJ [22] which have completely modular implementation for traits.

3.5. State Machines in Traits

A state machine (SM) describes the behavior of some system element, for example, the life cycle of an instance of a class. However, there are several challenges with the practical use of state machines: They can grow complex, it can be hard to compose them from reusable parts, and it can be hard to create slightly-varying versions as specializations of a general case, for example in a product line.

There is only a small amount of literature on building state machines from reusable component state machines, or on composition and generalization/specialization of such machines [16,57,72]. As described in the problem section (Section 1.1), current reuse and extension mechanisms for state machines are typically limited in terms of the following items:

- State machines are generally defined as elements of classes, so reuse and extension follow the rules and limitations of inheritance (subtyping).
- Composite states cannot be built based on already-defined state machines. In other words, composite states cannot be defined and reused separately.
- Although state machines in different domains may have isomorphic structures, the different vocabulary (e.g., different event and state names) in each domain prevents reuse.

In fact, it seems reasonable to expect that elements with similar behavior ought to be able to reuse the same state machine, especially if there was a parameterization mechanism to allow for small differences. It should similarly be possible to reuse behavior expressed as a state machine to build more complex behavior. In this section, we explain how state machines are defined in Umple traits and used in combination with state machines in other traits or classes.

3.5.1 Definition of State Machines in Traits

A trait can have zero or many state machines, each with a unique name. The definition of state machines in traits follows the same rules and constraints that exist for them in classes (described in Section 2.1.5).

Listing 34.a shows a trait called *T1* (lines 1-10) with two state machines *sm1* (lines 2-5) and *sm2* (lines 6-9). Use of trait *T1* in class *C1* results in class *C1* having those two state machines as native state machines. In general, if a class already has state machines with completely distinct names to those being introduced via traits, the introduced state machines are just 'flattened' into the class, i.e,. they are treated as though they were coded directly in the class.

Introduced machines that have names duplicating existing state machine names are composed (merged) with the existing machines, and the resulting composed machines are flattened into the class (described in Section 3.3.3). The same concept is applied when traits are used by other traits. Listing 34.b expresses the same model designed in Listing 34.a, in which class *C1* has two state machines, but it uses trait *T2* (line 15) to obtain the same state machines. Trait *T2* provides state machine *sm2* from its own definition (lines 9-12) and state

machine *sm1* from trait *T1* (line 8). This use of traits by other traits allows building more complex traits (and hence more complex state machines) from simpler ones (e.g., Figure 21).

Listing 34. State Machines in traits

a

```
1  trait T1 {
2    sm1{
3      s0 {e1-> s1;}
4      s1 {e0-> s0;}
5    }
6    sm2{
7      s0 {e1-> s1;}
8      s1 {e0-> s0;}
9    }
10 }
11 class C1 {
12   isA T1;
13 }
14
15
16
```

b

```
trait T1 {
  sm1{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
trait T2 {
  isA T1;
  sm2{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
class C1 {
  isA T2;
}
```

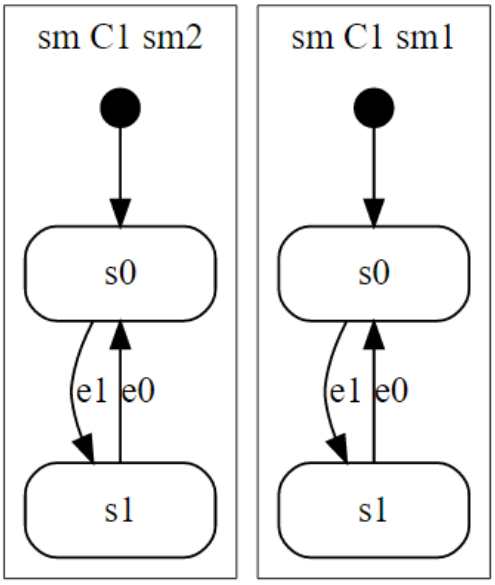


Figure 21. The diagram corresponding to Listing 34

3.5.2 The Relationship between State Machine, Provided, and Required Methods

As pointed out before, basic traits are composed of required and provided methods. Clients need to satisfy required methods in order to obtain benefits of provided methods. In other words, provided methods need those required methods to behave correctly. Therefore, modeling elements in traits should be served as provided functionality. State machines in traits clearly match this rule and are considered as provided functionality. More concretely, *any event in a state machine is considered as a provided method*, and so can satisfy the required methods of used traits.

Listing 35. Satisfaction of required methods through state machines

```
1  trait T1{
2    Boolean m1(String input);
3    Boolean m2();
4    sm1{
5      s1{ e1(String data) -> /{ m1(data); } s2;    }
6      s2{ e2 -> /{ m2(); } s1; }
7    }
8  }
9  class C1{
10   isA T1;
11   sm2{
12     s1{ m1(String str) -> s2;}
13     s2{ m2 ->  s1;}
14   }
15 }
```

State machines are supposed to encapsulate their own actions and guards so they can be reused as a piece of functionality. For example, a guard can be defined as a reference to an attribute in the trait or to a parameter of the event, so when the state machine is reused the attribute and parameter are reused as well. The same perspective is true for actions. In real world cases, state machines may need to obtain those conditions and actions from the context (e.g., clients) in which they are reused. State machines hence require those conditions and actions to behave correctly. This requirement is in alignment with the notion of required methods of traits. Therefore, required actions and guards of state machines can be expressed as required methods of traits. Therefore the required behavior of them can also be satisfied by other state machines in the clients.

It should be noted that if guards, actions, and activities of state machines are defined as methods (not inline code blocks directly in the state machine), they are not anymore required methods. They will be treated as provided methods, and traits' rules related to provided methods will be applied to them.

Listing 35 shows an example in which required methods are satisfied by events of state machines. As seen, trait *T1* has two required methods, *m1(String)* and *m2()*, called in actions of transitions *e1(string)* and *e2* in states *s1* and *s2* of state machine *sm1* respectively. Class *C1* uses trait *T1* and must satisfy those required methods. Class *C1* does not have any concrete method to satisfy the required methods, but it has state machine *sm2*. State machine *sm2* has two event *m1(String)* and *m2* which satisfy the required methods of trait *T1*.

If state machines are used to satisfy the required methods, there is a limitation in return types of required methods. All required methods must have Boolean as their return types, otherwise, events cannot satisfy them. The reason for this limitation is that all event automatically obtain Boolean as their return types by the Umple compiler.

Listing 36. Template parameters with state machines in traits

```
1  trait T1<TP>{
2    sm{
3      s1{ e1(TP p1)-> s2; }
4      s2{ e2(String p1, TP p2) -> s1; }
5    }
6  }
7  class C1{/*implementation*/}
8  class C2{
9    isA T1<TP=C1>;
10 }
```

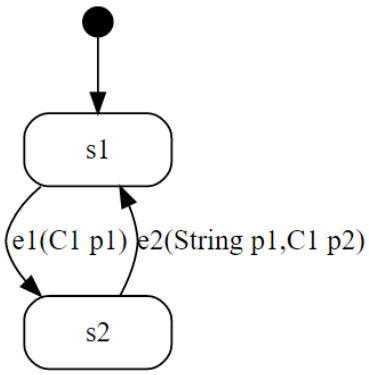


Figure 22. State machine diagram for the class C1 in Listing 36

3.5.3 Template Parameters in State Machines

Like the way template parameters defined in traits are used for required and provided methods, they can also be used in collaboration with state machines. Template parameters can be used to define the types of parameters for events. They can also be used in code blocks related to actions and activities, as described in Section 3.4.9.3.

Listing 36 illustrates an example of how template parameters can be used with events of state machines. State machine *sm* in trait *T1* has two events. Event *e1* has a parameter of type *TP* (line 3) and event *e2* has two parameters with types String and *TP* (line 4). Class *C2* uses trait *T1* (line 9) and binds class *C1* to the template parameters *TP*. The flattened state machine of class *C2* is depicted in Figure 22.

3.5.4 Operators

As with operators on methods described earlier (Section 3.4.11), certain operators can be applied to traits' state machines when they are used in clients. These provide mechanisms to improve flexibility, assign state machines to specific states, and resolve conflicts caused by name collisions. These operators follow the same structure defined for operators on methods.

3.5.4.1 Changing the name of a state machine

This operator is used to change the name of a state machine when it is to be reused by a client. This operator can also be mixed partially with the keeping operator to provide better flexibility. The syntax for this operator is as follow:

```
(+) stateMachineName as newName
```

When a state machine with a given name is specified by the renaming operator, it must be available in the trait being operated on, either directly in the trait or another trait used by the trait. Otherwise, the Umple compiler raises error code 230. Listing 37 shows how names of state machines represented in Listing 34 (parts a and b) can be changed to *mach1* and *mach2*. As seen, in Listing 37.a both state machines that are available directly in trait *T1* have been renamed. State machine *sm1* renamed in Listing 37.b is defined in trait *T1* but accessible in trait *T2* because trait *T2* uses trait *T1*. The changed model related to Listing 37 is depicted in Figure 23.

Listing 37. Changing the name of state machines in Listing 34

a

```
1 class C1 {
2  isA T1<sm1 as mach1, sm2 as mach2>;
3 }
```

b

```
class C1 {
 isA T2<sm1 as mach1, sm2 as mach2>;
}
```

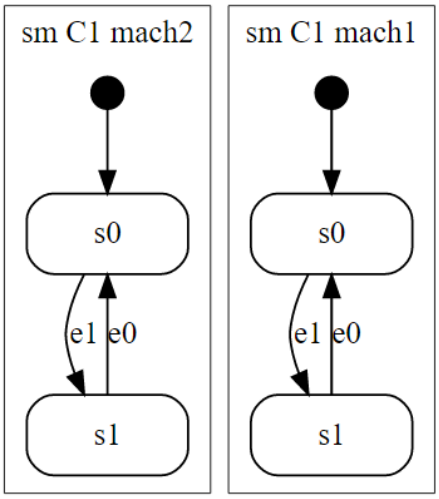


Figure 23. The diagram corresponding to modification in Listing 37

Listing 38 shows another example in which class *C1* uses the operator to rename the state machine *sm1* in trait *T1* in Listing 34 and also automatically remove other state machines, which are just state machine *sm2* in this case. The logic of this operator is exactly like the one described for renaming operator for methods in Section 3.4.11.2

Listing 38. Changing name of a state machine and removing others in Listing 34

```
1 class C1 {
2  isA T1< +sm1 as mach1 >;
3 }
```

There are two main scenarios for this operator. The first is merging: when a client already has a state machine and will obtain another state machine coming from the used trait. The two machines might have some of the same states but different functionality. The client wants to have all functionality *merged in one state machine* (described as we progress). In

this case, the operator is used to change the name of the incoming state machine from the trait, to *match* the name of the existing state machine. The result is that the new functionality will be merged into the existing state machine.

The second scenario is for avoiding conflicts. This occurs when a client has an existing state machine and wants to incorporate another one with different functionality, but that happens to have the same name. In this case, the client can change the name of the incoming state machine to be *different* from the existing state machine. This second scenario can be needed in various conflict-resolution scenarios.

As pointed out above, the state machine used in this operation needs to be accessible, otherwise, the Umple compiler raises error code 230 (e.g., Listing 39.a). Furthermore, the operator can be applied once on a specific state machine, otherwise, the Umple compiler raises the error code 290 (Listing 39.b).

Listing 39. Errors when changing improperly the name of state machines in Listing 34

a

```
1 class C1 {
2  isA T1<sm10 as mach1 >;
3 }
```

b

```
class C1 {
 isA T2<sm1 as mach1, sm1 as mach2>;
}
```

3.5.4.2 Changing the name of a state

This operator changes the name of a state *inside* a specific state machine. The operator covers both simple and composite states. The syntax used for this purpose is as follows:

```
stateMachineName.stateName.....stateName as newName
```

The state to be renamed is specified based on a series of names separated by dots, starting with the name of the state machine. If the state is a simple or composite state at the top level of a hierarchical state machine, then it comes directly after the name of the state machine. However, if it is deeper in the hierarchy, the chain of parents must also be specified. The last name in the series is always the name of the state to be renamed.

Listing 40 shows an example including a trait and class named *T1* and *C1*. Trait T1 has a state machine with a composite state named *s0* (line 3). Composite state *s0* has two internal states *s11* and *s12*. Class *C1* uses trait *T1* and changes the name of state *s0* to *state0* and the name of *s11* to *state11* (line 12). In order to specify the state *s0*, it is preceded by the

name of a state machine, which is *sm*. For state *s11*, the name of the state machine, the composite state, and the region name (implicitly called s0) precede it

Listing 40. An example that shows how to change the name of states

```
1  trait T1 {
2    sm{
3      s0{
4        e1-> s1;
5        s11{ e12-> s12; }
6        s12{ e11-> s11; }
7      }
8      s1{ e0-> s1;  }
9    }
10 }
11 class C1 {
12   isA T1<sm.s0 as state0, sm.s0.s0.s11 as state11>;
13 }
```

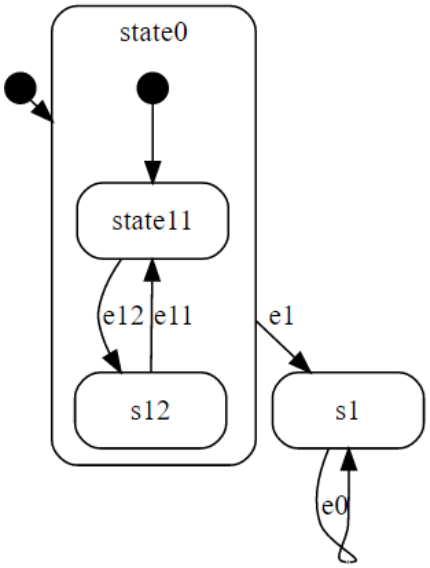


Figure 24. The diagram corresponding to modifications in Listing 40

In Umple, the name of the single region inside a composite state is set automatically to the name of the composite state. The operator applies also the same rule when it changes a composite state. Figure 24 shows the result of operators on the state machine in trait *T1* in Listing 40.

The first scenario for this operator is to change the vocabulary used for the names of states. This adds flexibility when a trait is specified in a generalized context and there is a

need to adapt names so as to be more domain-specific. For example, ‘tripEnded’ in a general transportation state machine becomes ‘landed’ in an adaptation to the airline domain, or ‘docked’ in an adaptation to the water transport domain.

The second scenario is when two states need to be merged, but they have different names. By changing the name of one so it matches the other, the algorithm knows how to merge them.

The third scenario occurs when there are two states in a state machine to be composed, but we want to keep those two states separate and prevent merging.

When a state is to be renamed, the state machine and state must be available in the trait, otherwise, the Umple compiler raises error code 230 (e.g., Listing 41.a). Furthermore, a state should not be renamed more than once. If this happens, the Umple compiler detects states and raises error code 229 (e.g., Listing 41.b).

Listing 41. Errors when changing improperly the name of state in Listing 40

a

```
1 class C1 {
2   isA T1<sm.m1 as state0>;
3 }
4
```

b

```
class C1 {
  isA T1<sm.s0 as state0,
         sm.s0 as state01>;
}
```

3.5.4.3 Changing the name of regions

This operator allows renaming a specified region. It is just like the operator used for changing the name of states. The difference is that the last name in the sequential series of names (separated by dots) is the name of a region to be changed. Regions in Umple (as in UML) have to be encapsulated in a state. If the region does not exist in the specified state, the Umple compiler raises error code 230. Furthermore, the new name must be unique in the list of available regions for the specified state, otherwise, the Umple compiler does not allow renaming and raises error code 237.

Since names of regions are set automatically by the Umple compiler and they are equal to the names of their initial states, renaming the name of a region must also be applied to its initial state. This is performed automatically by the operator. The applications for this operator are like those for changing the name of states. In particular, this operator is used when several regions are supposed to be merged or kept separate.

Listing 42. An example that shows how to rename a region

```
1  trait T1{
2    sm {
3      s1{
4        r1{ e1-> r11; }
5        r11{}
6        ||
7        r2{ e2-> r21; }
8        r21{}
9      }
10   }
11 }
12 class C1{
13   isA T1<sm.s1.r1 as region1,sm.s1.r2 as region2>;
14 }
```

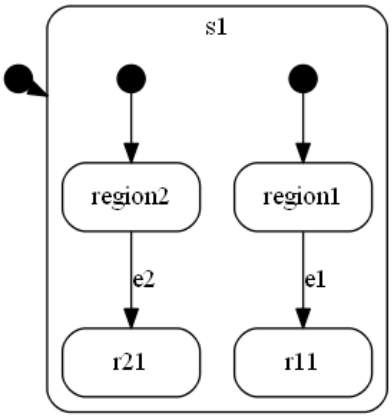


Figure 25. The diagram corresponding to modification in Listing 42

Listing 42 shows an example in which state machine *sm* in trait *T1* has a composite state *s1* with two regions *r1* and *r2*. Class *C1* uses trait *T1* and renames the name of those regions to *region1* and *region2* respectively (line 13). Figure 25 shows the result of the operations. As seen, now both initial states have the name of their regions.

3.5.4.4 Change the name of events

This operator is used to change the name of events that will trigger transitions in state machines. The syntax used for this operator is as follows:

```
(* | stateMachineName).eventName(argumentTypes) as new_Name
```

Through this operator, it is possible to rename an event related to a specific state machine or all state machines in a trait. For the first case, the modeler specifies the name of the state machine (*stateMachineName*). For the second case, an asterisk (*) is specified. The event name (*eventName*) must end with a pair of parentheses including any needed argument types. This operator does not allow changing the argument types because that would break the implementation of the event method. The operator is used mostly to change the event names based on a new domain's requirements. It can also be used to keep an event from being overwritten by the client's state machine and vice versa.

Listing 43. An example that shows how to change the name of states

```
1  trait T1 {
2    sm1{
3      s0 {e1(Integer index)-> s1;}
4      s1 {e0-> s0;}
5    }
6    sm2{
7      t0 {e1(Integer index)-> t1;}
8      t1 {e0-> t0;}
9    }
10 }
11 class C1 {
12   isA T1<sm1.e1(Integer) as event1, *.e0() as event0>;
13 }
```

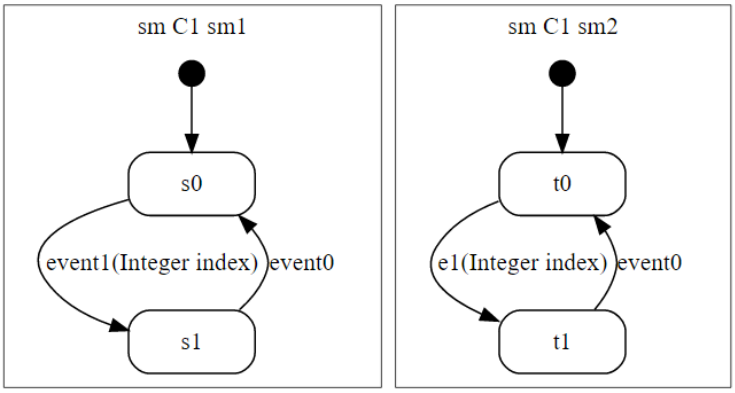


Figure 26. The diagram corresponding to modification in Listing 43

Listing 43 shows an example in which trait *T1* has two state machines *sm1* and *sm2*. These state machines have common events named *e1(Integer)* and *e0()*. Class *C1* wants to use trait *T1* with some changes in the name of events. It is required to rename all event names *e0()* to *event0()* and just change the event name *e1(Integer)* to *event1(Integer)* in state

machine *s1*. Line 12 depicts how class *C1* achieves it. Since the change on event *e0()* is going to happen in both the state machines, the symbol * has been used. However, the name of state machine *sm1* was used for the event *e1(Integer)* because we do not want to have it changed in state machine *sm2*. The result of those operations can be seen in the diagram shown in Figure 26.

When an event name is used with a specific name for its state machine, the state machine and event must be available in the trait, otherwise, the Umple compiler raises error code 231 (e.g., Listing 44.a). The same restriction is applied when the symbol * is used for the name of state machines. If the event is not available in at least one of state machines existing in the trait, then the Umple compiler raises error code 232 (e.g., Listing 44.b).

Listing 44. Errors when changing improperly the name of state in Listing 40

a

```
1 class C1 {
2   isA T1<sm1.e2() as event2>;
3 }
```

b

```
class C1 {
  isA T1<*.e2() as event2>;
}
```

3.5.4.5 Removing/keeping a state machine

This operator is used to remove or keep a state machine when a client uses a trait. In the removing mode, specified by the minus symbol '-', the indicated state machine is ignored and is not included in the client. In the keeping mode, specified by '+', only the indicated state machine is kept and the others are ignored. This operator can be used to keep the client free of un-needed detail or conflict. The syntax used for this operator is as follows:

```
(-|+) stateMachineName
```

Listing 45 shows an example in which trait *T1* has three state machines *sm1*, *sm2*, and *sm3*. Classes *C1* requires state machine *sm2* and *sm3* while class *C2* requires just state machine *sm2*. Class *C1* achieves this through removing state machine *sm1* from trait *T1* (line 13). Class *C2* obtains its required state machine through keeping just state machine *sm2* (line 16). Class *C2* could also achieve the same result through removing *sm1* and *sm2*. Using the keeping operator is more convenient when there are several state machines and modelers need just one them. Figure 27 shows the result of operations on classes *C1* and *C2* in Listing 45.

Listing 45. An example that shows how to remove or keep state machines

```
1  trait T1 {
2    sm1{
3      s0 { e0-> s0;}
4    }
5    sm2{
6      t0 { e0-> t0;}
7    }
8    sm3{
9      w0 { e0-> w0;}
10   }
11 }
12 class C1 {
13   isA T1< -sm1 >;
14 }
15 class C2 {
16   isA T1< +sm2 >;
17 }
```

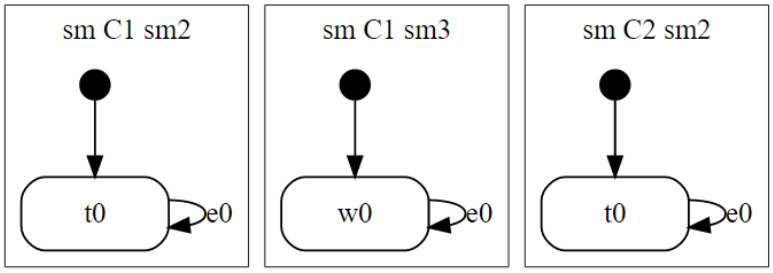


Figure 27. The diagram corresponding to modification in Listing 45

When a state machine is defined to be removed or kept, it must be available in the trait, otherwise, the Umple compiler raises error code 230. Listing 46 shows two cases which result in the error.

Listing 46. Errors when removing wrong state machine in Listing 45

a

```
1 class C1 {
2   isA T1< -sm10 >;
3 }
```

b

```
1 class C1 {
2   isA T1< +sm20 >;
3 }
```

3.5.4.6 Removing/keeping a state

This operator is used to remove or keep a simple or composite state when using a state machine in a trait. The syntax for this operator is as follows:

```
(-|+) stateMachineName.stateName.....stateName
```

This works much like removing/keeping a state machine, using a minus sign for removing and a plus for keeping. The symbols are followed by the name of the state. In the removing mode, this operation will delete all incoming and outgoing transitions of the state as well. In the keeping mode, the specified state will be kept and the remaining states will be removed. This also includes removing all transitions from other states to the specified state. This mode cannot be applied to the initial states, but if it is applied to other states, the initial state will *not* be removed (the reason for this is discussed later).

The operator is helpful for cases in which base state machines do not need the functionality of that specific state, or have the same state and do not want to merge it with the one coming from the reused trait. Another use is when clients use more than one trait and those traits have common state machines and states. These common states might have different functionality and clients might want to keep one version.

Listing 47. An example that shows how to remove or keep states

```
1  trait T1 {
2    sm1{
3      s0 {
4        e1-> s1;
5        e2-> s2;
6      }
7      s1 {
8        e2-> s2;
9        e3-> s3;
10     }
11     s2 {
12       e3-> s3;
13       e2-> s2;
14     }
15     s3 {
16       e0-> s0;
17       e2-> s2;
18     }
19   }
20 }
21 class C1 {
22   isA T1< -sm1.s2 >;
23 }
24 class C2 {
25   isA T1< +sm1.s1 >;
26 }
```

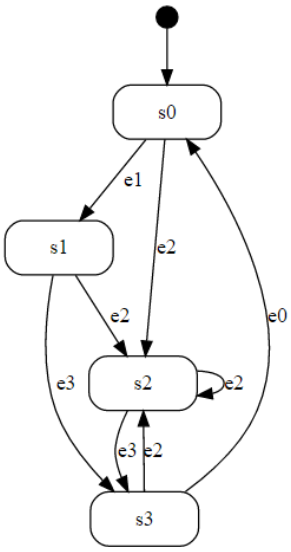


Figure 28. The diagram corresponding to state machine in trait T1 Listing 47

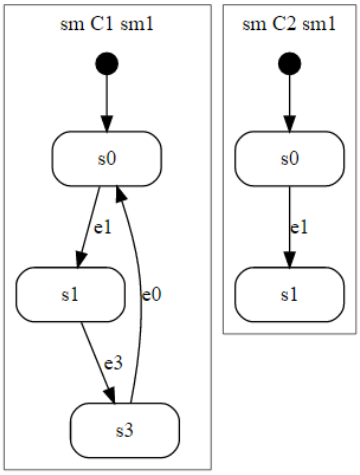


Figure 29. The diagram corresponding to state machines in class C1 and C2 in Listing 47

Listing 47 shows an example including trait *T1* with state machine *sm1* (line2). The graphical representation related to this state machine is depicted in Figure 28. Class *C1* uses trait *T1* and removes state *s2* from the state machine *sm1* (line 22). The result is shown in the left state machine in Figure 29. As seen, in addition to the state *s2*, all outgoing transitions (named *e2()*) from states *s0*, *s1*, and *s3* have been removed. The operator also removed incoming transition *e3()* from state *s2* to state *s3*.

Class *C2* uses trait *T1* and requires just state *s1* of state machine *sm1*. The result is shown in the right state machine in Figure 29. As seen, all other states except *s0* have been removed. The reason is that state *s0* is the initial state of state machine *sm1* and removing it

results in non-reachable states. The Umple compiler does not allow this situation. Furthermore, all transitions coming from other states to *s1* have been removed (in this case, there is none) except transitions coming from the initial state *s0*. The operator also removes the outgoing transitions *e2()* and *e3()* of state *s1* to other states *s2* and *s3* except the ones going to the initial state (in this case, there is none). It should be pointed out that in some scenarios it might make sense to even remove the initial state when the keeping operator is used. However, we decided to keep keeping and removing operators consistent with each other.

When a state is defined to be removed or kept, the state and its state machine must be available in the trait, otherwise, the Umple compiler raises error code 230 (e.g., Listing 48.a). Furthermore, the operator cannot be applied to the initial state of a state machine or to the initial state of a composite state because it will cause other states to become unreachable. If the modeler applies it in such cases, the Umple compiler raises error code 233 (e.g., Listing 48.b)

Listing 48. Errors when removing wrong state machine in Listing 45

a

```
1 class C1 {
2   isA T1< -sm1.s20 >;
3 }
```

b

```
1 class C1 {
2   isA T1< -sm1.s0 >;
3 }
```

3.5.4.7 Removing/keeping a region

This operator is used to remove or keep a region of a state machine. The syntax used for this operator is exactly like the one defined for removing or keeping a state, except that the last name in the dot-separated chain specifies the name of the region. This operator is utilized in cases similar to those explained for removing a state. It can also be used to make a composite state a simple state by reducing the number of regions to zero.

Listing 49 shows an example in which two classes *C1* and *C2* use trait *T1* and manipulate its state machine's regions. State machine *sm* has a composite state *s1* with three regions *r1*, *r2*, and *r3*. Class *C1* removes region *r1* from the composite state *s1* (line 20) while class *C2* keeps region *r2* (line 23). As seen in both cases, the region name appears after the name of state machine *sm* and the state *s1*. Figure 31 depicts the result of this operator on classes *C1* and *C2*. In comparison to the diagram in Figure 30, we can see, since region *r1* has been removed in class *C1*, the incoming transition *e2()* has also been removed automati-

cally. In class *C2*, all outgoing transition from region *r2* to other regions, including *e2()* and *e4()*, have been removed automatically in addition to region *r1* and *r3*.

Listing 49. An example that shows how to remove or keep regions

```
1  trait T1{
2    sm {
3      s1{
4        r1{ e1-> r11; }
5        r11{}
6        ||
7        r2{
8          e2-> r11;
9          e3-> r21;
10         e4-> r31;
11       }
12       r21{}
13       ||
14       r3{e5->r31; }
15       r31{}
16     }
17   }
18 }
19 class C1{
20   isA T1< -sm.s1.r1 >;
21 }
22 class C2{
23   isA T1< +sm.s1.r2 >;
24 }
```

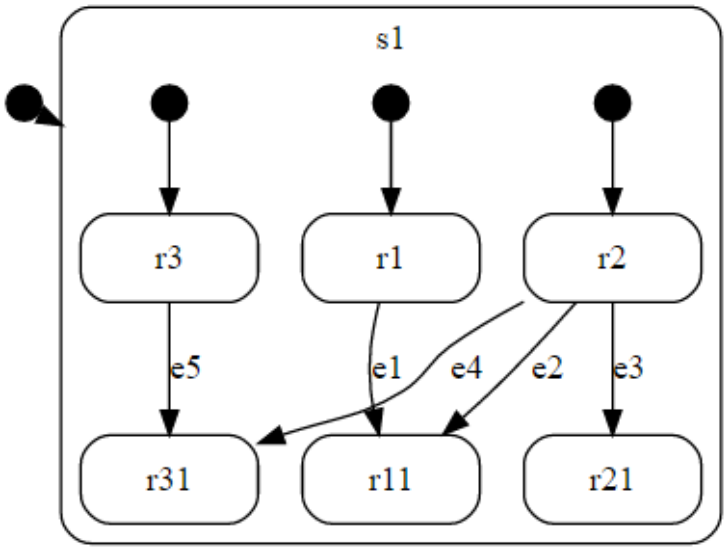


Figure 30. The diagram corresponding to state machine in trait T1 in Listing 49

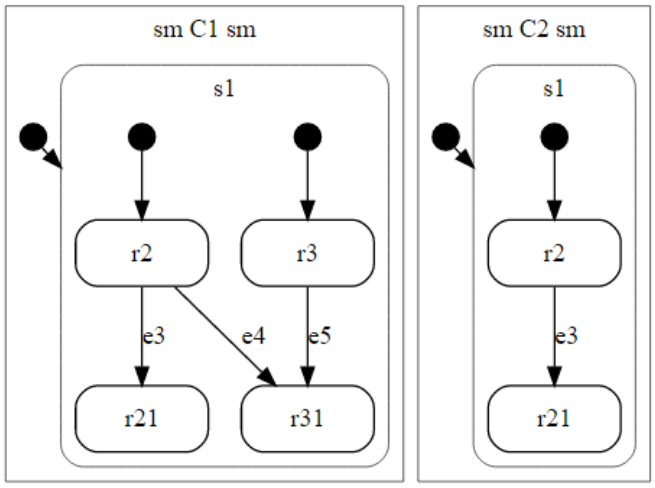


Figure 31. The diagram corresponding to state machine in class C1 and C2 in Listing 49

When a region is defined to be removed or kept, the state machine and the region must be available in the trait, otherwise, the Umple compiler raises error code 230. Listing 50 shows two examples resulting in the error, which in both cases the region *r12* does not exist in state *s1* related to the state machine in Listing 49.

