Queued and Pooled Semantics for State Machines in the Umple Model-Oriented Programming Language

By

Aliaa A. Alghamdi

MSc Thesis

Presented to the Faculty of Graduate and Postdoctoral Studies in
partial fulfillment of the requirements for the degree

Master of Science in Systems Science

School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

Abstract

This thesis describes extensions to state machines in the Umple model-oriented programming language to offer queued state machines (QSM), pooled state machines (PSM) and handing of the arrival of unexpected events. These features allow for modeling the behavior of a system or protocol in a more accurate way in Umple because they enable detecting and fixing common design errors such as unspecified receptions. In addition, they simplify the communication between communicating state machines by allowing for asynchronous calls of events and passing of messages between state machines. Also, a pooled state machine (PSM) has been developed to provide a different policy of handling events that avoid unspecified receptions. This mechanism has similar semantics as a queued state machine, but it differs in the way of detecting unspecified receptions because it helps handling these errors. Another mechanism has been designed to use the keyword 'unspecified' in whatever state of a state machine the user wants to detect these errors. In this thesis, the test-driven development (TDD) process has been followed to first modify the Umple syntax to add 'queued,' 'pooled,' and 'unspecified' keywords to Umple state machine's grammar; and second, to make a change to the Umple semantics in order to implement these extensions in Umple. Then, additional modifications have been made to allow for Java code generation from those types of state machines. Finally, more test cases have been written to ensure that these models are syntactically and semantically correct. In order to show the usefulness and usability of these new features, an example is shown as a case study that is modeled using the queued state machine (QSM) besides other small tests cases.

Acknowledgements

I would like to take this opportunity to thank Dr. Timothy C. Lethbridge, my co-supervisor, for his support and assistance throughout this M.Sc. thesis, as well as Dr. Gregor von Bochmann, my head supervisor. Thanks also to my research group, Complexity Reduction Software Engineering (CRuiSE). Sincerest thanks to my father, mother and siblings for their constant support, encouragement and understanding during this endeavour. Finally, I am very thankful to my sponsor Al Baha University for its financial support during the academic program.

Table of Contents

List of Figures

List of Tables

List of Listings

Abbreviations

CFSM: Communicating Finite State Machine.

EFSM: Extended Finite State Machine.

FDT: Formal Description Techniques.

FIFO: First-In First-Out.

FSM: Finite State Machine.

HFSM: Hierarchical Finite State Machine.

MDD: Model-Driven Development.

PSM: Pooled State Machine.

QSM: Queued State Machine.

SDL: Specification and Description Language.

TDD: Test-Driven Development.

UML: Unified Modeling Language.

Umple: **UML P**rogramming **L**anguage, Si**mple**, and A**mple**.

Chapter 1 Introduction

Complex computer-based systems have become essential nowadays. Most such systems comprise a number of entities that operate concurrently and cooperate with each other via communications (Van der Schoot, 1999).

The behavior of concurrent systems has a tendency to become very large and complex. It is therefore hard to model, design, and specify the functionality of such systems and assess whether they meet their requirements (Van der Schoot, 1999).

Important aspects of concurrent systems are communication protocols, wherein the entities communicate according to standard rules (Van der Schoot, 1999). The complexity of these systems is increasing due to the trends toward distributed computing and computer networks.

To handle this complexity, formal methods of specification and analysis of communication software have been used. One such approach is finite state machines (FSM) (Bochmann, 1978). FSM models are widely used for describing and specifying systems in different areas such as sequential circuits, distributed systems, communication networks, and communication protocols. They can also be used to model the behavior of business objects and user interfaces.

A finite state machine (FSM) can be described as follows: Each FSM has a finite number of states and transitions between these states. A finite state machine moves from one state into another when a specified event occurs. During the transition it can perform various actions if specified: an exit action from the origin state, a transition actions, and an entry action for the destination state.

When multiple state machines are used in a distributed system, they can be used to control transmission and reception of communication by exchanging messages over FIFO channels through which the signals flow (Gouda & Chang, 1984).

Umple (Lethbridge, Forward & Badreddin, 2012) supports defining state machines in a textual manner to be used for modeling and implementing various kinds of systems and processes. Umple is a model-oriented programming language developed at the University of Ottawa. It is a textual language that adds Unified Modeling

Language (UML) state machines, and class models to well-known object-oriented programming languages such as Java, PHP, C++ and Ruby. It also supports software patterns such as singleton (Almaghthawi, 2013).

Badreddin (2012) discussed in his PhD thesis the development process for the syntax and semantics of state machines in Umple. He completed the design and implementation of the first version of state machines in Umple. Moreover, he illustrated how the modeling concepts and code of state machines and other system aspects can be integrated in the same artifact; this allows developers to model and write code at the same time (Badreddin, 2012).

However, in the version of Umple as it stood at the time the current thesis research was started, there was no feature that allowed for state machines to communicate via FIFO queues. Events were implemented as synchronous method calls, and, the same thread that called the event was responsible for processing the state transition, as well as any entry, exit and transition actions. This implementation could be very useful to control the state of a single object, but was impractical in multi-threaded software, and could give rise to deadlock, among other problems.

Therefore, we set out to enhance the code generated from Umple so events would be queued; the caller would enter the event in the queue and then continue. This would allow state machines to communicate asynchronously by passing messages in a FIFO manner, and where the output message of one state machine becomes the input for another. The caller of an event message does not have to wait for the state machine to process all of its actions, as had been required in basic state machines in Umple. We call this new extension queued state machines (QSM).

Also, we add so-called *pooled* state machines (PSM). Pooled state machines extend QSM as follows. In original Umple state machines, and queued state machines, if an event arrives while in a state that is not programmed to respond to that state, then the event is ignored and discarded. This problem is called unspecified reception. In PSM, the event is retained at the front of the queue until the system enters a state in which the event can be consumed. Events further back in the queue are processed in the meantime. Among other things, PSM can handle events arriving at unexpected times.

A third feature added is the capability to handle unspecified receptions by designating a transition to be taken when an unexpected event arrives.

An additional feature added to Umple in this thesis is the ability to allow state machine events to have parameters of any type. This makes Umple conform more closely to UML specifications, in which signals and call events can have parameters.

In order to achieve these extensions to Umple, we followed a Test-Driven Development (TDD) methodology. Test-driven development encourages short development cycles or iterations and requires writing tests before releasing any incremental change (Patrick, 2006). The process essentially is to write test code before writing functional code. When these tests are first run, they fail because the desired behavior does not exist yet. To pass these tests, new code is produced. Now, running test code again passes. Finally, the new code is refactored to acceptable standards and to remove any duplications that may occur (Ambler, 2012).

Refactoring the existing design locally enables changing the part of the design that is affected by the new feature a developer wants to implement. This process results in enhancing the quality of the design and makes it easier to work with it in the future. In fact, TDD helps derive the design of the code as well as validate it (Ambler, 2012).

Therefore, we go through several steps as a whole cycle consists of extending Umple sate machine grammar with new syntax, instantiating Umple metamodel, as well as generating appropriate code in the target language (Java) in order to implement those new features.

1.1 Goal of the Thesis

The main goal of this thesis is to extend the Umple model-oriented programming language in order to provide for queued and pooled state machines, a well as event arguments and unspecified reception transitions. The thesis will show how we implemented this feature following the test-driven approach.

The resulting generated code will be more complex but it will work in a multi-threaded environment and allow asynchronous calls to events.

1.2 Research Questions

The current thesis will address the following three research questions:

- How can the Umple Language be extended to have queued and pooled state machines?
- How could the messages be encoded when they are passed between state machines?
- How can one deal with cases of unspecified receptions, for example, error messages?

1.3 Thesis Outline

The thesis is documented and organized in the following manner.

In Chapter 2, we provide some background information related to the Umple model-oriented programming language, showing its development infrastructure and its different tools. We also give an overview of the methodologies used to developing Umple.

In Chapter 3, we focus on the literature review of formal description techniques (FDTs) used for specifying and describing communication protocols and distributed systems. We provide a brief overview of some general design errors, and we outline some proposed solutions for handling them.

In Chapter 4, we discuss our design of queued state machine (QSMs), and we also provide the design of a pooled state machine (PSMs). We show Java code for QSM and PSM, and we discuss the different design alternatives we have for QSM and PSM. In addition, we demonstrate the syntax and semantics of QSM and PSM in Umple.

Chapter 5 discusses the implementation of QSM and PSM in Umple showing some examples of generated Java code from simple and composite QSM and PSM. We also discuss various issues related to the design and implementation of QSM and PSM. Besides, we give an overview of proposed unspecified reception mechanisms in Umple.

In Chapter 6, we show some test cases of QSM, PSM, and the unspecified reception mechanism in Umple. Also, we provide an example showing the usefulness of QSM in Umple.

Finally, in Chapter 7, we conclude our thesis and discuss different possibilities for future work.

Chapter 2 Umple: A Model-Oriented Programming Language

2.1 Introduction to Umple

Umple is a model-oriented programming language developed at the University of Ottawa (Badreddin & Lethbridge, 2012). Created in 2008, Umple became an open-source project hosted by Google Code in 2010. The word "Umple" takes its name from three concepts: **UML P**rogramming **L**anguage**;** Si**mple**; and A**mple** (Forward, 2010).

Umple is a textual language that adds Unified Modeling Language (UML) state machines and class diagrams to well-known programming languages such as Java, PHP, C++ and Ruby (Almaghthawi, 2013). In addition, Umple supports selected software patterns and code tracing (Badreddin & Lethbridge, 2010). It generates high-quality code for all the above-mentioned features and also generates diagrams, metrics, and many other artifacts. It enables developers to draw or edit UML diagrams; thus, they can move back and forth between diagrams and Umple code (Forward, 2010).

The goal of Umple is to reduce the gap between model-centric and code-centric developers. It does this by allowing the modeler to model visually or textually while allowing (code-centric) programmers to continue to program in the way they are used to, but incorporating more abstract model-level features in their code (Forward, 2010). One way of looking at Umple is that it enhances programming languages with model-oriented syntax that raises the level of abstraction to better support designers' intentions. In other words, the Umple textual syntax supports textual representations of high-level system abstractions, as well as low-level algorithmic specifications, in the same development artifacts at the same time (Badreddin, Forward & Lethbridge, 2010).

Umple's capabilities have been shown to enable developers to create programs more effectively and efficiently (Forward, 2010) than using conventional code or model diagrams alone. In particular, the system can be created with fewer keywords than other object-oriented programming languages, such as Java (Forward, 2010).

2.2 Layered Architecture of Umple

As shown in Figure 2.1, Umple and systems written in Umple can be viewed in the context of three metamodeling layers. These correspond to the OMG M2, M1 and M0 layers. The top layer (M2) is the Umple metamodel that has been defined in Umple itself; this represents all the classes in the Umple compiler used to create and compile models. The complete metamodel can be viewed at http://metamodel.umple.org.

The second layer (M1) is that of Umple models that are instances of the Umple metamodel, and describe arbitrary software systems. These describe the Umple elements, classes, associations, attributes, etc. that are part of any Umple program.

The bottom (M0) layer is the elements which are instances of the Umple model representing the objects, values, states, links, etc. found in a running program (Almaghthawi, 2013).

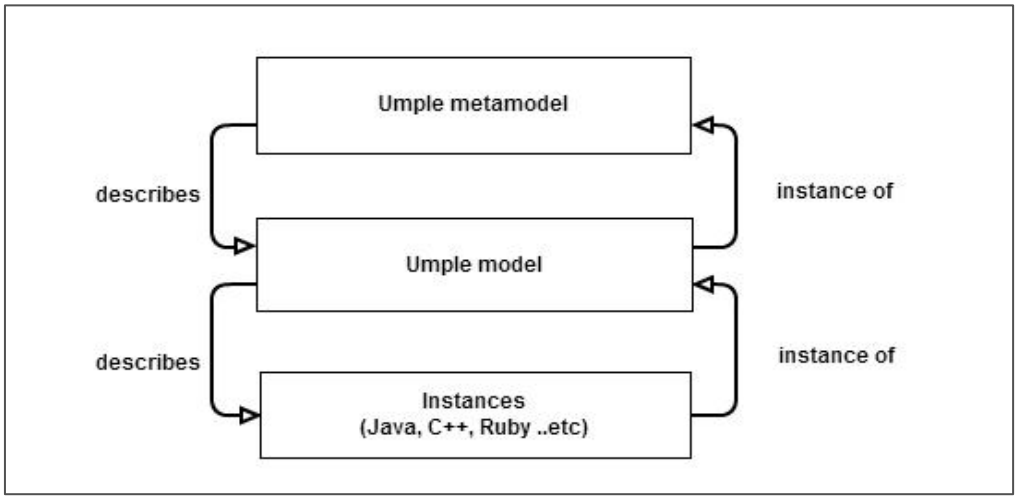


Figure 2.1: Umple metamodeling architecture by Almaghthawi, 2013

2.3 Advantages of Using Umple

Software programmers and developers can benefit from using Umple because it has several advantages over similar tools.

Umple is written in a textual form that is added to code in languages like Java and C++. The addition of Umple code also means the program can be viewed as UML diagrams. Therefore, Umple increases program comprehensibility. Existing tools tend to separate diagrams from code, and reverse engineering to construct diagrams is often

only approximate. A UML diagram of an Umple system is guaranteed to be complete since there is always a direct mapping from the Umple to the UML.

Using a textual form of Umple allows users to develop the essential structures of a system more quickly than with drawing tools because the text looks like programming language code and can be typed and edited very rapidly. Moreover, when using Umple's web-based tool (UmpleOnline) to edit the Umple text, the changes are immediately reflected as changes to the UML diagram. The user can also edit the UML diagram itself, and the change appears immediately in the textual code (Lethbridge, Forward & Badreddin, 2012).

Umple can work with existing libraries in several programming languages such as Java, C++, Ruby and PHP. Users can extend existing classes even if they cannot see the source code. Also, the user can write arbitrary code to call APIs of existing libraries, as well as convert code written in any languages to Umple.

In addition, Umple supports generating readable, clean, and understandable state-of-the-art code. It also supports the generation of several different high-quality programming languages at the same time by maintaining language-independent files for the models and linking them to language-specific code for algorithms.

One of the Umple guiding principles is that it has to be able to implement other software patterns and other programming idioms besides UML concepts, such as generating code for a singleton, immutable or delegation pattern. This has the advantage of increasing the abstraction level of Umple (Lethbridge, Forward & Badreddin, 2012).

Umple eliminates the need for writing boilerplate code (e.g., for constructs, patterns, attributes and associations) because the Umple compiler generates such code. This leads to reducing the number of lines of code to be written in Umple, resulting in a more readable system so that users can better focus on the logical issues and critical parts of their code, rather than the low-level technical problems. Also, eliminating the boilerplate code reduces the likelihood of introducing bugs into the system by enabling the user to write bug-free code for complex model constructs such as associations and

state machines in a consistent way.

Umple also eliminates the need for a 'round-trip' process. In Umple, the generated code should not be edited or changed. If the generated code is not the type of code the user wants, the user can embed traditional code into the Umple text, or they can use Umple's aspect-oriented capabilities, which act to enforce constraints or change the semantics of the program. Umple hence eliminates round-tripping, although it supports code-to-model transformation by converting an existing system to Umple (this Umplification process is still under development).

Umple bridges the gap between modeling languages and programming languages. That is, it provides model-code duality. One of Umple's main philosophies is that it merges code and UML models. Umple allows developers to write a model-only file, a code-only file for a target language such as Java or C++, or they can mix a model with target language code in one file.

Umple provides synchronization between diagrams and corresponding text to represent the same underlying model, which allows for interchangeable manipulation. Also, it illustrates the strong relationship between a modeled system and its underlying implementation (Forward, 2010).

A user can incrementally refactor (umplify) a program in a base language that will pass through the Umple compiler unchanged by getting rid of complex code (for constructs like associations, state machines and patterns) and replacing it with simple Umple code. The user can do this little by little, testing at each step (Lethbridge, Forward & Badreddin, 2010).

Figure 2.2 shows that model-to-code transformation is possible with no round-tripping. Code-to-model transformation is also allowed, which is called "umplification" (Almaghthawi, 2013). Umplification process is currently developed by Miguel Garzon (PhD student working on the Umple Project).

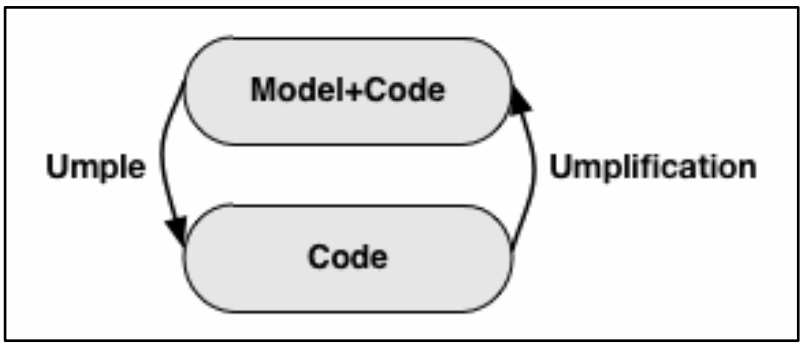


Figure 2.2: Umple model-oriented programming by Almaghthawi, 2013

As a result of all the above, Umple reduces the complexity of software engineering in such a way that it allows a developer to use a model-oriented programming approach to develop and maintain a system rapidly and at a high level of abstraction and hence to reduce the development time. The more Umple code is used, the more compatible, reliable and maintainable is the resulting program.

Lethbridge (2013) states that Umple has a number of properties including usability, scalability, and completeness that make Umple superior and more useful and simpler than other modeling tools.

2.5 Umple Features and Properties

The following are the core features that can be added to systems written in Umple:

1. **UML classes:** A user can define an object-oriented class, which is used as a type in a system (CRuiSE, 2013) (Almaghthawi, 2013).
2. **UML attributes:** An attribute in Umple is an entity that stores simple data (in contrast to an association, which stores data consisting of relationships). An attribute value can be accessed, modified or constrained by using specific methods associated with these attributes. Attributes in Umple are more abstract than fields in Java (CRuiSE, 2013) (Almaghthawi, 2013).
3. **Constraints:** Umple can support both basic OCL-type constraints and preconditions (CRuiSE, 2013) (Almaghthawi, 2013).
4. **UML associations:** An association is defined as the specification of a set of links

between classes as a language primitive. It allows for a large amount of code for the association to be generated in any of the supported programming languages (Java, C++, PHP, etc.). Umple represents associations in one of two ways: in-line (in one of the associated classes) or independently (outside either class). It generates specialized code for reflexive associations and association classes (CRuiSE, 2013) (Almaghthawi, 2013).