Listing 50. Errors when removing wrong state machine in Listing 49

a

```
class C1 {
  isA T1< -sm.s1.r12 >;
}
```

b

```
class C1 {
  isA T1< +sm.s1.r12 >;
}
```

3.5.4.8 Removing/keeping a transition

This operator is used to remove or keep a transition from a state machine. The syntax for this operator is as follows:

```
(-|+) stateMachineName.stateName.....stateName.
((eventName(argumentTypes) ([guard])?) | [guard?])
```

A minus symbol is followed by a transition that needs to be removed. A plus symbol is used to pick a transition to keep. A transition is defined by specifying the name of the state machine, states (including regions for nested states), event, and guard. The symbol *?* is not part of the operator and it is there to show which elements are optional. The name of the state

machine and states are mandatory. The event name (along with its arguments) depends on the type of transition.

If the transition is 'auto' (immediately taken on entry to the state or upon completion of a do activity) there is no need to specify it, otherwise, it must be specified. If a transition has a guard, it must be specified using the same syntax used when specifying transitions (inside square brackets). However, if a transition is auto and unguarded, it must be defined with an empty guard "[]" and without any event name. Actions and destination states are not part of the definition for this operator because the above definition suffices to uniquely select any transition. The operator is utilized when base state machines do not need a specific transition coming from the used trait. Furthermore, base state machines might want to extend a transition of a state, but it might already have a transition matching the given specification.

Listing 51. An example that shows how to remove or keep transitions

```
1  trait T1{
2    internal Boolean cond;
3    sm {
4      s1{
5        e2(Integer i)-> s2;
6        e3[!cond]-> s3;
7        e4[cond]-> s4;
8      }
9      s2{
10       [cond] -> s3;
11       [!cond]-> s4;
12     }
13     s3{
14       -> s1;
15     }
16     s4{
17      -> s1;
18     }
19   }
20 }
21 class C1{
22   isA T1< +sm.s1.e2(Integer) >;
23 }
24 class C2{
25   isA T1< -sm.s1.e4()[cond] >;
26 }
27 class C3{
28   isA T1< -sm.s2.[cond] >;
29 }
30 class C4{
31   isA T1< -sm.s3.[] >;
32 }
```

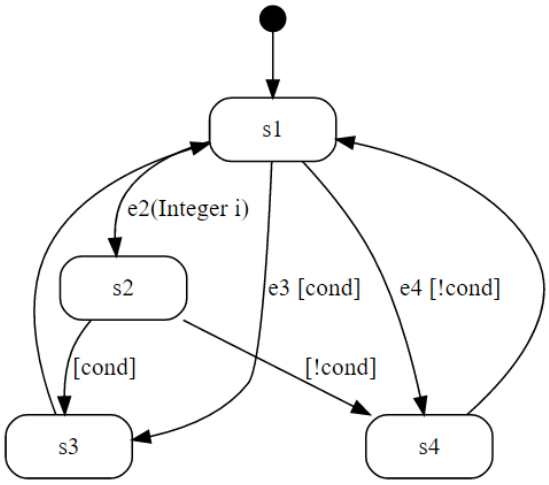


Figure 32. The diagram corresponding to state machine in trait T1 in Listing 51

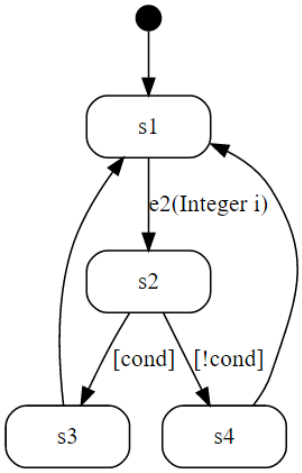


Figure 33. The diagram corresponding to state machine in class C1 in Listing 51

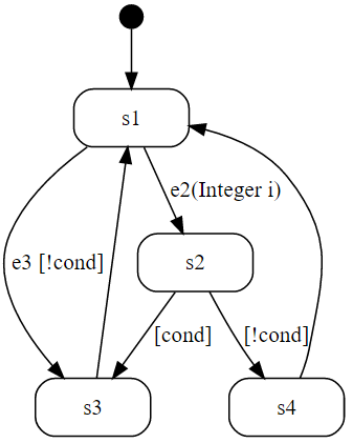


Figure 34. The diagram corresponding to state machine in class C2 in Listing 51

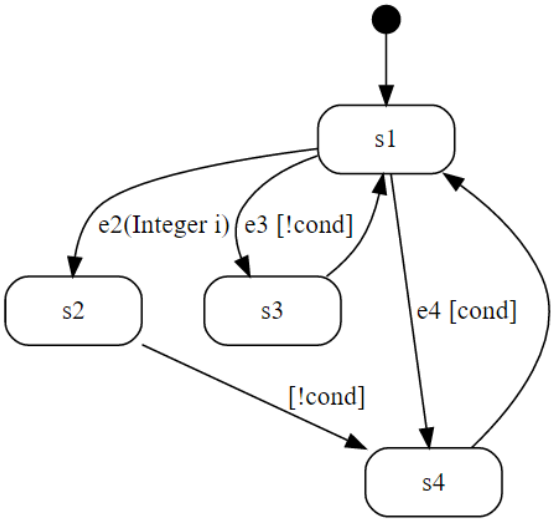


Figure 35. The diagram corresponding to state machine in class C3 in Listing 51

Different ways of using this operator to keep and remove transition are demonstrated in Listing 51. The graphical representation of the state machine *sm* in trait *T1* (line 3) is depicted in Figure 32. Class *C1* uses trait *T1* (line 22) and keeps the transition with the event name *e2(Integer)* from the state machine *sm* and state *s1*. Other transitions related to the state *s1*, which are *e3()* and *e4()*, are removed. Since the transition does not have a guard, brackets are not required in the specification of the transition. depicts the result of the operator in class *C1*.

Class *C2* also uses trait *T1* (line 25), but it removes the transition with the event name *e4()* and a guard on the variable *cond* from the state machine *sm* and state *s1*. This case shows how a transition with the event name and guard can be specified. The result of this operator in class C2 is depicted in Class *C3* uses trait *T2*, but it removes an auto transition from state *s2* with a guard on the variable *cond*. As seen, no name has been defined in the operator for the event and the guard is just defined inside brackets.hows the result of the operator in class *C3*.

Finally, class *C4* uses the trait *T1* (line 31), but it removes an auto transition without a guard. As seen, an empty bracket after the name of the state is used to specify the transition. As pointed out before (Section 2.1.5), Umple does not support non-deterministic state machines. Therefore, it is not possible to have more than one auto transition for a state. If that

happens, the Umple compiler detects it and automatically removes the last one defined in the Umple model. It also informs users about it. The result of the operator in class C4 is depicted in Figure 36.

Like other operators, the Umple compiler checks the validity of transitions defined in the operator and if they are not available in the state machines coming from traits, error code 231 is raised.

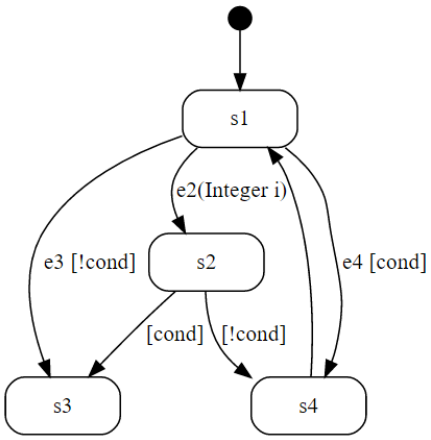


Figure 36. The diagram corresponding to state machine in class C4 in Listing 51

3.5.4.9 Extending a state by adding a state machine to it

This operator is used to assign a state machine to a specific state *inside* another state machine, hence turning that state into a composite state. The syntax used for this operator is as follows:

```
srcStateMachineName as
desStateMachineName.stateName.....stateName
```

This operator involves two state machines. The *srcStateMachineName* is found in the trait, and the *desStateMachineName* is found in the client. The state in the client can be simple or composite. This operator provides a practical mechanism to incrementally compose a state machine from various parts.

For example, simple ‘on/off’ pairs of states with events to toggle between them are fairly common and can be injected easily into destination states using this operator. If a composited state is extended with this operator, it will trigger our composition algorithm, illustrated in the next section.

Furthermore, the operator can be used to bring more than one state machine inside a state. More details regarding how the state accepts the assigned state machine will be illustrated later. Here, we just demonstrate how a simple state can be extended with another state machine.

Listing 52. Extending a state with another state machine

```
1  trait T1{
2    sm1{
3      m1{
4        t2-> m2;
5      }
6      m2{
7        t1-> m1;
8      }
9    }
10 }
11 class C1{
12   isA T1<sm1 as sm.s2>;
13   sm{
14     s1{
15       e2-> s2;
16     }
17     s2{
18       e1-> s1;
19     }
20   }
21 }
```

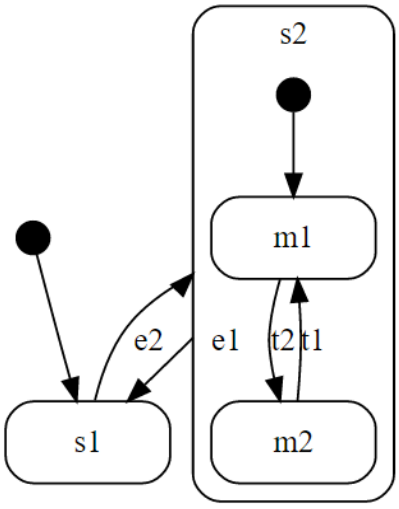


Figure 37. The diagram corresponding to state machine in class C1 in Listing 52

Listing 52 shows an example in which class *C1* has state machine *sm* with two states *s1* and *s2*. Trait *T1* has state machine *sm1*. Class *C1* needs to have state machine *sm1* activated when it is in state *s2*. Class *C1* achieves this by specifying the source state machine and destination state when it uses trait *T1* (line 12). The result of this operator can be seen in Figure 37.

Like other operators, the Umple compiler checks the validity of the source state machine and the destination state. If one of them does not exist or is invalid, the Umple compiler raises error code 230.

3.5.5 Composition

We have chosen to employ the word *composition* for the way state machines can be combined because it covers several related mechanisms. Through composition, we achieve typical state machine reuse, merging several state machines, and simple state extension by other state machines. Simpler traits can be composed with other traits to make more complex traits.

In the basic case, state machines being composed need to have the same vocabulary – in other words, the same names for the elements (state machines and states) that are to be considered the same. These state machines might have functionality that is either completely different or partially the same. There might be cases in which there is no need to compose state machines with the same names and even vice versa. This is achieved through the operators we have defined in the previous sections. In other words, modelers are given the freedom to decide which state machines must be composed. Trait developers or modelers have full control over the way elements of traits are composed. Therefore, composition based on names is required because traits or classes might have more than one state machine and during the composition process, it must be clear which state machines need to be composed.

Conceptually, during composition, the state machines existing in clients (called base state machines) are the ones receiving composition elements from state machines coming from traits. Technically, the latter ones are composed first and then the result state machine is flattened instead of the one defined in the clients. The composition of elements includes all concepts defined in state machines such as states, composite states, regions, transitions,

guards, activities, and actions. Elements in base state machines have a higher priority over the ‘same’ elements coming from used traits if any conflict is observed.

However, attributes used by guards within state machines deserve special attention. Attributes and state machines together hold the state of an object, therefore, overriding the attribute in a guard will tend to fundamentally change the semantics of the state machine. The Umple compiler, therefore, warns modelers that they may be introducing an error.

3.5.5.1 States

When two state machines are matched to be composed, states of these state machines should be composed. The following cases can happen among states (simple or composite) of the state machines:

- A state in the base state machine matches the one coming from used trait. Two states are considered to match if they have the same names and are at the same hierarchy level of their state machines, with the same names of the chain of parents. In this case, these two state machines are composed and the resulting state gets the same name (the composition of internal elements are described as we progress).
- A state exists in the base state machine, but it does not exist in the used trait. In this case, the state is added to the composed state machine.
- A state exists in the used trait, but it does not exist in the base state machine. The state is added to the composed state machine.

Listing 53.a and Figure 38 show an example in which class *C1* and trait *T1* have a common state machine named *sm* (line 2 and 9). The state machine in Trait *T1* has states *s1* and *s3* while the state machine in class *C1* has states *s1* and *s4*. Since state *s1* exists in both, they are composed and represented under the same name in the composed state machine. However, other states are unique and are added as-is to the composed state machine. Listing 53.b and Figure 39 show the Umple model of the composed state machine flattened in class *C1*.

When two states are composed, their entry actions are composed as well. If the base state has an entry action and the state coming from the used trait has also entry actions, then the ones coming from used traits are disregarded. If the base state does not have any entry

action, the actions coming from the used trait are considered as entry actions for the composed state.

Listing 53. Composition of states and the flattened composed state machine

a

```
1  trait T1{
2    sm{
3     s1{/*implementation*/}
4     s3{/*implementation*/}
5    }
6  }
7  class C1{
8    isA T1;
9    sm{
10    s1{/*implementation*/}
11    s4{/*implementation*/}
12   }
13 }
```

b

```
class C1{
  sm{
   s1{/*implementation*/}
   s3{/*implementation*/}
   s4{/*implementation*/}
  }
}
```

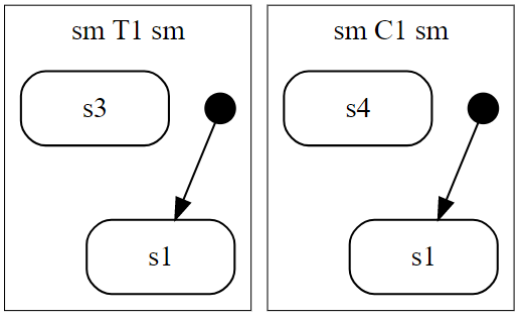


Figure 38. The diagram corresponding to state machine in class C1 and trait T1 in Listing 53.a

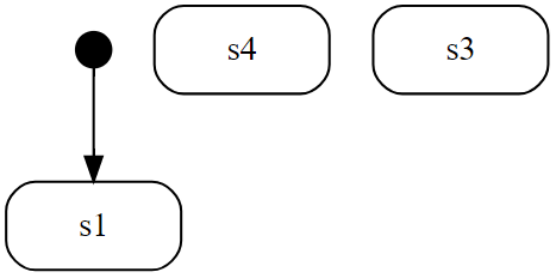


Figure 39. The diagram corresponding to state machine in class C1 in Listing 53.b

Listing 54.a shows an example in which different cases of composition related to entries are explained. State machines in class *C1* and trait *T1* have two states *s1* and *s2* that can be composed. The entry action of base states *s1* calls *action4()* while the entry of the same state in trait *T1* calls *action1()*. Since base functionality always has priority, then action *action1()* is disregarded. The base state *s2* does not have an entry action while the state *s2* in the used trait calls *action2()* in its entry. Therefore, the composite state calls *action2()* in its entry. Since there are no matching states for states *s3* and *s4*, they are added as-is to the

composed state machine. Listing 54.b shows the Umple model of the composed state machine flattened in class *C1*.

Listing 54. Composition of entry actions and the flattened composed state machine

a

```
1  trait T1{
2    sm{
3     s1{ entry /{action1();} }
4     s2{ entry /{action2();} }
5     s3{ entry /{action3();} }
6    }
7  }
8  class C1{
9    isA T1;
10   sm{
11    s1{ entry /{action4();} }
12    s2{ }
13    s4{ entry /{action5();} }
14   }
15 }
```

b

```
class C1{
  sm{
   s1{ entry /{action4();} }
   s2{ entry /{action2();} }
   s3{ entry /{action3();} }
   s4{ entry /{action5();} }
  }
}
```

A client can use more than one trait (and obtain more than one state machine) and so it is possible to have more than two states whose entry actions need to be composed with the base state's entry action. In this case, the composition algorithm needs to consider an order among them. Since we do not consider any order when traits are used by clients, this should be respected in the composition of actions. Therefore, the Umple compiler detects this case and raises error code 236. Note that this conflict happens if more than one state coming from used traits have an entry action.

Listing 55 shows an example in which composition of entry actions is considered as a conflict. The base state *s1* (line 14) in class *C1* has no entry action and so it can accept entry actions coming from state machines in *T1* and *T2*. However, there are two entries coming from used traits which cause a conflict. Note that if class *C1* did not have state machine *sm*, it would still be a conflict for composition. However, if state *s1* in class *C1* had an entry then this would not be a conflict because those entries coming from traits would be disregarded.

The composition of exit actions and do activities for matching states follows exactly the same rules explained for entry actions. Listing 56 shows a simple Umple model to demonstrate the rules for exit actions and do activities.

Consider that in our current implementation, we do not provide any operator for actions and activities because they are part of states and we have operators for states. Furthermore, we think having operators for actions and do activities could make the approach more complicated.

Listing 55. A conflict in composing entry actions of states

```
1  trait T1{
2    sm{
3      s1{ entry /{action1();} }
4    }
5  }
6  trait T2{
7    sm{
8      s1{ entry /{action2();} }
9    }
10 }
11 class C1{
12   isA T1, T2;
13   sm{
14     s1{}
15   }
16 }
```

Listing 56. Composition of entry actions and the flattened composed state machine

a

```
1  trait T1{
2    sm{
3     s1{ exit /{action1();} }
4     s2{ do   /{activity1();} }
5    }
6  }
7  class C1{
8    isA T1;
9    sm{
10    s1{ exit /{action2();} }
11    s2{ }
12   }
13 }
```

b

```
class C1{
  sm{
   s1{ exit /{action2();} }
   s2{ do   /{ activity1();} }
  }
}
```

3.5.5.2 Transitions

Each transition has a structure which makes it unique for each state. This structure includes the signature of the event and its guard. If base states do not have transitions with the same signatures that come from used traits, those incoming transitions are added to the base states.

This uniqueness of transitions guarantees statically that the composed state machines will be deterministic. If there are transitions in common, the ones coming from traits will be disregarded (this is the same 'override' semantics as is used in inheritance).

Listing 57. Composition of transitions and the flattened composed state machine

a

```
1  trait T1{
2    sm{
3      s1{
4        e1[x>0]-> s2;
5        e2-> s3;
6      }
7      s2{ e3-> s1;}
8      s3{ e4-> s2;}
9    }
10 }
11 class C1{
12   isA T1;
13   sm{
14     s1{
15       e1[x>0]-> s3;
16       e5-> s4;
17     }
18     s4{}
19   }
20 }
```

b

```
class C1{
  sm{
    s1{
      e1[x>0]-> s3;
      e2-> s3;
      e5-> s4;
    }
    s2{ e3-> s1;}
    s3{ e4-> s2;}
    s4{}
  }
}
```

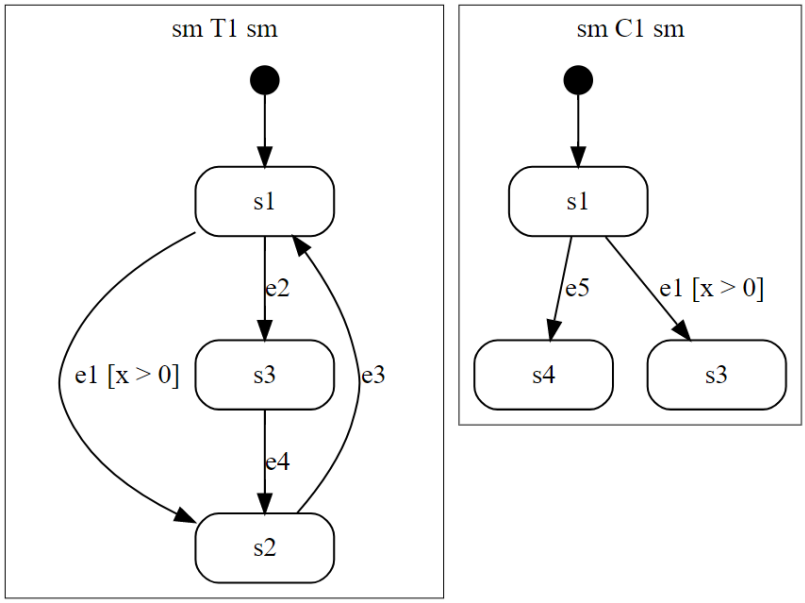


Figure 40. The diagram corresponding to class C1 and trait T1 in Listing 57.a

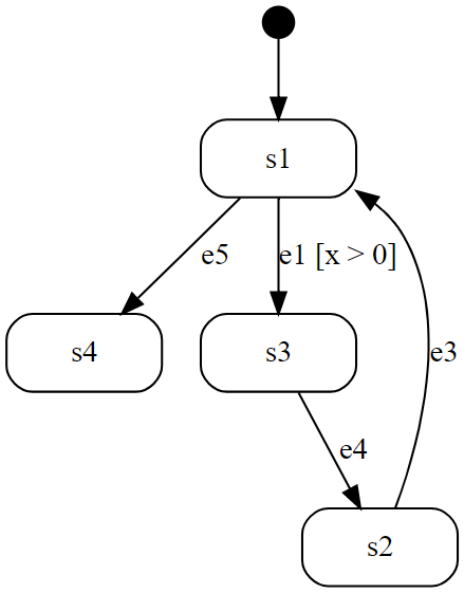


Figure 41. The diagram corresponding to class C1 in Listing 57.b

Listing 57.a and Figure 40 represent an example in which we describe how composition of transitions happens. As seen, class *C1* uses trait *T1* (line 12) and they have common state machine that needs to be composed. The base state machine and used state machine have common state *s1*. Both states have the transition *e1[x>0]* (lines 4 and 15), but their destination states are different. Therefore, the one coming from the used trait is disregarded. The used state *s1* (line 3) has the transition *e2* (line5), which does not exist in the base state *s1* and so is added to the list of transitions for the composed state *s1*. In the same manner, the base state *s1* has transition *e5* (line 16) which does not exist in used state *s1* and so it is also added to the list of composed state *s1*. Other states in the base and used state machines do not have a matching state and so they and their transitions are added to the composed state. Listing 57.b and Figure 41 show the Umple model of the composed state machine flattened in class *C1*.

As pointed out before, Umple does not support non-deterministic state machines and so composed state machines must be deterministic as well. Our composition algorithm automatically detects transitions that cause the composed state to be non-deterministic and raises error code 234.

Listing 58. An example regarding detection of non-determinism when composition occurs

```
1  trait T1{
2    sm{
3      s1{ e1[x>0] -> s2;  }
4      s2{ e2 -> s1; }
5    }
6  }
7  class C1{
8    isA T1;
9    sm{
10     s1{
11       e1 -> s3;
12       e3 -> s3;
13     }
14     s3{ }
15   }
16 }
```

Listing 58 shows an example regarding how a transition in a base state could be caused to have non-determinism. As seen, both state machines in class *C1* and trait *T1* have state *s1* and they need to be composed. The state *s1* in trait *T1* has transition *e1[x>0]* and the transition's destination is *s2*. The base state *s1* has transition *e1* without a guard and its destination state is state *s3*. This transition does not exist in the state *s1* coming from the trait, so it can be added to the composed state *s1*. However, this causes a situation in which the composite state machine can be in two states *s2* and *s3* simultaneously. Therefore, the composition is not allowed.

When transitions are composed, their actions need to be composed as well. The way actions are composed follows exactly the same rules defined for entry and exit actions of states. Listing 59.a and Figure 42 show an example in which different cases of composition are described. As seen, two transitions *e1[x>0]* and *e2* in state *s1* of base and used state machines need to be composed. The base transition *e1[x>0]* has action *action4()*, and the incoming transition has action *action1()*. Since there is an action defined in the base transition, the action *action1()* coming from the used transition is disregarded.

The base state *e2* does not have any action, but the incoming transition has action *action2()*. Therefore, action *action2()* is added to the composed transition *e2*. Furthermore, since the base and used transitions *e2* have different destinations, the one defined in the base state (state *s2*) is accepted as the valid destination. Transitions *e3* defined in state *s2* of trait

T1 and state *s1* of class *C1* are added with their actions to the composed states because there are no matching transitions for them. Listing 59.b and Figure 43 show the Umple model of the composed state machine flattened in class *C1*.

Listing 59. Composition of actions and the flattened composed state machine

a

```
1  trait T1{
2    sm{
3      s1{
4        e1[x>0]-> /{action1();} s2;
5        e2 -> /{action2();} s1;
6      }
7      s2{ e3-> /{action3();} s1;}
8    }
9  }
10 class C1{
11   isA T1;
12   sm{
13     s1{
14       e1[x>0]-> /{action4();} s2;
15       e2 -> s2;
16       e3 -> /{action5();} s2;
17     }
18     s2{}
19   }
20 }
```

b

```
class C1{
  sm{
    s1{
      e1[x>0]-> /{action4();} s2;
      e2 -> /{action2();} s2;
      e3 -> /{action5();} s2;
    }
    s2{ e3-> /{action3();} s1;}
  }
}
```

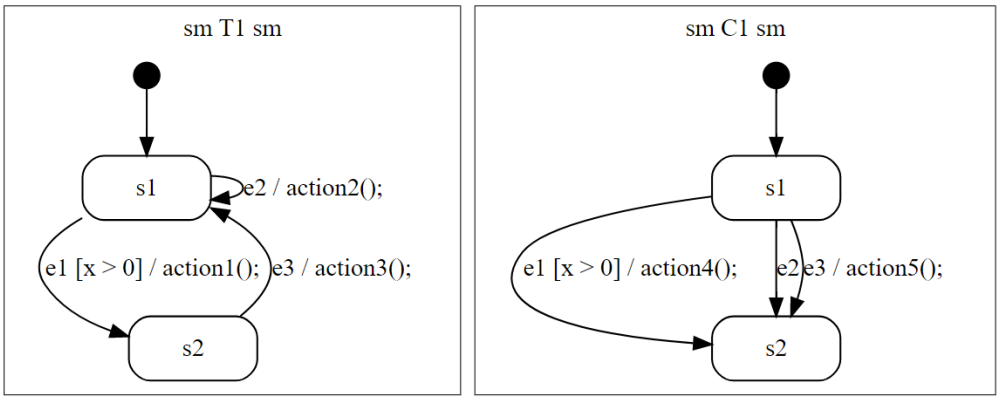


Figure 42. The diagram corresponding to class C1 and trait T1 in Listing 59.a

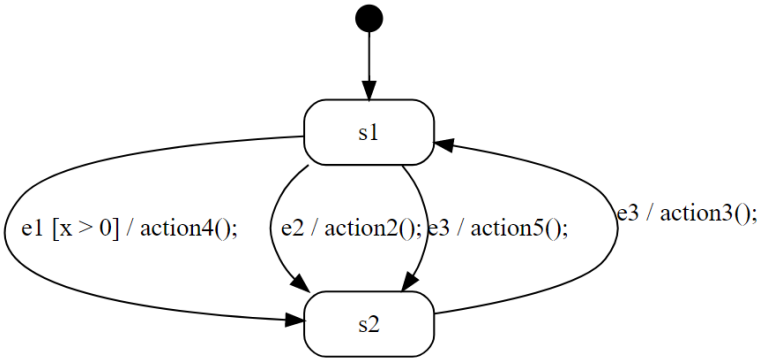


Figure 43. The diagram corresponding to class C1 in Listing 59.b

Finding matching transitions to be composed requires comparison of guards. Umple provides a grammar for Boolean expressions written for guards and constraints (an essential subset of OCL [109]). The Umple compiler parses Boolean expressions and builds constraint trees for them. In order to achieve a robust comparison, we have developed an algorithm that takes two guards (constraint trees) and precisely compares them (without paying attention to the order they are defined). This algorithm can currently detect if two Boolean expressions are completely equal statically. However, it is not able to detect if one Boolean expression logically implies another one (this is ongoing work), so such guards are currently treated as distinct. In the section on future work, we explain what kinds of approaches can be adopted to improve this capability.

3.5.5.3 The Keyword SuperCall

We explained that when states are matched, their transitions, entry actions, exit actions and do activities are composed. In the composition process, the base elements have always priority over the ones coming from used traits. In Listing 59.a, for example, we explained that the action of transition *e1[x>0]* in the base state *s1* (line 14) will be the one accepted in the composed state machine and the action coming from its matching transition (*action1()*) is disregarded.

However, there might be cases in which modelers want to have access to coming elements (in this case *action1()*). In order to satisfy this advanced requirement, we extended the syntax and semantics of composition with the keyword *superCall*. This keyword is used

in actions and activities and implies that actions or activities coming from matching elements must be executed at the point where the superCall keyword is encountered.

Listing 60. Composition of actions by the keyword superCall

```
1  trait T1{
2    sm{
3      s1{
4        entry /{action1();}
5        e1 -> /{action2();} s1;
6      }
7    }
8  }
9  trait T2{
10   isA T1;
11   sm{
12     s1{
13       entry /{superCall; action3();}
14       e1 -> /{action4(); superCall;} s1;
15     }
16   }
17 }
18 class C1{
19   isA T2;
20   sm{
21     s1{
22       entry /{ superCall; action5();}
23       e1 -> /{action6();superCall; action7();} s1;
24     }
25   }
26 }
```

Modelers can change the order in which the keyword appears in combination with other executable parts of base actions and activities so as to obtain full control of when the coming actions and activities must be executed. The incoming actions and activities can be executed before, after, or in the middle of the base actions and activities.

Listing 60 shows how the keyword can be used in a hierarchy among traits and classes to obtain all actions of used traits in a different order. The state machine *sm* in trait *T1* has state *s1* and this state has entry action *action1()*. Transition *e1* calls *action2()* when it is triggered. Trait *T2* uses trait *T1* (line 10) and has the same state machine and state. It wants to execute the entry action of used state *s1* (*action1()*) first and then execute its own action, which is *action3()*. It achieves this by having the keyword superCall before *action3()* (line 13). The state *s1* of trait *T2* also wants to have the same behavior for the action of its transition but in an opposite direction. It wants to execute first its own action and then one

coming from the used trait. In order to achieve this, it uses the keyword after its own action *action4()* (line 14).