5. **UML state machines**: Umple allows an unlimited number of state machines in a class, and an unlimited amount of state nesting. It allows guards written in the base language, as well as entry, exit and transition actions. Do activities trigger concurrent threads that can execute until an event arrives resulting in termination of a state. Since state machines are the focus of this thesis, we will discuss them more extensively later in section 3.1.11.
6. **Software patterns:** Immutable, delegation and singleton are supported as language primitives.
7. **Base-language methods:** Umple enables methods written in a base language to be embedded into the Umple program and passed to the compiler without being changed or modified while the rest of the code is generated by Umple (Lethbridge, 2013). In fact, separate bodies for the same method can be given for each target language.
8. **Aspect-oriented code injection:** Umple allows the use of before/after statements to enforce conditions on attributes, associations, constructors and the components of a state machine: A user or developer has the ability to inject code before or after any method such as the event methods of the state machine (as the state machine event methods are generated automatically from the state machine) (CRuiSE, 2013) (Almaghthawi, 2013). The purpose of using this technique is that sometimes a user or a developer wants some code to be executed when processing an event by a state machine regardless of what state the state machine is in.
9. **Mixins as a reuse mechanism:** This allows code that is developed independently to be injected into a set of classes. It also allows a user or a developer to compose a state machine from many separate files. Since Umple supports that every class can

have an unbounded number of state machines where each of them can be defined independently, an event in one state machine can trigger transitions in one or more state machines. Also, the actions and guards can be defined independently as simple functions that can be reused across a number of state machines or across classes and components (Badreddin, 2010). In addition, the state machine mixins can be achieved by defining a stand-alone state machine and then call the state machine inside a class.

10. **Tracing:** Umple supports a tracing language called Model Oriented Tracing Language (MOTL) that allows tracing and understanding the behavior of the systems developed in Umple to debug, monitor, and analyze them (Aljamaan et al., 2014). The advantage of using MOTL is to give a developer the ability to trace different UML entities using trace directives without making any change to the generated code (Aljamaan et al., 2014). It is a part of the Umple project that is developed by Hamoud Aljamaan (PhD student working on Umple Project). In our thesis, we have been able to benefit from the ability of MOTL to trace different Umple state machines. It is possible now to trace a whole state machine in Umple by specifying a trace directive with the name of a state machine which results in tracing all states of this state machine with any events and transitions (Aljamaan et al., 2014). In addition, the developer can trace a whole state machine that can be nested at several levels of depth or trace specific levels of substates. It is possible to trace different elements of a state machine specifically as a developer determines. For example, a developer can trace a certain state, transition, event, as well as specific attributes that can be constrained to occur in a specific state (Aljamaan et al., 2014).

Umple also supports importing and exporting of code to other forms of UML such as TextUML and Papyrus XMI (Almaghthawi, 2013).

Figure 2.3 gives a small sample of Umple. This shows a state diagram drawn by the UmpleOnline tool and the corresponding textual Umple form. This state machine is called 'bulb' and it has two states: 'On' and 'Off'. The state machine has one transition causing the state machine to transition from 'On' to 'Off' if the event 'push' is triggered.

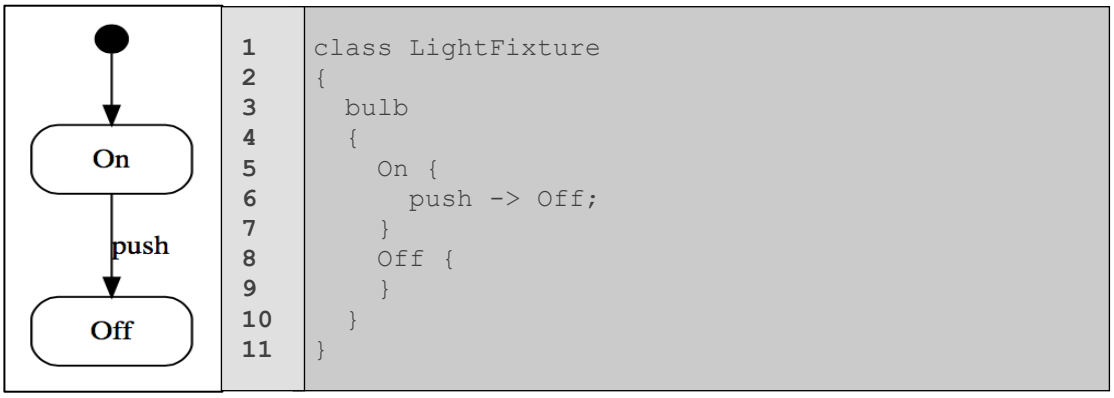


Figure 2.3: State diagram and Umple code as shown in UmpleOnline

2.6 Umple Tools

This section gives an overview of the current Umple development tools that are available to support viewing, editing and compiling Umple files to create an Umple model or system.

The key Umple tools are UmpleOnline, the command-line-based compiler, and an Eclipse Plugin, each of which was created for a specific purpose.

Figure 2.4 illustrates how to use UmpleOnline to write or load a model that in turns shows both textual and graphical representations of the same model.

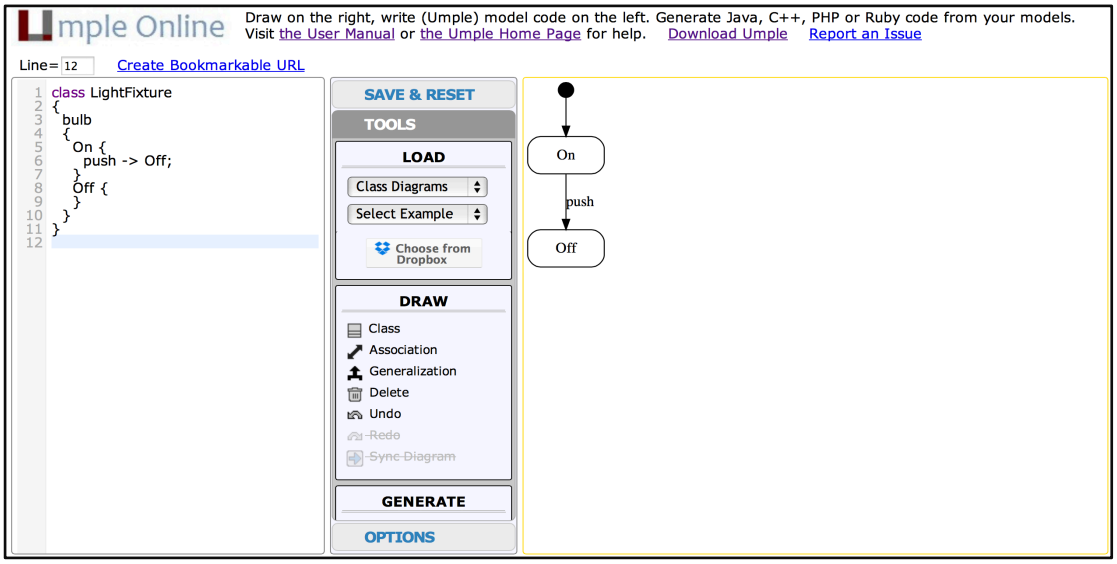


Figure 2.4: Textual and visual views of UmpleOnline

A description of all three tools is provided below.

2.6.1 Umple Command-Line Compiler

The Umple language can always be used from the command line. The Umple command-line compiler is preferred by traditional programmers who can use it to compile Umple files and generate the target code (Lethbridge, 2013). This tool is available as a Jar file and requires only an up-to-date installation of JVM Java 7 (Forward & Lethbridge, 2014).

An Umple file can be compiled using the command line by running this command:

```
java –jar umple.jar *.ump
```

Where * indicates the name of the Umple file and .ump indicates that code is written in Umple. As in a standard compiler, several files can be specified on the command line if desired.

Running this command compiles the Umple file and returns a notification message. If it compiles successfully, the message will be:

```
*.ump

Success! Processed *.ump.
```

If the compiler fails to compile the file, an error message is produced indicating the line on which the error occurred. For instance, to compile the Umple file saved as "LightFixture.ump" that is shown in Figure 2.4 using the Umple command-line compiler, we would run this command:

```
java –jar umple.jar LightFixture.ump
```

However, suppose there is a mistake in a state machine's syntax, such as forgetting to put a semicolon after the state "Off" in line 8 (push -> Off).

In tis case, the compiler will still compile the program but it will consider the state machine's code as 'extra code' and a warning message will be presented to indicate that there is likely something wrong at a specific line number. For instance, the warning notification in the above case is as follows:

```
LightFixture.ump

Warning 1006 on line 8 of file " LightFixture.ump":

State machine syntax could not be processed. It has been
considered as Extra Code

Success! Processed LightFixture.ump.
```

'Extra code' means that Umple believes the construct might be syntax from the base language (Java, C++ etc.), so it passes it through to the output files.

Suppose the error made is forgetting to put a closing curly bracket '}' at the end of the program. In that case, the compiler will fail to compile the program and an error message will be displayed showing the specific line number and file where the error occurs:

```
LightFixture.ump

Error 1502 on line 3 of file " LightFixture.ump":

Parsing error: Structure of 'class' invalid

Processed LightFixture.ump.
```

2.6.2 Umple as an Eclipse Plugin

Umple is available as an Eclipse Plugin that can be used like any other compiler in Eclipse and can be merged with other Eclipse-based modeling tools (Lethbridge, 2013). This gives developers the full power of the Eclipse environment while using Umple (Lethbridge et al., 2014).

2.6.3 UmpleOnline

UmpleOnline is a web-based tool that supports interactive editing of Umple both graphically and textually, ensuring that UML diagrams are kept synchronized with the Umple text (CRuiSE, 2013). In addition, it allows generation of high-quality code from Umple directly in a web browser to different programming languages and to various artifacts supported by Umple.

The center pane of UmpleOnline allows users to use Umple to load and save their models or Umple code (their work). It also enables them to explore various examples of Umple for models of different systems.

Furthermore, UmpleOnline allows users to type any Umple code into its left textual pane. Three seconds after the user stops typing in the pane, Umple will interpret the code and draw a corresponding class diagram in the right-hand graphical pane (Forward & Lethbridge, 2014). The text editor provides "syntax-assist," which helps users keep their code indented correctly as they type Umple text by highlighting matching parentheses and Umple keywords.

The right-hand graphical pane of UmpleOnline allows users to add a class on the canvas. They can also add attributes, generalizations, and associations. Several diagram formats are available including automatically-laid-out class diagrams, editable class diagrams, and automatically-laid-out state diagrams.

The bottom pane of UmpleOnline has a button to generate code and many other types of output. It shows the errors and warnings that may appear at the bottom of the page as users edit their text or diagrams (Almaghthawi, 2013).

UmpleOnline was created for several purposes. It is an effective tool used for educational purposes. Educators and students are not required to install UmpleOnline; they can access it if they have a browser and an Internet connection. In addition, they can install a local version of UmpleOnline and manipulate Umple files on their computers through a web browser (Almaghthawi, 2013). It has a variety of examples of complex models for different systems. In addition, they can create their own examples and models, save them on the server and then reload them later using bookmarks (Almaghthawi, 2013). Because of these features, it is recommended that software educators use UmpleOnline as a tool for teaching and demonstrating various UML design alternatives, and for showing the implication those designs have on the generated code of those alternatives (Forward, 2010).

In addition, UmpleOnline is considered to be a good demonstration tool. Once a model is created using UmpleOnline, one can see a view of both the visual and textual

representations of the same model (Forward, 2010). It can also be used to explore ideas quickly, giving the developer the ability to create rapidly and simulate basic models (Forward, 2010). A developer can also create a complicated model of a real case study using UmpleOnline, and initiate small projects (Lethbridge, 2013). Besides, UmpleOnline is an excellent tool for preparing professional-looking diagrams ready for publication (Forward & Lethbridge, 2014).

2.7 Umple Architecture

The Umple compiler acts like any other programming language compiler; it is tested and built in the same manner. In fact, the Umple compiler was written in Java and then rewritten in Umple itself (Forward, 2010). There are few tools that build their own compilers using their own languages; one example is the Pascal compiler, which since the mid-1970 has been written in Pascal (Cantu, 2008). In 1962, Hart and MIT developed the first self-hosting compiler for Lisp in Lisp itself (Hart & Levin, 1962).

Writing a compiler in the same language that is developed shows that this language can provide utilities that may not be found in the "base" language of the compiler and these features are well-suited for the compiler. For instance, most behaviors can be implemented using the abstract level of the Umple language where UML constructs can be used as well as other patterns, etc. It can be much easier to maintain the compiler when the same language is properly used demonstrating the usefulness of the tool. Moreover, writing the Umple compiler in itself helps the developer to customize and extend the language using high-level abstraction constructs.

The Umple compiler consists of several well-defined internal components including an Umple parser, which is the analyzer that helps the Umple compiler to parse the input Umple code into tokens, the Umple metamodel classes, which are the internal representation of Umple in which the tokens are used to populate and generate instances of the Umple metamodel from the parse tree, as well as several code generators used to generate various target base languages such as java, C++, PHP, and Ruby. Also, model-to-model transformation engines are used for generating the models of the systems built using Umple, and a synchronization engine that allows the

diagrams to be edited and then the resulting changes to apply to the text (Forward, 2010).

Figure 2.5 below illustrates the high-level components of Umple model processing.

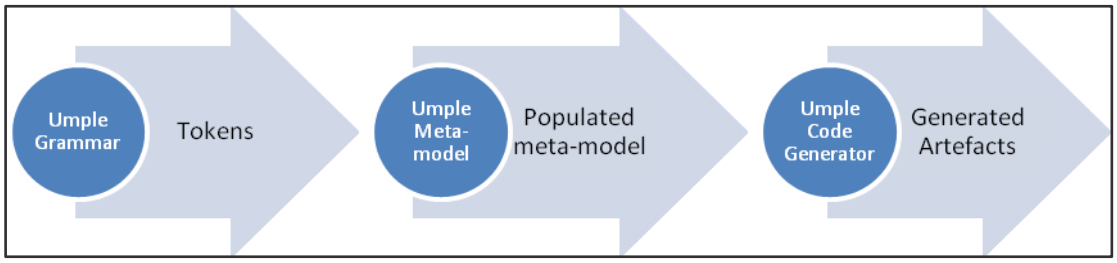


Figure 2.5: Umple well-defined components as shown by Badreddin, 2010

2.8 Development of Umple

The Umple language and processing tools were developed following a combination of agile, continuous integration, model-driven development, and test-driven development approaches to ensure that it provides high quality and flexibility.

Umple was developed in small iterations and increments that are compiled and frequently integrated. In addition, model-driven development is used to allow adding UML constructs to base languages besides generating high-quality object-oriented programming languages and other diagrams and artifacts. A test-driven development approach enables testing those components at multiple levels so that a fully tested and functional system is obtained.

2.8.1 Agile Development

Umple is developed using an agile method; a software development methodology that depends on iterative and incremental development combined with continuous integration and test-driven development approaches. The agile method has the idea of developing a system through repeated cycles and in small portions at a time, allowing developers to benefit from feedback during development, especially in earlier versions of the system (Dingsøyr, Dybå & Brede Moe, 2010). Design modifications are made, and new functionalities are added at each iteration of the development.

2.8.1.1 Agile Philosophy

The agile method is adaptive rather than predictive which means that the agile approach allows for a quick adapting to changes in requirements, and solutions evolve across collaboration between teams (Bakal, Althouse & Verma, 2012). It is also people-oriented rather than process-oriented where the role of the process is to support the development teams in their work not building their skills (Bakal, Althouse & Verma, 2012). In addition, the agile development method is based on an iterative approach in which the development tasks are broken in small portions with minimal planning. Each development task is allocated to a short timeframe called an iteration or sprint, and each iteration involves working in all stages: requirement analysis, design, coding, unit testing, and acceptance testing. The testing in an agile manner is usually done in parallel with coding or starting in early iterations of development life cycle, not in the last phase of the development life cycle as in the waterfall manner. In contrast, the waterfall model follows a sequential design process in which a separate testing phase is done upon the completion of the implementation phase (Dingsøyr, Dybå & Brede Moe, 2010).

In addition, the agile development methodology is an incremental approach in which each rapid iteration results in producing small software release (Dingsøyr, Dybå & Brede Moe, 2010).

The aim of the agile development methods is to enhance the quality of the system that can be achieved by dividing the development tasks into small units. So the team can produce frequent builds and perform intensive testing during each iteration which results in detecting and fixing bugs in a rapid manner. Therefore, the agile method focuses on high-quality development, testing, and collaboration between teams (Dingsøyr, Dybå & Brede Moe, 2010).

Continuous integration and test-driven development are two agile practices and tools that cover various areas such as requirements, design, modeling, coding, testing, etc. are used to improve the quality of systems and enhance the system agility (Bakal, Althouse & Verma, 2012).

2.8.1.2 Advantages of Agile Methodology

It reduces the overall risk and allow for rapid and flexible response to changes, which means that it makes it possible to add new features or implement new changes to the system because of the frequency of new increments produced. It also encourages teams to have face-to-face communication to discuss progress and get rapid feedback of what they have done, what they are doing, and what they will do in the future. Also, they discuss the features to be newly added or removed and the changes to be implemented based on changing requirements (Dingsøyr, Dybå & Brede Moe, 2010).

Further, the methodology of agile development results in a high-quality system with minimal bugs and defects because the system is tested intensively during the iterations of the development life cycle, which is achieved in least possible time. The agile method is more about coding rather than documentation (Dingsøyr, Dybå & Brede Moe, 2010).

2.8.2 Continuous Integration

Continuous Integration (CI) is a software development practice created for agile development that requires members of the team to integrate their changes to a larger code base frequently, often multiple integrations per day. These integrations are then verified through an automated build and testing to detect integration errors (Bakal, Althouse & Verma, 2012).

The goal of continuous integration is to reduce the risk of having an extended and difficult integration. This helps to get rapid feedback as the automated build for verifying the integrations helps in detecting integration errors as quickly as possible so that, if a defect is detected in the code base, it can be rapidly identified and corrected. The detecting of defects early in development leads to easily resolution of smaller and less complex defects (Rouse, 2008). Therefore, continuous integration helps reduce the problems of integration and enables development team to develop cohesive software more quickly (Bakal, Althouse & Verma, 2012).

2.8.3 Model-Driven Development (MDD) Methodology

Model-Driven Development (MDD) is an approach in which models are used as

specifications of software system in which the essential aspects of software are expressed. The models are then transformed in order to get corresponding source code. The transformation of the models constitutes the core of software development (Beydeda & Book, 2005). For example, a model transformation can facilitate the process that converts between different views of the system at an equivalent level of abstraction, or converts models between levels of abstraction, usually from a more abstract to less abstract view by adding more details supplied by the transformation rules (Beydeda & Book, 2005). The word 'driven' in MDD indicates that this approach enables models to be given a central and active role because they are at least as important as source code (Völter et al., 2013).

MDD is a less precise but common name for the discipline called model-driven software development (MDSD) (Völter et al., 2013).

Mellor et al. (2003) point out that the main idea behind MDD is to model a system at several layers of abstractions and from different views (as cited in Parviainen et al., 2009, p. 10). The models that are created for the system become the main artifacts of software development. By using generators or executing the created models at run-time, these models are transformed into running systems.

Models, modeling, and model-driven architecture (MDA) are considered the three basic concepts of the MDD approaches (Beydeda & Book, 2005).

Creating models for a software system is useful to identify a problem domain and design a solution in the solution domain. MDD models are abstract and formal at the same time. They have an exact meaning like programming code (Völter et al., 2013).

In addition, identifying relationships between the created models offers a network of dependencies that mark the process in which a solution is created, and also it assists in recognizing the implications of modifications at any point in that process (Beydeda & Book, 2005).

In MDD, the created models do not create documentation for software but they are considered to be part of the software, and they are equal to source code because of

the automation of their implementation (Völter et al., 2013). Moreover, defining a set of rules is required in order to automate many steps needed to transform one model representation to another, to trace between model elements, and to analyze essential features of the models (Beydeda & Book, 2005).

MDD targets finding domain-specific abstractions and enables accessing them through formal modeling (Völter et al., 2013). Völter et al. (2013) illustrate that there are three requirements that are required to be achieved in order to apply the "domain-specific model' concept; they are:

- Domain-specific languages are needed to enable the actual defining of models.
- Model-to-code transformation languages are needed.
- In order to run the transformations to generate executable code on various platforms, compilers, generators or transformers are needed.

By using models to represent the software and visualize the code and the problem domain, MDD intends to speed up the software development and makes it more cost efficient. MacDonald et al. (2005) state that MDD also aims at separating implementation technology from the business logic of the program (as cited in Parviainen et al., 2009, p. 10).

MDD implies more precise and clearer views of aspects while dealing with software development paradigm. It gives a clear description of the meaning of models, the separation of the domain-specific and technical implementation, the relationship between design and implementation, round-trip problems, architecture and generation, versioning and tests that if it is applied correctly, it will make the work of the developer much easier, and it will help avoid redundant code and improve software quality by using formalized structures (Völter et al., 2013).

According to Duby (2003), Object Management Group's (OMG) Model-Driven Architecture (MDA) is referred as software systems modeling, and it is a well-known example and standardization initiative of OMG focusing on MDD (as cited in Parviainen et al., 2009, p. 10). OMG works as an industry-driven consortium to develop standards for the implementation of MDD. MDA depends on a set of emerging standards to define

a set of models, notations and transformation rules (Beydeda & Book, 2005). The idea behind developing MDA was to allow the specification of system functionality (i.e. the processing and logic of the system) to be separated from the specification of its implementation technology. It then enables developers to focus on solving the problem rather than the dealing with the details of the implementation technology (as cited in Parviainen et al., 2009, p. 10). MDA as a way of modeling has several advantages because it enhances the quality, efficiency, and predictability of software development.

France & Rumpe (2007) state that MDA demonstrates the modeling of the system from three viewpoints (as cited in Parviainen et al., 2009, p. 11). The computation independent viewpoint is about the required features of the system, as well as the environment in which it operates. It results in computation-independent models (CIM). The platform independent viewpoint is concerned with the features of the system that are fixed and do not change as the system is used on different platforms. It specifies what the system does, and it results in platform-independent models (PIM). The integration of PIMs with the platform-specific details in the platform-specific viewpoint results in platform-specific models (PSM) that describe how the system is implemented.

2.8.3.1 Advantages of Model-Driven Development (MDD)

Mellor et al. (2003) argue that models can be used to increase productivity because building a graphical model using Unified Modeling Language (UML) is cheaper than writing functional code in Java, for example; however, using models may raise the degree of obstacles (as cited in Parviainen et al., 2009, p. 11). The list below shows that adopting MDD has several advantages.

According to Völter et al. (2013), MDD enhances the quality of software and helps improve the manageability of software complexity.

Regarding Umple, it supports MDD by allowing the developer to present and maintain a complex system as a model either textually or graphically, and then generate high-quality code from that system (Lethbridge, 2013).

2.8.4 Test-Driven Development (TDD) Methodology

Test-driven development (TDD) is an agile software development process that consists of a set of short development cycles or iterations (Patrick, 2006). It has been widely used within the agile process. It is a well-known development process used for open-source software projects; particularly ones concerned with adaptation of agile principles for continuous integrations of the system development which results in more manageable and understandable code (McConnell, 2014). This process is performed in short, and rapid iterations (Janzen & Saiedian, 2005).

The TDD approach results in enhancing the quality of the design and makes it easier to work with it in the future. In fact, TDD helps drive the design specification as well as directly validating it (Ambler, 2012).

The TDD process requires writing a set of automated tests, which, when first written, are expected to fail. The system is then changed so they pass. The automated tests become an aspect of the system's specification and a primary component of the system documentation; they are executed with every subsequent system change, providing rapid feedback about any bugs or unexpected side-effects of subsequent changes to the system. (Janzen & Saiedian, 2005).

Test-driven development was developed initially by Kent Beck (2003), and it has become widely accepted as a methodology in the software development community in combination with agile process models and the Extreme Programming (XP). Both agile process and test-driven development (TDD) are considered as basic methods working on iterative, incremental and evolutionary basis in the modeling process. In 2002, Beck claimed that following TDD for developing a system leads to simpler designs and increases the level of confidence.

An automated unit testing framework for Java called JUnit was developed by Erich Gamma and Kent Beck to implement TDD with Java. Implementing JUnit-like frameworks for various languages leads to creating a family of frameworks called xUnit, which enables programmers to write numbers of automated unit tests in order to initialize, execute, and make assertions about the code needed to be tested (Janzen &

Saiedian, 2005). Writing xUnit automated tests in the same language as the code under test results in allowing these tests to serve as first-class clients of the code, while the tests themselves actually serve as specification and documentation for the code. The test order does not matter because individual tests are written to be independent of one another. In addition, xUnit frameworks allow programmers to report the total number of successes and failures (Janzen & Saiedian, 2005).

According to Beck (2002), there are two main rules driven from adopting and using the test-driven development process:

1. Writing new code to enhance the functionality of the system only if an automated test written for this code has failed, and
2. Refactoring both the production code and test code to remove any duplication that may occur (Beck, 2002).

These above two rules drives the TDD mantra that is ***"Red-Green-Refactor***." The ***"Red Green Refactor"*** mantra refers to a process that indicates that using the TDD methodology enables focusing on small steps for writing software. It ensures that the process ends up with high-level productive and confident system by enhancing the production code (Boydens, Cordemans & Steegmans, 2010).

The development micro-cycle of TDD is visually shown in figure below:

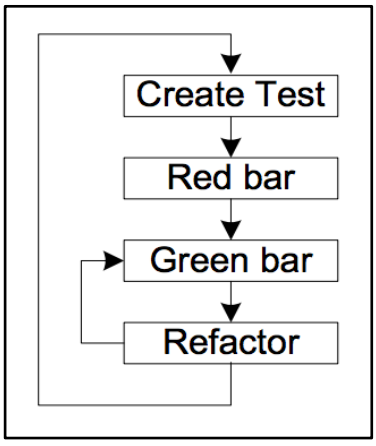


Figure 2.6: Test-driven development (TDD) cycle: Red-Green-Refactor by Boydens, Cordemans & Steegmans, 2010

Figure 2.7 illustrates steps that are driven by the above principal rules of TDD:

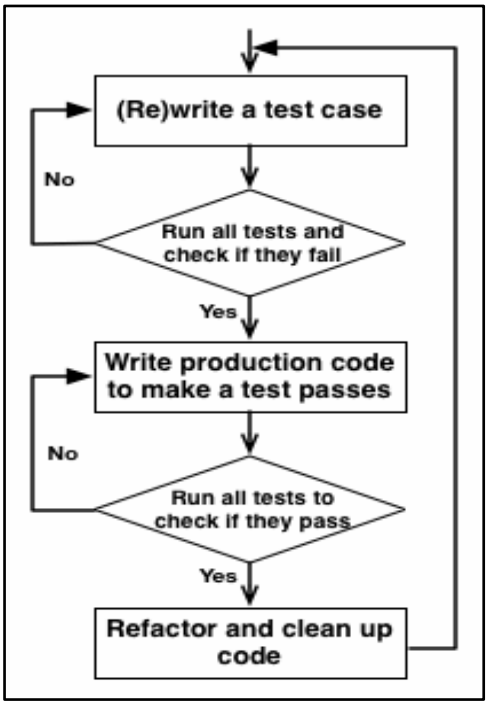


Figure 2.7: A graphical representation of the necessary steps for the test-driven development (TDD) cycle, using basic flowchart

2.8.4.1 Benefits of Test-Driven Development (TDD) Methodology

TDD is suitable for developing large systems where there are large numbers of developers working on developing and implementing the systems. This results in a large number of functional code lines and test code lines that would be running in a short time (Forward, Badreddin, Lethbridge & Solano, 2012).

In fact, TDD makes it easier for new developers to contribute to the software. This process supports adding a minimum code that is enough to make tests pass rather than writing a large number of lines for functional code which as a result can help detecting and resolving defects whenever they exist (Forward, Badreddin, Lethbridge & Solano, 2012).

In addition, TDD leads to improved code quality by enabling continuous regression testing (Beck, 2003). That is, automated unit tests will be run when the code is enhanced or maintained to detect and identify any new defects, as well as to control the uniformity of software releases (Beck, 2003).

To sum up, following TDD for developing and implementing software leads to improving quality, reducing debugging effort, promoting simplicity (Forward, Badreddin, Lethbridge & Solano, 2012), and enhancing flexibility of the software systems (Karai, 2009).

2.8.5 Benefits of Using a Combination of Agile, Continuous Integration (CI), and Test-Driven Development (TDD) Methodologies

The continuous integration (CI) approach provides some improvements over classic agile development. It allows development teams to be agile in response to rapid changes and at the same time it helps the teams to work effectively and efficiently in their domain. Whenever they have completed a change, their contributions are integrated, and components work well together. If there is a problem or if the component is not integrated correctly, tests created through test-driven development help discover and fix the problem quickly (Bakal, Althouse & Verma, 2012).

2.9 Building the Umple System Using a Test-Driven Development (TDD) Strategy

2.9.1 Umple Testing Infrastructure

The Umple development system includes several components; the Umple parser, the Umple metamodel, and several code generators used to generate various target base languages such as java, C++, PHP, and Ruby (Forward, 2010).

Currently, there are more than 3445 test cases that cover testing all the components of the Umple infrastructure, ensuring that they go through all the four levels of testing: parsing, metamodel, template, and semantic testing. A summary of tests can be seen at http://qa.umple.org

Therefore, testing the Umple infrastructure is done at several levels, which can be summarized as follows:

1. **Parser testing:** the testing process in Umple starts with this level to ensure that the Umple Abstract Syntax Tree (AST) is built correctly in order to be parsed into tokens (Almaghthawi, 2013).

2. **Metamodel testing:** the testing at this level is to verify that the metamodel of Umple is constructed from the Abstract Syntax Tree (AST), that is, the tokens are used to populate the metamodel properly (Almaghthawi, 2013).
3. **Template or code generation testing:** this testing is to check that the generated code such as Java, C++, PHP, and Ruby, is syntactically correct according to the languages' syntax, and it matches what is expected.
4. **Language-oriented semantic or testbed testing:** in this phase, the testing is to verify that the generated code behaves properly and correctly according to our expectations, and some other tool-oriented testing (Almaghthawi, 2013).

Essentially, in our work, we were interested in doing all types of testing to test the features we added to Umple state machines, which are queued state machines, pooled state machines and the unspecified reception handling mechanism (more details about these features are illustrated in Chapters 4 and 5). As we needed to modify the Umple grammar and Umple metamodel to accommodate the changes, we added a number of test cases to ensure our Umple code was parsed correctly and generated the tokens from adding the new keywords. Also, we added more test cases to ensure that the metamodel instance populated by processing these tokens was correct. Also, we wrote test cases to ensure that generated code from given Umple programs was valid, which in our case was Java code, and it was syntactically correct as we compared the expected code versus the actual code. Regarding these tests, we ensured that the syntactic translation of the Umple metamodel instance into the generated base language was correct. Finally, we added test cases to the Java testbed of Umple code in order to ensure that the generated code was semantically correct and behaves as expected. Therefore, this shows we went through all the levels of tests for our thesis.

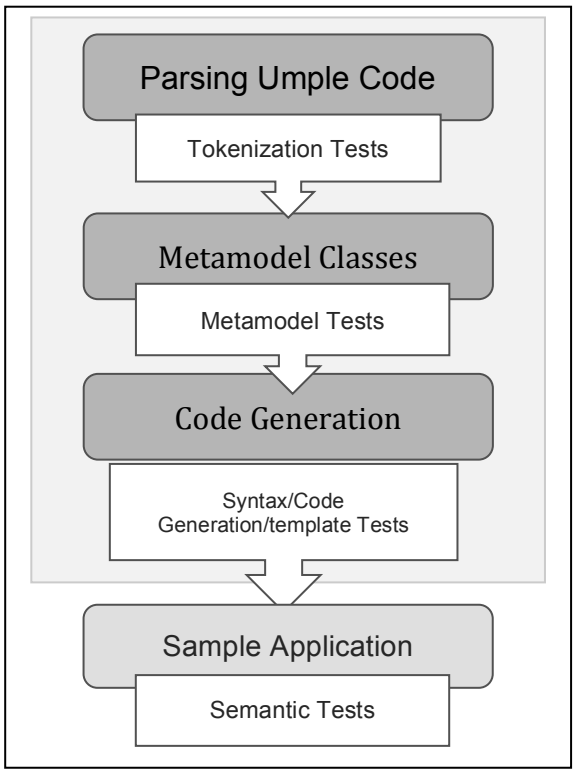


Figure 2.8: Umple testing infrastructure by Almaghthawi, 2013

Figure 2.8 shows the different levels of the Umple testing process, and it illustrates that the first three levels are done within the scope of Umple.

As illustrated in Figure 2.8, the first three levels of the Umple testing process are done within the scope of Umple, which means that we are only capable of testing Umple, not the set of possible systems built using Umple.

2.9.2 Test-Driven Development Methodology (TDD) and Umple

The development process of Umple follows an agile approach that prompts an iterative and incremental software development throughout the life-cycle development of the Umple infrastructure.

The earliest versions of the Umple compiler were written in Java, then the latest versions of Umple were written using Umple itself in a model-driven development fashion. Figure 2.9 shows an example of the continuous integration process of Umple that depends TDD (McConnell, 2014).

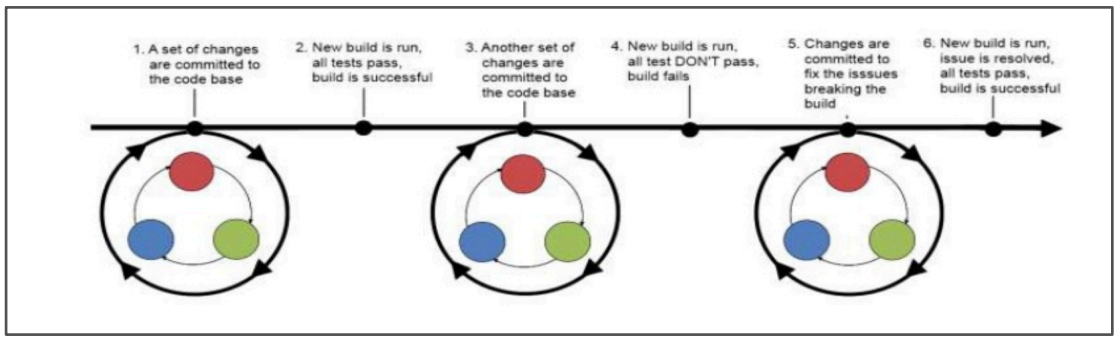


Figure 2.9: Umple continuous integration process relies on TDD by McConnell, 2014

As shown in the figure above, the development of Umple goes through several procedures. If an error occurs while a set of changes are committed to the source repository causing a build failure (some test does not pass), then immediate action must be taken by committing other changes to the source repository in order to fix the issues that break the build. Committing these changes should be made before committing or integrating any other changes to the repository. Then a new build is run automatically, hopefully showing that the build is successful, and all issues are resolved, as all tests pass (McConnell, 2014).

A brief description of how we can integrate new tests and make a change in Umple is listed in (Lethbridge, Forward & Badreddin, 2012).

All the automated tests in Umple must pass 100% after running the build. Once we run the tests automatically including a full build, we can see an automatically generated website with all the test results which enable us to find failing tests if there are. This results in assuring the quality of the code. This web page is located on the build server at: http://qa.umple.org (Lethbridge, Forward & Badreddin, 2012).

Unit Test Results.

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
3684	0	0	100.00%	40.659

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Figure 2.10: Umple automated testing report

Currently, there are over 3680 tests that cover all areas of the testing process of Umple. The automated quality process is used to run these tests to ensure they all pass and the model and code are consistent as illustrated in Figure 2.10.

Following the TDD approach to develop Umple brings many benefits to Umple. The process that follows TDD is not just about testing, but also about designing the system in a modular fashion. It is also about capturing the intention of the software or automated tests that can be easily verified by re-running the test suit so that little effort is wasted. It is much better than the act of manually testing and modifying or debugging an application until it works which will just benefits the developers (Forward, 2010). Capturing the testing process through automation enables all developers to benefit because the knowledge gained about true behavior of the system can be easily re-run and re-verified. In addition, we can capture the debugging effort of a new code generation behavior in our automated tests and then change the underlying Umple language to replicate that behavior natively because Umple is implemented in itself (Forward, 2010). Moreover, following TDD enables contributors to Umple to make sure that the code produced for implementing a new feature is reliable and maintainable, and also TDD makes it easier for new developers to contribute to Umple in the future (McConnell, 2014). Also, following this approach ensures a high quality to the Umple infrastructure since it ensures that 100% tests pass which means there are fewer errors, bugs, and a more enhanced system.

In addition, writing automated tests first during the development cycle allows developers of Umple to:

- Plan changes needed to implement a new feature or to enhance the existing feature prior to the actual implementation of this feature.
- Make sure that the changes produce the expected behavior.
- Thoroughly test the changes on the local environment of the developer before committing those changes to the source repository.
- Ensure that there is no side effect of those changes on other areas of the source code of Umple.