Listing 61. The flattened composed state machine of Class C1 in Listing 60

```
1  class C1{
2    sm{
3      s1{
4        entry /{action1();action3();action5(); }
5        e1 -> /{action6();action4();action2();action7();} s1;
6      }
7    }
8  }
```

Class *C1* uses trait *T2* and has the same state machine and state names defined in trait *T2*. It wants to achieve the same scenario for the entry action of state *s1*, described for the trait *T2* while it was using trait *T1*. However, it wants to execute the action of transition *e1* coming from trait *T2* in the middle of its own two actions, *action6()* and *action7()*. For the entry action, class *C1* uses the keyword before its own action (line 22). Class *C1* uses the keyword in the middle of actions for the transition *e1* in order to achieve the behavior it wants (line 23). Listing 61 shows the Umple model of the composed state machine flattened in class *C1*.

When clients use more than one trait, using the keyword superCall can cause a conflict. This happens because an order of execution is required. It is worth noting that the conflict explained here is different from the problem that exists in multiple inheritance. In multiple inheritance, the ambiguity comes from the fact that there are two methods, for example, coming from two different superclasses with the same name. The issue with superCall is about the execution order of actions coming from used traits. We cannot decide which action should be executed first when they are composed in a client. Please note that when this conflict happens, modelers can use operators to fix it.

Listing 62 shows an example in which a conflict exists and the Umple compiler detects it and raises error code 235. As seen, class *C1* uses two traits *T1* and *T2* and there is matching state *s1* and matching transition *e1* needed to be composed. Since the keyword has been used in the base transition *e1*, it indicates that the actions of matching transitions need to be executed, but there are more than one action to be executed. Therefore, the Umple compiler prevents these models from being composed.

Listing 62. A conflict while the keyword superCall is used

```
1  trait T1{
2    sm{
3      s1{ e1 -> /{action1();} s1; }
4    }
5  }
6  trait T2{
7    sm{
8      s1{ e1 -> /{action2();} s2;}
9      s2{}
10   }
11 }
12 class C1{
13   isA T1,T2;
14   sm{
15     s1{ e1 -> /{superCall; action3();} s3; }
16     s3{}
17   }
18 }
```

Note that if one of transitions coming from trait T1 or T2 did not have an action, then the composition would be valid because there would be no conflict in order of execution. This semantic is also applied to clients in which more than two traits are used. The same conflicts can also happen for activities, entry, and exit actions of states. They are detected and reported under error code 236.

3.5.5.4 Simple and composite state composition

Regions are another element of state machines that need to be composed. If the base state is simple and the incoming state is composite, then the composite state's region is added to the composited state. Therefore, the composed state will be composite. The same rule applies when the base state is composite and the incoming state is simple. If both states are composite they are composed with the same rule, but another special rule is applied to them (regarding their initial states) that is discussed in the next section.

Listing 63.a shows an example in which those two cases are explained. As seen, two state machines *sm* in class *C1* and trait *T1* need to be composed. The base state *s1* in class *C1* is simple and state *s1* in trait *T1* is composite. Therefore, the composed state *s1* will be a composite state including all elements inside state *s1* in trait *T1*. The base state *s2* in class *C1* is composite while the state *s2* in trait *T1* is simple. Therefore, the composed state *s2* will be

a composite state including all elements inside state *s2* in class *C1*. Listing 63.b shows the flattened state machines of class *C1* in Listing 63.a.

Listing 63. Composition of simple and composite states

a

```
trait T1{
  sm{
    s1{
      e1 ->  s2;
      t1{ e2 ->  t2;}
      t2{}
    }
    s2{}
  }
}
class C1{
  isA T1;
  sm{
    s1{}
    s2{
      e3 ->  s1;
      t1{ e4 ->  t2;}
      t2{}
    }
  }
}
```

b

```
class C1{
  sm{
    s1{
      e1 ->  s2;
      t1{ e2 ->  t2;}
      t2{}
    }
    s2{
      e3 ->  s1;
      t1{ e4 ->  t2;}
      t2{}
    }
  }
}
```

3.5.5.5 Composition and rules relating to the initial state

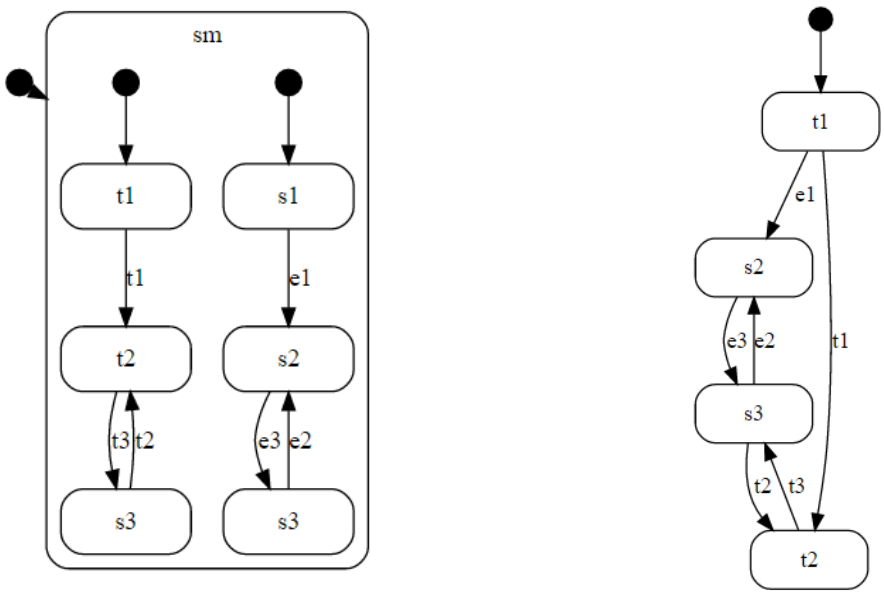
As expressed in Section 2.1.5.9, the initial state is the one listed first in a state machine. If two state machines are being composed and have the same initial state, composition is straightforward: The result is a single state machine. If the initial states are different, however, the question arises: Which of them would now become the initial state following composition? This can arise when two independent state machines are composed, or when two state machines with the same name are added as substate machines of a state. When such a situation happens, the Umple compiler informs modelers and raises warning code 228.

Listing 64 shows an example in which class *C1* uses two traits *T1* and *T2* that have two state machines with the name *sm*. The state machine *sm* in the trait *T1* has the initial state *s1* while the one in the trait *T2* has the initial state *t1*. Both state machines have a common state called *s3*. The solution adopted by Umple is to put each state of the two state machines into different *regions*. This keeps both state machines independent and composition is once

again straightforward. When a common event is triggered, it causes transitions in both regions. Figure 44.a shows the result of this solution on the example in Listing 64. Both regions can have states with the same name other than the start state. In Umple, we impose a rule that the start states of a set of regions must each have different names.

Listing 64. Using two state machines with different initial states

```
1  trait T1{
2    sm{
3      s1{ e1 ->  s2;}
4      s2{ e3 ->  s3;}
5      s3{ e2 ->  s2;}
6    }
7  }
8  trait T2{
9    sm{
10     t1{ t1 ->  t2;}
11     t2{ t3 ->  s3;}
12     s3{ t2 ->  t2;}
13   }
14 }
15 class C1{
16   isA T1,T2;
17 }
```



a b

Figure 44. The composed state machine of class C1 in Listing 64

However, what if the user wanted in fact *not* to create regions, but to compose the two state machines, and pick one of the start states? The solution is to rename one of the start states during the composition so they are both the same. This is accomplished using the re-naming ('as') operator. In order to apply this option on the example in Listing 64, line 16 must be replaced with the following:

```
isA T1 <sm.s1 as t1>, T2;
```

The result is depicted in Figure 44.b. As seen, the algorithm detects the common states (which are *s3* and *t1*) and composes them. The operator could be applied to *T2* instead of *T1*. In that case, the composed state machine would be the same as depicted in Figure 44.b except that the name of initial state would be *s1*.

A similar situation occurs when a client wants to extend a state machine coming from a used trait. In that case, the state machine inside the client needs to directly specify an initial state that matches the one coming from the used trait. This initial state is called a *dummy* initial state. For example, Listing 65, shows an example in which class *C1* uses the trait *T1* in Listing 64 and wants to extend its state machine's behavior (changing the transition going out of the state *s3*). The class *C1* defines a state machine called *sm*, which matches the name of the used one, and needs to add a 'dummy' initial state in it to ensure the merging occurs rather than region creation. The same approach is also applied when a composite state is to be extended by clients.

Listing 65. Setting the common initial states when a state machine is extended

```
1  class C1{
2    isA T1;
3    sm{
4      s1{}
5      s3{ e2 ->  s1;}
6    }
7  }
```

3.6. Associations in Traits

An association is a useful mechanism at the modeling level that specifies relationships among instances of classifiers. An association implies the presence of certain variables and provided methods (such as ones defined in Table 2) in both associated classifiers. Other

methods, as well as traits, can hence refer to the implied methods. Since an association can be seen as a set of implied provided methods, it would be logically possible to extend the concept of traits to incorporate associations. However, prior to our work, there was no such a mechanism in traits.

An association can only be between a class and another class or between a class and an interface; in other words, defining an association between two interfaces is not allowed. This must also be accounted for in the definition of traits. The reason for this is that associations imply that at least one end must maintain the state of the links between instances. For unidirectional associations (navigable in one direction only), only one end maintains this state, so the other end can be an interface. For bidirectional associations, both ends must maintain the state, so both ends must be classes.

Having associations in classes is considered a kind of limitation on fine-grained reusability because such classes cannot then be used alone in other systems. This happens because the other systems also need the associated classes or interfaces. Furthermore, the nature of associations is defined based on exact names. When class *A*, for example, with an association with another class or interface *B* is to be used in a different system, then that system must have a class or interface with the exact name *B*. In order to overcome this issue, we extend traits to have associations with template parameters.

Listing 66. An issue with associations among classes

```
1  class C1{
2    0..1 -- * C2;
3    /*implementation*/
4  }
5  interface I1 {
6    /*implementation*/
7  }
8  class C2{
9    isA I1;
10   /*implementation*/
11 }
```

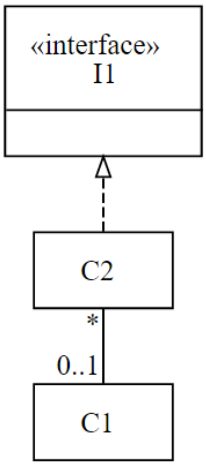


Figure 45. The diagram corresponding to Listing 66

Listing 66 and Figure 45 show an example explaining the case in which a class cannot be reused alone. In class *C1* (lines 1-4) there is a bidirectional association between *C1* and *C2*. Class *C2* (lines 8-11) implements interface I1 (lines 5-7). If we would like to use class *C1* in another system we have to transfer class *C2* and interface *I1* as well. There might be a class in the new system that can satisfy all features of class *C2*, but we cannot use it because it is forced to have exactly the same name in the new system. It is also impossible to change the name of the compatible class to *C2* because there will be inconsistency among elements of the new system.

In addition, if we change the name and apply it to all other parts of the system, the new name may be out of the domain of the system and so create understanding challenges. The same issues related to class *C1* can happen to class *C2* because it depends on class *C1*.

Associations are defined syntactically in the same way they are defined for classes. Traits can make associations with interfaces, classes, and template parameters. If one end of the association is a template parameter, the binding type must be checked to make sure it is compatible with the association. For example, if a trait has a bidirectional association with a template parameter, the binding value cannot be an interface and it must be a class. This is an extra constraint applied to template parameters. Breaking this constraint is reported to modelers using error code 213.

Listing 67. An example of associations in traits

```
1  interface I1 {
2    /*implementation*/
3  }
4  trait T1 <RelatedClass isA I1> {
5    0..1 -- * RelatedClass;
6    /*implementation*/
7  }
8  class C1{
9   isA T1<RelatedClass=C2>;
10  /*implementation*/
11 }
12 class C2{
13   isA I1;
14   /*implementation*/
15 }
```

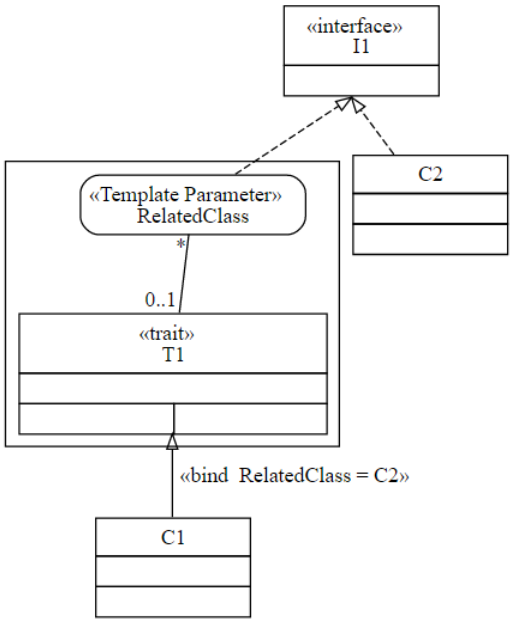


Figure 46. The diagram corresponding to Listing 67

In order to avoid the limitations in Listing 66, we redesign it with a trait depicted in Listing 67 and Figure 46. There are again two classes *C1* and *C2* and an interface *I1*. We added a trait (lines 4-7) with a template parameter *RelatedClass* restricted to implement interface I1. Furthermore, we added the same association that was in class *C1* in Listing 66 to the trait, but parameter *RelatedClass* was substituted for concrete name *C2*. This association applied to class *C1* in line 8, in which class *C2* has been bound to parameter *RelatedClass*. As a result, the association is available for both classes *C1* and *C2*. In this case, if we want to use class *C1* in another system we do not need to have exactly a class named *C2*. We need a class that implements interface I1, and we simply need to bind it to parameter *RelatedClass*. The name of the class is not fixed anymore in this approach and the proper candidate from the target system can be used with class *C1*. Class *C2* can be reused independently too because there is no concrete relationship between it and other elements of the system.

When an association is defined, APIs (set of methods) are generated in the class to allow such actions as adding, deleting and querying links of associations. Traits may use these APIs inside provided methods to achieve their needed implementation. With this approach, we increase cohesion because we will have more related provided methods inside traits and reduce coupling at design time because of having template parameters. Furthermore, the ab-

straction level of traits will be increased because we will not use attributes to establish relationships and instead utilize the more abstract notion of associations.

Listing 68. Observable pattern with traits and their associations

```
1  class Dashboard{
2    void update (Sensor sensor){ /*implementation*/ }
3  }
4  class Sensor{
5    isA Subject< Observer = Dashboard >;
6  }
7  trait Subject <Observer>{
8    0..1  -> * Observer;
9    void notifyObservers() { /*implementation*/ }
10 }
```

Listing 68 and Figure 47 depict a simple version of the observer pattern [43] implemented based on traits. As can be seen in line 7, the concept of the subject in the observer pattern has been implemented as trait *Subject*, which gets its observer as a template parameter. A direct association has been defined in trait *Subject* (Line 8), which has a multiplicity of zero or one on the *Subject* side and zero or many on the *Observer* side. This association lets each subject have many observers and it can also be used for when observers do not need to know the subject. The trait has encapsulated the association between the subject and observers and then applies it to proper elements when it is used by a client.

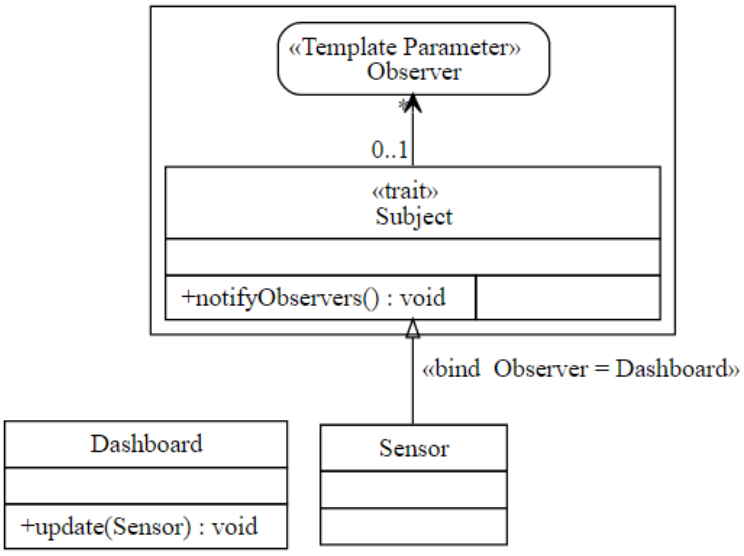


Figure 47. The diagram corresponding Listing 68

As each subject must have a notification mechanism to let observers know about changes, there is a provided method *notifyObservers()* for this. This method obtains access to all observers through the association. Two classes *Dashboard* and *Sensor* play the roles of observer and subject. Class *Dashboard* has a method named *update(Sensor)* (line 2) used by the future subject to update it. Class *Sensor* obtains the feature of being a subject through using trait *Subject* and binding *Dashboard* to parameter *Observer*.

3.7. Required Interfaces in Traits

Required functionality of classic traits is defined in terms of required methods. However, there are shortcomings with this approach. The first shortcoming is that there is no way to reuse a list of required methods. For example, consider a case in which there are traits that happen to have the same set of required methods but different provided methods. In this case, there is duplication due to repeated listing of the same methods. Furthermore, if there are several traits that must always have the same list of required methods, an inconsistency could be introduced in the design by changing just one of them and not all.

Listing 69 and Figure 48 describe an example of this shortcoming. As can be seen, traits *T1* and *T2* have two common required methods *method1()* and *method2()* (lines 2-3 and 7-8). They also have different provided methods *method3()* (line 4) and *method4()* (line 9) respectively. The required methods of traits *T1* and *T2* must always be kept the same because we want to have composed trait *T3* (line 11), which brings together provided methods of two traits (line 12). These provided methods must always achieve their functionality from the same required methods. Since traits *T1* and *T2* are two separate traits and can be extended separately, there is no way to guarantee those required methods will be kept the same during software maintenance. In other words, there is the possibility one of those traits might be modified without applying the change to the other one.

From a different perspective, another shortcoming appears when we know the clients of traits and that they must implement certain interfaces in order to have a correct implementation for the required methods. In fact, there is no way to put dependencies at the semantics level on clients through traits because required methods are simply syntactically captured. This suggests that it might be a good idea to put a restriction on clients that specifies the *interfaces* they must implement. Such a restriction would ensure that traits are not used in cli-

ents that just happen to have methods with the same signature. This is important because having reusable elements that work correctly with a minimum number of errors should be the designers' responsibility and not the user's [68]. Later, we will show a solution to this.

Listing 69. Duplication and potential inconsistency in required methods

```
1  trait T1{
2    abstract void method1();
3    abstract Double method2();
4    Float method3(){/*implementation*/ }
5  }
6  trait T2{
7    abstract void method1();
8    abstract Double method2();
9    Float method4(){/*implementation*/ }
10 }
11 trait T3{
12   isA T1, T2;
13 }
14 class C3{
15   void method1(){/*implementation*/ }
16   Double method2(){/*implementation*/ }
17 }
18 class C1{
16   isA C3, T1;
20 }
21 class C2 {
22   isA C3, T3;
23   Boolean method3(){/*implementation*/ }
24 }
```

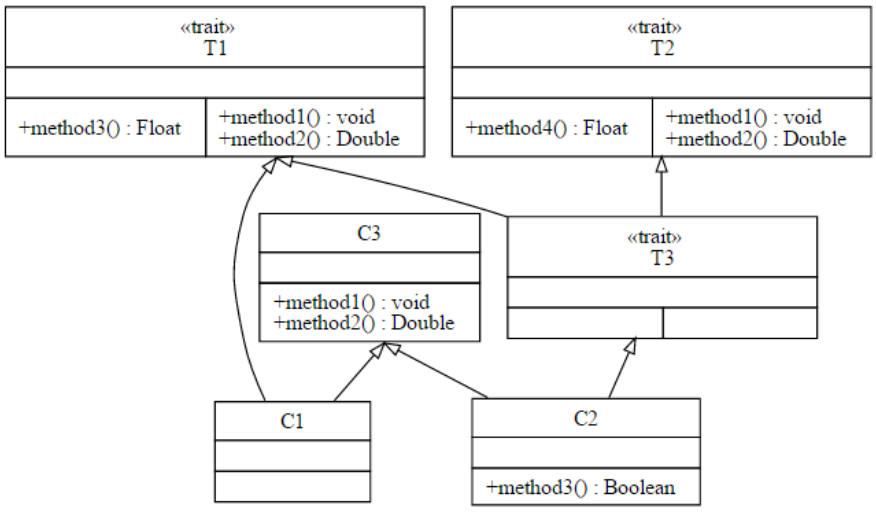


Figure 48. The diagram corresponding to Listing 69

Listing 70. Traits with incomplete set of required methods

```
1  interface I1{
2    void method1();
3    Double method2();
4  }
5  trait T1{
6    abstract void method3();
7    abstract Float method4();
8    void method5(){/*implementation*/ }
9  }
10 class C1{
11   isA I1, T1;
12   void method1(){/*implementation*/ }
13   Double method2(){/*implementation*/ }
14   void method3(){/*implementation*/ }
15   Float method4(){/*implementation*/ }
16 }
17 class C2{
18   isA T1;
19   void method3(){/*implementation*/ }
20   Float method4(){/*implementation*/ }
21 }
```

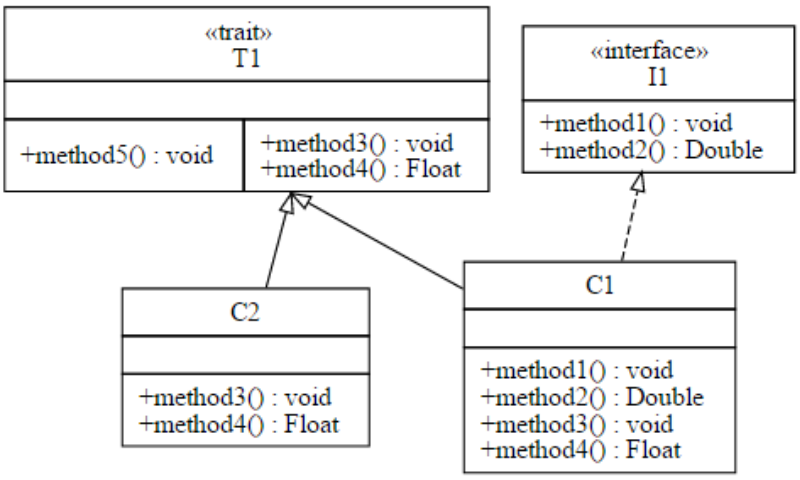


Figure 49. The diagram corresponding to Listing 70

Listing 70 and Figure 49 depict this shortcoming, in which two classes *C1* (line 10-16) and *C2* (line 17-21) have the ability to satisfy required methods of the trait *T1*. In addition, imagine that the correct satisfaction of required methods depends internally on clients which implement interface *I1* (line 1-4). According to the specification (that is not clear

here), trait *T1* should not be used by class *C2*, while it has been used. This gives rise the idea that there should be a mechanism to let traits specify more precisely what they want.

Traditionally, there is no straightforward way to apply a mechanism to avoid this issue. Of course, it can be done implicitly by defining all abstract methods of interfaces as required methods, but again in that case, there will be duplication and the issue of inconsistency. Furthermore, this makes traits have lots of required methods, which results in them being less readable and understandable.

Listing 71. Introduction of required interfaces to reduce duplication and inconsistency

```
1  interface I1{
2    void method1();
3    double method2();
4  }
5  interface I2 {
6    isA I1;
7    Boolean method3();
8  }
9  trait T1{
10   isA I1;
11   Float method3(){/*implementation*/ }
12 }
13 trait T2{
14   isA I1;
15   Float method4(){/*implementation*/ }
16 }
17 trait T3{
18   isA T1,T2;
19 }
20 class C3{
21   void method1(){/*implementation*/ }
22   Double method2(){/*implementation*/ }
23 }
24 class C1{
25   isA C3, I1, T1;
26 }
27 class C2 {
28   isA C3, I2, T3;
29   Boolean method3(){/*implementation*/ }
30 }
```

In order to address these issues, we extend traits with *required interfaces*. Using these, traits can either put extra restrictions on clients or just manage their required methods in a more modular and reusable way. Traits may use already-existing interfaces or new interfaces may be written to accomplish the desired modularization. Furthermore, developers

will be able to create a hierarchy of interfaces to optimize the reusability. Traits define their required interfaces through the keyword *isA* followed by the name of interfaces and a semi-colon.

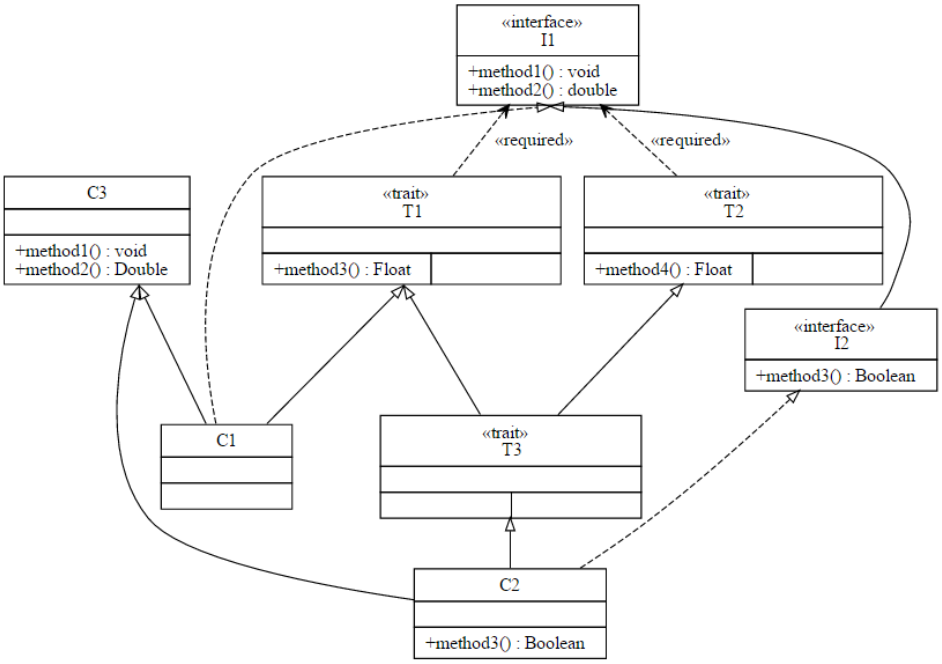


Figure 50. The diagram corresponding to Listing 71

When a class uses traits, it needs to implement the required interfaces of those traits, otherwise, the Umple compiler detects missing interfaces and raises error code 222. If a trait uses other traits with required interfaces, those required interfaces are added to the set of required interfaces of the trait and final clients are required to implement all of those required interfaces.

The new design for the example in Listing 69 is shown in Listing 71 and Figure 50. There is now a hierarchical design for required methods in terms of interfaces, making it reusable and consistent. Traits *T1* and *T2* now have the same required interface (lines 10 and 14) and if there is a modification in the required interface it will be applied to both. Classes *C1* and *C2* have to implement interfaces *I1* (Line 25) and *I2* (line 28) to be able to use trait *T1* and *T2*.

3.8. Transformations and Code Generation

Model-driven development allows and recommends developers to model systems abstractly and focus on high-level functionality without concern for implementation details. When possible, the goal is to generate implementation code automatically, a process either called model-to-code transformation or code generation. One designs systems with abstract elements in order to focus on business problems instead of technology, to have fewer errors, and to increase the speed of development.

In this section, we discuss and describe how systems which are modeled using traits can be implemented using automatic code generation. We discuss different strategies that depend on the type of target language and then describe our own automatic code generation used by Umple.

As pointed out before, traits were first introduced and implemented in Squeak [54] and then in other languages like PHP [108]. These constitute the first group of languages that have native keywords or structures for representing traits; their compilers are aware of traits and can analyze them. The second group of programming languages such as Ruby [110] and Javascript [100] support traits, but without specific keywords for them. In these cases, developers adapt other structures of the language to implement traits. However, several of the most important languages such as Java and C++ do not support traits at all. There has been some research towards adding traits to these languages, but traits have never become a part of their standard versions [21,33,65,83,85,86]. When traits are represented in models, there will be three options for implementing the modeled systems corresponding to the three groups of programming languages described above.

The first option applies to programming languages that directly support traits. In this case, automatic code generation is straightforward because there will be a one-to-one mapping from traits in the modeling level to traits in the implementation level. It is worth pointing out, however, that currently these languages do not support associations, required interfaces, and state machines as proposed in this thesis. Therefore, there are not direct one-to-one mapping for these concepts.

The second option is associated with programming languages that do not have a direct keyword for traits but provide structures used to mimic traits. The implementation for these languages will be a little bit different than for the first group because there will not be a

direct mapping between traits in modeling and implementation levels. However, the mapping will happen conceptually because each trait will be implemented with the required structures in the generated language. In this automatic code generation, using best practices (for implementing traits) will play the most important rule because we would like to use the minimum combination of structures and to have as modular as possible a representation of the implementation of traits. This would improve the process of understanding of the final systems for who are code-oriented or need to inspect the final system.

The final option, for languages not supporting traits at all, is to directly base code generation on the idea that compilers typically use flattening to inject provided methods into clients. After the compiler does this, all elements of traits are treated as real elements of clients, and clients have access to those elements just like any other elements. Code generation from model-based traits can do this directly for programming languages that do not support traits.

This third approach, however, requires greater intelligence in analyzing the traits at the model level to ensure the validity of the final systems. Since we chose the final option, in Umple the compiler does considerable analysis of the traits and provides many warnings and error messages when syntactic and semantic problems are identified in the defined traits. Appendix I, for example, shows a summary of error and warning code we detect to make sure the final model (system) will be valid.

It should be noted that the approach we describe in this thesis implies that there should not be any *round-trip* model transformation. This means that there should be just a direct transformation from model to code and developers should not modify the generated code. When there is a need for modifications, they should be first applied to the model and then the code generation should be reapplied to update the final system. This approach is just like the standard approach for compiling a high-level programming language and is the preferred approach in model-driven development. Allowing round-tripping (taking modifications of generated code and applying them to update the model, then regenerating the code) would be too complex in the context of traits; it has always been against the Umple philosophy too.

The reason for the complexity of round-triping for traits is that it is hard to find duplicated elements in the generated code. Even if it is possible, it might even be harder to re-

late generated code to its modeling elements. For example, state machines and associations are modeling elements that are implemented by different techniques at the code level and so it is hard to express whether specific code represents a state machine at the modeling level or not. Being able to achieve this capability is considered as future work.

Since Umple supports code generation for languages such as Java and PHP, we wanted to provide a mechanism to be reusable by all code generators. Otherwise, we had to re-implement our composition algorithm and validity checks in each code generator. Therefore, we introduced a layer of the model-to-model transformation inside the Umple compiler that applies flattening to the Umple compiled model and so there is no need to modify already designed code generators. This also allows us to provide a module extension to Umple that can be enabled and disabled easily. This was also critical to provide a mechanism to switch easily between the flattened and normal model in our cloud-based IDE. The model-to-model transformation obtains an Umple model as its input and returns an Umple model that has all composition algorithms and flattened elements applied to it (described in Section 3.3). The final Umple model then is delivered to the code generators to generate the code for specific languages chosen by modelers.

The Umple models are the master file, and we do not want developers to edit generated code. However, we provided a simple mechanism to track traits in the generated code because it helps when people want to inspect or to certify the generated code. It also provides information to debuggers. There is a traceable annotation for each provided method which indicates exactly the name of a trait from which this method comes.

All validity checks and the composition algorithm for state machines have been committed to the Umple compiler on Github and are freely accessible to investigation or adaptation by other modeling languages.

3.9. The Traits Metamodel

One of key factors in understanding a model-driven technology is its metamodel. In this section, we describe the metamodel of traits. The metamodel of traits is constructed in combination with some of the modeling elements already defined in the Umple compiler. Figure 51 shows a simplified version of traits' metamodel. This metamodel is used to describe the nature of elements utilized to implement traits in Umple. The comprehensive metamodel of

traits along with the Umple compiler can be found in its online version [50]. The classes in the metamodel are as follows. The first eight of these existed prior to our work, but have been extended in our work:

UmpleElement: This is an abstract class that is one of the top-level items found in an Umple metamodel. Currently, it has one subclass which is *UmpleClassifier*. It includes information regarding the positions of the element, package name, extra code, and so on.

Method: This represents the concept of methods in Umple. Methods can be abstract and have return types and parameters. If a method is not abstract, then it can have a body that indicates its execution logic. A method can have several bodies, each one representing an implementation in a different programming language.

UmpleClassifier: This is an abstract class that is a subclass of *UmpleElement*. It includes dependencies on external libraries, *UmpleModel*, and so on. It has a zero-to-many association with class Method. *UmpleClassifier* also manipulates the token obtained through the Umple parser. This class has three subclasses, *UmpleInterface*, *UmpleClass*, and *UmpleTrait*.

Attribute: This represents the concept of attributes in Umple. An attribute has a type and name and it can also have modifiers.

Association: This represent the concept of associations in Umple. It encapsulates ends of an association, role names, and multiplicities.

StateMachine: This represents the concept of state machines in Umple. It encapsulates all elements that can be defined in a state machine such as states, transitions, and so on.

UmpleInterface: This represents the concept of interfaces in Umple. It is a subclass of *UmpleClassifier*.

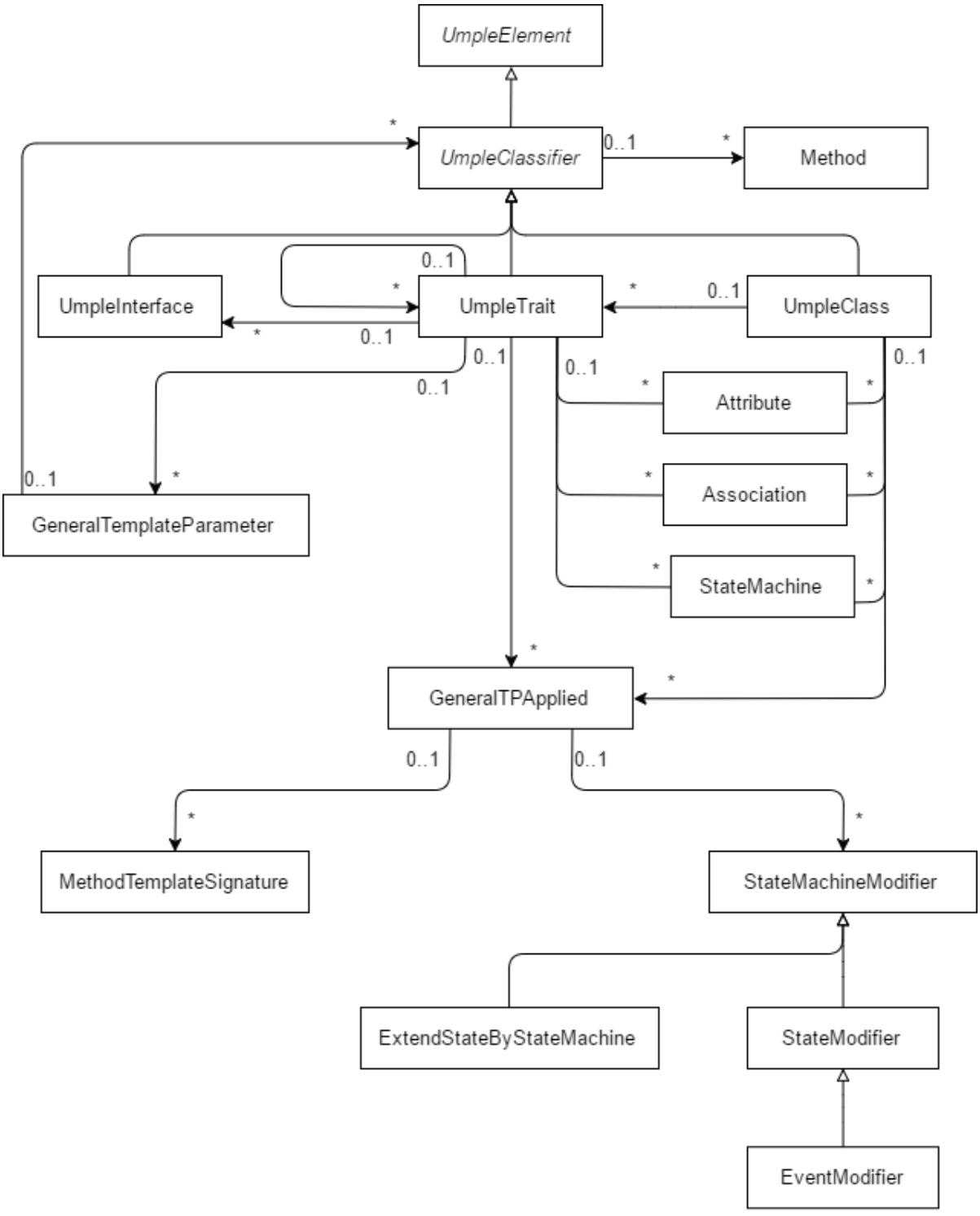


Figure 51. A simplified metamodel of traits

UmpleClass: This represents the concept of classes in Umple and is a subclass of *UmpleClassifier*. It has a zero-to-many association with classes *Attribute, Associations, StateMachine, UmpleTrait* (which represent traits), and *GeneralTPApplied* (which encapsulates values and operators applied to traits). It also includes other elements defined for classes in Umple. A zero-to-many association with *UmpleTrait* satisfies the fact that classes as clients can use as many traits they want. A zero-to-many association with *GeneralTPApplied* expresses that fact that many operations can be applied for each trait. The number of *UmpleTrait* and *GeneralTPApplied* for each class is equal.

UmpleTrait: This represents the concept of traits in Umple and is a subclass of *UmpleClassifier*. A trait can have a zero-to-many association with classes *UmpleInterface, GeneralTemplateParameter, GeneralTPApplied, Attribute, Association, StateMachine,* and *UmpleTrait*. The associations with the classes are used for the following purpose:

- UmpleInterface: to save the list of required interfaces defined for the trait.
- GeneralTemplateParameter: to save the list of template parameters defined for the trait.
- GeneralTPApplied: to save values and operators applied to each trait used inside the trait.
- Attribute: to save the list of attributes defined in the trait.
- Association: to save the list of associations defined in the trait.
- StateMachine: to save the list of state machines defined in the traits.
- UmpleTrait: to save the list of used traits by the trait. A trait cannot have itself in this list.

GeneralTemplateParameter: This presents the concept of template parameters for traits. *GeneralTemplateParameter* encapsulates the name of the parameter and holds a zero-to-many association with *UmpleClassifier*. The association is used to specify constraints defined for the parameter. Although *UmpleTrait* is a subclass of *UmpleClassifier,* the values for this association can just be interfaces and classes.

GeneralTPApplied: This class represents all values and operators applied to a trait while it is used by a client. Because of it, both kinds of clients, *UmpleTrait* and *UmpleClass*, have an association with it. *GeneralTPApplied* has a zero-to-many association with

MethodTemplateSignature and *StateMachineModifier*. It also holds the values bound to template parameters. The associations with the classes are used for the following purpose:

- MethodTemplateSignature: to save all operators applied to provided methods.
- StateMachineModifier: to save all operators applied to state machines

MethodTemplateSignature: This represents the class encapsulating an operator applied to a trait. It includes the signature of the provided method and modifiers such as symbols '-', '+', and `as`.

StateMachineModifier: This class represents the class encapsulating operators applied to state machines of traits. *StateMachineModifier* by itself deals with any operator applied directly to the state machine and not to its internal elements. These operators are renaming, removing, and keeping state machines. Other operators are subclasses of this class because they need the state machine information for all of their operations. The subclasses of *StateMachineModifier* are classes *ExtendedStateByStateMachine* and *StateModifier*.

ExtendedStateByStateMachine: This class encapsulates the operator used to extend a state. It includes the name of destination machine and the required hierarchy of states. The name of source state machine is stored in its superclass.

StateModifier: This represents all operators involving states. It holds the hierarchy of states and other values specific to each operator. The name of state machine and type of modifier are stored in its superclass.

EventModifier: This represents all operators applied to transitions. It holds the signature of the event and the guard expression. The name of the state machine, the hierarchy of states, and the type of modifier are stored in its superclass.

3.10. Summary

In this chapter, we first introduced the requirement of model-based traits, described our algorithms, and explained the semantics of activities performed in these algorithms. Then, we explained how model-based traits can be defined in Umple. We described the syntax and semantics of model-based traits in Umple. We explored how model-based traits can be implemented in object-oriented programming languages, which is achieved automatically in Umple. The metamodel of traits was also explained.

In the next chapter, we discuss research that is in the close relationship with our work. This helps to understand how those research affected our work and how our work could improve them. Since there is no direct work about having traits at the modeling level (as long as we are aware of), we discuss research conducted in the domain of traits, inheritance, and composing modeling elements such as state machines.

Chapter 4. Related Work

In this chapter, research that is most closely related to our research is explored. Each research project is first described briefly and then the research's contributions to the present research and vice versa are investigated. We categorized related work into two sections. The first is about basic traits and the second is about state machines and how they are merged or extended for developing systems. Most of the related research to the first section has already been described in Section 2.2, but we will add some other aspects not discussed in that section.

4.1. Other Research about Basic Traits

In this section, we explore some research that has been conducted regarding traits and their basic capabilities.

4.1.1 Traits in Statically-typed Class-based Languages

The main issues and the possible strategies for integrating traits into statically-typed class-based languages are explored in depth by Nierstrasz et al. [74]. The first issue to be considered is: What is the relationship between traits and types? Furthermore, if there is a relationship (integration) between classes and traits, how should trait methods be typed? In statically-typed languages types must be defined in advance, but this could prevent traits from being generic enough to be applied to many different classes, some of which may not be known at compile time. Nierstrasz et al. also expressed that there are many other trade-offs that arise when traits are added to another language. For example, implementing traits by compiling them away is an easy solution, but it makes debugging harder.

To resolve these questions, Nierstrasz et al. [74] proposed a formal flattening-based model and strategy to add traits to these kinds of languages. Although this model is not comprehensive, it could resolve several of the sophisticated issues (pointed above) related to the integration. The authors have used this model to examine how traits can be integrated into C#. Nierstrasz et al. also tried to simulate traits in C++ using template parameters and (virtu-

al) multiple inheritance. They discuss the consequences of such a strategy: C++ allows implementing trait-like composition by using a combination of nested templates and multiple inheritance with virtual base classes. It can also simulate the renaming (alias) operator by disciplined use of scope modifier ::, but the removing (exclusion) operator cannot be achieved in the same way that is equivalent from a compositional point of view. Nierstrasz et al. also expressed that this kind of traits (represented as template classes) cannot be distributed and linked as a library and the source code of traits must be present at compile time.

The difference between our work and that of Nierstrasz et al. is that we support traits at the modeling level and when it comes to implementation at the code level, we do not use template parameters and multiple inheritance. We use single inheritance and then flatten elements of traits to classes. Our approach also provides native operators for traits to resolve conflicts and even change the visibility of provided methods. Flattening may cause duplication in classes that use traits, but Nierstrasz et al. also indicated that their version of traits cannot be compiled separately and they need to be compiled in the context of each class, which results in duplication in object-code.

Having typed trait inheritance is explored by Liquori and Spiwack [65,66] in which they developed an extension called Featherweight-trait Java (FTJ) for Featherweight Java (FJ) [53]. The main goal of the work was to introduce typed trait-based inheritance as a simple way to provide a simple type system that typechecks traits when imported in classes. In this extension, traits do not have state and are used as behavioral bricks to build classes. The authors have placed special emphasis on dealing with the classical issue of multiple inheritance called the “diamond” conflict. This work could be considered as a first-step to adding traits to statically-typed class-based languages, in particular, Java.

In our work, we do not consider traits as types, but we use traits to build classes as well. Not having traits as types can be mitigated in our approach by the fact that there is no need to implement required methods of traits in abstract classes. Therefore, when a trait is used by an abstract class it can be considered as an identical type for the trait. Furthermore, our approach does not just resolve the problem of multiple inheritance regarding methods, it also resolves the problem of multiple inheritance when modeling elements are considered in design of systems instead of methods.

Supporting traits in Java was also explored by Quitslund [82,83]. In that research, the goal was to explore how to resolve barriers of reuse in Java through traits via a case study of Java Swing. These barriers include the lack of multiple inheritance, inaccessible private inner classes, non-extensible final classes, and synchronized variations. In addition, traits were directly implemented as an extension to mini-Java (MJ), a subset of Java, by devising a compiler (TMJC) that works by translation, taking source extended with traits, and translating it to pure mini-Java. An Eclipse-based environment for this was created [84]. In this, a programmer can move freely between views of the system with or without its traits. The advantage of representing traits as classes is that existing Java development tools can be used.

Our approach implement traits in Java with pure Java structures (classes and interfaces). However, we do not use classes to model traits. Our approach instead has its own syntax and semantics for traits and performs all validity checks before transformation of traits to pure Java code. This also enables us to have modeling elements as part of traits, which is not possible in Quitslund's work. Having separate syntax and semantics also removes the ambiguity that arises when classes are used for two different purposes. We also have an environment in the cloud that allows moving freely between views. We do not have an Eclipse plugin to achieve such benefits when Umple is used in Eclipse. However, modelers can generate the diagrams' scripts and render them locally in their machines.

Emerson et al. [71] proposed an implementation for Java based on their study of java.io libraries. In their research, traits are represented as stateless Java classes. Required methods are expressed as abstract methods. Classes representing traits can be used with other classes to produce composite classes. According to their implementation, a class can be used both through inheritance and through composition. This implementation is in contrast with their earlier proposal of type traits [82] in which traits were special program units.

In our work, we also consider abstract methods as required methods, but we do not consider it for classes because our approach has a syntax for traits so there is no mixing this with the concept of classes. The composition in our approach is achieved among traits and classes while it is performed through classes in Emerson et al.'s work. Furthermore, their approach also does not deal with modeling elements. It is worth noting if such implementation is necessary for some Java projects, the only modification required in our approach is to de-

velop a new code generator to map our defined metamodel to Java code. In this case, our metamodel and rules defined for traits would be reusable as well.

Attempts in the direction of exploring traits in Java resulted in the research of Denier [33] in which AspectJ has been utilized as a subset of aspect-oriented programming [56,111]. The key idea behind this was a feature called inter-type declaration in aspect-oriented programming. The inter-type declaration mechanism offers a subset of structural reflection, which allows for extension or modification of classes at compile time. The proposed solution could implement most of properties of traits but it was not able to provide full support regarding conflict resolution. The main reason for this shortcoming was the lack of fine-grained operators in AspectJ. Although aspects could implement capabilities of traits, there are some philosophical differences that we will discuss in Section 4.2.6 when we talk about state machines.

XTRAITx, a language for pure trait-based programming, was introduced by Bettini and Damiani [21]. Their research achieves complete compatibility and interoperability with Java without reducing flexibility of traits. It also provides an incremental adaptation of traits in existing Java projects based on Eclipse. In this implementation, classes play the role of object generators and types, while traits only play the role of units of code reuse and are not types. In this research, there are several operations for traits (as conflict resolution methods) including method alias, method restrict, method hiding, and method/field rename.

Trait-based programming has all operators we have in Umple for basic features of traits (except changing visibility of provided methods), but it does not support enhanced features we have for Umple including required interfaces, associations, template parameters, and state machines. Like trait-based programming, we also do not consider traits as types and use them as units of code and model reuse instead of just code reuse. Trait-based programming provides traits for Java, but traits in Umple are implemented in all languages generated by the Umple generator. As far as we are aware, this feature is only available in our approach through Umple.

4.1.2 Stateful Traits

We talked about stateful traits [18] in Sections 2.2.1 and 2.2.5, and extend that discussion here. The reason stateful traits were introduced is that if required accessors are going to be

public (which is a case in the Smaltalk implementation, for example), they can violate the encapsulation of client classes. Furthermore, if a trait is modified to have additional state elements, new required accessors will be propagated to all client traits and classes. Therefore, this introduces a form of fragility that traits were intended to avoid.

This challenge is resolved in stateful traits by letting traits define instance variables (or attributes) [18]. Instance variables are purely local to the scope of a trait unless they are explicitly made accessible by the clients of the trait. In other words, instance variables are private to a trait unless the client decides to either access them or merge instances variables coming from multiple traits at composition time. The client also can map those instance variables to possibly new names. The new names will be private to the scope of the client. The conflicts regarding the names are avoided and instance variables from disjoint traits can be merged by the clients. In this approach, the clients still retain the control of the composition. Bergel et al. [18] also proved [19], by means of a formal object calculus, that adding state to traits preserves the properties of the system when the traits are flattened.

Traits in Umple support attributes, but they are not local to traits. In other words, Umple has stateful traits that explicitly make attributes accessible to clients. The main reason for having attributes accessible comes from the fact that we flatten all traits elements at the modeling level to clients so it is not possible to have two attributes with the same name for a class. We believe there are some cases in which having attributes local is beneficial, but it can be achieved differently in Umple through having unique names for attributes. This unique name can be built by combining the name of attribute with the name of its trait. Since the name of traits are unique in Umple, we will obtain unique names for attributes automatically.

4.1.3 Traits and Software Product Lines

Application of traits in software product lines (SPL) was investigated by Bettini et al. [22,23]. Bettini et al. used traits along with records [27] to model the variability of the state part of products explicitly. In their approach, class-based inheritance is ruled out and classes are built only by composition of traits, interfaces, and records. They introduced Featherweight Record-Trait Java (FRTJ), which ensures type-safety of a SPL by type-checking its

artifacts only once and ensuring type-safety of an extension of a (type-safe) SPL by checking only the newly added parts.

Damiani et al. [32] also propose an approach for SPLs of trait-based programs. It is based on delta-oriented programming (DOP). In their approach, program modifications are expressed by delta modules and formulated by using the trait composition mechanism. The approach has been realized in the programming language DELTATRAITJ. The approach provides a flexible inter-product and intra-product code reuse.

Our work can be extended to be used in the context of SPLs because of the capability it has in providing reusable assets (we consider additional exploration in this area as future work); other researchers are looking into the best way to achieve that. Furthermore, our work deals with modeling elements for traits such as state machines, but the work of Damiani et al. [32] does not cover this dimension.

4.2. Research on State Machines in Traits

Our work is unique in the manner it reuses and extends state machines. In this section, we explore related work regarding reuse, extension, and composition (including merging) among state machines and their clients.

4.2.1 Resolving Inconsistencies

State machines can be used in early phases of software development to describe or extract requirements and design alternatives. Merging state machines can be used to identify, track, and resolve inconsistencies [37,46,59,87]. State machines are also used to build a full representation of the behaviour of the system under development.

Easterbrook et al. [38] proposed a framework called Xbel supported by a multi-valued model checker, Xchek, to merge and reason about multiple inconsistent state machine models. These inconsistent state machines can arise from the different perspectives of stakeholders. The framework acts as an exploration tool to support requirement negotiation. It does not restrict analysts to use a specific method of merging models or handling inconsistencies. It helps to reason about merging inconsistent models (or disagreements) that can affect the critical properties of the final system.

Our approach deals with state machines at design time and it assumes that conflicts related to different perspectives of stakeholders have already been resolved. Conflicts that arise during reusing state machines are deemed to be like conflicts that happen in programming languages. Our approach has been enriched with practical operators to deal with these kinds of conflicts.

Sabetzadeh et al. [89] used a category-theoretic approach for representation and analysis of inconsistency in graph-based viewpoints generated in requirements. Their approach is able to parameterize inconsistency through lattices. The designer must explicitly identify interconnections between viewpoints before the merge process happens. In our approach, traits can partially be considered as viewpoints. However, it is possible to define required behavior in traits and also modify elements when they are reused in clients. In our approach, composition of elements is performed based on implicit rules defined in the Umple compiler. For example, state machines with the same names are composed. However, modelers can use provided operators to explicitly interconnect elements (indeed, overwrite implicit rules we have defined as part of our implementation).

4.2.2 Merging of State Machines

Uchitel al. [98] proposed model merging as an approach in order to compose partial behavioral models described by different stakeholders that may have different views or concerns. They formally defined model merging as observational refinement and argue that merging consistent models results in a minimal common refinement. Since minimal common refinements are not unique, modelers must participate in the merging process. Their approach also provides an algorithm for checking consistency between two models. In their approach, the goal is to augment the knowledge about the final system by obtaining the common refinement knowledge from different partial behavior descriptions. In other words, the goal is to preserve the behavioral properties instead of the model structure.

One of the differences of our approach compared to Uchitel et al. [98] is that our approach preserves the model structure. The reason is that our approach is used to build the final system instead of providing knowledge at the early stages. Therefore, any change in the model structure, if needed, must be performed by modelers. Furthermore, composition in our approach can be considered as the process of selecting the most appropriate common refine-

ment. In order to enable modelers to participate in the composition process of our approach, required operators have been defined.

Nejati et al. [72] described two operators, match and merge, to manage models in model-based development. The operator *match* is used for finding relationships between models and merge is used for combining models in terms of the existing relationships. In their approach, hierarchical state machines can be matched and merged in terms of both structural and semantic information. Their approach preserves the behavioral properties of models. The approach has been designed for models developed independently and can be used for states reached from the initial states. The tool used for their approach has not been maintained since the publication date.

In our approach, we also have these two operators needed to compose elements. For matching, structural information of elements is used in order to make sure embedded implementations (such as actions and activities) are not going to be affected because of behavioral merging. Nejati et al.'s approach [72] can be deemed as an informational extension to our approach by which modelers can be informed about elements that have the same behavior. Then, if the results make sense to modelers, a proper operator can be used to map and merge them. We consider our systematic mapping and composing better than implicitly merging based on behavior because it can be tracked statically later by modelers. In other words, when operators are used to explicitly interconnect elements to each other for merging purpose, they become part of the static model that can be tracked. This is not possible when merging is performed implicitly because elements are matched dynamically by the matching algorithm and it is not part of the model.

Sabetzadeh et al. [90] also propose a framework in which relations between models are first-class artifacts in merging models. They describe the framework by applying it to merging state machines, and also a tool [90] they developed for this purpose. The paper answers questions about how to specify the relationships between models when we want to merge them. This paper points out two important concerns about the specification of model relationships. The first one concerns the flexibility in the way that concepts from a set of models are interrelated to each other. The second expresses that there should be different merging algorithms for different phases in the development process because each phase has its own concerns and requirements. They have implemented a proof-of-concept tool called

TReMer based on their framework. The advanced version of the tool, TREMer+ [91], considers merging models as a way to check the global consistency of distributed models. The idea is to avoid the complexity of checking the consistency of the final system through pairwise checking rules and to perform the checking when models are merge.

Our approach covers two concerns pointed by Sabetzadeh et al. [90] through providing operators to make interconnection among state machines elements (if needed) and an algorithm based on traits, which allows reusing, extending, and composing state machines at the design and implementation level (described in Sections 3.3.3 and 3.5). This is one of our reasons why we opted not to consider behavioral matching because, at this level of modeling, which results in the final implementation, structural properties are a valid source of composition. Furthermore, our approach supports two levels of consistency checking: at the definition and composition levels. The first checks consistency without paying attention to the used traits (state machines). It detects conflicts such as nondeterministic events. The second is done when the composition happens and includes detecting variable conflicts.

Walkinshaw et al. [101] worked on an approach to compare models based on their structure and behavior (language). This approach uses a scoring mechanism to find the similarity of states. In their approach, states do not have labels and the score is computed by matching the surrounding network of states and transitions. In fact, Walkinshaw et al. have used binary classification performance measures to quantify the differences among models. Their research tries to resolve the issue with existing language comparison approaches, in which they fail to effectively and reliably compare the full language because the essentially impossible sequences are not considered alongside the possible sequences. Indeed, the techniques are primarily motivated by the specific challenge of evaluating the accuracy of state-machine reverse-engineering techniques.

In our approach, we consider that states have labels and when they are combined with the states' hierarchy level (and region names), unique identifications are achieved for states. This is the main factor for merging states. The algorithm of Walkinshaw et al. [101] can be used when modelers want to explore possible equal states in the composite final system based on the structure and behavior. Then, they may use the results to merge states explicitly.

Pradel et al. [81] present a comparison method for state machines extracted from specification miners, in particular, method-ordering constraints. The idea is to provide an

evaluation mechanism which allows comparing different mining approaches objectively instead of subjectively. The framework has two steps. In the first step, the specification of the API is expressed in a lightweight formal language, utilized to generate a reference finite state machine (FSM). Then, the reference FSM is compared with the FSM mined by the approach. Pradel et al.'s approach uses a k-tail algorithm [24] to merge states. Our work can be differentiated from that of Pradel et al. [81] in that there is no need to know exactly how two state machines inside traits are equal because what is important is to know if two state machines (or states) are the same or not. If they are not, the composition algorithm is going to take care of it. Furthermore, if the matching algorithm of our approach was expressed based on the k-trail algorithm, the value of k would be one.

4.2.3 Software Product Lines

State machine merging has also been explored in software product lines [80]. Beidu et al. [16] provided a framework called FeatureHouse for composing feature-oriented state machines. The framework has two tools which allow generating a feature structure tree (FST) for every feature and composing FSTs using superimposition. The input of the framework is a feature state machines (or state machine fragments) and the outputs are two different models. One model is used to facilitate the composition of future feature state machines and the second is the whole product line, with optional features guarded by presence conditions. The authors have also proposed a feature merge expression that allows incrementally modifying feature transition systems [10]. Umple currently does not support feature models [55], but our approach could satisfy the merging part of Beidu et al. [16]'s framework. This exposes the fact that our approach also has the potential to be used in the domain of software product engineering.

4.2.4 Distributed and Object-Oriented Systems

Development methods of distributed systems have also utilized state machines. ROOM [95] is a methodology for developing distributed real-time systems. It is based on two complementary modeling paradigms, modeling dimensions, and abstraction levels. Modeling dimensions are composed of Structure, Behavior, and Inheritance while abstraction levels consist of System, Concurrency, and Detail. It has been indicated that in the behavior part of ac-

tors, inheritance has been added to the basic statechart formalism. For example, a subclass can refine the behavior of a superclass by decomposing what is a leaf state in the parent into a nested state machine.

Our approach provides such a feature through traits. However, it surpasses the limitations of inheritance (described as different scenarios in Section 1.1.1). Required methods can be mapped to ports, defined in ROOM, but they are not forced to trigger an event of state machines. Each message in ROOM is assigned to a port, but our approach has a direct messaging mechanism. In our approach, structural boundaries for state machines are provided so as to allow using, integrating, and moving them easily in or between software systems. This is accomplished in ROOM through assigning them to actors. ROOM also lacks the ability to track event flows that span multiple objects over time [94]. This issue in our approach can be moderated by the fact when the composed state machine is obtained for a class and all events can be tracked for one object. This can be completed by a feature of Umple which allows generating sample tables of sequences for a state machines. Of course, if tracking at the execution time is required, it can be achieved by the model-based tracing feature of Umple [7].

Objectcharts [30] is another approach for the design of object-oriented software systems in which state machines are used to characterize the behavior of classes. It proposes a hierarchical process to extend state machines and uses single inheritance. Although it provides clear steps regarding how to reuse state machines, it lacks features such as the ability to reuse a state machine freely in different levels of hierarchy (such requirements discussed in Section 1.1). Our approach follows the hierarchical process defined in Objectcharts, but it does not have the limitations of inheritance.

4.2.5 State Reduction

State machine composition in any form might end up generating many states; this might not be preferable for cases in which memory allocation is important. State reduction can be done through merging equivalent states. The responsibility is typically given to modelers to determine the states that need merging and to give them the same names so the algorithm can take care of merging. Avedillo et al. [11] propose a state assignment algorithm which uses state splitting and merging techniques during state assignment in order to minimize the area of the combinational component after logic reduction. In our approach, we do not minimize

the merged state machines because the number and name of state machines are important. Avedillo et al.'s approach can be applied to the final composed state machine produced from our approach.

4.2.6 Aspect-orientation

State machine composition has been researched in aspect-oriented software development [3,56] as well. Elrad et al. [40] proposed to extract crosscutting behavior from state machines and model them as aspects. Aspects and base state machines are modeled with orthogonal regions and utilize broadcasting and propagating events to communicate with each other. Mahoney et al. [67] improved Elrad et al. [40]'s method by allowing events to act as aliases for other events in order to specify the order in which events should be handled.

Zhang [102] used aspect-oriented techniques to extend the behavior of a state machine. Zhang concentrated on the execution history of state machines as a candidate for pointcuts. This brings a rich mechanism to specify pointcuts inside state machines. Ge et al. [45] incorporated aspects into UML state machines by having core and aspect state machines designed inside composite states. Then, the final model is woven based on the relative location of core composite states and aspect composite states. In fact, the final model by itself is modeled as another state machine with those composite states and other decisions states. The composite states communicate with each other through message passing.