Chapter 3 Literature Review

In this chapter, we discuss several technologies and languages that are relevant to this thesis. We start by discussing communication protocols; since they are a major application of our work, and we discuss a variety of languages that can be used to represent them, including SDL, Spin, and UML state machines.

3.1 Protocols and Formal Description Techniques (FDTs)

Communication protocols (simply called protocols) are sets of conventions and rules used to govern and control communication within or between different entities and processes of a distributed system or computer network. They govern the meaning and format of messages, packets and data exchanged within or between entities of a system, either by peer entities within a layer or by entities between different layers of a distributed entities (Venkataram, 2014).

The description of a protocol has an important role in all stages of a protocol design (Bochmann & Sunshine, 1980). According to Bochmann (1990), due to its essential role in the development life cycle of a distributed system, protocol specification is developed in connection with communication services throughout the design of the system. As illustrated by Bochmann and Sunshine (1980), the protocol specification describes the internal structure of the protocol, which means that it describes the operation of each entity within a layer such as internally initiating actions such as timeouts, and also the response of inputs and messages from its users and the other entities through the lower layer service.

The protocol specification needs to be validated in order to derive large parts of code for all implementations of system components (Bochmann, 1990) that concentrate on the real coding of the protocol. According to Venkataram (2014), this can be achieved by following software engineering aspects. The protocol specification is also used as a reference during the implementation phase in order to select test cases for conformance testing and test result evaluation (Bochmann, 1990).

There are two ways for describing and defining communication protocols and distributed systems: informal and formal methods. The design and development of communication protocols using the informal or formal methods involves the

specification, validation, and implementation of the protocols (Venkataram & Manvi, 2005). Some informal methods are used to describe and represent the communication protocol such as representing the sequence of message exchanged using Message Sequence Chart (MSC) or a flowchart (Venkataram & Manvi, 2005).

As these informal methods have some drawbacks, several formal specification and description languages (FDTs) are used to overcome those disadvantages and to be used for the descriptions of distributed systems and communication protocols.

Due to the increase of complexity and variety arising from protocols that govern data communication in distributed systems, computer networks, and communication technologies, the communication protocols have become harder to design, understand, and analyze. Therefore, formal description techniques (FDTs) such as SDL, Estelle, and LOTOS have been developed for different reasons, and they have been applied in various domains. They were used for developing and designing the description of distributed systems and communication protocols to handle that complexity (Bochmann, 1990, p.167).

Venkataram and Manvi (2005) defined the formal description techniques (FDTs) as methods that are used to facilitate the protocols design and implementation in a quick and sufficient way. The formal description techniques (FDTs) are mathematically based techniques that introduce strictness and reliability into the different steps of the protocol development process (Babich & Deotto, 2002).

According to Babich and Deotto (2002), the purpose of using the formal description techniques (FDTs) is to simplify the process of designing, validating, and implementing the development process of communication protocols. They also aim to speed up the process of developing a communication protocol and to ensure the final implementation is consistent and conforms to its specification. FDTs show how they can be used in conveniently managing the development process of communication protocols that results in producing highly reliable software.

FDTs do not replace the traditional informal methods for developing the communication protocols, instead, they integrate with them with deeper understanding of the system's behavior, early detecting and handling of error, clear documentation of the process, and mathematical correctness proofs for high-integrity systems (Babich &

Deotto, 2002).

FDTs are based on two basic approaches. They can depend on some theoretical models or they can be specified by using high level programming language (Venkataram, 2014). Also, a hybrid models can be used for designing and developing of communication protocol.

The various description techniques used to define different behavior features described above are state transition models including finite state machines (FSM), communicating finite state machines (CFSM), and Petri nets, formal grammars, and algebraic calculi which are used for describing the specified behavior of the communication protocol and distributed systems (Bochmann, 1990). The other languages used for the same purpose besides describing the structure of the protocol specifications are high-level programming languages, abstract data types, and temporal logic (Bochmann, 1990). In addition, hybrid models such as extended finite state machine (EFSM) can be used, which are defined as different extensions added to some of the formal specification languages such as FSM by combining them with high-level programming languages or abstract data type to describe parameter values (Bochmann, 1990). Moreover, there are some language standards used for the design and implementation of communication protocols such as specification and description language (SDL) (based on FSM combined with more extensions), UML statecharts (an object-based variant of Harel statechart (Harel, 1987) that is extended by UML) (OMG, 2013), Estelle (based on EFSM combined with extended Pascal), and LOTOS (based on Calculus of Communicating systems CCS) (Bochmann, 1990).

3.2 Finite State Machine (FSM)

Finite state machine (FSM) is a powerful and simple model that captures the essential behavior of a large set of systems such as communication protocols and control systems (Bochmann, 1978). FSMs have also been used for modeling reactive systems, hardware digital systems, software engineering, compilers, network protocols, and study of computation and languages (Harel, 1987). Bochmann (1978) provided a systematic analysis of the behavior of two FSMs communicating with each other. One of the possible communication mechanisms is “direct coupling” where each component

has specific types of transitions that are directly coupled with transition types in the other component in the system (Bochmann, 1978).

The other communication mechanism described by Brand and Zafiropulo (1983) is that the processes can be modeled explicitly using FSMs, and the channels can be modeled implicitly using queues that have unbounded capacity that allows for an arbitrary messages in transit.

There are two categories: Mealy and Moore FSMs. They are theoretically similar but practically have different characteristics ("Finite State Machines," 2001).

Each transition of a Mealy machine has an input interaction and an output interaction, and its current state and current input interaction uniquely determine the value of its output interaction (Mealy, 1955). In other words, when the machine is in the state, and an input interaction (event) arrives that matches the input interaction labeled on a transition, then this transition will be executed. It causes the machine to transition to a new state, and the output interaction (which later came in UML to be called the transition action) will be produced. The output becomes the input to another FSM (i.e. the output of the machine is a signal/event) (Bochmann, 2008).

In a Moore machine, its output interaction (which came to be called in UML the do action) is based only on the state rather than the transition, which means that the output is not produced during the transition (Moore, 1956). Instead, the Moore machine produces a continuous output (i.e. a continuous value) depending on the state (Bochmann, 2008).

Unified Modeling Language (UML) and Umple can handle both machines because they can have actions associated with transitions and do-actions in states that may produce output values.

3.2.1 Advantages of Finite State Machine (FSM)

The importance of using FSM model is that it describes a protocol as a state machine at a low level of abstraction that can be easily understood (Holzmann, 1991). Also, using FSM to specify the protocol means that each process in the system makes about the others.

The finite state machine (FSM) model is an easy-to-use graphical language, which makes it an effective and convenient model ("Finite State Machines," 2001). Additionally, the finite state machine (FSM) allows for defining all external and internal events that the system may respond to which means it spends more time on designing before coding (Bollig, 2006). This allows for easily modifying a program if there is any need to do so such as adding new features to its states (Bollig, 2006). Also, it allows for easily testing a program because the testing can be broken down into small tests for each state, which leads to high-quality system (Bollig, 2006). Besides, the finite state machine (FSM) can be used to model concurrency in a system (i.e. each FSM describes a concurrent component of the system) ("Finite State Machines," 2001).

Modeling by a finite state machine approach is used to check specification and validation of communication protocols (Bochmann, 1978). Using a formal specification method is effective for the validation, testing, and implementation of such protocols (Bochmann, Gerber & Serre, 1987). This process allows for a formal analysis of protocols before their implementation. It verifies that the protocol specification realizes the service specification and checks that the protocol specification is free from design errors such as deadlocks and unspecified receptions.

In order to verify the design of two or more state machines communicating by asynchronous messages passing, a global behavior is driven from the behavior of these communicating state machines. This is called reachability analysis- it determines the global behavior and identifies all global system states that are reachable from the initial state of the system. Automated tools, such as Spin have been used for this purpose.

During reachability analysis, the following design errors may be detected:

- Unspecified receptions
- Deadlocks: all processes wait for a message while none is in transit.
- Non-executable transitions and/or states: transitions that will never be executed, or states that will never be reached.

3.2.2 Representation of Finite State Machine (FSM)

A finite state machine (FSM) is represented in some different ways that are considered to be useful for various purposes. One of these ways is a transition table,

which is used to describe some algorithms and systems operating on finite state machines (Saunders, Coulson, and Folse, 1992). The rows of the transition table correspond to the states of the machine, and the columns correspond to the input symbols that may occur. The entries in the table are the next states that the machine enters and receives the given inputs. In addition, the table represents the output symbols corresponding to the transitions of the transition-assigned (Mealy) machine (Saunders, Coulson, and Folse, 1992).

Even though the transition table is practical for describing some aspects of the system, the visualization of a finite state machine is more human-oriented representation (Saunders, Coulson, and Folse, 1992). A representation named the state transition diagram is a labeled graph that visualizes and depicts a state machine by representing its states as nodes and its transitions as arcs. In the state transition diagram, each arc is labeled with an input symbol that causes the transition. It may also be labeled with an output symbol that is produced by the transition given that the machine is transition-assigned (Mealy) machine (Saunders, Coulson, and Folse, 1992).

To illustrate, the state transition diagram shown in Figure 3.1 represents a transition-assigned (Mealy) machine (Saunders, Coulson, and Folse, 1992). It has four states (q0, q1, q2, q3), two input symbols (0, 1), and four output symbols (0, 1, 2, 3).

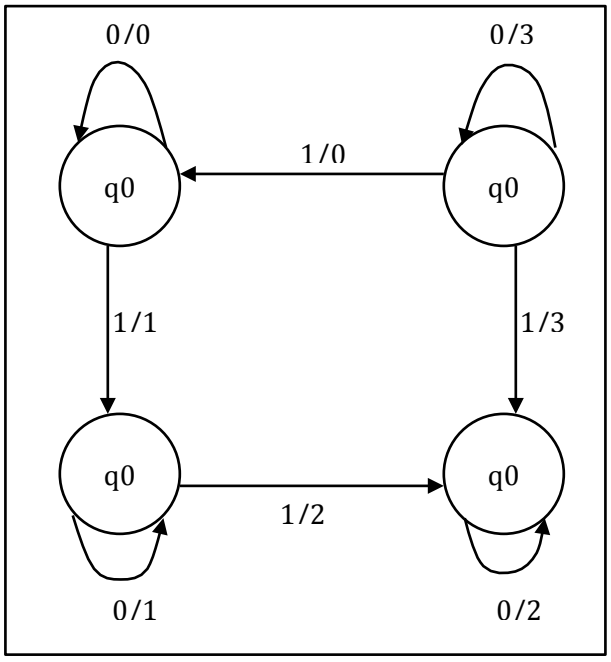


Figure 3.1: State transition diagram as shown in UmpleOnline

The transition table shown in Table 3.1 represents the state machine depicted in Figure 3.1. The columns of the table represent the input symbols, and the rows represent the states.

Table 3.1: A transition table

	0	1
q0	q0/0	q1/1
q1	q1/1	q2/2
q2	q2/2	q3/3
q3	q3/3	q0/0

3.2.3 Extended Finite State Machine (EFSM)

As a simple state machine provides limited features that may not be suitable for complex models, an extended finite state machine (EFSM) is introduced to allow more formal concepts and features (Bochmann, 2013).

As explained by Bochmann (1978), the two main limitations of a finite state approach are the explosion of state space and the inheritance in the finite state modeling of the transmission medium. The first problem can be resolved by using some validation tools. The second problem is useful only if a small number of messages exist in transit. The proposed solutions for these limitations as explained by (Bochmann, 1978) is to use the proposed method that combines both a finite state machine model and the assertions of high-level programming language. That allows for an extension to a finite state machine by using regular expressions to model message queues without enforcing a limit on the number of messages in transit.

One of these concepts is to allow state machine input and output to have parameters; thus, an event can be a signal, a change in some condition, or the passage of time (Hogrefe, 2013).

Several specification languages based on the model of EFSM have been developed. For example, Estelle is based on the EFSM model that shows a system consisting of several interconnected components; each is represented as a module communicates via input/output interactions with other modules in the system (Budkowski, & Dembinski, 1987).

3.2.4 Hierarchal Finite State Machine (HFSM)

Harel (1987) provided extensions to the conventional state machines and state transition diagrams: hierarchy, and concurrency, which can be considered a shorthand notation. These extensions called hierarchal finite state machine (HFSM) or Statecharts (Harel, 1987) can be used to describe complex system behaviors in a compositional and modular manner.

Due to the fact that some systems have a very large set of states and transitions, using a flat and sequential FSM should be avoided because it becomes hard to represent and analyze those systems (Girault, Lee & Lee, 1999). Therefore, a hierarchy provided by Harel (Harel, 1987) can be used to resolve this problem.

To clarify, HFSM is defined where a state can be further nested into another finite state machine. The benefit of using HFSM is to reduce the number of states. As illustrated in (Girault, Lee & Lee, 1999), FSM can be used to describe the behavior of a module in a concurrency model that FSM can be nested within (Girault, Lee & Lee, 1999). Equally, a subsystem in a concurrency model can become active when it is nested within a state of FSM (Girault, Lee & Lee, 1999).

3.2.5 Communicating Finite State Machine (CFSM)

In some cases, because of their complexity, it becomes quite sophisticated to specify and model distributed systems and communication protocols in form of FSM regardless of their conceptual simplicity (Klemm, 1996). Therefore, CFSM can be used to model such systems since it has the same potential functionality of finite state machine (FSM), but it is extended with the support of data handling and asynchronous communication ("Finite State Machines," 2001).

CFSM is actually a FSM extend with support for data handling and asynchronous communication. CFSM has locally synchronous behavior and globally asynchronous behavior. It is useful for embedded systems and communications protocols that are used for a description of the control structure of specifications written in languages like Statecharts and SDL (Hierons, 2001).

The model of CFSMs was introduced by Brand and Zafiropulo to investigate a model of communication protocols with regard to certain protocol properties (Brand &

Zafiropulo, 1983). They proposed a model for designing and analyzing communication protocols that is based on finite state machines (FSMs).

CFSMs describe all component processes of a system in which each process is modeled by one or several FSMs, and a full-duplex, error-free, FIFO channel to represent the interconnecting channel along with their desirable properties that are responding to only the events that occur (Brand & Zafiropulo, 1983).

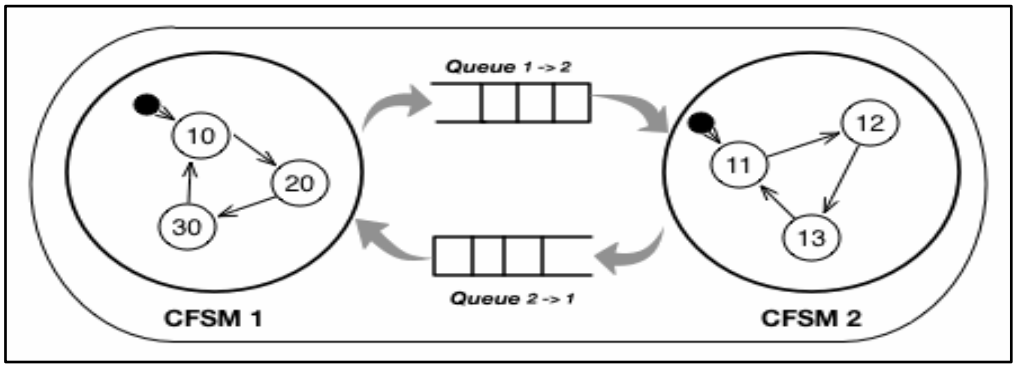


Figure 3.2: Communicating Finite State Machines (CFSMs) by Klemm, 1996

The processes communicate by sending and receiving messages with no assumptions regarding the time that a message spends in a channel before it is delivered to its destination. It shows protocol parties and the communication medium as separate entities (Brand & Zafiropulo, 1983).

Brand and Zafiropulo (1983) assumed that the proper abstraction of the channels is to have queues with unbounded capacity because the protocols may operate on different channels with different capacities allowing for arbitrary number of messages in transit. As explained by Brand and Zafiropulo (1983), using this proposed model of communication protocols have an advantage over Petri nets in that it is more powerful in modeling channels such as FIFO property, but it is less powerful in modeling processes (Brand & Zafiropulo, 1983).

Bollig (2006) expressed CFSM as a model consists of a set of FSMs that are connected pairwise with reliable, unbounded, FIFO channels. They share one global initial state and several global final states. Each FSM has a set of transitions that are labeled with send or receive actions (Bollig, 2006). A send action puts a message at the end of the channel where a receive action is taken provided that the required message is found in the channel (Bollig, 2006). This model can be extended to allow for CFSMs to

send specific synchronization messages which results in more expressive power of that model (Bollig, 2006).

A simple example was shown by Bochmann (1978) to describe the alternating bit protocol defined by Bartlett, Scantlebury, and Wilkinson in ("A note on reliable full-duplex transmission over half-duplex links", 1969). It demonstrates the use of the proposed notation they provided in (Bochmann, 1978) for describing of finite state machines of this communication protocol.

3.2.5.1 Communication Mechanisms For CFSM

Besides queuing of messages (which this thesis concentrates on), there are other mechanisms for communication between CFSMs. One of these mechanisms is shared variables communication. This mechanism is applied to Moore machine where a particular machine accesses the local variables of the other machine (i.e. the variables that implicitly represent the current state) in which its transition has an enabling predicate or enabling condition that requires the later machine to be in a specific state.

The other mechanism is synchronous communication (also called direct coupling, rendezvous interaction, or message passing without queuing) that expresses the interaction between FSMs. It is a form of communication where two FSMs participate at a point of communication simultaneously and the two transitions are represented as one transition of the overall system (Bochmann, 2013).

A method call is another type of communication that is implemented in original Umple state machine. The state machine responds to the occurrence of events implemented as public functions, which return a Boolean value. These functions are called by any component of the system. The functions return true if the event results in executing the transition, otherwise they return false (Badreddin, 2012).

3.3 Specification and Description Language (SDL)

Specification and Description Language (SDL) is a formal language that has been used for system analysis and design and communication protocol specifications since the early 1980ies. It is used for event driven, real-time and communicating systems (Babich & Deotto, 2002). The main applications of SDL are communication protocols

and telecommunication systems (Babich & Deotto, 2002).

SDL was developed by Telecommunication standards sector (ITU-T) of the International Telecommunication Union (ITU) between 1976 and 1992 and it has been evolved since then (Babich & Deotto, 2002).

The latest version of the ITU-T Specification and Description Language is SDL-2010. This takes the place of the previous version SDL_2000, It is based on object-oriented principles and is combined with UML in such a way that the UML models are integrated with the SDL-2010 models (SDL Forum Society, 2013).

SDL-2010 provides textual and graphical representations of the structure, behavior, and data of communication systems and protocols in addition to physical descriptions (SDL Forum Society, 2013). It describes distributed systems using finite state machines in which the systems are communicating through channels by sending and receiving signals. The behavior of each system in SDL is described as a set of processes; each process is represented as an extended finite state machine (SDL Forum Society, 2013). The processes of the system operate concurrently and interact with each other via signals. To illustrate, each process can be in a state waiting for an input signal or it is in a state transition. During a state transition, outputs can be generated, or signals can be sent. Each process has its own input queue in which an input signal to a process in a state transition is stored in a FIFO sequence, and it is processed asynchronously (SDL Forum Society, 2013).

The example in (Hogrefe, 2013) describes the input queue mechanism of a process in SDL. It explains the order of removing the signals from input queue in order to initiate the transitions to move a machine from one state to another (Hogrefe, 2013). It shows that when a signal is at head of a queue and the state cannot respond to it, it will be discarded unless the SAVE operator is attached to the state, then this signal will be saved for later consumption (Hogrefe, 2013). The signals will be continually removed from the input queue and processed until there is no kept in the queue.

3.4 Unified Modeling Language (UML)

The Unified Modeling Language is a standardized notation for object-oriented software systems that is based on a set of diagrams in which each diagram describes

aspects of the structure and the behavior of a software system.

The UML standardization is managed and coordinated by the Object Management Group (OMG) (Babich & Deotto, 2002).

UML statechart diagram concentrates on the concept of event-ordered behavior of an object (OMG, 2013). The different elements of the state machine are the following:

Regions: If a state machine at topmost level or a state has two or more regions, then these regions are orthogonal to each other. A part of behavior can be represented by a region that may execute concurrently with its orthogonal region. Each region comprises of a set of states and transitions determining the behavioral flow within such region. It also has its own initial and final states (OMG, 2013).

States: A state machine consists of a set of states for which each state represents a situation of the execution of the state machine's behavior held by some invariant conditions implied through the name of the state. There are three types of states defined by UML; simple, composite, and submachine states (OMG, 2013).

In Umple, a string attribute is controlled by a state machine, which has an unbounded number of states. In Umple, the first state is made by default as the start state and the end state is a state that does not have any outgoing transitions (Badreddin, 2012).

- **State entry:** The state may have associated entry behavior that is executed upon entering to the state (OMG, 2013).
- **State exit:** The state may have associated exit behavior that is executed upon exiting from the state (OMG, 2013).
- **State doAcitivity:** The state may have associated doAcitivty behavior that starts execution upon entering the state but after executing the state entry if it is defined and executes concurrently with any other behavior defining n the state until it completes or the state exit executes (OMG, 2013). In Umple, doActivity represents a long-running computation while in a state that are executed by a separate thread. This in turns allows the state machine to stay live and respond to other events; even while the do activity is running. The do activity is terminated by a transition out of the state (Badreddin, 2012).
- **State history:** The state history is associated with region of composite states for keeping track of the state configuration it was in when it was last exited in order

to enable easily returning to the same configuration when the region becomes active again (OMG, 2013).

- **Final state:** It is a special type of a state that a region has to indicate that such region has completed (OMG, 2013).

Transitions: A single directed arc represents a transition that originates from a source state and terminates on a target state that indicates a fragment of the state machine behavior (OMG, 2013). Umple supports the syntax for different types of state machine transitions: timed transitions, instantaneous transitions, and reflexive transitions (Badreddin, 2012).

Transition can be associated with:

- **Events:** The semantics of state machine events in Umple is to implement the event-handling functions. If an event-handling function executes, it would first check the current state of the state machine, call any entry, execute the transition actions, and call exit actions, if such actions are specified.
- **Guards:** The guard constraint is evaluated before the transition is enabled. If the guard constraint is evaluated to false, then the transition will be disabled. If it is evaluated to true, then the transition is enabled (OMG, 2013).
- **Actions or Effects:** During the transition, the number of actions can be executed. Actions are executed when the transition is fired (OMG, 2013).

3.5 Umple State Machine

An implementation of state machines in Umple builds on UML 2.4.1 specifications, but it does not thoroughly follow them, since Umple has adopted the philosophy that it has the freedom to try out new ideas (Badreddin, 2012). It supports most of the UML 2.4.1 semantics, including events, signals, guards, transitions, actions, entry actions, exit actions, nested states, concurrent states, and some other features (Badreddin, 2012).

The simple state diagram in Figure 3.2 is drawn using UmpleOnline, and it illustrates the main elements of the state machine: two states, s1 and s2; two transitions: two events, e1 and e2; one guard; and one transition action.

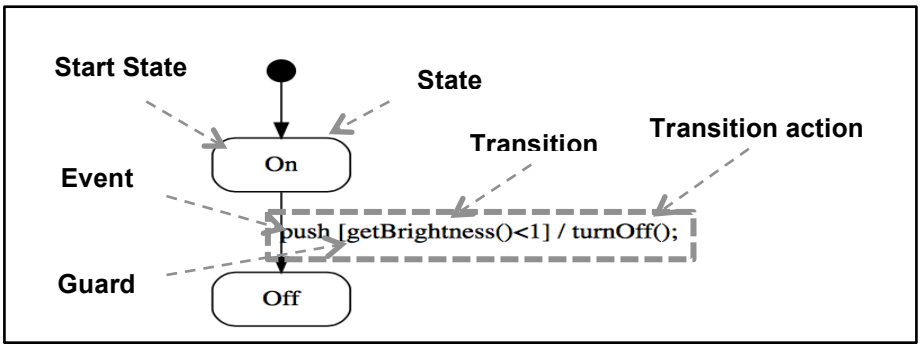


Figure 3.3: A simple state machine as shown in UmpleOnline

3.5.1 TCP/IP Simulation Example

To illustrate the use of Umple state machine for developing of the communication protocol, a TCP/IP simulation example can be used which consists of a set of states and transitions to describe the behavior of this protocol. This up-to-date example can be found in UmpleOnline. This example was built based on information provided in (Uijen, 2009).

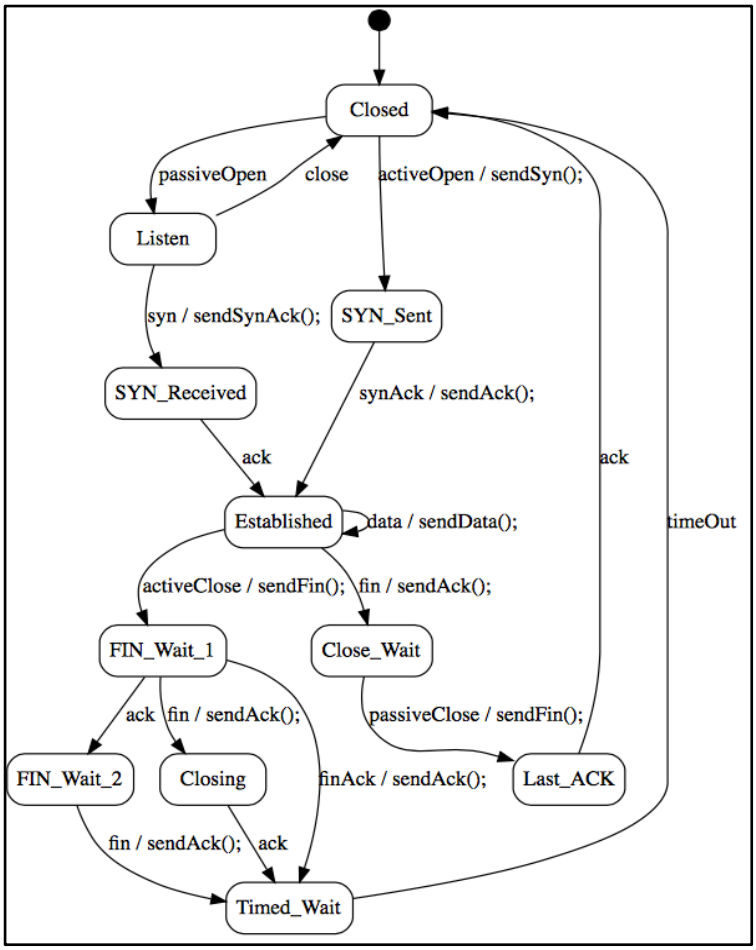


Figure 3.4: State machine of TCP/IP simulation

The Umple code for this example is provided below showing Umple syntax for state machine of TCP/IP simulation.

```
1  // UML state machine and simulation of a the TCP-IP protocol
2
3  namespace tcp_simulation;
4  class Tcp {
5    depend java.util.LinkedList;
6    depend java.util.Queue;
7    depend java.lang.Thread;
8    depend java.io.*;
9    depend java.lang.InterruptedException;
10   depend java.io.IOException;
11
12   // Tcp flags
13   const String SYN="SYN";
14   const String ACK="ACK";
15   const String FIN="FIN";
16   const String RST="RST";
17   const String SYNACK="SYNAck";
18
19   // Queues for adding and removing Tcp flags
20   //LinkedList messages= new LinkedList();
21   Queue_Tcp q=new Queue_Tcp();
22   BufferedReader in=null;
23   lazy String sentence;
24   connection{
25     Closed{
26       passiveOpen->Listen;
27       activeOpen ->/{ sendSyn();}  SYN_Sent;
28     }
29
30     Listen{
31       syn ->/{ sendSynAck();} SYN_Received;
32       close -> Closed;
33     }
34
35     SYN_Received{
36       ack -> Established;
37     }
38
39     SYN_Sent{
40       synAck->/{ sendAck();} Established;
41     }
42
43     Established{
44       fin ->/{ sendAck();} Close_Wait;
45       activeClose->/{ sendFin();} FIN_Wait_1;
46       data->/{sendData();} Established;
47     }
48     Close_Wait{
49       passiveClose ->/{ sendFin();} Last_ACK;
50     }
51     Last_ACK{
52       ack -> Closed;
53     }
54     FIN_Wait_1{
55       ack ->FIN_Wait_2;
56       fin ->/{ sendAck();} Closing;
57       finAck ->/{ sendAck();} Timed_Wait;
58     }
```

```
61  FIN_Wait_2{
62        fin ->/{ sendAck();} Timed_Wait;
63      }
64      Closing{
65        ack -> Timed_Wait;
66      }
67      Timed_Wait{
68        timeOut->Closed;
69      }
70    }
71
72    public synchronized void sendSyn(){
73      try{
74        q.putMessage(SYN);
75      }catch (InterruptedException e) {}
76    }
77
78    public synchronized void sendSynAck() {
79       try{
80         q.putMessage(SYNACK);
81       }catch (InterruptedException e) {}
82    }
83
84    public synchronized void sendAck() {
85      try{
86        q.putMessage(ACK);
87      }catch (InterruptedException e) {}
88    }
89
90    public synchronized void sendData() {
91      in=new BufferedReader(new InputStreamReader(System.in));
92      String sn;
93      try{
94        sn=in.readLine();
95        try{
96          q.putMessage(sn);
97        }catch (InterruptedException e) {}
98      } catch (IOException ioe){}
99    }
100
101   public synchronized void sendFin() {
102     try{
103       q.putMessage(FIN);
104     }catch (InterruptedException e) {}
105   }
106 }
```

Listing 3.1: Umple syntax for TCP/IP simulation state machine

The TCP/IP simulation system consists of (11) states. The system starts in the initial state Closed. Each state of the TCP/IP state machine has at least one transition. The whole state machine has (17) transitions; that is, for each state, an event triggers a transition to the next states. A complete sketch of TCP/IP is given in Appendix A.1.

3.5.2 Communicating Mechanisms For State Machines in Umple

First, the current mechanism is method call. Problems:

1. It assumes there is only one active process in the system that calls a given state machine (no concurrent input events).
2. If one called transition produces as output a call on the state machine that made the original call (input), then that latter machine has two concurrent methods activated (very bad practice).

Making the methods synchronized does not work because in case 1 there may be deadlocks, and in case 2 there will be a deadlock.

To avoid such problems, we introduce the queuing mechanism of CFSM into Umple. In addition, the pooled approach of CFSM is provided for handling the unspecified receptions, which in turns reduces these problems. We also provide other mechanisms for handling the unspecified receptions in Umple.

In my thesis, I have realized the implementation of queued and pooled cases in Umple state machines in next chapters.

3.6 Related Work

In this section, a literature review is provided to study the handling of queues in three different finite state machine generators: State Machine Compiler (SMC), Sparx Systems Enterprise Architect (EA), and the SDL generator tool Real Time Developer Studio (RTDS).

3.6.1 State Machine Compiler (SMC)

State Machine Compiler (SMC) is a free, open source Java-based tool. SMC has been designed to work on any platform that supports Java 1.7 or better (Rapp, 2014). SMC enables a developer to write a state machine specification textually using the state machine language, and then SMC generates the State Pattern classes for the developer. It then uses the State patterns classes to generate the executable code in one of 14 languages (e.g. Java, C, C++, PHP) to implement the state machine (Rapp, 2014).

SMC allows for viewing and editing the whole finite state machine in a single file making it easy to get the design advantage of the State Pattern without writing the state classes (Rapp, 2014).

The goal of SMC is to support active objects of an application by providing an FSM. A finite state machine approach in SMC starts with an active object that receives asynchronous events and might send these events as well (Rapp, 2014).

Rapp (2014) emphasized that the philosophy about the state machines in SMC is different from the UML philosophy; thus, SMC does not follow the Harel/UML statechart specification. Therefore, there are some UML statechart features that are not supported by SMC. For example, hierarchical state machine, composite states, deferred events, compound transitions, and completion events are not supported by SMC.

Transition Queues:

The main problem that might occur in a state machine when using SMC is to issue a transition from within a transition action (Rapp, 2014). To solve this issue, SMC proposes using timers and transition queues. It is suggested not to use transition arguments if using transition queues to avoid any difficulties or errors that might occur in the state machine code.

The generated Java code from a class that has an associated FSM will then have a transition table private member and transition queue private member to place transitions in for later execution (Rapp, 2014) as follows:

```
private static HashMap _transition_map;
private LinkedList _transition_queue;
```

Also, it will have a class static block to fill the transition table with transition methods as follows:

```
static {
  try {
    Class context = AppClassContext.class;
    Method[] transitions = context.getDeclaredMethods();
    String name;
    int i;
```

```
    for (i = 0; i < transitions.length; ++i) {
      name = transitions[i].getName();
      // Ignore the getState and getOwner methods.
      if (name.compareTo("getState") != 0 && name.compareTo("getOwner") !=
0) {
_       transition_map.put(name, transitions[i]);
      }
    }
  } catch (Exception ex) {}
}
```

Finally, a transition method is added to queue up a transition only if it detects the call outside a transition, and then dequeue a transition and execute it (Rapp, 2014) as follows:

```
private synchronized void transition(String trans_name) {
  // Add the transition to the queue.
  transition_queue.add(trans_name);
  // Only if a transition is not in progress should a transition be
issued.
  if (_fsm.isInTransition() == false) {
    String name;
    Method transition;
    Object[] args = new Object[0];
    while (_transition_queue.isEmpty() == false) {
      name = (String) _transition_queue.remove(0);
      transition = (Method) _transition_map.get(name);
      try {
        transition.invoke(_fsm, args);
      }  catch (Exception ex)
      {
        // Handle exception.
```

```
      }
    }
  }
  return;
}
```

In Umple, we define a queued state machine (QSM) which is a state machine that has a queue in which events are injected to be processed for later execution. The Java generated code from QSM is described in Chapter 4.

Default transitions:

In addition, SMC deals with a case of unexpected events by defining "Default" transitions that are placed in a state to back up all transitions (Rapp, 2014). These transitions enable objects to handle and recover unexpected events and continue execution normally. The "Default" transition may have guards, and it may not have transition arguments (Rapp, 2014).

Moreover, if a state receives a transition that is not defined in that state, a special state named 'Default" can be defined. This special state contains transitions that may have guards and argument features. If a user does not define a Default state or a Default transition, and there is no relevant transitions defined in a state machine when a cetain event occurs, SMC will throw a "Transition Undefined" exception (Rapp, 2014). In Umple, we have a special transition named 'unspecified' to handle unexpected events. More information about this special transition is presented in Chapter 5.

Transition Arguments:

Furthermore, SMC enables defining argument lists for state machine transitions. The transitions can receive an argument list that is typically used in the guard conditions or possibly in transition actions (Rapp, 2014). When the same transition is defined with multiple guards and different argument lists are used, SMC will result in incorrectly generated code (Rapp, 2014). In Umple, a transition's arguments are also supported.

3.6.2 Sparx Systems Enterprise Architect (EA)

Enterprise Architect has been developed by Sparx Systems to specify, design, construct and document software systems projects (Sparx, 2014). It is a visual platform built on UML 2.5 specifications to be used for various generalized modeling purposes. It includes a set of features and capabilities that cover all aspects of development (Sparx, 2014). Sparx Systems has developed EA for 15 years (Sparx, 2014). It is intended for a broad range of industries in 160 countries.

EA supports generating and reverse engineering of source code for state machines and other models in programming languages such as Java, C, and C++ (Sparx, 2014). The UML modeling in EA supports and depends on code engineering; that is, a user can generate code from a model, and create and update the model from code. It also allows for debugging, compiling and visualizing executable code generated from the model (i.e. it converts the actual code execution and calls into visual diagrams). In addition, the EA tool allows for round-trip and synchronization of code in various programming language; therefore, code generation templates provided by EA enable customizing generated source code(Sparx, 2014).

Enterprise Architect supports modeling UML state machines that are based on Harel State Charts (Sparx, 2014). In EA, a state machine demonstrates the movement of an element (a Class) between states. It also classifies the behavior of this element depending on transition triggers and conditioning guards. The state machine diagrams include a set of elements that a user can use for modeling the behavior of dynamic systems. State machine diagrams elements include states, initial states, final states, history states, decision points, junctions, entry points, exit points, terminations, and forks/joins (Sparx, 2014). Also, connectors used for state machine diagrams are transitions and object flows. The transition defines the logical movement from a state to the next state in a state machine (Sparx, 2014). It can have a guard, effect, or trigger (event). The trigger can be of the following four types: Call, Change, Signal, or Time.

In addition to those state machine diagram elements, it is possible to add composite states and regions to a state machine diagrams by first creating a composite state element and then subdividing the element with regions.

Enterprise Architect not only supports generating, compiling and executing Java code from State Machines, but also it supports simulating State Machines in visual models during execution (Sparx, 2014).

EA does not generate Java code for queued transitions explicitly but while the execution of a simulation of a state machine, the triggers are queued to the end of the list of the simulation events. The status of a trigger can be a used trigger (i.e. fired), a lost trigger (i.e. fired but has no effects), a signaled trigger (i.e. fired and consumed by one or more transitions), or not signaled trigger (i.e. have not fired yet). The EA tool also enables a user to add parameter values to a signal trigger each time the simulation fires the trigger (Sparx, 2014).