Kienzle et al. [57] implemented a method to design reusable assets based on three notations: class diagrams, state machines, and sequence diagrams. They make use of a packaging mechanism called an aspect model to group structure, state, and message behavior. Their method also allows an aspect to use another aspect by means of inheritance in order to increase the reusability of aspects. The template parameters are added to the aspect model to maximize flexibility and reuse.

Although aspect-orientation is flexible and has received some attention by researchers [6,57,61], there are important methodological and technical differences between traits and aspects. Aspects were introduced to deal with crosscutting concerns and provide a mechanism to modulate and reuse them. Traits were introduced to deal with the problem of inheritance regarding not being able to get the maximum benefit of behavior defined inside classes. Furthermore, in aspect-orientation, aspects are elements that specify to which modeling

elements their behavior must be applied, but in traits, it is *clients* (traits or classes) that decide which traits must be reused. This overcomes an issue (that is also claimed to be the benefit) with aspects which is exactly what code is affected by an aspect. This effect is not easy to determine and can be changed unexpectedly. The differences can be summarized as follows:

I. Aspects model crosscutting concerns and their advice models can affect (cross-cut) more than one element in the model under development. Even though this is a great feature, it can be problematic. As a system is incrementally developed, it can be possible to have new elements that match pointcuts of some already-defined aspects, but they may not be part of the core concerns for which those aspects have been designed. This results in hidden side effects. For example, an aspect defined with a regular expression to log activities can be caused to log security data because a security module might add or rename an activity which accidentally matches the regular expression. Traits do not suffer from this side-effect because they just affect their direct clients.

II. Aspects and traits are used in different methodological ways. In aspect-orientation, core concerns (e.g., classes and state machines) are not responsible for specifying what they require to reuse. In fact, aspects specify what must be reused by core concerns (e.g., through binding templates). However, in trait-orientation clients decide what they need to reuse. One of the consequences of aspect-orientation is that developers need to utilize tools to debug or understand, for example, whether or not there is an aspect applied to an element, however, this can be interpreted easily in trait-orientation because elements directly express the list of reused traits.

III. The way an aspect reuses another aspect is different from the way traits use other traits. An aspect reuses another aspect through inheritance, which does not allow the detailed level of reuse among aspects. It also does not allow modifying the behavior coming from super aspects (to some extent this can be done through template parameters). However, traits can reuse other traits (more than one) with much better flexibility.

IV. Aspects allow specifying how advice models should be woven to the core concerns, but there is no such mechanism in traits. Traits follow some defined rules that govern how elements inside traits must be reused and composed with clients. Not having aspect weaving options (before and after) in traits may affect how elements inside traits can be reused. However, the simplicity coming from this can definitely help enable traits to be adopted by practitioners, in particular, object-oriented developers.

It is important to express that we have experienced items I and II through questions that we have received from new developers who joined the Umple team to help develop the Umple compiler itself. The Umple compiler has been developed using Umple and it benefits from the use of aspects. Several times it has been pointed out by developers that they implement methods, but those methods show some unpredicted behavior. They were not aware that their methods match some pointcuts of Umple compilers' aspects (item I). Having an IDE support, for sure, could be helpful in this case for Umple. However, this reveals that aspect-oriented models depend on support of specific IDEs. If there is no IDE support for them, it will be challenging to understand them. Trait-based systems can be explored without IDE features (for example, with a simple text editor). This is considered as one of traits' advantages.

In the same manner, they pointed out that there are some classes that do not have specific methods, but the final model shows them and when we introduced aspects to them, they asked us about a tool to show how many aspects are applied to a specific class (items I and II). The same conclusion as before is reached in this case: that aspect-oriented programs depends on specific IDE features to be understood well.

4.3. Summary

In this chapter, we discussed research conducted regarding classic traits. We described which languages support them and in which domains traits have been used. Afterwards, we focused on research conducted regarding reusing, extending, and merging state machines. These studies showed how our approach can be improved and how our approach may improve oth-

ers. For example, state reduction can be used in final phase of work to reduce number of states required to represent state machines of a system.

In the next chapter, we demonstrate how we evaluate our approach through implementing several cases studies. We also show how we concluded that our approach is practical. Finally, we explain how we adopted test-driven development to make sure our implementation of the approach in Umple is valid.

Chapter 5. Case Studies

In this chapter, we explain ways we have demonstrated and evaluated the practicality and effectiveness of our work by way of case studies. We have considered several goals as follows:

Firstly, we want to evaluate whether we can effectively model a system with the concepts of traits added to Umple. This is achieved through a case study that shows how traits can be used in Umple to model a functional system. The case study is explained in Section 5.1.

Secondly, we are interested to know if we can model and implement a system with traits that has already been developed with Umple without traits. This includes the process by which we should detect traits. In fact, the goal is to make sure our approach is capable of being used in large systems. Furthermore, we want to compare the systems with and without traits, to observe benefits and drawbacks of the version with traits. This part is explained in Section 5.1.3.

Thirdly, we want to evaluate whether we can effectively model a system with state machines defined in traits. This includes evaluating the syntax of state machines in traits and also the composition algorithm we have developed for state machines. For this purpose, we want to implement a system which already uses state machines, so we can see how many opportunities for reusable state machines our approach can uncover. This evaluation is described in Section 5.3.

Finally, we want to make sure what has been applied to the Umple compiler including syntax, semantics, and the composition algorithm are performing formally as they have been defined. Furthermore, the semantics and syntax of traits will not be changed accidentally as the Umple compiler extends over time. This is covered in Section 0.

5.1. A Geometric System

In this section, we implement a geometric system based on our approach, as a part of our evaluation. The rationale, design and results of the implementation are covered in the next sections.

5.1.1 Goals

The main goals of this case study are summarized as follows:

1. To confirm that we can define traits and their provided and required methods.
2. To confirm that traits can be reused by classes that already have a superclass.
3. To confirm that traits can be composed of other traits.
4. To confirm that we can achieve general traits by using template parameters.
5. To confirm the flattening algorithm works for methods.
6. To confirm that if methods are defined in traits they have better reuse opportunity than if they are defined in classes.
7. To confirm that if we develop a system at the modeling level with traits, we can generate a functioning system for it in the Java programming language.

5.1.2 Implementation

In this section, we describe how we used Umple traits to implement the geometric system. It is important to note that we believe traits should be used when they bring benefits regarding flexibility, reusability, and avoiding multiple inheritance. It is possible to make more extensive use of traits, to the point of using them to introduce every single method. We have not chosen that style, instead using traits when they are most useful, and using classic object-oriented design when it already works well.

In Figure 52, a part of this system's hierarchy is depicted. We have simplified the case study, hiding classes, leaves, attributes, and methods that are not necessary for the discussion about traits we provide. Full code and diagrams of the complete case study can be viewed in UmpleOnline [116]. In the Load menu, select the Geometric System. By default a

class diagram is shown, without methods and/or traits. To show methods and traits, go to the Options menu and click on the appropriate checkboxes.

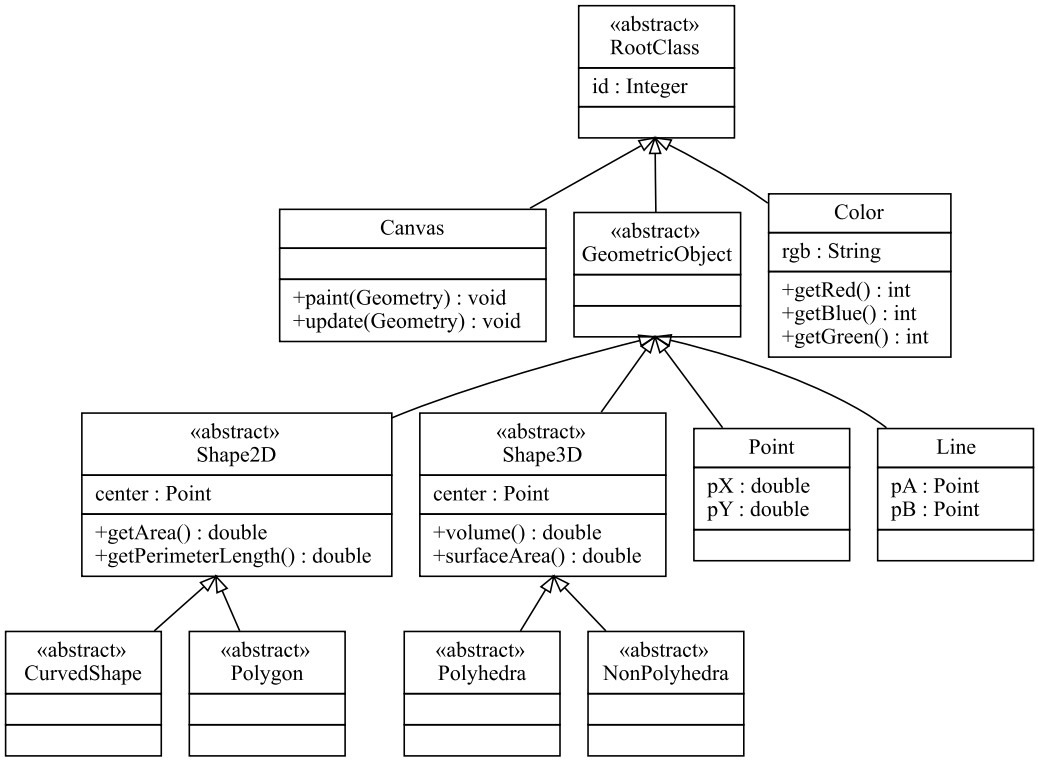


Figure 52. The hierarchy of the graphical system

As shown in Figure 52, there is a superclass named *RootClass* with three subclasses specifically *Canvas*, *GeometricObject*, and *Color*. The class *Canvas* is responsible for drawing geometric objects and the class *Color* keeps the necessary features related to color. The class *GeometricObject* is an abstract class for all geometric objects and has four subclasses including *Shape2D*, *Shape3D*, *Point*, and *Line*. The class *Shape2D* is a superclass for two classes, *CurvedShape* and *Polygon*. The class *Shape3D* is also a superclass for two classes, *Polyhedra* and *NonPolyhedra*. These abstract classes have their own subclasses (leaves). For example, the classes *Circle*, *Rectangle*, *Sphere*, and *Cube* are subclasses of the classes *CurvedShape*, *Polygon*, NonPolyhedra, and *Polyhedral* respectively. The class *Line* is straight, has no thickness, and extends in both directions without end.

One of the features that the system must have is to allow comparing two objects (e.g., shapes, color, etc.) regarding whether they are equal or not. We call this the equality feature.

For example, the system needs to compute whether or not two points are equal. Furthermore, the system must also allow comparing objects regarding being bigger or smaller. We call this the comparability feature. For instance, we need to compute whether or not a cube is smaller than another cube. There are cases in which we cannot have the comparability feature for classes (because there is no consistent way to compute it) and they must have just the equality feature. However, all classes that need the comparability feature must also have the equality feature. For example, points and lines can only have the equality feature while a circle must have both features. In this system, there are also some other classes that do not need these features (e.g., *Canvas*).

Listing 72. Te traits related to the equality and comparability features

```
1  trait TEquality<TP1>{
2    Boolean isEqual(TP1 object);
3    Boolean isNotEqual(TP1 object){
4      return isEqual(object) ? true : false;
5    }
6  }
7  trait TComparable<TP2>{
8    isA TEquality<TP1=TP2>;
9    Boolean isLessThan(TP2 object);
10   Boolean isLessAndEqual(TP2 object) {
11     return (isLessThan(object) && isEqual(object)) ? true : false;
12   }
13   Boolean isBiggerThan (TP2 object){
14     return isLessAndEqual(object) ? false : true;
15   }
16   Boolean isBiggerAndEqual(TP2 object){
17     return (!isLessThan(object) && isEqual(object)) ? true : false;
18   }
19 }
```

In order to implement these features, we have designed two traits named *TEquality* and *TComparable*. Their Umple code is depicted in Listing 72. The trait *TEquality* provides a provided method named *isNotEqual()* and requires a required method named *isEqual()*. It also has a template parameter named *TP1* used in the arguments of the required and provided methods. This feature allows us to have the same or suitable type for the arguments of methods which are going to be used in clients. This removes the need for casting of types in the body of methods and allows static type-checking during compilation.

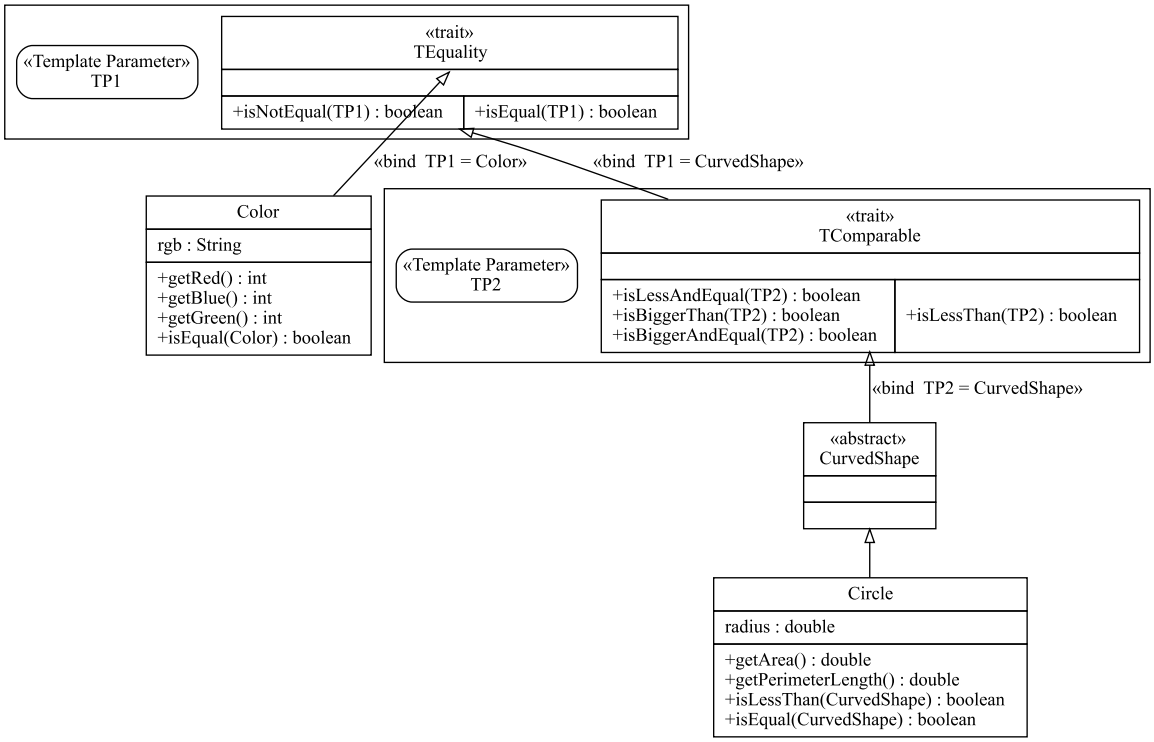


Figure 53. Use of the trait TEquality and TComparable

The trait *TComparable* uses trait *TEquality* to obtain functionality needed for its provided methods in addition to the required method named *isLessThan()*. It provides three provided methods named *isLessAndEqual()*, *isBiggerThan()*, and *isBiggerAndEqual()*. It also has a template parameter named *TP2* passed into the parameter of *TEquality*. In other words, this trait has one required method in the body and obtains another one through the trait *TEquality*. It also has three provided methods in the body and obtains one through the trait *TEquality*. To give the exact or proper type to the template parameter of these traits, they should be applied to the exact class or the first common superclass. Therefore, the trait *TEquality* is applied to the class *Color*, *Point*, and *Line* while the trait *TComparable* is applied to *Polygon*, *CurvedShape*, *Polyhedra* and *NonPolyhedra*.

Figure 53 shows part of the diagram in which the class *Color* uses the trait *TEquality*. It implements the required method of the trait, which is *isEqual()*. The class *CurvedShape* is abstract and uses the trait *TComparable*. It does not have enough information to implement the required methods of the trait so it keeps them as abstract methods and forces leaves to implement them (*Circle* in this case). It should be noted that there also are other ways to de-

sign and apply these traits. For example, the trait *TEquality* and *TComparable* could be completely distinct and we would then apply *TEquality* to the *GeometryObject* and the trait *TComaprable* to other mentioned classes. The key point here is that traits can give developers a variety of options while designing systems.

Another feature that we want to have is color and its related functionality for geometric objects. We also want to have an additional color for the edges of shapes. However, there are some shapes that mathematically do not have edges. We have designed two traits named *TDrawable* and *TDrawableWithEdge* for these purposes. The related Umple code is depicted inListing 73.

Listing 73. Traits related to the equality and comparability features

```
1  trait TDrawable {
2     0..1 -> * Color color;
3     Integer getRed(){
4       return getColor(0).getRed();
5     }
6     Integer getBlue(){/*implementation */}
7     Integer getGreen(){/*implementation */}
8     void applyTransparency(Integer p){/*implementation */}
9     void applyPattern(Integer type){ /*implementation */}
10    void applyColorFilter(Integer f){ /*implementation */}
11 }
12 trait TDrawableWithEdge{
13    isA TDrawable;
14    Integer getERed(){
15      return getColor(1).getRed();
16    }
17    Integer getEBlue(){/*implementation */}
18    Integer getEGreen(){/*implementation */}
19    void applyETransparency(Integer p){ /*implementation */}
20    void applyEPattern(Integer type){ /*implementation */}
21    void applyEColorFilter(Integer f){ /*implementation */}
22 }
```

The trait *TDrawable* gives the general meaning of color to all geometric objects while *TDrawableWithEdge* provides edge color functionality to appropriate shapes. The trait *TDrawable* has an association with the class *Color* and does not have any required method. It provides several provided methods related to color in which three of them are wrappers for methods in the class *Color*. All the functionality of the trait *TDrawable* depends on the first object color of the association (e.g., line 4).

The traits *TDrawableWithEdge* uses the trait *TDrawable* and adds provided methods related to the color of edges. All the functionality of trait *TDrawableWithEdge* depends on the second object color of the association defined in trait *TDrawable* (e.g., line 15). These two traits do not have any required methods, which shows that it is possible to consider traits as a mechanism to implement libraries. To have these feature in the system, the trait *TDrawable* is applied to the class *GeometricObject* because all shapes must have a color. The trait *TDrawableWithEdge* is applied to the class *Shape2D* and *Polyhedra* because instances of the class *NonPolyhedra* do not have edges to be colored.

Since the class *Canvas* should draw geometric objects along with their color, there should be a mechanism to let this class know about the changes in the properties so that it can update the canvas. This feature can be achieved through the *Observable* pattern. In order to implement this pattern, we reuse the trait *Subject* introduced in Listing 68. The class *GeometricObject* uses this trait and assigns the class *Canvas* as a binding type to the parameter *Observer*.

The indicated features so far are general features that other projects or classes might also want to use. Having them in terms of traits allows developers to reuse them without worries about the complexity of the class hierarchy. We can change or remove the names of provided methods either because of specifics of the domain or in case of conflicts.

5.1.3 Results

We modeled the system in Umple and generated Java code for this system. We compiled the system and the Java compiler passed the compilation without any error. We also manually checked to make sure each Java class has the functionality that it has obtained through traits at the modeling level and their type and signature are correct.

The following shows the satisfaction of goals defined in Section 5.1.1:

1. We showed that we could define traits and their required and provided methods. We could also validate that if a client does not implement required methods of traits, they cannot reuse it.
2. We showed that traits can be used by classes that already have a superclass.

3. We showed that traits can use other traits to achieve some of their functionality.
4. We showed that traits can be defined in a general manner with template parameters. These traits can be used by several classes.
5. We confirmed that our flattening algorithm works for methods by being able to see methods of traits are accessible in Java classes in the generated code.
6. We showed that if methods are in traits, they can be reused in cases that are not possible with inheritance. This shows that we have improved reuse by adopting the approach.
7. We showed that a system modeled by traits can be implemented in the Java programming language. This is demonstrated by generating and compiling the system in Java.

5.2. Re-engineering the Umple Compiler and JHotDraw

In this section, we describe re-engineering of two systems, the Umple compiler and JHotDraw [106], such that they use traits. The goals, design, and results of these case studies are in the following subsections.

5.2.1 Goals

The main goals of these two case studies are summarized as follows:

1. To validate that our approach and implementation is scalable and practical.
2. To confirm that model-based traits can be used in the implementation of software systems with well-defined requirements.
3. To confirm that a final system generated from a traits-based model has the same behavior if the system is developed without traits.
4. To explore further what benefits we can achieved if traits are applied to already developed systems.
5. To reconfirm that if we develop a system at the modeling level with traits, we can implement it in the Java programming language.

5.2.2 Implementation

In this section, we describe how we re-engineered two systems, the Umple compiler and JHotDraw [106] to use traits and automatically regenerated them in the Java language.

It is important to point out that the advantages of traits associated with better composition and reuse have been recognized by languages such as Scala [104], Squeak [54], Perl [105], Fortress [8], and PHP [108]. Identification of traits is still challenging in this area and there are manual and semi-automatic approaches and tools for this purpose [20,64]. However, there is not yet a comprehensive approach or tool that can identify traits based on all clues like method cancellation [9] or duplication. They were also developed for specific languages, which makes it difficult to adopt a systematic approach.

Detecting *all possible* traits, in order to maximize the effect on reusability and LoC (Line of Code), is not necessary to achieve our goals. Therefore, we just focus on detecting traits based on the exact duplicate methods. We also describe traits' advantages as learned in the case studies, as well as some static metrics, although the static metrics are not the key contribution of the approach nor the main factors of the evaluation.

The first system we re-engineering is the Umple compiler, which is written in Umple. We simply added traits directly to it. The second case study, JHotDraw was written in Java, so we had to first transform it to Umple. We did this through a tool called the Umplificator which allows us to transform software systems from Java to Umple [44]. The transformed system retains the same properties as the original since the Umple compiler transforms it back to its original target language.

Detection and implementation of traits for our two case studies was a manual process. Duplicate code (clones and near clones) were detected using CodePro Analytix [112]. CodePro Analytix is a Java tool for Eclipse developers who are concerned about improving software quality and reducing development cost. It provides a feature to find clones, but was not initially designed to detect traits.

In the first round of the process, we detected methods sharing the same signature and body. Each method was assigned to a trait and then its required methods were discovered. We followed this approach because we wanted to have fine-grained traits and then compose them into composite traits. When the owner of a method had implemented special interfaces that were crucial for the methods, we considered them as required interfaces.

Table 3. Static metrics of Umple and JHotDraw

	LOC (.java)	LOC (.ump)	Types	Percentage by Kinds	Methods
Umple	100613	39782	1133	7.2% interface, 92.7% class	2018
JHotDraw	80535	77647	1068	5.7% interface, 94.2% class	6893

Next we looked for methods which had a) the same number and order of parameters but different types (to see if we can use template parameters), and b) the same body. We again assigned each method to a trait and discovered the required methods and required interfaces. Template parameters were added to traits according to the number of differences in types. Then names of different types were replaced with template parameters. When special restrictions were detected that are needed for binding the types of template parameters, they were applied to parameters.

Finally, we integrated traits with other modeling elements of the systems and generated the final systems automatically in the Java language. We ran the pre-existing test cases of those systems on the newly regenerated systems.

5.2.3 Results

All of the test cases passed in the new systems. This confirms that it is possible to use traits at the model level and transform them into target languages while keeping the system's behavior exactly the same as the system without modeling.

Static metrics of these two systems before and after applying traits are depicted in Table 3 and Table 4 respectively. As can be seen in Table 4, traits were detected for Umple that resulted in 0.84% deduction in LOC. In other words, 335 line of code were saved because of traits. All traits have one required method on average (minimum zero and maximum nine) and have been applied at least to two clients on average (minimum two and maximum four).

There are 68 detected traits for JHotDraw, which resulted in 1.36% (1062 LOC) reduction in code volume. Each trait has at least two required methods on average (minimum zero and maximum three) and have been applied at least to two clients on average (minimum

two and maximum three). The improvement in JHotDraw is more than the improvement we had in the Umple compiler because there was more interface implementation in JHotDraw as compared to the Umple compiler. This is summarized in the fifth column of Table 3. There are 7.2% interfaces and 92.7% classes in Umple while 5.7% interfaces and 94.2% classes in JHotDraw.

Table 4. Traits specification for Umple and Jhotdraw

	Traits	**Required Methods (avg)**	**Clients (avg)**	**Saved LOC**	**Saved LOC**
Umple	17	1.0	2.3	335	0.84%
JHotDraw	68	2.5	2.1	1062	1.36%

While we were detecting traits in these systems we made some interesting observations. Some of these were confirmations of already-made points in other scientific papers, while the remainder are related to our improvements to traits. Firstly, it was confirmed that code generation is an essential part of model driven development. The benefits of traits would not be present if we were not generating code, and were just using them for documentation, as is often the case for models in industry [78].

Secondly, having big methods clearly decreases reusability. We detected several clones internally within many methods using CodePro Analytix [112], but they were mixed up with other implementation code, preventing us from being able to straightforwardly make those methods into provided methods of traits. If the methods had been more fine-grained, we would have likely been able to create more reusable traits and further reduce the number of lines of code. This issue was particularly noticeable in cases where an interface was implemented by several classes.

Thirdly, further refactoring of big methods is likely to lead to additional traits, hence further code reduction. Fourthly, as indicated before, we used duplicate methods as a clue for traits, if we used other factors (such as considering different types of clone [88]), we would be able to detect more traits.

Having relatively small 1% reduction in code volume through the use of traits in the two re-engineered systems described above may seem to suggest the use of traits might not be worthwhile. Having code savings is nice, but not necessary for there to be a valid contri-

bution. The benefit of having the traits goes beyond mere code volume reduction; with the introduction of traits, we have reduced the possibility of errors due to duplicate code, and have improved the understandability of the system. For example, in our evaluation systems, we found in one duplication case that there was a comprehensive comment regarding the functionality for just one of them and nothing for others. Therefore, developers, who will read the clone instance that does not have the comment, would have to employ lots of cognitive effort to understand it.

In these case studies, we have not detected state machines through the reverse engineering process because Umplificator does not yet support detecting state machines from Java code and representing them in Umple. Therefore, we might be able to achieve even better percentage of code reduction by representing a significant amount of code through state machines.

The following shows the satisfaction of goals defined in Section 5.2.1:

1. We have used traits to model two systems which have in average 90000 LoC. This showed that our approach can be used to develop reasonably large systems.
2. We modeled two systems which are actively under development and have well-defined requirements. This shows our approach is capable of handling such systems.
3. We modeled two systems with traits and then generated those automatically. These systems passed the test cases that were designed for the same systems without traits. This confirms that a system modeled with traits can have the same behavior as the same system without traits. This guarantees completeness with respect to the test sets.
4. Due to the reduction in code volume, we have some evidence that traits could reduce cognitive effort when a system needs to be studied by developers. However, we think more experimental studies need to be conducted in this direction.
5. We were able to generate two open-source systems from our models and run them. This again shows that we can use traits at the modeling level and automatically implement them in the Java programming language.

5.3. CORDET Framework

In this section, we implement an industrial framework called the CORDET Framework [77] to explore how having state machines in traits is practical and how they can be used in real scenarios. Although the framework is industrial and large, it is not possible to give all scenarios in which we can show the application of our defined operators. In fact, different combinations of operators can happen in different domains and scenarios.

5.3.1 Goals

The main goals of this case study are summarized as follows:

1. To confirm that it is possible to build systems using state machines in traits and also that it is practical.
2. To confirm that state machines in traits give opportunities for reuse.
3. To confirm that our flattening algorithm (including composition) for state machines is working correctly.
4. To confirm that state machines defined in traits and used to build a system, can be transformed to a target programming language.
5. To reconfirm that if we develop a system at the modeling level with traits, we can generate and execute Java for it.

5.3.2 Implementation

The CORDET Framework [77] is a specification of a generic architecture for distributed service-oriented embedded systems. The framework specifies components that implement the generic service concept. The service concept of CORDET is based on the service concept of the “Packet Utilization Standard” or PUS [107]. The behavior of components is defined based on state machines. These state machines are defined using the FW profile [75]. The modeling concepts of the FW profile have been implemented in the C language [76]. The linear behavior in state machines is modeled by *procedures,* which are a restricted version of

activity diagrams in UML. The notion of procedures has been defined in the FW profile as well.

Generally, frameworks should provide mechanisms whereby target applications can modify some of their behavior so as to meet their own specific needs. This mechanism is achieved in the CORDET framework through *adaptation points*. An adaptation point is specified by the stereotype <<*AP*>>. Adaptation points can be defined in actions or guards of state machines and in procedures.

The focus of the framework is for reuse at the framework level and not at the component level, but reuse and adaptability at the component level inside the framework may be achieved by single inheritance. Having the framework based on our approach breaks this reuse limitation and makes the components much more reusable.

We have chosen this framework for our evaluation for three reasons. The first is that it exploits state machines and single inheritance to describe and reuse behavior of components. Since our approach proposes to deal with limitations of reusing state machines, the framework is a great candidate to express how our work can improve reusability and flexibility. The second reason is that we have wanted to apply our approach to an industrial-strength system: CORDET has been applied to several real systems and is being maintained. Additionally, we are familiar with one of the base projects related to the framework, called OBS Framework Design Patterns [120], and have applied it to a real system. This could be a big advantage when understanding this industrial framework.

In the following, we explain the behavior of the main components in the framework and describe how our approach is applied to them so as to achieve the same functionality but with better reuse. We have used in our implementation the same names for elements defined in the framework in order to help to track outputs and differences. Other components of the framework are modeled based on the same approach used for the main components.

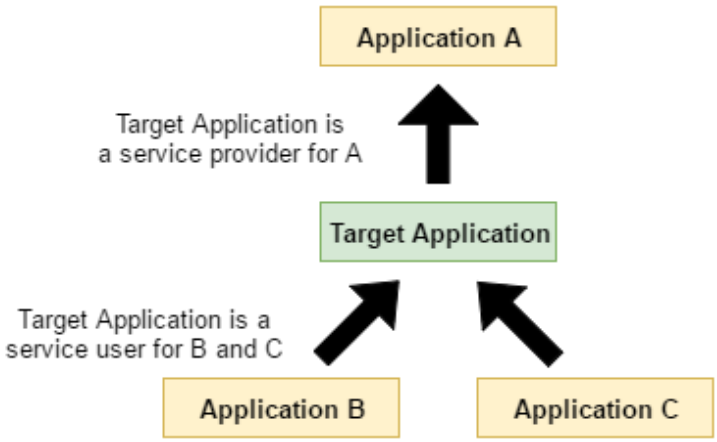


Figure 54. Applications as Providers and Users of Services, Alessandro et al. [77]

The service concept in the framework is defined as a set of capabilities that an application offers to other applications. An application is considered as a provider of service to other applications or a user of service from others. Figure 54 depicts the concept in which the green target application plays both roles while application A plays the role of a user and application B and C play the role of providers. The capabilities are managed through *commands* and *reports*. A command is a data exchange between a service user and a service provider to perform a particular activity within the service provider. A report is also a data exchange between a service provider and a service user to provide information related to the execution of a service activity.

Components in the framework need to have specific behavior, depicted in Figure 55 as a state machine called Base State Machine. Once a component is instantiated, it will be in the state CREATED and will need proper initialization based on the specification of the target application. The process of initialization called CIP (Component Initialization Procedure) is depicted on the left side of Figure 56. If this procedure finishes successfully, the component will be in the state INITIALIZED. In this state, the reset procedure called CRP (Component Reset Procedure) is executed and its successful execution puts the component in the state CONFIGURED. The CRP makes a component get its default values, and its process is depicted on the right side of Figure 56.

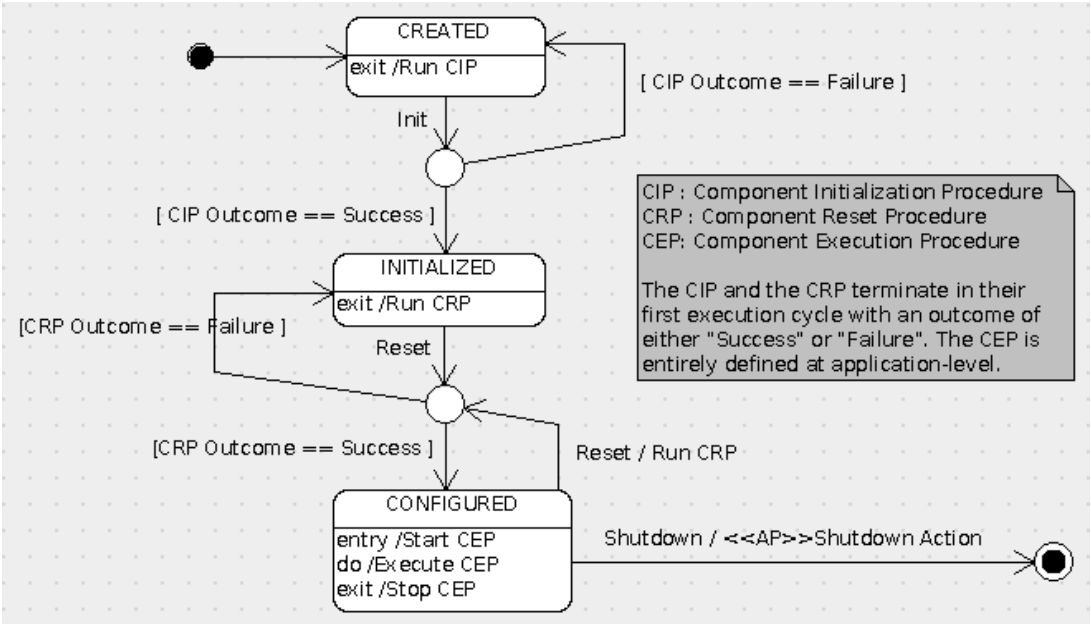


Figure 55. A Base State Machine, Alessandro et al. [77]

Once the component is in state CONFIGURED, it should execute the procedure assigned to it, which completely depends on the target application. In the framework, it is called CEP (Component Execution Procedure) and there is no defined procedure for it. The logic of how the procedure should be dealt with is described through the do activity and entry and exit actions.

However, the framework specifies that this can also be done with an embedded state machine. In that case, *Start*, *Execute*, and *Stop* actions, which are part of the FW profile, are not needed anymore. This method is the one adopted in this thesis because we wanted to model the system with state machines as much as possible. Finally, the component ends its execution by receiving the event *shutdown*. It is valuable to indicate that the state CONFIGURED is the state in which the meaningful behavior of components such as commands and reports is defined. We will get to this point as we progress.

Regarding the adaptation points specified by stereotypes <<AP>>, there is one point in Figure 55 called *Shutdown Action* and there are four adaptation points in Figure 56 called *Do Initialization check*, *Do Initialization Action*, *Do Configuration check*, and *Do Configuration Action*. Their names are self-explanatory.

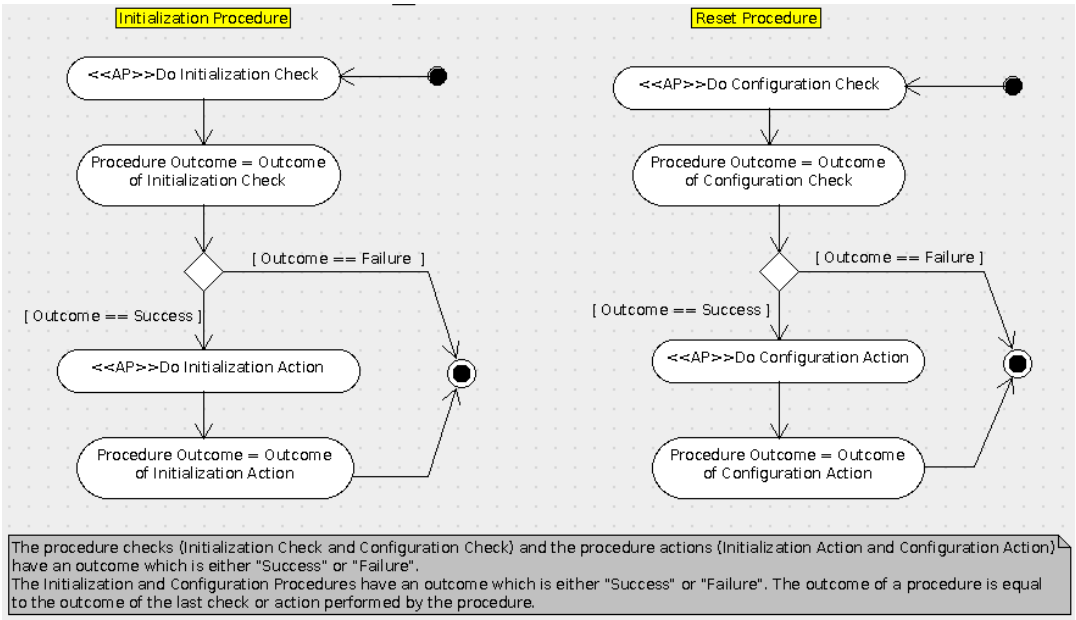


Figure 56. Initialization and Reset Procedures, Alessandro et al. [77]

Listing 74 shows the Umple model related to the specification depicted in Figure 55 and Figure 56. In order to model the specification based on our approach, we first need to define a trait, called *BaseStateMachine* (line 1) and then define the Base State Machine inside the trait, called *baseStateMachine* (line 9-31). The definitions of states and transitions follow the specification. Two states *transientCCIP* (line 14) and *transientCCRP* (line 22) are defined in order to express the two decision points in Figure 55.

Two procedures (*CIP* and *CRP*), defined as parts of the Base State Machine depicted in Figure 55, are modeled through two methods *runCIP()* (line 30-35) and *runCRP() (line 36-39).* Procedures are modeled through methods in Umple. Umple has a feature that allows using programming language syntax to precisely specify sequential procedures. In this case, we implement the *CIP* and *CRP* procedures using Java syntax. Having procedures as methods allows seamlessly using them inside the state machine for purposes they have been defined (lines 11 and 19). These two methods were defined as private methods so they will not be accessible through public interfaces of clients, but clients will be able to use them through the public interfaces of *baseStateMachine*. Furthermore, these two methods will be considered as provided methods of the trait *BaseStateMachine.*

Listing 74. The base state machine based on traits

```
1  trait BaseStateMachine{
2    internal Boolean cipOutcome = false;
3    internal Boolean crpOutcome = false;
4    Boolean initializationCheck();
5    Boolean initializationAction();
6    Boolean configurationCheck();
7    Boolean configurationAction();
8    void runShutdown();
9    baseStateMachine{
10     CREATED{
11       exit /{cipOutcome = runCIP();}
12       init -> transientCCIP;
13     }
14     transientCCIP{
15       [cipOutcome] -> INITIALIZED;
16       [!cipOutcome] -> CREATED;
17     }
18     INITIALIZED{
19       exit /{crpOutcome = runCRP();}
20       reset -> transientCCRP;
21     }
22     transientCCRP{
23       [crpOutcome] -> CONFIGURED;
24       [!crpOutcome] -> INITIALIZED;
25     }
26     CONFIGURED{
27       reset -> transientCCRP;
28       shutdown -> /{runShutdown();} END;
29     }
30     final END{}
31   }
30   private Boolean runCIP(){
33     if (initializationCheck()) return initializationAction();
34     return false;
35   }
36   private Boolean runCRP(){
37     if (configurationCheck()) return configurationAction();
38     return false;
39   }
40 }
```

Adaptation points are elegantly modeled by required methods of traits. This forces clients to provide their own implementation according to the needs of the target application. The four adaptation points in Figure 56 are modeled by the required methods *initializationCheck()*, *initializationAction()*, *configurationCheck()*, and *configurationAction()* (lines 4-7). The adaptation point in Figure 55 is modeled through the required method *runShutdown*() (line 8). Since adaptation points are modeled as required methods they can be seamlessly

used inside other methods and also state machines' actions and guards. Two attributes *cipOutcome* and *crpOutcome* (lines 2-3 are used to keep the results of two procedures *runCIP*() and *runCRP()* (line 11 and 19) respectively in order to make proper decisions inside the state machine.

One of the features that frameworks generally provide to the target application is a default implementation for adaptation points. Default implementations are mostly used for testing or rapid prototyping. Such a perspective has also been defined in the CORDET framework and it is implemented by having subclasses that provide default implementations. In our approach, it can be achieved by a class using a trait (*BaseStateMachine*) and then providing default implementations for adaptation points or using the trait by another trait and then providing default implementations for adaptation points. The first approach reduces the reusability because the default implementation needs to follow the limitation of inheritance, while the second approach offers the same flexibility as the trait *BaseStateMachine* provides.

Listing 75. The default implementation for adaptation points

```
1  trait BaseStateMachineDefault {
2    isA BaseStateMachine;
3    Boolean initializationCheck() {
4      return true;
5    }
6    Boolean initializationAction(){
7      return true;
8    }
9    Boolean configurationCheck(){
10     return true;
11   }
12   Boolean configurationAction() {
13     return true;
14   }
15   void runShutdown(){/*do necessary tasks*/}
16 }
```

The second perspective is depicted in Listing 75. As seen, the trait *BaseStateMachineDefault* just uses the trait *BaseStateMachine* and provides default implementations of adaptation points. Now, we have two reusable assets that can be reused by other clients in ways than would be possible just using inheritance (The limitations of inheritance were explored in Section 1.1.3).

Next, we concentrate on the behavior of the *command* component. A command encapsulates both the actions that need to be executed and the conditional checks that determine whether or not the command either is sent or executed. A command's life cycle is divided into two parts named the user and provider sides. We concentrate on the provider side here. The conditional checks are triggers for switching between different states and they need to have zero logical execution time (i.e. they must consist of a sequence of steps that are executed in one single execution of the procedure [75]). This makes it possible to have those checks as parts of guards in state machines.

There are two checks called *acceptance* and *ready* checks. The acceptance check returns true if the command has been received and can be accepted. The ready check returns true when the execution of the command can start. These checks are application specific and are considered as adaptation points. There are also four actions as follows:

- **startAction**: should be executed at the beginning of a command execution. The output specifies whether the command should be processed or aborted.
- **progressAction**: encapsulates the actions performed on one execution step. The output of this action specifies whether the command needs to be continued, failed, or terminated.
- **terminationAction**: encapsulates final actions that need to be executed before the command is terminated. The output of this action specifies whether the command will be terminated or aborted.
- **abortAction**: encapsulates the finalization actions that need to be executed in case of a command failure.

The above actions are also application specific and so are considered as adaptation points. The command switches among its states based on defined checks and actions. The command needs to have the behavior of the Base State Machine defined in Figure 55 and its own specific behavior. In fact, the behavior of the command should be defined inside the state *CONFIGURED*.

In order to achieve this, the framework proposes single inheritance. This means that the behavior of the command attaches to its superclass's behavior and there is no straightforward way to reuse it independently (as we discussed in Section 1.1.1). In our approach,

this can be accomplished in two ways. The first way, depicted in Listing 76, is to use the trait *BaseStateMachine* inside the trait *CrFwInCmd* (line 2) and then extend the state *CONFIGURED*. The new trait needs to define a state machine with the name *baseStateMachine* (line 3) and then specify the dummy initial state (line 4), which is *CREATED*. The trait then defines a state called *CONFIGURED* and puts the behavior of command in it. The behavior of command has not been defined in Listing 76 because it will be defined in our second way.

Listing 76. Building command's behavior without assignment

```
1  trait CrFwInCmd{
2    isA BaseStateMachine;
3    baseStateMachine{
4      CREATED {}
5      CONFIGURED {/*behavior of the command.*/}
6    }
7  }
```

The second way, which is our recommended one, is to design the state machine related to the command completely independently. Then, compose the final state machine through assigning this state machine to the state *CONFIGURED*. Listing 77 depicts an independent state machine for the command inside the trait *CrFWInSM*. The state machine *status* (line 11) has exactly the same behavior defined inside the composite state *CONFIGURED* in the previous way.

As noted, there is no need to give the same name, *baseStateMachine*, to this new state machine and also there is no need to know about the initial state and the hierarchy level at this level of definition. The trait is completely self-defined with its own requirements and can be reused in different components as needed. Therefore, the main difference between the first and second ways is that in the first one, it would not be possible to reuse the behavior independently. We also need to follow the same names for the initial state, regions, and composite states, otherwise, we cannot extend the proper state.

.

Listing 77. Building commands' behaviour with assignment

```
trait CrFWInSM{
  internal Boolean startActionResult = false;
  internal Boolean progressActionResult = false;
  internal Boolean terminationActionResult = false;
  Boolean acceptanceCheck();
  Boolean readyCheck();
  Boolean startAction();
  String progressAction();
  Boolean terminationAction();
  Boolean abortAction();
  status {
    RECEIVED{
      check[acceptanceCheck() ] -> ACCEPTED;
      check[!acceptanceCheck()] -> ABORTED;
    }
    ACCEPTED{
      execute[readyCheck()] -> STARTED;
      execute[!readyCheck()] -> PENDING;
    }
    PENDING{
      [readyCheck()] -> STARTED;
      [!readyCheck()] -> PENDING;
    }
    STARTED{
      entry/{startActionResult = startAction();}
      [!startActionResult] -> ABORTED;
      [startActionResult] -> INPROGRESS;
    }
    INPROGRESS{
      entry/{ progressActionResult = progressAction(); }
      [progressActionResult == "continue"] -> INPROGRESS;
      [progressActionResult == "completed"]-> TERMINATING;
      [progressActionResult == "failed"] -> ABORTED;
    }
    ABORTED{
      entry /{abortAction();}
    }
    TERMINATING{
      entry/{terminationActionResult = terminationAction(); }
      [terminationActionResult] -> TERMINATED;
      [!terminationActionResult] -> ABORTED;
    }
    final TERMINATED{}
  }
}
```

Listing 78. Building final state machine of the command

```
1  trait CrFwInCmd{
2    isA BaseStateMachine;
3    isA CrFWInSM <status as baseStateMachine.CONFIGURED>;
4  }
```

Listing 78 shows how the composition is performed to obtain the final behavior for the command – exactly the same one depicted in Listing 76. The trait *CrFwInCmd* uses two traits *BaseStateMachine* and *CrFWInSM* (line 2-3). It is worth pointing out that we could design *CrFwInCmd* as a class and also use default implementations for those two state machines which are *BaseStateMachineDefault* and *CrFWInSMDefault*. Line 3 in Listing 78 shows that the independent state machine *status* is assigned to the state *CONFIGURED*. There is no need to specify from which trait a state machine and its state come because all state machines are considered like local state machines of the trait *CrFwInCmd*.

Reporting is another service in the framework; we focus on parts related to the command here. This enriches the behavior of the command by permitting the users of commands to be aware of the different statuses of a command. Therefore, appropriate actions (or reports) need to be defined for this purpose. There are four reports that need to be sent back to the user. They are application-specific and are considered as adaptation points in the framework. These are as follows:

- **acceptanceAckReport**: sends true if a command is accepted by the provider, otherwise, it sends false.
- **startAckReport**: sends true if a command starts its execution, otherwise, it sends false.
- **progressAckReport**: sends three values, *continue*, *completed*, and *failed* based on the execution status of a command.
- **terminationAckReport**: sends true if a command terminates as expected, otherwise, it sends false.

Listing 79 show the Umple model related to the reporting functionality. The trait *CrFWInSMWithReport* uses the trait *CrFWInSM* (line 2) to get the main functionality of the command. Then, it models the state machine *status* with new required states, guards, transi-

tions, and actions. For example, the action *acceptanceAckReport* was added to the transitions of the state *RECEIVED* (line 9 and 10). There is no need for new states, transitions, or guard in this case. Two state machines, coming from the client itself and the trait *CrFWInSM*, are merged automatically with each other so that the final state machine can be obtained from it.

Listing 79. The command state machine with reporting capability

```
1  trait CrFWInSMWithReport{
2    isA CrFWInSM;
3    void acceptanceAckReport(Boolean value);
4    void startAckReport(Boolean value);
5    void progressAckReport(String value);
6    void terminationAckReport(Boolean value);
7    status {
8      RECEIVED{
9        check[acceptanceCheck()]-> /{acceptanceAckReport(true);} ACCEPTED;
10       check[!acceptanceCheck()]-> /{acceptanceAckReport(false);} ABORTED;
11     }
12     STARTED{
13       [startActionResult]-> /{startAckReport(true);} INPROGRESS;
14       [!startActionResult]-> /{startAckReport(false);} ABORTED;
15     }
16     INPROGRESS{
17       [progressAction()=="continue"] ->
                            /{progressAckReport("continue");} INPROGRESS;
18       [progressAction()=="completed"] ->
                            /{progressAckReport("completed");} TERMINATING;
19       [progressAction()=="failed"] ->
                            /{progressAckReport("failed");}ABORTED;
20     }
21       TERMINATING{
22       [terminationActionResult]->
                            /{terminationAckReport(true);} TERMINATED;
23       [!terminationActionResult]->
                            /{terminationAckReport(false);} ABORTED;
24     }
25   }
26 }
```

Listing 80. Building final state machine of the command

```
1  class CrFWInWithReport{
2    isA CrFwInCmd;
3    isA CrFWInSMWithReport<status as baseStateMachine.CONFIGURED>;
4  }
```

As seen, there is no need to repeat the definitions of the states and transitions not affected by the reporting functionality. The trait *CrFWInSMWithReport* has the entire func-

tionality related to the command and reporting, but it does not have the behavior of the Base State Machine. To obtain this, the same model depicted in Listing 78 is used, but the trait *CrFWInSM* is replaced with *CrFWInSMWithReport*. It also is possible to use *CrFwInCmd* if it is preferred to encapsulate the trait BaseStateMachine. Listing 80 depicts the Umple model related to this definition. The composition algorithm takes care of merging necessary states and transitions.

5.3.3 Results

In this case study, we were able to model state machines of the CORDET framework in traits and use traits to extend state machines based on the specification of the framework. We manually evaluated the components of the framework which were generated in Java. The results showed the components have the functionality defined for them as specified in the framework.

The following shows the satisfaction of goals defined in Section 5.3.1:

1. We confirmed that it is possible and practical to build a system based on state machines in traits. This comes from the fact that we could model specifications of the framework based on state machines and generated code for the CORDET framework.
2. We showed that state machines in traits can be reused more freely in different scenarios than when they are modeled inside classes and reused based on the concept of inheritance. Being able to model state machines in traits and reuse them to build a system is a clear sign of this.
3. We confirmed our flattening algorithm (including composition) for state machines is working as we expected. This is observed from the fact that we extended and composed state machines in traits to achieve the specification defined in the framework. The final system was what we expected, therefore, it is a fair validation for this goal. However, we believe our case study may not cover all cases needed to evaluate the composition part of the flattening algorithm. Therefore, more case studies still need to be developed as future work.

4. We confirmed that if a system is developed by state machines in traits, we can automatically transform it to a target language (in this case Java), that can be executed.
5. We were able to generate the framework components from our trait-based models and run them. This again showed that we can use traits at the modeling level and automatically implement them in the Java programming language.

5.4. Testing at the Modeling Level

In this section, we describe our test-driven development process for validating that requirements for model-based traits were implemented correctly.

5.4.1 Goals

The main goals of this case study are summarized as follows:

1. To confirm that the implementation correctly matches the specification
2. To make sure as Umple progresses, new extensions will not contradict traits syntax and semantics (i.e. to avoid regressions)

5.4.2 Implementation

Umple, from its inception, was developed using a comprehensive test-driven development approach. Separate layers of tests are applied at the syntactic level, semantic analysis (metamodel-population) level, code generator level, and by execution of resulting systems. The Umple compiler has over 6000 tests that are run every time anyone tries to rebuild the compiler. This ensures the Umple compiler remains valid as it is incrementally extended by new features.

We continued Umple's test-driven-development approach when adding traits. The following outlines how we have followed Umple's multi-layer testing philosophy. The first layer of tests concerns the syntax of all elements we have added to traits such as state machines, operators, and so on. These kinds of test cases make sure that all keywords and operators are parsed properly by the Umple compiler. They also guarantee that the parsing does not break when others make extensions in the Umple grammar. The second layer of tests deals with semantics. We break this into several categories:

The first category ensures that the core semantics of traits is correctly applied based on requirements we described in Section 3.1. For example, if a trait has a provided method, the provided method must be part of any client that uses the trait. Other tests in this category make sure that operators applied to a trait by a specific client will not affect other clients of the trait. For example, if a provided method is renamed by a client, the renaming must be performed for that client and other clients should obtain the provided method with its original name. In other words, these test cases check that model transformation that is performed at the Umple compiler level is a correct transformation.

The second category of semantic test cases is about detecting conflicts that can happen between traits and their clients. For example, a trait cannot use itself or a trait cannot make a bidirectional association with interfaces. These test cases also cover validations related to operators. For example, if an operator wants to change the name of a state machine, it must be available in the used trait.

Finally, we have dedicated a category of semantic test cases for the composition algorithm we have developed for state machines. This makes sure the semantics we defined for the composition of state machines is valid after model transformation. This category is a complicated one because test cases cover a different combination of using state machines together or in collaboration with other traits elements.

We have developed 218 test cases [121] overall (including 426 assertions) and they are executed every time the Umple compiler is built. All test cases are passed in the current implementation. Since most of elements defined in traits, such as state machines and associations, have been tested as part of the Umple compiler (see the previous section), there was no need to develop test cases for them. This has reduced tremendously the number of test cases required for developing traits in Umple. Furthermore, the Umple compiler serves as one of our integration tests because it includes traits we have developed for it. In fact, the Umple compiler uses its own traits to achieve its functionality.

We have not developed any test case for code generation related to different target languages because they have their own test cases. As long as we give a valid model to those code generators, the output of them will be valid as well. As pointed out before, using model transformation at the Umple compiler level was a strategic decision to have traits accessible in different programming languages supported by Umple. Therefore, there was no need for

any unique test cases regarding traits for code generators. Our developed systems as part of this validation process are good indicators that the Umple code generator works properly, otherwise, we would not be able to execute those system modeled by traits.

5.4.3 Results

We were able to develop test cases for requirements of model-based traits and integrate them to the testing process of Umple.

The following shows the satisfaction of goals defined in Section 5.4.1:

1. Our implementation passed all test cases extracted from requirements. This showed our implementation is in compliance with requirements.
2. We integrated our test cases into the testing process of Umple and therefore if changes happens in the syntax and semantics of traits, those test cases will inform developers.

5.5. Summary

In this chapter, we implemented several systems to validate our work. The first system validated that basic features of traits implemented in Umple are working correctly. The second and third systems confirmed that model-based traits can be used to build the same system that was built without traits. The forth system investigated whether state machines in traits can be utilized to model a system. In all these systems, we also validated that we can implement systems automatically in the Java programming language. In all these systems, reuse functionality or models are encapsulated in traits, increasing flexibility and reducing limitations in comparison to inheritance (the issue with inheritance was discussed in Section 1.1). Finally, we demonstrated that we have used test-driven development to make sure our implementation is complete with respect to test cases developed from requirements.

Although we evaluated several dimensions of our work, there are cases that we were not be able to evaluate and consider them as future work. For example, we did not evaluate the time that might be gained because of using traits and how easy traits are to be learnt and

manipulated by programmers. In order to answer these kinds of questions, we need to design an experimental study in which a group of people develop a system with and without traits.

The results of our evaluations in this chapter can be summarized as follow:

- We showed that we can define traits and their required and provided methods.
- We showed that traits can be used by classes that already have a superclass.
- We showed that traits can use other traits to achieve some of their functionality.
- We showed that general traits can be defined through template parameters.
- We confirmed that our flattening algorithm works for methods.
- We showed that if methods are in traits, they can be reused in cases that are not possible with inheritance.
- We showed that a system modeled by traits can be implemented in the Java programming language.
- We showed that our approach can be used to develop reasonably large systems.
- We showed that a system modeled with traits can have the same behavior of the same system without traits.
- We showed that it is possible and practical to build a system based on state machines in traits.
- We showed that state machines in traits can be reused more freely in different scenarios than when they are modeled inside classes and reused based on the concept of inheritance.
- We showed our flattening algorithm (including composition) for state machines is working as we expected.
- We showed that if a system is developed using state machines in traits, we can automatically transform it to a target implementation language.

In the next chapter, we discuss features that our approach offers regarding reusability. We also explain what challenges we faced while we were developing our approach and what challenges developers might face when they want to adopt our approach. This may help people who want to use ort approach or want to adopt traits into their own modeling language.

Chapter 6. Discussion and Challenges

In this chapter, we first explain how our extended features can provide better reusability and then express some challenges that we have faced so far during our research or that developers may face while using traits with the extended features.

6.1. Discussion Regarding Reusability

In this section, we discuss how our approach can provide better reusability. The goal is to show how each part of our solution along with other preliminary features of traits plays its role in software reuse. For this purpose, we will make connections between specific parts of our approach and the most common concepts in the majority of reuse techniques [58]: abstraction, selection, specialization, and integration as defined in Biggerstaff and Richter's framework [25].

6.1.1 Abstraction

Abstraction plays a central role in software reuse so if we would like to reuse software artifacts effectively, concise and expressive abstractions are essential [58]. Abstraction reflects the point that concealment of details and focusing on the most important factors facilitate reusability. Indeed, having a high level of abstraction helps us to have more reusable elements. In our approach, we concentrate on traits at the modeling level, which is a higher level than traditional programming. This helps us to put implementation concerns aside and focus more on reusing functionality. The implementation concerns will be resolved through automatic code generation.

We introduce modeling elements to traits that increase both abstraction of traits and reusability opportunity of those elements. Reuse of those elements is not limited anymore by the constraints of inheritance as discussed in Section 1.1.3. Furthermore, instead of obtaining the functionality of traits only through methods, such functionality can be expressed with more abstract elements such as state machines. This results in more abstract traits.

Required interfaces can be used in the design of other parts of systems to have or create proper clients. This helps developers to understand requirements of traits more easily or put restrictions on clients in a modular way. Associations are also more abstract than classic code in which developers have to define instance variables and implement the necessary APIs.

6.1.2 Selection

Selection deals with locating, understanding, comparing, and selecting reusable software artifacts. This is covered by our approach in two ways.

Firstly, developers can select proper traits (e.g., from a repository) according to their need and use them inside classes or other traits. They may gain understanding of the traits by looking at either their required methods and interfaces or the interfaces needed by template parameters. Secondly, developers are also allowed to select from the provided methods of each trait.

6.1.3 Specialization

Specialization focuses on generalized or generic artifacts and specializes them by inheritance, parameters, transformation, constraints, and some other forms of refinement.

In our approach, inheritance is available through the composition mechanism: Traits can extend the behavior of their super-traits (composing traits), and classes can extend their traits under the flattening mechanism. Particularly notable benefits of specialization can be observed when modeling elements in traits (e.g., state machines and associations) are specialized in sub-traits. For example, when there is an association in a composing trait and there is the same association in the composed trait, then multiplicities of the composed trait will affect the composing one.

Moreover, template parameters allow developers to define generic traits and specialize generic traits by binding specific types to parameters. They also let one put restrictions on types of parameters by defining required interfaces for parameters. Template parameters are used with associations to provide a better configuration mechanism to associate clients of traits with other classes.

Finally, Umple lets developers customize visibility and names. Renaming provided methods allows for the development of domain-specific vocabularies, which helps developers create more-adaptive systems. It is important to mention that customization is also used for conflict resolution.

6.1.4 Integration

Integration considers how reusable elements will be integrated into a software system effectively. In other words, how developers combine a collection of selected and specialized artifacts into a complete system. This is achieved by knowing more about artifacts' interfaces. In our approach, there are two complementary ways that developers can understand traits. The first is required methods and interfaces, which reveal how a trait can be used in what classes or traits. The second one is required interfaces for template parameters, which indicate allowable bindings for the parameters. These two features can also be considered a way to select traits (discussed in Section 6.1.2). Furthermore, as a common way of helping developers to know much about reuse artifacts, we can assign comments to either traits or each element of traits.

6.2. Challenges

There are three categories of challenges regarding our research: Challenges our work will help developers overcome, challenges we faced when developing traits, and challenges that might be faced by those trying to use our traits in models.

6.2.1 Developers' Challenges Our Work Should Help With

The first category of challenges is the technical challenges we are allowing software engineers to overcome if they use our approach. The following is a partial list:

Avoiding Multiple Inheritance: developers generally want to avoid multiple inheritance for a variety of reasons, yet at the same time reduce duplication in their model, but there is a lack of techniques to overcome this if the developer wants to work at the modeling level.

Moreover, developers who use traits may want to define semantics-level restrictions on traits and make them as a part of traits' usage. On the other hand, our new feature regarding having associations inside traits opens a new design technique which is portable and flexible. For example, we depicted in Listing 68 how the observable pattern can be redesigned.