3.6.3 Real-Time Developer Studio (RTDS)

PragmaDev Real Time Developer Studio (RTDS) = supports both SDL and SDL-RT. It is used to model and develop real-time and embedded software (Pragmadev: Real time developer studio, 2014).

RTDS can be used for all stages of the development cycle. It provides users with a graphical interface that enables them to manage projects, design models, simulate the models, and generate code from the models (Pragmadev: Real time developer studio, 2014).

RTDS supports the SDL standard of the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T), and also supports the SDL-RT standard, which is a mix of the SDL standard, UML, and the C language (Pragmadev: Real time developer studio, 2014).

SDL contains an action language with an execution semantics in addition to its graphical notations. For this reason, SDL is considered as a formal language and its models are executable. However, SDL-RT is semi-formal because it combines the SDL standard with C code (Pragmadev: Real time developer studio, 2014).

RTDS tool allows development of the requirements, structural and behavioral models and designs; it also supports model checking capabilities, traceability information, code generation, and testing (Pragmadev: Real time developer studio, 2014).

RTDS supports generating code for the whole system or sub-parts of the system such as a process, a task, or a block in order to compile and execute it. The generated code can be documented for further customization (Pragmadev: Real time developer studio, 2014).

RTDS only enables generating C or C++ code from SDL models; it does not support generating Java code from these models. Thus, the comparison of the generated code of the SDL models with the generated code of Umple queued state machines is not applicable at this level. This is because our work only enables generating Java code for queued state machines in Umple. However, the other team members in our group are working on generating C++ from the queued state machines (Pragmadev: Real time developer studio, 2014).

3.7 Handling Unspecified Reception in Specification Languages

To handle Unspecified Reception errors, the following ways can be used:

- Input is dropped and ignored which is the semantics of SDL and original Umple state machines. In certain designs, this can be a valid choice. For example if a state machine is modeling user interaction, and certain buttons are to be ignored in certain states, then it is reasonable to just drop events that correspond to ignored button presses.
- Complete behavior for all inputs by adding "error" state and transition corresponding to the inputs that were non specified in the original specification will lead to this error state. In an error state, all inputs will lead back to this error state. This is what we have done in Umple with the addition of the 'unspecified' keyword.
- Queue models defined for communicating subsystems or components. Bochmann (2013) listed three queuing models in which the definition of the properties of the composed system depends on: a single input queue for each component, multiple queues, one for each source component, and an input pool. The message transmission can be reliable or unreliable and the messages can be received in order as they are sent or not. For the third model that is input is deferred and placed in a buffer pool, - inputs are left in the buffer pool and will

not be consumed in this state of the specified component. When the state changed appropriately, they can be consumed. The SAVE operator in SDL, Deferred Event in UML, and General Message Pool are used for this purpose. In Umple we have added the concept of a pooled state machine. Details of these are described in the next subsection.

3.7.1 Original Semantic of Umple State Machines in Case of Unspecified Reception

In original semantics of Umple state machines, the default behavior for handling unspecified reception error is to return false and ignore an unexpected event if the current state does not respond to it.

In addition, in the case of extended semantics of Umple state machines, unspecified reception occurs when a message exists at the head of queue and there is a set of reception transitions but the message in the queue is different from the messages expected by the reception transition.

3.7.2 Save Operator in SDL and Deferred Event in UML

In current SDL semantics, each state machine is assumed to have a single input queue where received messages from different sources are stored. If an unspecified reception event occurs, a message at the head of a queue will be dropped.

SDL supports the SAVE operator which is concerned with non-ordered signal reception. A SAVE operator in SDL matches the deferred event of UML (Selic & Rumbaugh, 1999). The main idea of SAVE operator is to postpone the consumption of a specified signal for future processing until the following transition is executed (Babich & Deotto, 2002). In the SDL grammar, the SAVE operator specifies a set of identifiers for signals and remote procedures whose instances are not related to the process in a state to which save is attached and they have to be saved for future processing (Babich & Deotto, 2002).

The SAVE operator changes the order of signal consumption because it saves the events that cannot be triggered in a state where this operator is defined (Hogrefe, 2013). In addition, the SDL SAVE construct is added for each signal that may arrive in the input queue earlier that expected to prevent deadlocks in the SDL specifications (Hogrefe, 2013).

Unspecified reception in UML is an unexpected event reception which is recognized as a semantic variation point. The default behavior in UML when unspecified reception occurs is to discard this event if the deferred construct is not defined (Lilius & Paltor, 1999). However, in UML state machines, there is a special mechanism for deferring events in states (Selic & Rumbaugh, 1999). This can be achieved by including a clause 'deferred / {event list}' in a state (Selic & Rumbaugh, 1999). This advanced constructor is used to solve the problem: when an event usually does not enable any transitions, it is kept waiting until the next state, meaning that the transition selection mechanism does not discard this event in this case. In the UML standard, a set of deferrable events can be attached to a state. So if the event that does not enable any transitions occurs, it is pushed into the deferrable list and once the system move to a next state, this event will be dispatched and processed if it enables any transitions (Lilius & Paltor, 1999). Upon entry to such state, the UML state machine will automatically recall any saved event(s) that are no longer deferred and process them as if they have just arrived (Selic & Rumbaugh, 1999).

Chapter 4: Queued State Machines (QSM) and Pooled State Machine (PSM) in Umple

4.1 Enhancements to Umple State Machines

We extend the semantics of Umple state machines to add queued state machines (QSM) and pooled state machines (PSM).

These mechanisms can be used for communicating FSMs (CFSMs), but can also be useful when there is only one state machine, with other code triggering events.

A queued state machine (QSM) in Umple is a state machine that works in a multithreaded environment and supports an event queue. The code that creates events will add each event to a queue. A separate thread processes the events in the queue in first-in-first-out (FIFO) order and performs any actions and necessary state changes.

A pooled state machine (PSM) in Umple works in the same way as the queued state machine (QSM) except in the way it deals with the case of unspecified reception. As with QSMs calls to event methods of PSMs return immediately. But the handling of events is not done strictly in FIFO order as in QSMs, but in the order in which the events are required by the transitions from the current state. In other words, if an event is at the front of the queue, but the current state does not have a corresponding transition, then the next event in the queue is considered (and so on).

QSM and PSM in Umple can be used in the case when we have only one state machine that can operate normally while the other thread triggers events. In other words, one thread will take care of implementing the behavior of the state machine while the other thread handles event triggering. In addition, QSM and PSM can be used for communicating finite state machines (CFSMs). They perform thread safe operations. They can provide better handling of signals and asynchronous communication semantics to simplify communication between state machines interacting with each other by asynchronous calls of events and passing of messages.

The main features of the queued and pooled state machines are:

1. Adding the events to the queue/pool as they are called.
2. In QSM, processing the events in the same order as they are received (i.e. in FIFO order).
3. The event-processor thread processes the events until there are none left in the queue/pool (or none in the pool that can be handled), and then it waits for the next event to arrive.
4. If an event occurs in a QSM that does not match any event in the current state (unspecified reception), and there is no transition labeled 'unspecified', then the event is taken off the queue and ignored (i.e. it is not processed). If an 'unspecified' pseudo-event exists in the current state then this transition is taken instead.

Queued and pooled state machines extend the original semantics of state machines in Umple, in the following called Basic state machines, which is considered to be appropriate mostly for simple applications. Basic state machines in Umple work in a single thread, they process events immediately as they are received. When an event occurs and there is no state anywhere in the (queued, pooled or basic) state machine that accepts it, the compiler will detect this event and indicate to the user that this symbol cannot be found.

When the state machine's event occurs, the state machine code continues in the same thread as the caller. A problem occurs when there are two machines and the second machine creates an event for the first machine in a transition that is triggered by the first machine. As shown in Figure 4.1, there are two method calls active on the same machine at the same time (which is bad programming practice). In fact, during the transition of SM1 triggered by b, the transition of SM2 is called, which in turn calls the transition c of SM1 (which has not yet completed its b transition) – two transitions are active on SM1 at the same time.

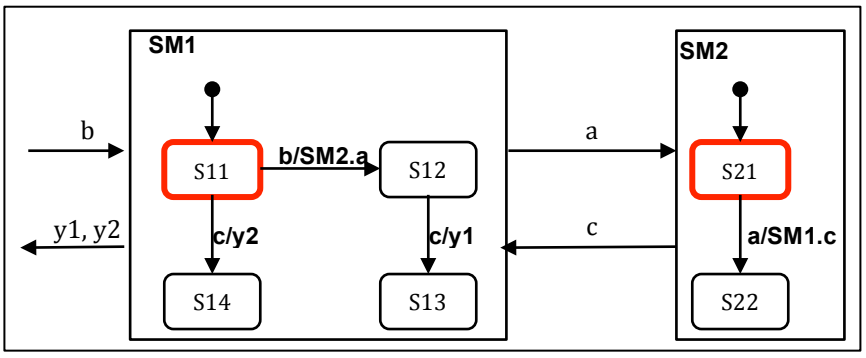


Figure 4.1: Example of the state machines in which two event method calls active on the same machine at the same time

To illustrate the differences between the basic and extended semantics of Umple state machines, an example of two state machines is shown in Figure 4.1. Two questions needed to be addressed to distinguish between the actual and expected behaviors of those two machines, which are:

"What happens when an input event 'b' is processed? What output to the environment would have been produced and which state would the machine (SM1) be after an input event 'b' has been completely processed?"

The expected behaviors of both machines are as follows. The first machines SM1 is initially in state (s11) and the second machine SM2 is initially in state (s12). When an input event 'b' occurs, the action of this transition is executed: an event 'a' of the state machine SM2 is called and the event 'b' is processed; the machine SM1 moves from state (s11) to state (s12). Then the event 'a' of the machine SM2 is processed. The action of this transition includes calling the event 'c' of the machine SM1. It then moves the machine SM2 from state (s21) to state (s22). Then, the event 'c' is the processed. The machine SM1 transitions to state (s13) and produces the output 'y1' to the environment.

In Umple basic state machine, Umple uses procedure call semantics to process input events. That means each transition of the state machine is realized as a method that is called when the event occurs.

Implementing the above example in Umple basic state machine causes the transitions b and c to be active at the same time. This is due to the fact that Umple implements the basic state machines in a single thread environment.

The actual behaviors of the Umple basic state machines: SM1 and SM2 are not as expected. During the call of the 'b' transition, the method 'a' is called on the same object; that is, there are two processes executing in the same object simultaneously, the waiting of the 'b' method, and the execution of the 'a' method. Thus, the actual behaviors of SM1 and SM2 are as follows. When the event 'b' occurs, the method 'a' is called, the event method 'c' is called and SM2 moves to (s22). When the method 'c' is called, the machine SM1 is still in state (s11) because the method 'b' is still executing. Thus, the method 'c' is called; the machine SM1 moves from (s11) to (s14), and the 'y2' output is produced to the environment. The final outputs indicate that there is a problem in the implementation of these state machines SM1 and SM2 in Figure 4.2.

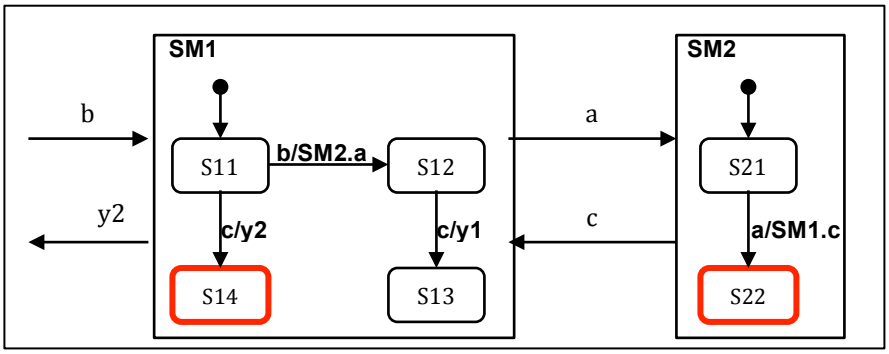


Figure 4.2: The final outputs of the Umple basic state machines' behaviors

The proposed solution for this problem is to use Umple Queued State Machines (QSM) because they are implemented in a multithreaded environment, which will work for this kind of concurrency.

The actual behaviors of SM1 and SM2 when using QSMs are shown in Figure 4.3, 4.4, 4.5, and 4.6. When the input event 'b' occurs, the event 'a' is injected in the queue until the event 'b' is completely processed. The process transitions SM1 from (s11) to (s12). Then, the event 'a' is taken off the queue (it is consumed). The process of the event 'a' performs calling of an event 'c'. The event 'c' is injected in the queue. After the event 'a' is completely processed to move SM2 from (s21) to (s22), the event 'c' is taken off the queue (it is consumed). It moves SM1 from (s12) to (s13) and produces 'y1' to the environment. Figure 4.6 shows the final outputs of the implementation of queued SM1 and queued SM2.

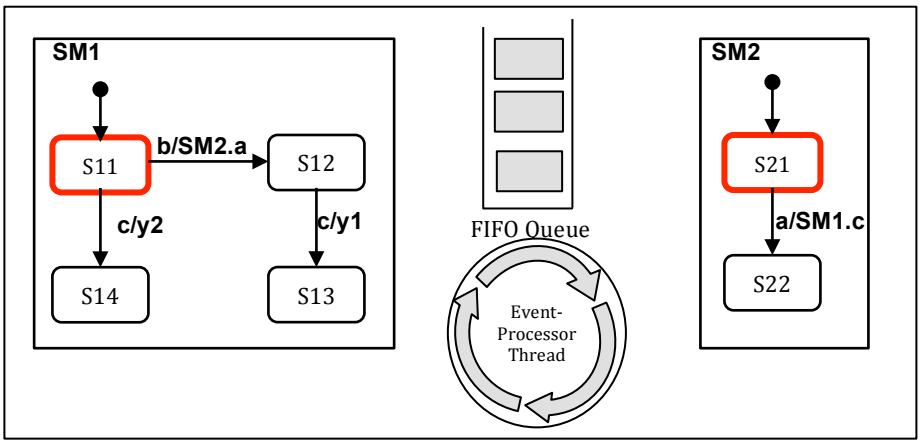


Figure 4.3: The first step of the QSMs implementation

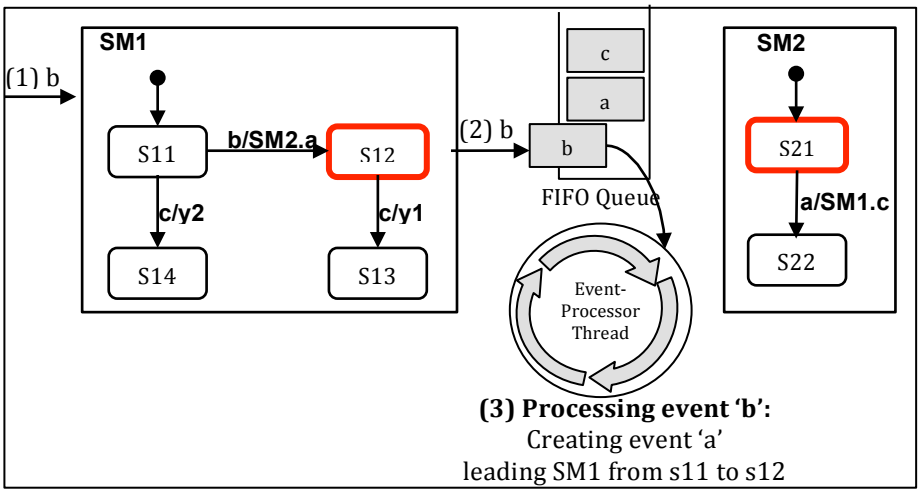


Figure 4.4: The second step of the QSMs implementation

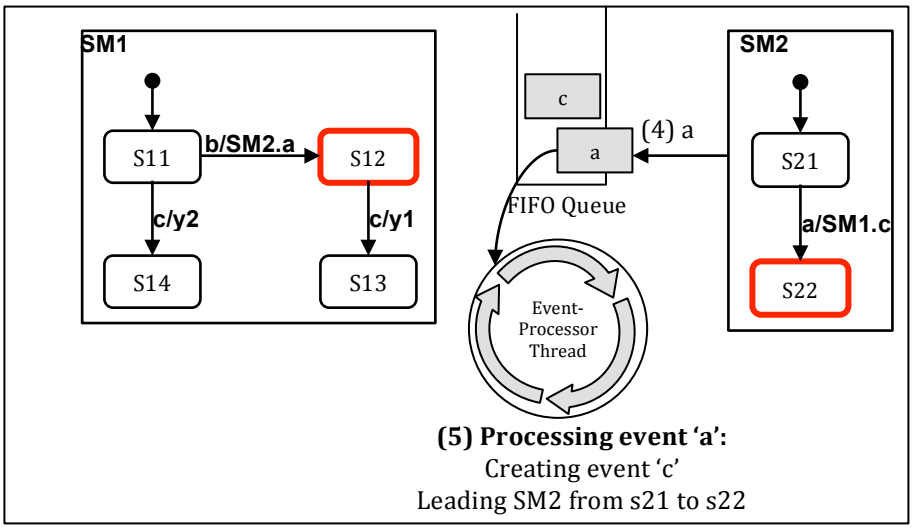


Figure 4.5: The third step of the QSMs implementation

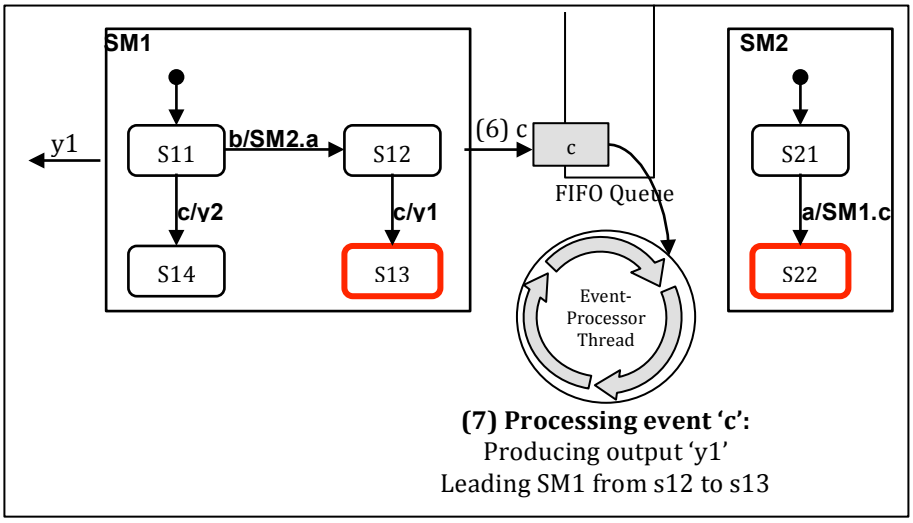


Figure 4.6: The final step of the QSMs implementation

The other problem that may occur when using Umple basic state machines is the unspecified reception design error.