Developers who use state machines to develop their systems in an incrementally way need to worry about how they compose those modelled state machines. They also need to make decision regarding in which classes those state machines must be defined so as to be able to reuse them better in other classes (if it is required). Such concerns were discussed in Section 1.1.1. Our approach tackles those concerns and allows developers to focus more on the way they design the logic of state machines. When it is required to compose state machines, it will be possible with traits. Developers can even adjust those reused state machines based on slightly different requirements of new systems (or classes).

Finally, being able to implement reusable modeling elements in different programming languages is challenging for developers. They need to make sure elements they reuse while modeling can be properly transformed to executable code. In our case, this is performed automatically and developers do not need to worry about it.

6.2.2 Challenges We Faced in This Work

The second category of challenges are the technical challenges we faced during our research including design and implementation phases.

Our first challenge was to develop a usable syntax for traits. We had to define the necessary keywords inside Umple and there were two options 1) using specific keywords for each concept 2) reusing other keywords available in Umple. Since, the main philosophy of Umple is simplicity, we defined a minimum of new keywords and reused already-existing keywords. For example, instead of having the keywords "required" and "provided" for required and provided methods, we consider abstract methods as required methods and normal methods as provided method. We also re-used the 'isA' keyword in several contexts, where doing so makes sense. The main reasons for this decision are as follows:

- Avoiding having lots of different keywords and forcing the modeler to remember which is for which.

- The isA rule is applied quite nicely if we name traits well. For example, "isA Comparable" (traits Comparable defined in Listing 72) clearly specifies the client wants the functionality of being comparable.
- It is often interchangeable whether the reused unit is class, interface, or trait; using the same keyword allows the client to be unchanged if, for example, we turn a superclass into a trait.
- In existing languages, there are many ways of specifying reuse of elements; we wanted something simple and distinct, and which also is abstract

However, some have doubted the wisdom of our choice of the 'isA' keyword. For example, readers of the code do not instantly know whether the reused element is a superclass, interface or trait. However, we think it should not be important for developers to know what kind of element is being reused. The most important thing is the functionality obtained.

The next challenge we faced was developing semantics of traits in the modeling context. The new semantics must be compatible with original Umple/UML semantics and be usable and meaningful at the modeling level. Traits semantics had to be extended to admit modeling elements as a part of traits definitions. Exploring the soundness of the approach in numerous scenarios was challenging. We had to explore the elements' positive and negative effects and then see whether or not each item we introduced is in compliance with the original definition of traits. Most incompatibilities came from the fact that finally those elements must be flattened to clients and there must be less conflict and more effectiveness for those elements.

Developing rules that show when traits are correct (and producing appropriate error messages in other cases) was also another challenge. Our comprehensive list of error and warning generated as part of the validation phase is depicted in Table 5 in Appendix I. We wanted to check the original rules of traits in addition to the ones we promoted for model-based traits in an automatic way. This caused us to check both flattened and normal models. However, it resulted in a good mechanism for our implementation in which we can show both flattened and original models of the final system as alternative views. Having these two views ought to have a positive effect on developers regarding their understanding of the final system [26].

Adding and composing state machines was also challenging. State machines in traits can be reused more than once by clients and so we needed to make sure that operators and the composition algorithm will not affect the base definition of traits. Therefore, it was required to clone traits (deep cloning) when they are reused by clients and then pass them to the composition algorithm to compose them with other state machines. We first tried to use the Java cloning mechanism to deal with it but soon noticed that state machines have associations with other Umple classes which are not used when state machines are used in traits. Therefore, we developed specific cloning methods for elements of state machines that we were interested in. This also helped improve memory use when it comes to the cloning part of state machines.

Implementing operators for state machines and other elements was challenging as well because modelers should not be able to apply conflicting operators to elements. Furthermore, there should not be an order in which operators could be applied. This required us to perform deep analysis every time a new operator was added to traits.

6.2.3 Challenges Expected to Be Faced by Adopters

The third category of challenges is those that might be encountered by adopters of our work. We believe that the challenges will be minimal. Traits provide a layer of functionality sitting on top of the existing modeling language and so our work allows rational copying of reusable elements (flattening) in a controlled way. Previous research has shown that flattening can be done with elements such as methods and attributes. We showed how flattening can be performed just as easily with associations and state machines.

Regarding the learnability of our concepts, developers define traits in a way very similar to how they define classes, albeit with a few unique details related to traits. Developers will need to learn the rules of applying traits in various class hierarchy contexts and how to resolve conflicts through rename/remove operations. These rules are simple, as indicated earlier in this thesis. Furthermore, if developers make mistakes, the compiler will raise relevant warnings and error messages – and as mentioned in the last section, the developer can see the flattened version of the system to verify their compositions are performed as expected.

Our syntax has been designed to be straightforward and conflicts are detected automatically by Umple's compiler. The compiler helps modelers regarding the conflict resolu-

tion through concrete examples. In our lab, developers showed that they were able to understand how to use traits in their design after a 30-minutes presentation. Of course, an empirical study ought to be conducted to assess usability of our technique; this is future work.

6.3. Summary

In this chapter, we made a connection between features of our work and principles of reuse present in the majority of reuse techniques. We argued that our approach brings more abstraction to traits, provides mechanisms for developers to select the reuse elements they need, allows developers to specialize reuse elements when they need more customization, and finally offers ways for developers to reason about how to integrate traits into a software system.

In the next chapter, we give an overview of our work, summarizing what we have achieved. Finally, we discuss some potential future work.

Chapter 7. Conclusion and Future Work

In this chapter, we summarize what we have achieved during this research and also discuss future work.

7.1. Summary and Conclusion

In this thesis, we implemented an enhanced mechanism for reuse based on traits. We extended traits to be abstract and coherent in order to provide better reusability and integration with model-driven software development. The contributions can be summarized as follows:

Enhancement of Umple to support traits: We have extended Umple with the fundamental structure for supporting traits. This extension includes defining traits, required methods, provided methods, attributes, and template parameters. In order to obtain these features, we extended the Umple grammar which allows the Umple parser to parse syntax related to traits. The Umple metamodel has also been extended to allow having traits as modeling elements. Then, we extended the Umple compiler (the analysis phase) to build a valid instance of a metamodel for any system developed by traits.

Required interfaces as a new abstraction in traits: We have extended the basic definition of traits to be able to define additional semantic requirements on their clients through having required interfaces. This prevents traits from being used in clients that just satisfy the signature of required methods. Traits can also use a hierarchy of interfaces for better management of their requirements.

Adding associations to traits: Associations, one of the key modeling elements in UML, were added to traits. Traits can have static and dynamic associations with other elements. Static associations are defined when traits make associations directly with a class or an interface. These associations finally will be between candidate clients and those classes and interfaces. Dynamic associations are defined when a trait makes an association with a template parameter. Therefore, clients can specify the other side of association when they use

traits. This kind of associations makes traits more reusable and flexible. Furthermore, traits are able to present some of their provided methods implicitly in terms of associations, which is more abstract than explicit provided methods. Taken together, all these items enhance how traits can be used in model-driven software development.

Adding state machines to traits: State machines, a second key UML modeling construct, were also added to traits. In order to maximize the benefits of state machines in traits, we developed an algorithm that composes state machines of traits when they are used by traits or classes. We also combined state machines with template parameters in order to increase reusability of traits. We introduced the concept of superCall for state machines actions and activities which allows much better flexibility when state machines are composed. Furthermore, we allow state machines to be used as a way to extend a simple state in order to produce composite states. The events of state machines in traits (or classes) serve as provided methods that can be used to satisfy the required methods of used traits. This shows how modeling and implementation synergistically work together.

Operators on trait inclusion: We defined and implemented required operators to resolve conflicts when they happen, and also allow controlling granularity and reusability of traits. These operators allow renaming and selective inclusion.

Traits in the Umple user-interface: In order to increase the way traits and their elements can be explored, we provided an automatic way to represent traits graphically and switch between trait-based and class (flattened) diagrams.

Programming-language independence of traits: Our model transformation at the Umple compiler level allows to have traits at the modeling level and generate implementations in major programming languages such as Java and C++. Although a key impetus for our work is to bring traits to the modeling level, the work we have performed can also allow our work to be applied to code that consists of only methods, which looks just like Java or C++, or both at the same time.

Case studies demonstrating the applicability of our work on traits: We evaluated Umple traits and our enhanced features through different cases studies. The first case study demonstrated that basic features of traits can be utilized in Umple in the same manner as they are used in programming languages to develop systems. Then, we extracted traits from two systems based on Umple. We re-implemented those systems with traits defined in Umple and

achieved the same functionality they had before using traits. This was confirmed by the fact that systems generated from model-based traits could pass all test cases that have already been developed for them (when they were developed without traits). This demonstrates that traits can be used in large and model-based systems. This, in turn, allows us to use our approach as a generic extension providing traits in languages such as Java. The last case study evaluated state machines in traits and our composition algorithm through implementation of a framework. We demonstrated how state machines in traits can be used to build systems and achieve more reusable state machines. These reusable state machines can be easily used in different projects or classes in a different level of hierarchy.

7.2. Future Work

There are many opportunities to extend this work or use traits for different applications. Some suggested future work is as follows:

Traits provide a significant flexibility regarding reuse, so they are good candidates to be utilized for model-based software product lines. Although other researchers are investigating various ways to model and generate product lines, our work has capabilities that might prove particularly beneficial. Umple traits offer modeling elements such as associations and state machines which are good matches for this purpose. We want to extend syntax and semantics along with a good methodology to enable Umple traits to be used for model-driven software product lines. The possible option are to achieve this is to extend Umple to support feature models that can be expressed based in part on state machines and traits.

It would be productive to explore the use of traits to the GoF design patterns [43] and other design patterns. A preliminary study has shown positive results regarding this.

Although we have fully implemented and extensively tested traits syntax and semantics in Umple, it would be beneficial from a formal software engineering perspective to have a formal definition for our trait extensions like the one defined for basic traits [35]. This also includes the formalization of the entire structure and composition algorithm (discussed in Section 3.3) in combination with other elements of Umple. This may facilitate the way Umple traits can be adopted by other programming or modeling languages.

During our case studies, we had to use third-party tools to find duplicated code (clones) in source code and then convert them to traits. It would be beneficial to have an

Umple extension to let developers automatically discover traits in their already-developed systems in Umple and then have them represented by traits automatically. The tool could even be extended to allow detecting traits in other languages such as C++.

Our composition algorithm currently is not capable of fully analysing Boolean expression to determine when two or more are equivalent, or one subsumes another. Hence there remain opportunities to better enable merging of transitions involving guards. We want to improve our algorithm to tackle this issue. Furthermore, two equal Boolean expressions defined in guards might statically be equal, but they might never be equal at runtime because of constraints on the values of involved variables. The formulae become more complicated when non-Boolean variables, Boolean variables, and function calls are mixed together. These cases are satisfaction issues and have to be dealt with by model checking techniques.

For these purposes, tools like Choco [113] can be adopted. Choco can be a great candidate because a) it is a Java library and so can be used directly during the composition process; and b) it just needs variables and satisfaction criteria and those can easily be obtained from the model of the compiled Umple file at runtime.

Another perspective on the above issue is to detect implications after composition through using a model checker. Then, we can iterate over the models and apply desired actions on them. For example, we can simplify the guards so they can be detected by our native comparison algorithm. This process can be performed automatically in Umple, which is part of on-going research by another Ph.D. students in our Lab. Indeed, Umple provides a transformation to the formal representation of classes and state machines in nuXmv [4] so models can be explored in that tool. The plan is to integrate the model-checking process with our approach.

We introduced traits into Umple (a text-oriented modeling language). We also adopted a simple graphical representation for traits (described in Section 2.2.10) based on the previous research conducted by Schärli et al. [93] and extended it with our notations. It would be interesting to study how graphical model languages can adopt our traits as part of their modeling elements.

During this research we focused on modeling languages (or frameworks) which are compatible with the concepts of object-orientation and UML. It would be interesting to study whether the philosophy of traits can be used in other modeling frameworks. We suggest

studying which elements of those modeling languages can be represented in traits, what kinds of conflicts might happen, and what operators are required.

Although we have applied our approach to a real world system, we have not yet fully explored the capabilities of all our operators in case studies. Applying the approach to more systems in which state machines have an important role can reveal better scenarios for reusing traits.

Finally, it would be important to conduct an experimental study to explore how usable is our approach when practitioners develop systems based on traits. The results of such a study may provide feedback on how a methodology should be designed for trait-based software systems or what kinds of new operators might need to be added to model-based traits.

References

[1] Abdelzad, V. Extended traits for Model-Driven software development. *CEUR Workshop Proceedings*, CEUR-WS (2015), 1–5.

[2] Abdelzad, V. and Lethbridge, T.C. Promoting traits into model-driven development. *Software and Systems Modeling*, (2015), 1–22.

[3] Abdelzad, V. and Shams Aliee, F. Aspect-Oriented Software Development versus Other Development Methods. *Journal of Theoretical and Applied Information Technology 31*, 2 (2011), 147–152.

[4] Adesina, O., Lethbridge, T., and Some, S. A fully automated approach to discovering non-determinism in state machine diagrams. *10th International Conference on the Quality of Information and Communications Technology (QUATIC'2016)*, (2016), 73–78.

[5] Adesina, O.O. Integrating formal methods with model-driven engineering. *Models 2015 Doctoral Symposium*, (2015), 86–92.

[6] Alam, O., Kienzle, J., and Mussbacher, G. Concern-Oriented Software Design. *16th International Conference, MODELS 2013*, (2013), 604–621.

[7] Aljamaan, H., Lethbridge, T.C., and Garzón, M.A. MOTL: a textual language for trace specification of state machines and associations. *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, 2015, 101–110.

[8] Allen, E., Chase, D., Hallett, J., et al. The Fortress language specification. *Sun Microsystems 139*, (2005), 140.

[9] Arévalo, G. Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis. *LMO '03*, (2003), 47–59.

[10] Atlee, J., Beidu, S., Fahrenberg, U., and Legay, A. Merging Features in Featured Transition Systems. *in Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation*, (2015), 38–43.

[11] Avedillo, M.J., Quintana, J.M., and Huertas, J.L. State merging and state splitting via state assignment: a new FSM synthesis algorithm. *IEE Proceedings - Computers and Digital Techniques 141*, 4 (1994), 229.

[12] Badreddin, O., Forward, A., and Lethbridge, T.C. Model oriented programming: an empirical study of comprehension. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, IBM Corp. (2012), 73–86.

[13] Badreddin, O., Forward, A., and Lethbridge, T.C. Exploring a Model-Oriented and Executable Syntax for UML Attributes. *Software Engineering Research, Management and Applications, Studies in Computational Intelligence 496*, (2013), 33–53.

[14] Badreddin, O., Forward, A., and Lethbridge, T.C. Improving Code Generation for

Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity. *Software Engineering Research, Management and Applications, Studies in Computational Intelligence 496*, (2013), 129–149.

[15] Badreddin, O., Lethbridge, T.C., Forward, A., Elasaar, M., and Aljamaan, H. Enhanced Code Generation from UML Composite State Machines. *Modelsward*, (2014), 1–11.

[16] Beidu, S., Atlee, J.M., and Shaker, P. Incremental and Commutative Composition of State-Machine Models of Features. *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, (2015), 13–18.

[17] Benedicenti, L., Succi, G., Valerio, A., and Vernazza, T. Monitoring the efficiency of a reuse program. *ACM SIGAPP Applied Computing Review 4*, 2 (1996), 8–14.

[18] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R. Stateful traits. *Advances in Smalltalk, Proceedings of 14th International Smalltalk Conference (ISC 2006), LNCS*, (2007), 66–90.

[19] Bergel, A., Ducasse, S., Nierstrasz, O., and Wuyts, R. Stateful traits and their formalization. *Computer Languages, Systems & Structures 34*, 2–3 (2008), 83–108.

[20] Bettini, L., Bono, V., and Naddeo, M. A trait based re-engineering technique for Java hierarchies. *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, (2008), 149–158.

[21] Bettini, L. and Damiani, F. Pure trait-based programming on the Java platform. *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools*, ACM Press (2013), 67–78.

[22] Bettini, L., Damiani, F., and Schaefer, I. Implementing software product lines using traits. *the ACM Symposium on Applied Computing*, (2010), 2096–2102.

[23] Bettini, L., Damiani, F., and Schaefer, I. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming 97*, 3 (2015), 282–308.

[24] Biermann, A.W. and Feldman, J.A. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers C-21*, 6 (1972), 592–597.

[25] Biggerstaff, T. and Richter, C. Reusability Framework, Assessment, and Directions. *IEEE Software 4*, 2 (1987), 41–49.

[26] Black, a. P. and Scharli, N. Traits: tools and methodology. *Proceedings. 26th International Conference on Software Engineering*, (2004), 1–11.

[27] Bono, V., Damiani, F., and Giachino, E. Separating type, behavior, and state to achieve very fine-grained reuse. *Electronic proceedings of Formal Techniques for Java-like Programs (FTfJP)*, (2007), 1–15.

[28] Bracha, G. and Cook, W. Mixin-based inheritance. *ACM SIGPLAN Notices 25*, 10 (1990), 303–311.

[29] Chung, W., Harrison, W., Kruskal, V., et al. Concern Manipulation Environment

(CME). *Proceedings of the 27th International Conference on Software Engineering*, (2005), 666–667.

[30] Coleman, D., Hayes, F., and Bear, S. Introducing Objectcharts or how to use Statecharts in object-oriented design. *IEEE Transactions on Software Engineering 18*, 1 (1992), 8–18.

[31] Cruise Group, U. of O. Umple Flattening algorithm. 2017. https://github.com/umple/umple/blob/master/cruise.umple/src/UmpleInternalParser_CodeTrait.ump.

[32] Damiani, F., Schaefer, I., Schuster, S., and Winkelmann, T. Delta-trait programming of software product lines. *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA, Springer LNCS 8802*, (2014), 289–303.

[33] Denier, S. Traits Programming with AspectJ. *RSTI-L'objet 11*, 3 (2005), 69–86.

[34] Domínguez, E., Pérez, B., Rubio, Á.L., and Zapata, M.A. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology 54*, 10 (2012), 1045–1066.

[35] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A.P. Traits: A Mechanism for Fine-grained Reuse. *ACM Transactions on Programming Languages and Systems 28*, 2 (2006), 331–388.

[36] Duggan, D. and Techaubol, C.-C. Modular mixin-based inheritance for application frameworks. *ACM SIGPLAN Notices 36*, 11 (2001), 223–240.

[37] Easterbrook, S. and Nuseibeh, B. Using Viewpoints for Inconsistency Management. *in BCS/IEE Software Engineering Journal*, (1996), 31–43.

[38] Easterbrook, S.M. and Chechik, M. A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. *23rd International Conference on Software Engineering (ICSE'01)*, (2001), 411–420.

[39] El-Fakih, K., Yevtushenko, N., and Bochmann, G. V. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering 30*, 7 (2004), 425–436.

[40] Elrad, T., Aldawud, O., and Bader, A. Aspect-Oriented Modeling: Bridging the Gap Between Implementation and Design. *the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, (2002), 189–201.

[41] Frakes, W.B. and Kang, K. Software reuse research: status and future. *IEEE Transactions on Software Engineering 31*, 7 (2005), 529–536.

[42] Frogley, T. An introduction to C++ Traits. *Overload Journal #43, http://accu.org/index.php/journals/442*, (2001).

[43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., 1995.

[44] Garzon, M. and Lethbridge, T.C. Exploring How to Develop Transformations and Tools for Automated Umplification. *Proceedings of the19th Working Conference on Reverse Engineering*, IEEE (2012), 491–494.

[45] Ge, J., Xiao, J., Fang, Y., and Wang, G. Incorporating aspects into UML state machine. *3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, IEEE (2010), V2-505-V2-508.

[46] Grundy, J. and Hosking, J. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering 24*, 11 (1998), 960–981.

[47] Harel, D. and Politi, M. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. Mcgraw-Hill, 1998.

[48] Http://cruise.eecs.uottawa.ca/umple/APISummary.html. Umple API References. 2017.

[49] Http://cruise.eecs.uottawa.ca/umple/GrammarNotation.html. Umple Grammar. 2017.

[50] Http://cruise.eecs.uottawa.ca/umple/umple-compiler-classDiagram.shtml. Umple metamodel. 2017.

[51] Http://cruise.eecs.uottawa.ca/umple/UmpleGrammar.html#generate. Supported languages and outputs in Umple. 2017.

[52] Https://git-scm.com/. Git. 2017.

[53] Igarashi, A., Pierce, B.C., and Wadler, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*, 3 (2001), 396–450.

[54] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices 32*, 10 (1997), 318–326.

[55] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University*, November (1990), 161.

[56] Kiczales, G., Lamping, J., Mendhekar, A., et al. Aspect-oriented programming. *the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241*, (1997), 220–242.

[57] Kienzle, J., Abed, A.W., and Klein, J. Aspect-oriented multi-view modeling. *Proceedings of the 8th ACM international conference on Aspect-oriented software development - AOSD '09*, ACM Press (2009), 87.

[58] Krueger, C.W. Software reuse. *ACM Computing Surveys (CSUR) 24*, 2 (1992), 131–183.

[59] van Lamsweerde, A., Darimont, R., and Letier, E. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering 24*, 11 (1998), 908–926.

[60] Lee, E.H.-S. Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language. *Stanford University, thesis*, (1988), 508.

[61] Lee, J. and Hsu, K.H. GEA: A goal-driven approach to discovering early aspects. *IEEE Transactions on Software Engineering 40*, 6 (2014), 584–602.

[62] Lethbridge, T. Teaching modeling using Umple: Principles for the development of an effective tool. *Proceedings of the 27th IEEE Conference on Software Engineering Education and Training (CSEE&T)*, (2014), 23–28.

[63] Lethbridge, T.C., Abdelzad, V., Orabi, M.H., Orabi, A.H., and Adesina, O. Merging Modeling and Programming Using Umple. *ISoLA*, (2016), 187–197.

[64] Lienhard, A., Ducasse, S., and Arévalo, G. Identifying traits with formal concept analysis. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, ACM Press (2005), 66–75.

[65] Liquori, L. and Spiwack, A. *Featherweight-trait Java: A trait-based extension for FJ*. RR-5247, INRIA Sophia Antipolis - Méditerranée; INRIA, 2004.

[66] Liquori, L. and Spiwack, A. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems 30*, 2 (2008), 1–32.

[67] Mahoney, M., Bader, A., and Elrad, T. Using Aspects to Abstract and Modularize Statecharts. *International Workshop on Aspect-Oriented Modeling*, (2004), 1–8.

[68] Meyers, S. The Most Important Design Guideline? *IEEE software 21*, 4 (2004), 14–16.

[69] Meyers, S. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[70] Morisio, M., Ezran, M., and Tully, C. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering 28*, 4 (2002), 340–357.

[71] Murphy-Hill, E.R., Quitslund, P.J., and Black, A.P. Removing duplication from java.io. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press (2005), 282–291.

[72] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. Matching and merging of statecharts specifications. *Proceedings - International Conference on Software Engineering*, (2007), 54–63.

[73] Neysian, B.S. and Babamir, S.M. Automatic verification of uml state chart by bogor model checking tool: Automatic formal verification of network and distributed systems. *Conference Proceedings of 2015 2nd International Conference on Knowledge-Based Engineering and Innovation, KBEI 2015*, (2016), 797–802.

[74] Nierstrasz, O., Ducasse, S., Reichhart, S., and Schärli, N. *Adding Traits to (Statically Typed) Languages*. Technical Report IAM-05-006, Institut f¨ ur Informatik und Angewandte Mathematik University of Bern, Switzerland, 2005.

[75] Pasetti, A. and Cechticky, V. The Framework Profile - FW Profile -. 2013, 1–71. https://fwprofile.googlecode.com/files/FWProfile_Def_1.3.pdf.

[76] Pasetti, A. and Cechticky, V. *The Framework Profile C1 Implementation - User Manual* -. PP-UM-COR-00001, Revision 1.2.0, P&P Software GmbH, Switzerland, 2013.

[77] Pasetti, A. and Cechticky, V. The CORDET Framework - Definition -. 2015, 1–88. http://pnp-software.com/cordetfw/cordetfw.pdf.

[78] Petre, M. "No shit" or "Oh, shit!": responses to observations on the use of UML in

professional practice. *Software & Systems Modeling 13*, 4 (2014), 1225–1235.

[79] Pilitowski, R. and Derezińska, A. Code Generation and Execution Framework for UML 2.0 Classes and State Machines. In *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Springer Netherlands, Dordrecht, 2007, 421–427.

[80] Pohl, K., Böckle, G., and Van Der Linden, F. *Software product line engineering: Foundations, principles, and techniques*. Springer-Verlag Berlin Heidelberg, 2005.

[81] Pradel, M., Bichsel, P., and Gross, T.R. A framework for the evaluation of specification miners based on finite state machines. *2010 IEEE International Conference on Software Maintenance*, IEEE (2010), 1–10.

[82] Quitslund, P.J. *Java Traits — Improving Opportunities for Reuse*. Technical Report CSE-04-005, OGI School of Science & Engineering Oregon Health & Science University, 2004.

[83] Quitslund, P.J. and Black, A.P. Java with Traits — Improving Opportunities for Reuse. *Proceedings of the 3rd International Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (ECOOP)*, (2004), 45–49.

[84] Quitslund, P.J., Murphy-Hill, E.R., and Black, A.P. Supporting Java traits in Eclipse. *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, ACM Press (2004), 37–41.

[85] Reichhart, S. *Thesis: Traits in CSharp*. Software Composition Group, University of Bern, Switzerland, 2005.

[86] Reppy, J. and Turon, A. Metaprogramming with traits. *Proceedings of the 21st European conference on Object-Oriented Programming (ECOOP)*, (2007), 373–398.

[87] Robinson, W.N. and Pawlowski, S.D. Managing requirements inconsistency with development goal monitors. *IEEE Transactions on Software Engineering 25*, 6 (1999), 816–835.

[88] Roy, C.K., Cordy, J.R., and Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming 74*, 7 (2009), 470–495.

[89] Sabetzadeh, M. and Easterbrook, S. Analysis of inconsistency in graph-based viewpoints: a category-theoretical approach. *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, (2003), 12–21.

[90] Sabetzadeh, M., Nejati, S., Easterbrook, S., and Chechik, M. A relationship-driven framework for model merging. *Proceedings - ICSE 2007 Workshops: International Workshop on Modeling in Software Engineering, MISE'07*, (2007), 2–7.

[91] Sabetzadeh, M., Nejati, S., Easterbrook, S., and Chechik, M. Global Consistency Checking of Distributed Models with TReMer +. *30th International Conference on Software Engineering (ICSE'08)*, (2008), 815–818.

[92] Sakkinen, M. Disciplined inheritance. *European Conference on Object-Oriented Programming (ECOOP)*, (1989), 39–56.

[93] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A.P. Traits: Composable Units of

Behaviour. *ECOOP – European Conference on Object-Oriented Programming, volume 2743 of Lecture Notes in Computer Science*, Springer Berlin Heidelberg (2003), 248–274.

[94] Selic, B. An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. *Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, (1993), 335–344.

[95] Selic, B., Gullekson, G., McGee, J., and Engelberg, I. ROOM: an object-oriented methodology for developing real-time systems. *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, IEEE Comput. Soc. Press (1992), 230–240.

[96] Shulga, T.E., Ivanov, E.A., Slastihina, M.D., and Vagarina, N.S. Developing a software system for automata-based code generation. *Programming and Computer Software 42*, 3 (2016), 167–173.

[97] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices 21*, 11 (1986), 38–45.

[98] Uchitel, S. and Chechik, M. Merging partial behavioural models. *ACM SIGSOFT Software Engineering Notes 29*, 6 (2004), 43–52.

[99] Ungar, D., Chambers, C., Chang, B.W., and Hölzle, U. Organizing programs without classes. *Lisp and Symbolic Computation 4*, 3 (1991), 223–242.

[100] Van, C.T. and Miller, M.S. Traits.js: robust object composition and high-integrity objects for ecmascript 5. *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, ACM Press (2011), 1–8.

[101] Walkinshaw, N. and Bogdanov, K. Automated Comparison of State-Based Software Models in Terms of Their Language and Structure. *ACM Transactions on Software Engineering and Methodology 22*, 2 (2013), 1–37.

[102] Zhang, G. Towards Aspect-Oriented State Machines. *Asian Workshop on Aspect-Oriented Software*, (2006), 1–5.

[103] Ziemann, P., Hölscher, K., and Gogolla, M. From {UML} Models to Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science 127*, 4 (2005), 17–33.

[104] Scala. http://www.scala-lang.org/.

[105] Perl. https://www.perl.org/.

[106] JHotDraw 7. *2004*. http://www.randelshofer.ch/oop/jhotdraw/.

[107] ECSS, Ground Systems and Operations - Telemetry and Telecommand Packet Utilization Standard. 2003.

[108] Traits in PHP. 2014. http://www.php.net/manual/en/language.oop5.traits.php.

[109] *Object Constraint Language, version 2.4*. Object Management Group, 2014.

[110] Traits in Ruby. 2014. http://ruby-naseby.blogspot.ca/2008/11/traits-in-ruby.html.

[111] AspectJ. 2014. https://www.eclipse.org/aspectj/.

[112] CodePro Analytix. 2014. https://developers.google.com/java-dev-tools/codepro/doc/.

[113] Choco. *Http://www.choco-solver.org/*, 2016.

[114] Umple User Manual. *Cruise Group, University of Ottawa*, 2017. http://www.umple.org.

[115] Umple, Model-Oriented Programming. 2017. http://www.umple.org.

[116] UmpleOnline. 2017. http://www.try.umple.org.

[117] Unified Modeling Language (UML)- Object Management Group. 2017. http://www.omg.org/spec/UML/.

[118] GitHub Umple. 2017. https://github.com/umple/Umple.

[119] Umple Traits Grammar. *Cruise Group, University of Ottawa*, 2017. https://github.com/umple/umple/blob/master/cruise.umple/src/umple_traits.grammar.

[120] OBS Framework Design Patterns. 2017. http://www.pnp-software.com/ObsFramework/doc/indexDesignPatterns.html.

[121] Umple Traits Test Cases. 2017. https://github.com/umple/umple/blob/master/cruise.umple/test/cruise/umple/compiler/UmpleTraitTest.java.

Appendices

Appendix I:

Table 5. List of errors and warnings (W) raised by the Umple compiler for traits

Error or warning Code	Description	Explained in
200	The name of a trait must be alphanumeric and start with an alpha character, or _ .	3.4
201W	The name of a trait should start with a capital letter.	3.1
202	The used traits by clients must exist in the system.	3.4.1
203	The name of a trait must be unique in the system.	3.1
204	A trait cannot use itself.	3.1.3
205	A trait cannot use itself even indirectly through its used traits.	3.4.3
206	A type bound to a template parameter of a trait by a client must implement interfaces defined as constraint on the parameter.	3.1.7.2
208	A trait must satisfy required methods of its used traits.	3.1.3
210	A client cannot obtain more than one provided method from used traits that have the same signature.	3.4.10
211	When a client uses a trait, it cannot apply removing/keeping and renaming provided methods more than once on the same provided method.	3.1.9.1
212	If a client indicates the signature of a method as a part of using a trait, that method must exist in the trait.	3.4.11.1, 3.4.11.2
213	A trait cannot have a bidirectional association with an interface obtained through a template parameter.	3.6
214	The name of template parameters must be unique.	3.1.7.2
215	The clients must bind a type to an existing template parameter.	3.1.7.2
216	The type of a template parameter cannot be specified more than once.	3.1.7.2
217	The name and type of attributes in traits must be kept unique all the time.	3.1.7.2
218W	A client and its used trait cannot have attributes with the same	3.4.7

	names.	
219	A client must bind types to all template parameters of its used traits.	3.1.7.2
220	When a provided method of a trait is renamed, the new name should be unique in the set of provided methods.	3.1.9.2
221	When a client binds a type to a template parameter of its used trait, the type must exist in the system.	3.4.9.2
222	When a client uses a trait, it must implement the required interfaces of the trait.	3.7
223	When there is a constraint on a template parameter, used class and interfaces in the constraint must exist in the system.	3.4.9
224	A multiple inheritance constraint cannot be applied to the binding type of a template parameter.	3.1.7.2
225	A type bound to a template parameter of a trait by a client must be a subclass of the class defined as constraint for the parameter.	3.1.7.2
228W	When two state machines or regions are composed and have different initial states, this informative warning is raised.	3.5.5.5
229	The name of a state machine in a trait cannot be renamed more than once by a client.	3.2.4.2
230	When an operator is applied to a state machine or state in a trait, the state machine or state must available through that trait.	3.5.4.1-7
231	When an operator is applied to an event of a state machine in a trait, the event must exist in the state machine.	3.2.4.4, 3.5.4.8
232	When an operator is applied to an event of all state machines in a trait, the event must exist at least in one of the state machines.	3.5.4.4
233	The initial state of a state machine cannot be removed by an operator.	3.5.4.6
234	The composition algorithm will not compose transitions of state machines that cause the composed state machine becomes nondeterministic.	3.5.5.2
235	When a superCall is used in actions or activities, it must refer to a unique action or activity.	3.5.5.3
236	The composition algorithm cannot compose two entries coming from two different used traits for a specific state.	2.1.5.1, 3.5.5.3
237	When a region is renamed, the new name must be unique for the including state.	3.2.4.3

Appendix II

This appendix describes the technical requirements used to develop traits for model-driven development (MDD). The following is the description of columns related to the requirement table.

- **ID:** A unique identification code used to identify the requirement. It is also supposed to be used as a reference when other requirements need to refer to this requirement.
- **Requirement Description:** An imperative statement of the requirement. The modal verb “must” and “should” are used in the description to show the functionality of the requirement is mandatory. The optional requirements use modal verbs “may” and ”might” in their description.
- **Justification:** Explains the rationale for why the requirement is important when providing a traits capability in MDD.
- **Source**: Indicates from which source the requirement has been extracted. It has three values as follows:
  - original def.: Indicates that the requirement comes from the original definition of traits.
  - ext: Indicates that the requirement comes from an extension to the original definition that might be proposed by other researchers.
  - new: Indicates it comes from the research performed with the objective of introducing traits in the MDD context (implemented in Umple).
- **Satisfied:** Indicates whether the requirement has been implemented in Umple traits.
- **Test case:** Indicates whether the requirement has been covered by test cases developed for validation and verification of the design and implementation of traits in Umple.

ID	Requirement Description	Justification	Source	Satis-fied	Test case
R001	Traits must specify required methods.	Some reusable assets depend on the context in which they are reused. They need to be able to call other specific functions (methods) to be able to provide the functionality they promise. Traits are reusable assets and they can be reused in different contexts. Therefore, they need to specify their required functions.	original def.	yes	yes
R002	Required methods of traits must be satisfied by classes.	Since provided functionality needs required methods to provide promised functionality, the required methods must be available (implemented) in any classes that want to have the provided functionality. It is also invalid to call (invoke) a method that in turn is using another method which is not available in the system.	original def.	yes	yes
R003	Traits must offer provided methods for clients.	Traits are considered as reusable assets and therefore they need to bring value to their clients. This value can be provided by offering methods that provide functionality to clients.	original def.	yes	yes
R004	Traits must be able to use required methods inside their provided methods.	Traits can define required methods, and therefore those required methods should be used by provided methods. Otherwise, there is no benefit of having the concept of required methods.	original def.	yes	yes
R005	Traits must be able to use provided methods inside other provided methods.	Traits might define some local methods to modulate some functionality required by several other provided methods. Therefore, it must be possible to achieve such a degree of modularity inside traits.	original def.	yes	yes
R006	Traits' names must be unique in the system under development (there is an exception, check requirement R047)	Traits can be reused several times in the system under development, therefore, their names must be unique within the set of clients using each trait so as to be able to properly refer to them or reuse them.	original def.	yes	yes
R007	The signature of provided methods must be unique.	Provided methods are used by clients or other provided methods of traits; therefore, their signatures are required to be unique within the set of clients using each trait.	original def.	yes	yes
R008	The signature of required methods must be unique.	Required methods are implemented by clients and used in provided methods of traits; therefore, their signatures are required to be unique within the set of clients using each trait to validate that they are satisfied in clients and uniquely referred to in provided methods.	original def.	yes	yes
R009	Traits must be able to define attributes.	In some situations, provided methods might need to communicate with each other through attributes. For example, keeping the state of a variable. Therefore, traits should be able to define those attributes.	ext.	yes	yes
R010	The name and type of attributes in traits must be unique.	An attribute is used by its name in provided methods, therefore, its name and type must be unique within the set of clients using any trait. Not having unique names and types can cause conflicts when they are compiled by compiler as well.	ext.	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
R011	Traits must be able to define their required attributes through required methods	Traits might prefer not to have attributes to keep states of their activities and request them from clients that use them. In that case, the required methods must return a type which is compatible with the required attribute type.	original def.	yes	yes
R012	Traits must be able to define one or many template parameters that can allow a type referred in a trait to be referred by a different name in a client. The maximum number is 255.	There might be situations in which traits cannot be reused as a result of differences in types of provided methods. Therefore, having template parameters will allow tackling these cases. The limit of 255 is imposed by the needs of Java as a target language.	ext.	yes	yes
R013	Traits must be able to define constraints on template parameters regarding what interfaces must be implemented by the bound types.	This is required to make sure the bound types can be used without fault in traits.	ext.	yes	yes
R014	Traits must be able to define constraints on template parameters regarding what superclass bound types must have.	This is required to make sure the bound types can be used without fault in traits.	ext.	yes	yes
R015	Traits must be able to use template parameters for types of parameters of required methods.	In order to design general traits, general required methods must be defined. Therefore, it must be possible to use template parameters as types for parameters of required methods. In other words, having template parameters for required methods makes them adaptable.	ext.	yes	yes
R016	Traits must be able to use template parameters for types of parameters of provided methods.	Designing general traits requires having general provided methods. Therefore, it must be possible to use template parameters as types for parameters of provided method. In other words, having template parameters for provided methods makes them adaptive.	ext.	yes	yes
R017	Traits must be able to use template parameters for types of attributes.	General provided methods of traits might need to have general attributes. Therefore, it must be possible to used template parameters as types for attributes.	ext.	yes	yes
R018	Traits must be able to use names of bound types of template parameters as strings in the code found in method bodies.	In model-oriented technology it is possible to have model and code together, there might be a need to allow executable code to use bound types. For example, being able to create an instance of a bound type. This feature allows the achievement of such a capability.	new	yes	yes
R019	Traits must be able to use other traits.	Traits need some functionality to achieve to their goal. That functionality might be obtained through using other traits. Therefore, it must be possible for them to use other traits.	original def.	yes	yes
R020	When a client uses another trait, the client's provided methods must have higher priority over the same	When a client uses another trait, it might be possible to have methods with the same signature in the used trait. The client requires their own provided methods, therefore, the ones coming from used traits need to be disregarded. Furthermore,	original def.	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	methods coming from the used trait.	the client might have those methods to intentionally overwrite incoming methods.			
R021	When a client uses another trait, the client's attributes must have higher priority over the same attribute coming from the used trait.	When a client uses another trait, it might get attributes with the same names and types from the used trait. The client requires their own attributes, therefore, the ones coming from the used trait need to be disregarded. Note: this case can be problematic because the attribute might be manipulated by several unrelated provided methods. This situation must be reported to the developer.	ext.	yes	yes
R022	A trait should be able to satisfy the required methods of their used traits with their own provided methods.	When a trait uses other traits, it might want to satisfy required methods of those used traits. Therefore, the trait needs to define methods for them. These methods in fact are provided methods of the trait.	original def.	yes	yes
R023	A trait does not need to satisfy required methods of their used traits.	Traits are not final elements in the design of systems and they will be used finally by classes, therefore, it is logical to be able to postpone satisfaction of those required methods. This can be useful when a trait uses other traits in order to have more functionality, without worries about their requirements.	original def.	yes	yes
R024	When a trait uses more than one trait, it cannot obtain two or more methods with the same signature from those traits (there is an exception, check R025)	The name of provided methods in traits must be unique. Therefore, it should not be allowed to have two methods with the same name.	original def.	yes	yes
R025	When a trait uses more than one trait, it can accept two or more methods with the same signature from those traits if they come from the same source trait somewhere in the use hierarchy of traits. In this case, one of them must be considered in the set of provided methods.	The name of provided methods in traits must be unique. However, if there are several methods which are coming from the same source, keeping one and disregarding the rest should be possible. Technically, all other provided methods will be able to use the method and the result will be valid.	original def.	yes	yes
R026	When a trait uses more than one trait, it cannot obtain two or more attributes with the same signature from those traits (there is an exception, check R027)	The name of attributes in traits must be unique. Therefore, it should not be allowed to have two methods with the same name.	ext.	yes	yes
R027	When a trait uses more than one trait, it can accept two or more attributes with the same signature from those traits if they come from the same source trait in the trait use	The name and type of attributes in traits must be unique. However, if there are several attributes which come from the same source, keeping one and disregarding the rest should be possible. Technically, all other provided methods will be able to use the attributes and the output will be valid as well.	ext.	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	hierarchy. In this case, once of them must be considered in the set of attributes.				
R028	A class must be able to reuse traits.	Classes are central concept in designing object-oriented systems. Since traits are considered as reusable assets, classes need to be able to reuse them.	original def.	yes	yes
R029	A class must be able to reuse traits even it has a superclass	One of the benefits of traits is that they can be reused when their functionality is required. Therefore, there should not be any issue when a class reuses them while it has a superclass.	original def.	yes	yes
R030	A class must be able to implement interfaces through using traits.	Traits have provided methods which bring functionality to their clients. Provided methods are like concrete methods in classes. Therefore, when classes reuse traits those provided methods should be able to implement abstract methods of interfaces. This will be possible if provided methods and abstract methods in interfaces and traits have the same signature.	original def.	yes	yes
R031	When clients (classes or traits) use traits, they must be able to remove provide methods.	When clients use traits, there might be conflicts because of name collision of provided methods. Furthermore, clients might not need specific provided methods. Therefore, it is logical to be able to remove conflicting or unrequired methods. In other words, it is required to be able to control granularity of traits.	original def.	yes	yes
R032	When clients (classes or traits) use traits, they must be able to remove attributes.	When clients use traits, there might be conflicts because of name collision of attributes. Furthermore, clients might not need specific attributes. Therefore, it is logical to be able to remove conflicting or unrequired attributes.	ext.	no	no
R033	When clients (classes or traits) use traits, they must be able to rename provided methods.	When clients use traits, there might be conflicts because of name collision of provided methods, and clients might want to keep both conflicting methods. Furthermore, clients might need to use provided methods under specific names that are more domain specific. Therefore, it is logical to be able to rename conflicting provided methods.	original def.	yes	yes
R034	When clients (classes or traits) use traits, they must be able to rename provided attributes.	When clients use traits, there might be conflicts because of name collision of attributes. Furthermore, clients might need to use attributes under specific names that are more domain specific. Therefore, it is logical to be able to rename conflicting attributes. This is also the way to keep the accessing methods of both conflicting attributes.	ext.	no	no
R035	When clients (classes or traits) use traits, they must be able to keep only specific provided methods.	When a trait has many provided methods, it might not be practical to use several remove operators to eliminate all but the provided methods that are needed. This feature allows the developer to deal with this situation.	new	yes	yes
R036	When clients (classes or traits) use traits, they must be able to keep specific attributes.	When a trait has many attributes, it might not practical to use several remove operators to eliminate all but the attributes that are needed. This feature allows the developer to deal with this situation.	new	no	no
R037	When clients (classes or traits) use	Clients, specifically classes, might need to have provided methods with a specific	new	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	traits, they must be able to change the visibility of provided methods.	visibility. For example, they might want to keep provided methods private while those provided methods have been defined as public in traits. The opposite scenario might also be possible.			
R038	When clients (classes or traits) use traits, they must be able to change visibility of attributes.	Clients, specifically classes, might need to have attributes under specific visibility. For example, they might want to keep attributes private while those attributes have been defined as public in traits. The opposite scenario might also be possible.	new	no	no
R039	When a client removes a provided method from the used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore the signature of provided methods must remain untouched even though they might be used in different clients with the remove operator.	original def.	yes	yes
R040	When a client renames a provided method from the used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore the signatures of provided methods must remain untouched even though they might be used in different clients under the rename operator.	original def.	yes	yes
R041	When a client keeps only a particular provided method from the used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore the signatures of provided methods must remain untouched even they might be used in different clients under the keeping operator.	new	yes	yes
R042	When a client removes an attribute from the used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore attributes must remain untouched even though they might be used in different clients under the remove operator.	ext.	no	no
R043	When a client renames an attribute from the used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore attributes must remain untouched even though they might be used in different clients under the rename operator.	ext.	no	no
R044	When a client keeps only a particular attribute from a used trait, it should not affect other clients that use the same trait.	Traits are reusable assets and therefore attributes must remain untouched even though they might be used in different clients under the keeping operator.	new	no	no
R045	When a client uses a trait and the trait has provided methods, those methods must be considered as native methods of the client.	Clients can obtain their required functionality through provided methods of their used traits. This functionality must be used freely in the clients. Therefore, the most flexible way is to consider provided methods as native elements of clients and so all rules relative to native elements can be applied to them.	original def.	Yes	yes
R046	When a client uses a trait and the trait has attributes, those attributes must be considered as native at-	Provided methods of traits uses attributes to achieve some functionality. Clients obtain their required functionality through provided methods. When traits are used by clients their provided methods become clients' native methods, therefore, at-	original def.	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	tributed of the client.	tributes need to be considered as native elements of clients so provided methods could behave properly.			
R047	Traits must be able to be defined under the same name in different files or the same file, with different parts of the two definitions merged.	Developing systems in a way that code and model can be arranged in different files is a good practice. Therefore, it must be possible to achieve this with traits. This feature is called mixin in Umple modeling language.	New	yes	yes
R048	Provided methods of a trait can satisfy required methods of other traits.	A client might use a trait that has some required methods. Those required methods might have been implemented by provided methods of other traits. Therefore, the client must be able to reuse those traits in order to satisfy required methods of the traits. This increases the flexibility of using traits.	original def.	yes	yes
Requirements related to required interfaces					
R200	Traits must be able to define their required interfaces. The maximum number is 255.	Required functionality of classic traits is defined in terms of required methods. They can also be used to put restrictions on classes which will use traits. However, some classes might have methods with signatures that can satisfy those required methods but traits have not been defined for those classes. In fact, there is a need for a new layer of constraint by which traits can semantically specify their final classes. The limitation of 255 is imposed by Java as a target language.	new	yes	yes
R201	The required interfaces of traits must be implemented by classes that use them.	Required interfaces have been defined for traits to make sure the correct classes will have access to them, therefore, classes need to implement required interfaces.	new	yes	yes
R202	Traits cannot implementation inter-faces of their used traits.	Classes implement interfaces and they can use traits to implement them, therefore, the constraint must be put on final clients, which are classes. In fact, the constraint is on classes to make sure that they are right types when traits are used inside them. Traits are not types so it is not required for them.	new	yes	yes
R203	When traits use other traits, the lat-ter's required interfaces become required interface of the former.	Final clients of traits are classes therefore traits are not responsible to implement interfaces. Therefore, the required interfaces of their used traits must become their required interfaces as well.	new	yes	yes
R204	When clients use traits, they cannot remove their required interfaces.	Clients cannot remove required interfaces when they use traits because in that case the functionality those traits promised will not be valid anymore.	New	yes	yes
R205	When clients use traits, they cannot rename required interfaces.	Clients cannot rename required interfaces when they use traits because in that case the functionality those traits promised will not be valid anymore.	new	yes	yes
R206	When a class uses several traits and all of them have the same required interfaces, one implementation of those require interfaces must satisfy the required interfaces of all used traits.	Defining required interfaces is a way that traits put a constraint on classes. If a class uses several traits and those traits have the same required interfaces, imple-menting those required interfaces by the class satisfies all required interfaces of the traits. This is correct technically because a class cannot implement an interface twice.	new	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
R207	When a trait uses several traits and all of them have the same required interfaces, those required interfaces form a set (no duplication).	Traits cannot implement required interfaces of their used traits. Furthermore, one implementation of required interfaces by classes is enough to satisfy required interfaces of used traits (check R206). Therefore, there is no need for duplication of required interfaces. Furthermore, it is wrong technically to have more than one required interface with the same signature for a trait.	new	yes	yes
Requirements related to associations					
R300	Traits must be able to define associations.	In order to integrate traits into the model-driven development process, it should be possible to represent functionality of traits in terms of modeling elements. Associations are one of the most-used modeling elements and so adding them to traits is beneficial.	new	yes	yes
R301	Traits must be able to define associations with template parameters.	In order to have more general associations inside traits, it is necessary to have associations with elements that can be adaptive. Having associations with template parameters is a way to achieve this.	new	yes	yes
R302	When a trait makes bidirectional associations with template parameters, the bound values must only be classes.	Bidirectional associations need references in both ends of associations. Interfaces cannot have attributes to keep the references. Therefore, if one end of an association is a template parameter, the bound type cannot be an interface.	new	yes	yes
R303	When a trait makes directional associations with template parameters, the bound types can be interfaces, classes, and primitive types.	Directional associations need a reference at one end. When associations are defined in traits, the end in which the reference will be saved is traits. Therefore, if another end of an association is a template parameter, the bound type can be any type.	new	yes	yes
R304	Traits must be able to make bidirectional associations with classes.	Traits can have associations; therefore, it should be possible to have bidirectional associations with classes. The benefit is that classes can have dynamic associations with all clients of traits.	new	yes	yes
R305	Traits cannot make bidirectional associations with interfaces.	Traits can have associations, but directional association is not allowed for interfaces because of need to store data regarding the linked objects. Therefore, such a limitation needs to be respected in traits.	new	yes	yes
R306	Traits must be able to make directional associations with interfaces	Traits can have associations; therefore, it should be possible to have directional associations with classes. The benefit is that traits can encapsulate objects of classes without needing to disclose the type of their clients to those classes.	new	yes	yes
R307	Traits must be able to remove associations of their used traits if it is needed.	Clients might use more than one trait and so it is possible to have conflict in associations obtained from those used traits. Therefore, traits need to resolve that conflict. Furthermore, clients might not need some associations of their used traits, therefore, they should be able to remove them.	new	no	no
R308	Traits must be able to rename role name of associations in their used	Clients might use more than one traits and so it is possible to have conflict in associations obtained from those used traits. However, clients might want to keep	new	no	no

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	traits if it is needed.	those associations under different names. Therefore, traits need to be able to re-name associations. Furthermore, clients might want to rename associations of their used traits to be closer to the domain in which they are reused.			
R309	Traits must be able you modify the multiplicity of associations in their used traits if it is needed.	Multiplicity of associations is one of the dynamic aspects of them, therefore, being able to modify multiplicity of associations in used traits, will make traits more flexible reusable assets.	new	no	no
R310	When a client uses a trait and the trait has associations, those associa-tions must be considered as native associations of the client.	Clients obtain associations of traits when they use them. Along with associations many functions come. For example, we can add an element to an association or remove an element. Therefore, those associations must be considered as native associations for clients.	new	yes	yes
Requirements related to state machines					
R400	Traits must be able to define state machines as one of their elements.	Modern software development approaches use modeling elements such state ma-chine to model functionality of systems. In order to have the benefits of traits in modern software development, it is required to provide functionality of traits in terms of state machines.	new	yes	yes
R401	The name of state machines inside traits must be unique.	A state machine can be referred to when traits are used and trait can have more than one state machine, therefore, their names must be unique. Furthermore, when state machines are implemented in OO programming languages, they will proba-bly be converted to instance variables. In these languages, it is invalid to have two instance variables with the same name.	new	yes	yes
R402	Events of state machines in traits must be considered like provided methods with their own parameters, but with a Boolean return type.	Events in state machines have signatures like methods. They can have parameters and return types. In fact, if a class or trait defines a state machine the events of the state machine will be the methods of the class or trait. Therefore, it is logical to consider events as provided methods. However, we consider a restriction on events regarding the fact that they need to return whether they caused a state tran-sition or not. This forces their return type to be Boolean.	new	yes	yes
R403	Event of state machines in traits can be used to satisfy required methods of used traits.	Since in requirement R402 we consider that events can be considered as provided methods, therefore, they can be used to satisfy required methods.	new	yes	yes
R404	When a trait is used by a client and the trait has state machines, those state machines must be considered as native state machines for the client.	Clients need some functionality and it can be obtained through state machines in traits. This functionality must be used freely in the clients. Therefore, the most flexible way is to consider state machines as native elements of clients and so all rules relative to native elements can be applied to them as well.	new	yes	yes
R405	When a trait is used by a class, the events of state machines must be	Since in requirement R402 we consider that events can be considered as provided methods, therefore, they can be used to implement abstract methods of interfaces	new	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	able to satisfy implementation of the interfaces that is supposed to be implemented by the class.	that are supposed to be implemented by classes.			
R406	When a client uses a trait that has a state machine, the client must be able to add new states to the state machine.	Traits can be composed of other traits, and classes can need state machines inside traits to have some extra states to better model the final required functionality. Therefore, they need to be able to add new states to the state machines of their used traits.	new	yes	yes
R407	When a client uses a trait that has a state machine, the client must be able to add new transitions to the state machine	Since in requirement R406 we consider that clients can add new states to the state machines of their used traits, therefore, it is required to be able to define transition for them as well.	new	yes	yes
R408	When a client uses a trait that has a state machine, the client must be able to change the destination of transitions.	Clients might need slightly different functionality of the one obtained from the state machines of their used traits. Therefore, they need to be able to change the destination of some transitions to achieve this.	new	yes	yes
R409	When a client uses a trait that has a state machine, the client must be able to change the action of transi-tions.	Clients might need slightly different functionality of the one obtained from the state machines of their used traits. Therefore, they need to be able to change the actions of some transitions to achieve this.	new	yes	yes
R410	When a client uses a trait that has a state machine, the client must be able to change the entry action of states.	Clients might need slightly different functionality of the one obtained obtain from the state machines of their used traits. Therefore, they need to be able to change the entry actions of some states to achieve this.	new	yes	yes
R411	When a client uses a trait that has a state machine, the client must be able to change the exit action of states.	Clients might need slightly different functionality of the one obtained from the state machines of their used traits. Therefore, they need to be able to change the exit actions of some states to achieve this.	new	yes	yes
R412	When a client uses a trait that has a state machine, the client must be able to change the do activity of states.	Clients might need slightly different functionality of the one obtained from the state machines of their used traits. Therefore, they need to be able to change the do activities of some states to achieve this.	new	yes	yes
R413	When a client uses a trait that has a state machine, the client must be able to add regions to composite states of the state machine.	Clients might need to extend functionality of some state machines in their used traits. This might be required to add new regions to some composite states. There-fore, having such a feature allows reuse of traits in a more adaptive way.	new	yes	yes
R414	When a client uses a trait that has a state machine, the client cannot	Guards are one of the factors used to distinguish one transition from another one. Therefore, it must not be possible change guards of transitions. This is a decision	new	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	change the guard of transitions.	made to make sure the way clients can extend the state machines in their used traits is valid and controllable.			
R415	When a client uses a trait that has a state machine, the client cannot change the event of transitions.	Events are one of the factors used to distinguish one transition from another one. Therefore, it must not be possible change events of transitions. This is a decision made to make sure the way clients can extend the state machines in their used traits is valid and controllable.	new	Yes	yes
R416	When a client uses a trait that has a state machine, the client must be able to add new actions to transitions while still using the transitions' already-defined actions.	Clients might need to add some functionality to actions of transitions in the state machines of their used traits. Overwriting them will result in not being able to have access to those actions defined before. Therefore, clients might duplicate those actions again. Therefore, it must be possible to achieve this without duplicating those actions.	new	yes	yes
R417	When a client uses a trait that has a state machine, the client must be able to add new entry actions to states while using the states' already-defined entry actions.	Clients might need to add some functionality to entry actions of transitions in the state machines of their used traits. Overwriting them will result in not being able to have access to those entry actions defined before. Therefore, clients might duplicate those entry actions again. Therefore, it must be possible to achieve this without duplicating those entry actions.	new	yes	yes
R418	When a client uses a trait that has a state machine, the client must be able to add new exit actions to states while using the states' already defined exit actions.	Clients might need to add some functionality to exit actions of transitions in the state machines of their used traits. Overwriting them will result in not being able to have access to those exit actions defined before. Therefore, clients might duplicate those exit actions again. Therefore, it must be possible to achieve this without duplicating those exit actions.	new	yes	yes
R419	When a client uses a trait that has a state machine, the client must be able to add new do activities to states while using the states' already defined do activities.	Client might need to add some functionality to do activities of transitions in the state machines of their used traits. Overwriting them will result in not being able to have access to those do activities defined before. Therefore, clients might duplicate those do activities again. Therefore, it must be possible to achieve this without duplicating those do activities.	new	yes	yes
R420	When a client uses a trait that has a state machine, the client must be able to extend a state with another state machines.	When a system is developed based on state machines it is common to incrementally improve or extend the state machines to achieve the required functionality. Typically this is performed in the way in which new state machines are created and then some of states are extended to be a composite state. It is possible to need that extended behavior in some other places. However, it is not possible with current techniques. Therefore, it is a big advantage for traits to be able to reuse state machines defined in other traits and consider them as internal behavior of their or other state machines. By doing this, state machines will be more reusable and also traits can extend their state machines incrementally.	new	yes	yes
R421	When a client uses a trait that has a state machine, the client must be	Clients can use traits and so there might be conflicts because of the name collision of state machines. Furthermore, clients might not need specific state machines.	new	yes	Yes

ID	Requirement Description	Justification	Source	Satisfied	Test case
	able to remove the state machine.	Therefore, it is logical to be able to remove conflicting or unrequired state machines. In other words, it is required to be able to control granularity of state machines in traits.			
R422	When a client uses a trait that has a state machine, the client must be able to remove states of the state machine.	When clients use several state machines and they need to be composed, the common state name might be source of conflict. Furthermore, clients might not need specific states. Therefore, it is logical to be able to remove conflicting or unrequired states from state machines of the used traits.	new	yes	yes
R423	When a client uses a trait that has a state machine, the client must be able to remove transitions of the state machine.	Clients can extend the behavior of the state machines in their used traits. This extension might be required to remove a transition. Furthermore, it can be possible to have a situation in which a client uses more than one trait and there are common state machines among them. Clients need to compose them but there may be one or more problematic or unnecessary transitions. Therefore, it should be possible to be able to remove transitions from state machines of the used traits.	new	yes	yes
R424	When a client uses a trait that has a state machine, the client must be able to remove regions of a composite states of the state machine.	Clients can extend the behavior of the state machines in their used traits. This extension might be required to remove a region. Furthermore, it can be possible to have a situation in which a client uses more than one trait and there are common state machines among them. Client needs to compose them but there may be one or more problematic or unnecessary region. Therefore, it should be possible to be able to remove regions from state machines of the used traits.	new	yes	yes
R425	When a client uses a trait that has state machines, the client must be able to keep one or several of those state machines.	It is possible to have traits with many state machines and clients might be interested in one or few of them. Using remove operators might not be a suitable approach. Therefore, it is beneficial to clearly specify which state machines are needed to be kept, removing others.	new	yes	yes
R426	When a client uses a trait that has a state machine, the client must be able to keep one or several of states of the state machine.	State machines in traits can have many states and clients might be interested in some of those states. Using remove operators might not be a suitable approach. Therefore, it is beneficial to clearly specify which states from state machines are to be kept, removing others.	new	yes	Yes
R427	When a client uses a trait that has a state machine, the client must be able to keep one or several of transitions of a states of the state machine.	State machines in traits can have many states and clients might be interested in some of those states. Using remove operators might not be a suitable approach. Therefore, it is beneficial to clearly specify which states from state machines are to be kept, removing others.	new	yes	yes
R428	When a client uses a trait that has state machines, the client should not be able to remove the initial states of the state machines.	Clients have this ability to remove states of state machines in their used traits. However, it must not be possible for initial states. The reason is that other states will not be reachable.	new	yes	yes
R429	When a client uses a trait that has a	The same way that clients cannot remove initial states of state machines in their	new	yes	yes

ID	Requirement Description	Justification	Source	Satis-fied	Test case
	state machine, the client should not be able to remove the initial state of the composite states of state machine.	used traits, they should not be able to do it for initial states of composite states of the state machines.			
R430	When a client uses a trait that has state machines with the same name, those state machines must be composed.	Clients might use several traits and those traits might have state machines. Clients might need to have those state machines represented under one state machine. This should be possible to achieve. Furthermore, those state machines might have the same names but slightly different functionality and clients want to have a composed state machine that have accumulation of that functionality. Therefore, composition of state machines is a good feature for traits to be supported.	new	yes	yes
General implementation requirements					
R501	When traits are used at modeling level, they should be implementable in object-oriented programming languages	In order to make model-based traits practical, it should be possible to execute such models. In order to execute the system, we need to generate code for the model. Since traits focus on systems designed based on object-oriented principles, the language of generated code should be object-oriented as well. Having this capability is critical for its adoption.	new	yes	yes