Ignoring Unspecified Events in Basic and Queued State Machines in Umple

Currently, the UML default behavior is to ignore that event. This is also the semantics of Umple basic and queued state machines, which is to ignore unspecified receptions. The default behavior is that the event method returns false and the state remains unchanged.

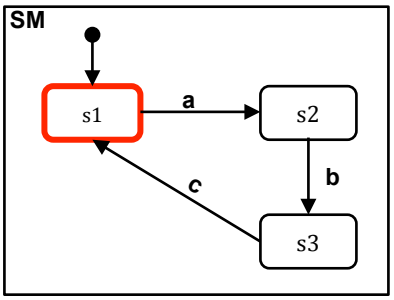


Figure 4.7: Example of unspecified reception problem in a state machine

The state machine shown in Figure 4.7 using Umple basic or queued state machine results in an unspecified reception error if the sequence of events executed is: <a, b, b, c>. If an event 'a' occurs, the machine SM moves from state (s1) to state (s2). When an event 'b' occurs, the transition is fired and the process performs the transition to state (s3). Now, the event 'b' occurs and because the machine is now in state (s3) not (s2), this event would be ignored, which means it would not be processed. The default behavior of the Umple basic and queued state machine is to ignore and discard

unspecified events. Finally, the event 'c' triggers the transition resulting in leading the machine SM to state (s1).

The execution trace of <a, b, b, c> is illustrated in Table 4.1.

Table 4.1: Execution trace for Umple code in Figure 4.1

Event	Current State	Next State	Note
a	s1	s2	a is consumed and processed
b	s2	s3	b is consumed and processed
b	-	-	b is ignored - Unspecified reception error
c	s3	s1	c is consumed and processed

Other Solutions For Detecting and Handling Unspecified Receptions in Umple

We have introduced three fundamental design alternatives to detect and handle unspecified receptions that may occur at any time, which are to the following:

- Use a special transition named 'unspecified' in a state, at any level of nesting, which would match any event that is not handled when it is encountered in a state. This solution is an extended semantics of a state machine. In SDL, a star ('*') is used to indicate any other message that may be received in that state of the machine (SDL-RT, 2013). The 'unspecified' transition has the same semantic where a message that arrives, and there is no transition to consuming it in the current state; it would be received and consumed by the 'unspecified' transition. The semantics of this special event have the same semantics as any other event of a state machine which means that it could perform any kind of action just like any regular event; it could be a self-transition; it could transition to some other state, or even it could be guarded. It can then do one of the following actions:
  - The 'unspecified' transition goes back to the same state, which means that the message is ignored.
  - The 'unspecified' transition has an action to display a message indicating that there was an unspecified reception.
  - The 'unspecified' transition can lead to an error state, which can have an instantaneous transition that leads back to the current state.
- Use the after/before code injection capability of Umple to inject code after or before events for all types of state machines (i.e. basic, queued, and pooled). For

example, a method to display a message indicating that there is unspecified reception can be injected after one or more event method handling. Thus, if an unspecified reception is detected, the message is displayed to indicate that there is an error.

- Use a pooled state machine (PSM) that would only take messages from the pool for which there is a transition in the current state to consume these messages. The semantics of PSM works for the entire state machine. The semantics of PSM is similar to the SAVE in SDL; however, the latter acts on a state-by-state basis and only for the explicitly SAVED inputs. The semantics of the SAVE is that for a given state a certain number of inputs (that would not be consumed) would be saved for later (remaining in the queue). When the machine moves to a new state, the saved messages will be treated first before any other messages in the queue (SDL-RT, 2013).

The Behaviors of the Umple Basic and Queued State Machines if the 'unspecified' Transition is Specified

To illustrate how to use the unspecified reception handler mechanism, how the semantics of each kind of the state machine is affected by this mechanism, what the actions are taken when the unspecified reception handler mechanism is used, and what outputs result from executing this mechanism with these types of the state machine, we consider the following example shown in Figure 4.8. 'unspecified' transitions are added to the states (s1) and (s3) of the state machine SM and change the default behavior of this state machine.

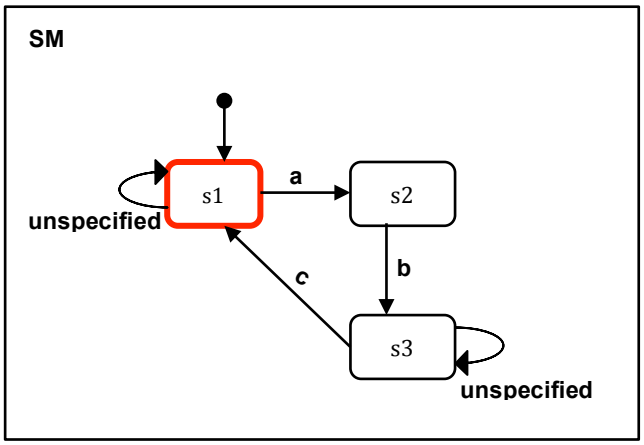


Figure 4.8: Using 'unspecified' transitions in states (s1) and (s3)

Table 4.2 shows the executions for all three types of state machines when the sequence of events called is <a, b, c, c, a, b>.

Table 4.2: Execution trace of the example in Figure 4.8

	Basic State Machine	Queued State Machine	Pooled State Machine
initial	s1	s1	s1
a	- a is called and processed. - transition to ‘s2.'	- a is called and processed. - transition to ‘s2.'	- a is called and processed. - transition to ‘s2.'
b	- b is called and processed. - transition to ‘s3.'	- b is called and processed. - transition to ‘s3.'	- b is called and processed. - transition to ‘s3.'
c	- c is called and processed. - transition to ‘s1.'	- c is called and processed. - transition to ‘s1.'	- c is called and processed. - transition to ‘s1.'
c	- c is ignored because it is unspecified reception. - ‘unspecified' method is called. - transition to ‘s1.'	- c is ignored because it is unspecified reception. - ‘unspecified' method is called. - transition to ‘s1.'	- c is pooled because it is unspecified reception. - state machine is in ‘s1.'
a	- a is called and processed. - transition to ‘s2.'	- a is called and processed. - transition to ‘s2.'	- a is called and processed. - transition to ‘s2.'
b	- b is called and processed. - transition to ‘s3.'	- b is called and processed. - transition to ‘s3.'	- b is called and processed. - transition to ‘s3.' - c is taken off the pool and is processed. - transition to ‘s1.'

The semantic of a pooled state machine (PSM) has a different strategy to handle the events of the state machine. When an event occurs and there is no transition to consume it, it would be saved in the pool. The state machine then continues its execution. Another event in the pool that matches a defined transition in the current state of the machine will be removed from the pool to be processed. When the machine reaches a state in which the saved event has a transition to consume it, the event is removed from the pool and is processed.

Figure 4.9 shows the behavior of the state machine SM when the sequence of events <a, b, c, c, a, b> occur. It shows that when the event 'c' occurs after the event 'b' triggering the transition to state (S3), the event 'c' is consumed and processed leading the state machine SM to state (s1). Then when the event 'c' occurs, it will be saved in the pool because there is no transition to consume it.

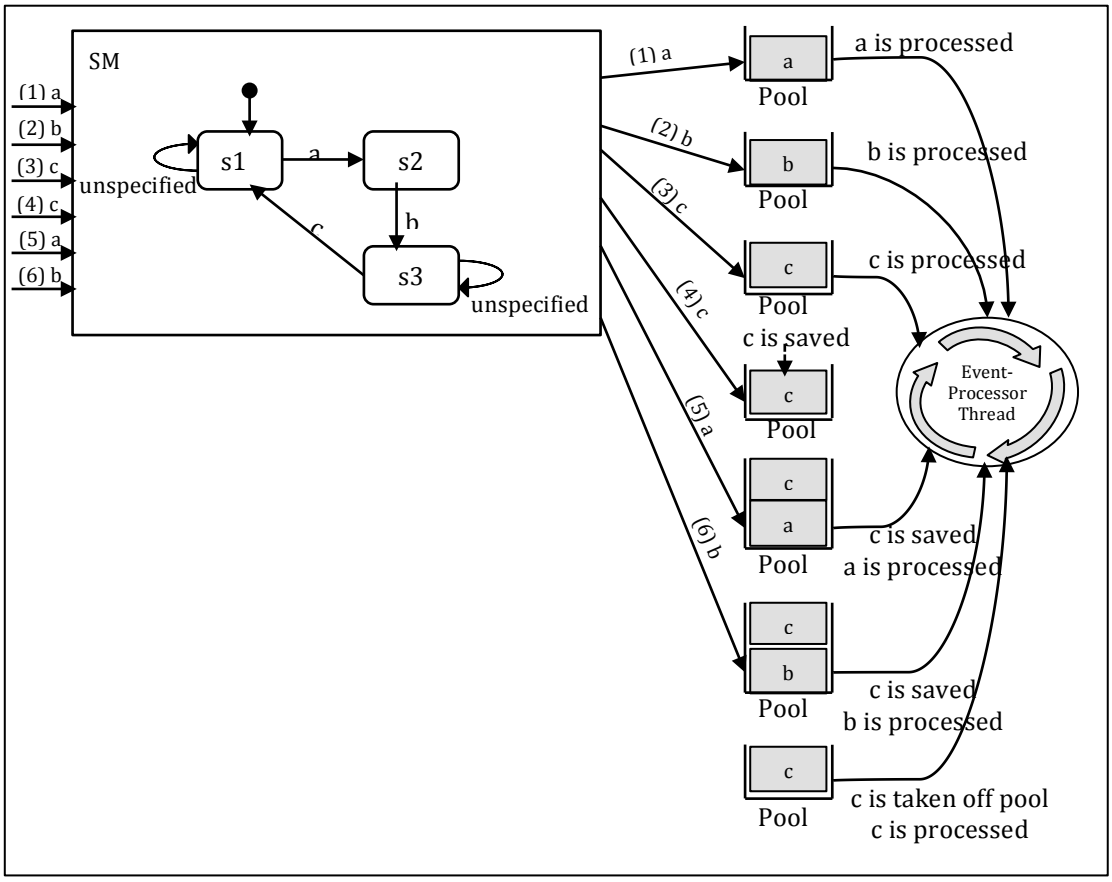


Figure 4.9: The behavior of the pooled state machine (PSM)

4.2 Queued State Machines (QSM) in Umple

A queued state machine (QSM) has a FIFO queue, in which the series of events to be executed are placed. The calling thread can then continue and do other tasks. A separate thread exists in each state machine, which removes the events from the queue one at a time and executes the corresponding transition method in the order they were inserted into the queue.

During the design of the queued state machines, we uncovered a number of issues related to the design decisions that we made, in order to end up with our final

design of this type of state machines.

Let us consider the following example of a state machine that has two states s1 and s2 and two transitions. In this state machine, an event e1 causes the machine to transition from state s1 to s2 and an event e2 causes the machine to transition back to state s1, as shown in the state machine diagram of Figure 4.10.

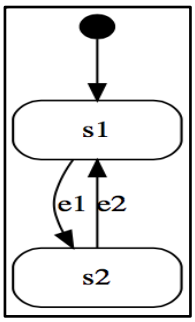


Figure 4.10: A state machine as depicted in UmpleOnline

In order to derive the implementation from this model, we used the UmpleOnline tool where we wrote Umple code to sketch this diagram (we can also draw the diagram without writing the text and the text will appear directly), and then we generated the Java code from this model. We modified this code to get the proposed implementation of the model.

The pseudocode of what the Java code should look like is shown in Listing 4.1 below:

```
1   public class X implements Runnable
2   {
3     create an enumeration for the types of messages accepted by the
4   state machine: Message type is defined as a string: *event_M*
5     declare a message queue
6     declare a event-processor thread
7     constructor
8     {
9       create instance of the message queue
10      start the event-processor thread to remove from queue
11    }
12
13    // Event method handlers
14    _e1()
15    {
16      check a current state then transitions to the next state
17    }
18    _e2()
19    {
20      check a current state then transitions to the next state
21    }
22    Define an inner class called Message to create object instances of
23  messages by passing their types and parameters as the arguments
```

```
25    class Message
26    {
27      constructor with two arguments:  message type and parameters
28    }
29    Define an inner class called MessageQueue that has two
30  synchronized methods for add and remove the events into the message
31  queue
32    *put method* to insert a message into the message queue and then
33  notify other threads.
34    *getNext method* to remove a message from the head of the message
35  queue. If the message queue is empty, then wait until an event is
36  added to the queue.  Otherwise, remove the message from the queue
37  and return it.
38
39    class MessageQueue {
40      declare and instantiate a message queue
41      //put method to add the message into the queue
42      synchronized put(Message m) {
43        add a message to the pool and then notify
44      }
45      // getNext method to remove the message from the head of the
46  message queue
47      synchronized getNext() {
48        while the queue is empty, then wait, otherwise, the messge is
49  removed from the head of the message queue
50      }
51    }
52
53    //messages accepted
54    e1() {
55      add a message to the message queue
56    }
57    e2() {
58      add a message to the message queue
59    }
60
61    event-processor thread run method
62    {
63      do forever using while(true) loop {
64        check if the message queue is not empty, if yes, remove a
65  message from the head of message queue. Otherwise, waiting for an
66  event to be added into message queue
67        switch (message type) {
68          compare message type with each *event name_M*
69          If it matches:
70            call event method _*name*
71            break
72        }
73      }
74    }
75  }
```

Listing 4.1: Pseudocode for queued state machine (QSM)

The following subsections discuss issues related to the design of a queued state machine implemented in Java code.

4.2.1 Multithreading Environment

A multi-threaded program in Java has an initial entry point (the main() method). Other threads have additional entry points (run() methods), which are run concurrently with the main() (He, 2013).

The main goals of making a program multithreaded are to improve perceived performance, responsiveness and throughput by making use of multiple CPUs, and also to simplify program design by separating concerns (He, 2013).

There are two methods for creating threads in Java. The first is to extend the Thread class in java.lang.Thread and then to override the run () method. The second method is implementing the Runnable interface and providing the implementation to the abstract method run() to specify the running behavior of the thread.

Creating Threads For the Java Code of Queued State Machine Class:

For the design of the Java code that implements a QSM, we use the second method of creating a thread. In the constructor of the class, we construct a new Thread instance with "this" as an argument to the constructor and then invoke the start () method that calls back run () in the Runnable object. The reason for implementing the Runnable interface is because Java does not support multiple inheritances. So, if a class already extends from some particular superclass, then it cannot extend Thread. Implementing Runnable will give a program more flexibility to extend other base classes.

Therefore, there are several threads that work simultaneously; each queued state machine has its own thread (i.e. event-processor thread), and the other thread is the thread of the process that sends a message along with its parameters if there are any.

Below is a piece of Java code generated from the QSM class illustrated in Figure 4.10.

```
1   import java.lang.Thread;
2   class X implements Runnable {
3     ...
4     public X()
5     {
6       setSm(Sm.s1);
7       queue = new MessageQueue();
8       removal=new Thread(this);
9       //start the thread of X
10      removal.start();
11    }
12  }
```

4.2.2 Enumerations for Incoming Message Types

State machines communicate with each other by passing messages that are encoded when they are passed as an object instance of Message that has a type and parameters. The types of incoming messages for each queued state machine are represented as enumerated values in the generated Java code. The "MessageType" attribute is declared in Java as a special data type enum that has a number of constant values. A variable of the enum type can be defined and which can be assigned to one of the enum constants as value.

The following code is an enumeration for the message types of the queued state machine:

```
//enumeration type of messages accepted by X
enum MessageType { e1_M, e2_M, e3_M, e4_M }
```

4.2.3 Inner Classes: MessageQueue and Messages Classes

To implements a QSM, we decided to create two classes; one is the Message class and the other is the MessageQueue class. Message and MessageQueue contents are exactly the same for all queued state machine classes. We have three design alternatives to represent these two classes in the Java code:

First: The first option is to externalize these two classes and have them used by all classes. In other words, this design allows for creating Message and MessageQueue as separate public classes that can be used by all classes that implement a QSM. They

would have to be instantiated by using templates because they need to operate on a specific enumerated type of messages. This design helps avoid creating a lot of nearly identical code in all classes because creating identical code in every class breaks the simplicity of the design.

Second: This option is to get rid of these classes completely and make their methods a part of the main class instead.

Third: The final option (and the decision taken) is to keep the same architecture, which is to have them as inner classes inside of the main class.

We create for each outer class that implements a QSM two inner classes; one is the Message class and the other is the MessageQueue class. Inner classes are created with the 'protected' access modifier. Protected inner classes restrict access from classes other than the ones in the same package and its subclasses. Inner classes are used for readability and to provide access to the private variables and methods of the outer class. They act like any other member of the outer class. We create instances of these inner classes from inside the outer class. The members of the inner classes can be accessed by the outer class no matter what their access modifier is. The outer class members can be accessed within inner classes directly, no matter what the access modifier is.

This is the best solution among the three options because when message parameters are involved, things become complicated with the other options. In particular, if one considers the case of a message pool (not FIFO) then there will be methods that depend on the state of the machine.

Below, we provide more details about these two classes, and our design decisions for defining these classes.

Message class: We defined an inner class called Message that is used to create a message that may or may not be associated with parameters.

This class has two attributes. The first attribute is 'type' that is used to define the type of the message. This attribute is of data type "MessageType" that is defined in the outer class as an enum type containing the set of all types of messages (events) as

constants. The second attribute is “param” which is defined as “Vector” type. This attribute is used to define the list of parameters of each message. Each parameter can be declared as any data type whether Integer, Strings, etc. Also, the message may have no parameters; the list then will contain only the null value.

Vector is used to define a list of parameters for the message (event). Vector is synchronized in such a way that if one thread is working on the Vector, no other thread can get hold of it (Singh, 2013). This means only one thread can perform an operation on a Vector at a time. Also, Vector can grow by doubling its size by default. It maintains the element insertion order and allows duplicated and null values (Singh, 2013).

Below, we illustrate a piece of Java code that demonstrates the definition and declaration of the ‘Message’ class:

```
1   protected class Message {
2     MessageType type;
3     //Message parameters
4     Vector<Object> param;
5     public Message(MessageType t, Vector<Object> p){
6       type = t;
7       param = p;
8     }
9     @Override
10    public String toString(){
11      return type + "," + param;
12    }
13  }
```

Message and its parameters in Java code are used to represent the event in the state machine and its arguments to be passed. For example, if a transition is defined as:

```
e1 (String eventName) -> s2;
```

The event is ‘e1’ and its argument is ‘eventName’ of type string. Thus, the message is of type ‘e1_M’ and its parameter is ‘eventName.'

To create an instance of the message we create a new instance of the Message class; its constructor has two arguments: type and param. For instance, for the above transition, we create a message as follows: First, we add a ‘param’, which in this case is (eventName) to a vector list as follows:

```
Vector v = new Vector(1);
v.add(0, eventName);
```

Then we create the instance of a message as follows - to be later added to the queue:

```
new Message(MessageType.e1_M, v);
```

MessageQueue class: We defined another protected inner class called MessageQueue that has two synchronized methods: “put” for adding the messages with their parameters to the queue, and ”getNext” to remove and consume the messages from the queue.

The queue of type (LinkedList) is defined and declared as an attribute in the ‘MessageQueue’ class to buffer the list of the messages in FIFO order, assuming that the messages are instances of the ‘Message’ class.

Synchronization is an important concept in concurrent programming. It allows threads to have mutual exclusion, to wait for a certain condition to become true, or to send signals to other threads. It is necessary for reliable communication between threads (Vogel, 2013).

Java provides a mechanism for synchronization called locks. When multiple threads execute a particular part of code at the same time, locks ensure it is executed by only one thread at a time. The simple way to ensure a specific method or class in Java is locked is to use the ‘synchronized’ keyword (Vogel, 2013). This can be applied to methods or blocks.

By using synchronized, the single thread can execute a synchronized part of code at a time and each thread that enters a synchronized block of code is able to see the impact of all previous changes guarded by the same lock that can be either a string or an object and which can only be executed by one thread at the same time (Vogel, 2013).

Below, we show a piece of Java code that demonstrates the definition and declaration of the 'MessageQueue' class for adding and taking out incoming messages of the queue:

```
1   protected class MessageQueue {
2     Queue<Message> messages = new LinkedList<Message>();
3
4     public synchronized void put(Message m)
5     {
6       messages.add(m);
7       notify();
8     }
9     public synchronized Message getNext()
10    {
11      try {
12        while (messages.isEmpty())
13        {
14          wait();
15        }
16      } catch (InterruptedException e){ e.printStackTrace();}
17      //The element to be removed
18      Message m = messages.remove();
19      return (m);
20    }
21  }
```

The put method:

The put method of the MessageQueue object is called in the outer class. When this method is called, it adds the incoming message into the queue, and notifies the event-processor thread that tries to get incoming messages from the buffer, if there are any.

The getNext method:

The getNext method is called by the event-processor thread in the run method. It has a continuous while loop that, if the queue is empty, invokes the wait method from the Object class, and event-processor thread which is trying to consume an incoming message from an empty queue will wait until a message is added to the head of queue. If the queue is not empty, then this method will skip the while loop and remove the message from the head of the queue, and send back a message to the "main class," if there is any.

4.2.4 Message Accepting Methods

When a QSM responds to the occurrence of events that are usually but not

always outside of the context of the state machine itself and created by any component of the system, a message-accepting method would then be called.

The message-accepting method is named after the event name. If the method has arguments, then it will create a Vector in which the arguments are added. Then, the incoming message would be added to the queue by calling the put method of the 'MessageQueue' object. For example, if a QSM has an event as:

```
e1 -> s1;
```

In Java code, the event-handling method (called when an event is taken from the queue in order to process a transition) is declared as follows:

```
public boolean _e1(){}
```

The message accepting method (that adds an event to the queue and is called by external code such as another state machine) is declared as follows:

```
public void e1(){}
```

If the method without the leading underscore is called, a message is added into the message queue by calling the put method of the message queue object. An example of a message accepting method for the event e1 with no parameters for the example of the QSM class shown in Figure 4.10:

```
1  //------------------------------
2  //messages accepted
3  //------------------------------
4  public void e1 ()
5  {
6    queue.put(New Message(MessageType.e1_M, null));
7  }
```

The following example of the message accepting method shows the case when the queued state machine's event e1 has one parameter that is (eventName) of the type 'String' as:

```
1  //------------------------------
2  //messages accepted
3  //------------------------------
4  public void e1 (String eventname){
5    Vector v = new Vector(1);
6    v.add(0, eventName);
7    queue.put(New Message(MessageType.e1_M, v));}
```

The example below of the message-accepting method shows the case when the queued state machine's event e3 has two parameters (id and nextState) of types 'Integer and String' as:

```
1  //------------------------------
2  //messages accepted
3  //------------------------------
4  public void e3 (Integer id, String nextstate){
5    Vector v = new Vector(2);
6    v.add(0, id);
7    v.add(1, nextState);
8    queue.put(New Message(MessageType.e3_M, v));
9  }
```

4.2.5 Removing a Message From the Queue – 'run' Method

The event-processor thread has a run method that is used to define the behavior of the thread that is to remove incoming messages from the queue and then process them.

```
1    @Override
2    public void run (){
3      boolean status=false;
4      while (true)
5      {
6        Message m = queue.getNext();
7        switch (m.type)
8        {
9          case e1_M:
10           status = _e1();
11           break;
12         case e2_M:
13           status = _e2();
14           break;
15         case e3_M:
16           status = _e3();
17           break;
18         case e4_M:
19           status = _e4();
20           break;
21         default:
22       }
23       if(!status){
24         // Error message is written or exception is raised
25       }
26     }
27   }
```

In the run method, a Boolean variable called 'status' is defined and declared to have a value 'false.' When an incoming message is removed from the queue and matches one of the message types to be processed, then the event-handling method will return 'true'.

A continuous while loop (infinite loop) in the "run" method is defined that invokes the ‘getNext’ method to get an incoming message from the message queue and process it if there is any. Otherwise, the ‘getNext’ method will wait until the next available message is ready. The while wait loop with notification is a standard way of synchronizing concurrent processes. The switch statement is used inside the while loop to compare the incoming message that is removed from the head of the queue with all the defined message types. It is always a best practice to have a default statement in a switch statement even if it is an empty statement. If the removed message matches one of the message types, it will then execute the event method that matches this message. If it does not, then the status will be kept assigned as ‘false' and then we will skip the switch statement and execute the ‘if' condition that is now valid. In the ‘if' condition body, there is a comment to indicate that this message will be ignored (i.e. it will not be processed). If a message is ‘ignored' by a state machine, then it does nothing according to UML (there is a comment saying that /* write an error message or raise exception */). In the current implementation of the basic state machine, it returns ‘false.'

A complete sketch of the Java code for the queued state machine model depicted in Figure 4.10 after applying all changes is provided in Appendix B.1.

4.3 Pooled State Machine (PSM) in Umple

A pooled state machine (PSM) has a different semantic as compared to queued state machines because it does not strictly obey FIFO order for consuming the messages passed between state machines. Instead, it removes the first message that matches an event that can be triggered in the current state by searching down all the messages in the queue. The pooled state machine hence resolves the unspecified reception situation.

The design of the pooled state machines also has a number of issues related to the design decisions that we made. If we take a look at the example mentioned above of a state machine that has two states s1 and s2 and two transitions in Figure 4.10, we followed the same steps we took for developing the QSM code generator to derive the implementation from this model.

The pseudocode of what the Java code should look like is shown below:

```
1   public class X implements Runnable
2   {
3     create an enumeration for the types of messages accepted by the
4   state machine: Message type is defined as strings: *event_M*
5     define a Map for the state machine that allows querying which
6   events are possible in each map by defining a set of message types
7   for every state of the state machine
8     declare a message pool
9     declare a event-processor thread
10    constructor {
11      create instance of the message pool
12      start the event-processor thread to remove from pool
13    }
14    // Event method handlers
15    _e1(){
16      check a current state then transitions to the next state
17    }
18    _e2(){
19      check a current state then transitions to the next state
20    }
21    Define an inner class called Message to create object instances of
22  messages by passing their types and parameters as the arguments
23    class Message{
24      constructor with two arguments:  message type and parameters
25    }
27    Define an inner class called MessageQueue that has two
28  synchronized methods for add and remove the events into the message
29  queue
30    *put method* to insert a message into the message queue and then
31  notify other threads.
32    *getNext method* when it is called, it is first called
33  'getNextProcessableMessage' method to search down the messages in
34  the pool and then remove a message from the pool that matches one of
35  the message types for the current state. Otherwise, it returns null.
36  If the message is null, then wait until an event is added to the
37  pool.
38    * getNextProcessableMessage method* to iterate through messages
39  and remove the first message that matches one of the Messages types
40  of the current state the state machine is in, otherwise return null
41  class MessagePool {
42      declare and instantiate a message pool
43      //put method to add the message into the pool
44      synchronized put(Message m)
45      {
46        add a message to the pool and then notify
47      }
48      // getNext method to remove the message from the pool
49      synchronized getNext()
50      {
51        call getNextProcessableMessage method
52        while (message==null) {
53          wait();
54          call getNextProcessableMessage method
55        }
56        return the message
57      }
58
59  // getNextProcessableMessage to iterate through the messages in the
60  pool
61      getNextProcessableMessage()
62      {
```

```
63  iterate through messages and remove the first message that matches
64  one of the message types for the current state in the state machine,
65  otherwise return null
66      }
67    }
68    //messages accepted
69    e1() {
70      add a message to the message pool
71    }
71    e2() {
72      add a message to the message pool
73    }
74    event-processor thread run method {
75      do forever using while(true) loop {
76        check if the message pool is not empty, if yes, remove a
77  message from the head of message pool. Otherwise, waiting for an
78  event to be added into message pool
79        switch (message type) {
80          compare message type with each *even name_M*
81          If it matches:
82            call event method _*name*
83            break
84        }
85      }
86    }
87  }
```

Listing 4.2: Pesudocode for pooled state machine (PSM)

The following issues cover the main points of the design of Java code for the implementation of the above state machine model that applies to pooling logic.

4.3.1 Enumerations of Message Types

In a PSM, we create and define enumerations of messages for all events of the state machines in the same way as for QSM. If one state or substate of the state machine has no regular or timed events, then an extra message type called 'null_M' is defined as the enumerated value in Message Type. For example, the enumeration of messages for the PSM is:

```
//enumeration type of messages accepted by X
enum MessageType { e1_M, e2_M, e3_M, e4_M, null_M }
```

4.3.2 Inner Class: MessagePool Class

The Message class is created for PSM in the same way as in the case of a QSM. To enable a PSM, we add a pooling mechanism to the getNext method in the MessagePool class.

To do that, a HashMap is created in which a set of messages is defined for every state of the machine. Below, we present the sample code for creating and initializing the ‘stateMessageMap' which maps any state to a set of message types (between the () separated by commas).

In the example below, there is one PSM, and each state responds to only one message type except for the final state that has no events. The resulting sets will be very small in this example but could be much larger in a more sophisticated example.

```
1   // Map for a X pooled state machine that allows querying which
2   events are possible in each map
3     public static final Map<Object, HashSet<MessageType>>
4   stateMessageMap = new HashMap<Object, HashSet<MessageType>>();
5     static {
6       stateMessageMap.put(Sm.s1,new
7   HashSet<MessageType>(Arrays.asList(MessageType.e1_M)));
8       stateMessageMap.put(Sm.s2,new
9   HashSet<MessageType>(Arrays.asList(MessageType.e2_M)));
10      stateMessageMap.put(Sm.s3,new
11  HashSet<MessageType>(Arrays.asList(MessageType.e3_M)));
12      stateMessageMap.put(Sm.s4,new
13  HashSet<MessageType>(Arrays.asList(MessageType.e4_M)));
14      stateMessageMap.put(Sm.s5,new
15  HashSet<MessageType>(Arrays.asList(MessageType.null_M)));
16  }
```

If one or more state or substates have no events, either regular or timed, then that state or substate will have ‘null_M’ message. We would use it by simply calling

`stateMessageMap.get(state).contains(msg.type)`

We use a HashMap to define a set of messages for each state of the state machine to store a pair of key and value (Singh, 2013). Note that HashMap does not consider the order for inserting and storing the returned key-values pairs (Singh, 2013).

The ‘getNext’ method calls ‘getNextProcessableMessage’ method to searches down the pool and find the first event that can be responded to in the current state in order to return a message that belongs to set of the messages of the state. The returned message will then be removed from the pool. So, this method is needed to test each event in the pool. If there is no message to be returned, then the pool blocks waiting for another event to be triggered. When a new event is being added to pool while the machine is blocked, the method to search each event in the pool will need to be called to determine if the block should be released.

A sketch of the getNext method to implement the pooling mechanism is:

```
1   public synchronized Message getNext(){
2     Message message=null;
3     try {
4       message=getNextProcessableMessage();
5       while (message==null)
6       {
7         wait();
8         message=getNextProcessableMessage();
9       }
10    } catch (InterruptedException e) { e.printStackTrace(); }
11    // return the message
12    return (message);
13  }
14  public Message getNextProcessableMessage(){
15    // Iterate through messages and remove the first message that
16    // matches one of the Messages list
17    // otherwise return null
18    for (Message msg: messages)
19    {
20      if(stateMessageMap.get(getSm()).contains(msg.type))
21      {
22        //The element to be removed
23        messages.remove(msg);
24        return (msg);
25      }
26    }
27    return null;
28  }
```

4.3.3 Removing a Message From the Pool – 'run' Method

The event-processor thread has to check which state the machine is in. Then it asks the pool to give the next message by providing it with the list of messages that are acceptable for this current state and processes this message. If there were no messages in the pool, it would wait until a message is added to the pool.

A complete sketch of the Java code for the pooled state machine model depicted in Figure 4.10 after applying all changes is provided in Appendix B.2.

To illustrate the semantics of the two types of the state machines; queued and pooled, we run the implementation code of model shown in Figure 4.10 to get the outputs. After that, we compare the results to see the different implementations of these state machines.

The following shows the comparison of queued and pooled state machines by highlighting the main difference of their semantics and by tracing the executions of their Java code.

In QSM, the main thread initiates the events and puts them on the queue. The event-processor thread takes the events of the queue and processes them (note that this may happen concurrently with the main thread producing more events) as follows:

- e1 is triggered and is added to the queue, then the event-processor threads takes e1 off the queue and it is processed leading to move the machine to s2.
- e2 is triggered and is added to the queue (which may happen in parallel with the actions of the event-processor thread above), then the event-processor threads takes e2 off the queue and it is processed to move the machine to s1.
- e2 is triggered and is added to the queue, then the event-processor threads takes e2 off the queue and it is not processed; this event is ignored as it is unspecified reception error.
- e1 is triggered and is added to the queue, and then the event-processor threads takes e1 off the queue and it is processed to move the machine to s2.

In PSM, the main thread initiates the events and puts them into the pool. The event-processor thread takes the events from the pool and processes them as follows:

- ✓ e1 is triggered and is added to the pool, and then the event-processor threads takes e1 from the pool to it is processed leading to move the machine to s2.
- ✓ e2 is triggered and is added to the pool (possibly in parallel with the above), then the event-processor thread takes e2 off the pool and it is processed to move the machine to s1.
- ✓ e2 is triggered and is added to the pool. There is no transition in state s1 to consume e2– therefore it is kept in the pool, and it is not processed at this time.
- ✓ e1 is triggered and is added to the pool. This event can be processed in s1 and the event-processor thread takes e1 off the pool and it is processed it to move the machine to s2.
- ✓ The state machine is in s2 and e2 is still in the pool. This event can be processed by firing the transition leading to s1. Therefore the event-processor thread takes e2 off the pool and it is processed to move the machine to s1.

The trace execution of above Umple examples for the queued, and pooled state machines can be shown in Table 4.3.

Table 4.3: Execution trace of Umple code for basic, queued, and pooled state machines

	Queued State Machine	Pooled State Machine
initial	s1	s1
e1	s2	s2
e2	s1	s1
e2	s1 (e2 is added to the queue, and then it is taken off the queue, but it will be ignored because it is unspecified reception.)	s1 (e2 is unspecified event. It is pooled but it will not be removed from the pool.)
e1	s2	s2 (e2 is taken off the pool and it is processed) s1

4.4 Syntax of Umple Queued State Machine (QSM) and Pooled State Machine (PSM)

In his Ph.D. thesis, Badreddin (2010) gives more details about the Umple state machine grammar. He shows the different notations and symbols being used for defining the syntax of Umple state machines. The grammar has been enhanced since that time as Umple state machines have been extended to have many other features such as history states, events arguments, queued state machines, pooled state machines, and the unspecified reception handler mechanism.

4.4.1 Allowing For a Queued State Machine in Umple (QSM)

In order to allow for QSM in Umple, we first made a minor change to the Umple state machine grammar. We modified it to accept this new feature by adding a keyword 'queued'. As shown in the grammar in Listing 4.3, the state machine can be defined as queued. The '?' symbol indicates that defining a state machine as *queued* is optional. Also, the grammar shows that we can have QSM in a class or standalone to be used by other classes. The notations and symbols used in the Umple state machine grammar are further explained in (Badreddin, 2010) (CRuiSE, 2013).

```
// State machine elements in Umple. See user manual page: BasicStateMachines
stateMachineDefinition : statemachine [=queued]? [=pooled]? [name] {
[[state]]* }
stateMachine : [ enum]]
   | [[inlineStateMachine]]
   | [[referencedStateMachine]]
   | [[activeDefinition]]
inlineStateMachine : [=queued]? [=pooled]? [~name] { ( [[comment]]
   | [[state]]
   | [[trace]]
   | [=||]
   | [[standAloneTransition]])* }
```

Listing 4.3: Umple grammar to specify 'queued' and 'pooled' keywords by CRuiSE, 2013

To specify a state machine to be queued, the user writes the keyword '*queued*,' followed by the name of the state machine, then a block in curly brackets '{}' containing a set of states and events. To define a standalone state machine, the above is preceded by the keyword '*statemachine.*' Inline state machines in Umple are defined inside a class, whereas standalone state machines are defined outside of classes, and later applied to multiple classes.

The examples below illustrate how we define a queued state machine (QSM) in a class (Listing 4.4), and how to define one as a standalone state machine (Listing 4.5). We do not define a class for this state machine.

```
1  class X {
2    queued sm {
3      s1 {
4        e1 -> s2;}
5      s2 {
6        e2 -> s1;}
7    }
8  }
```

Listing 4.4: Example of Umple code for inine queued state machine

```
1  statemachine queued sm{
2    State1{
3      e1 -> State2;}
4    State2{
5      e2 -> State1;}
6  }
```

Listing 4.5: Example of Umple code for stand-alone queued state machine

4.4.2 Allowing for a Pooled State Machine in Umple (PSM)

Likewise, we modified the Umple state machine grammar to allow for the *'pooled'* keyword in both inline and standalone state machines as shown in Listing 4.3.

4.4.3 Writing Queued and Pooled Keywords on the Same Line to Define a State Machine in Umple

Queued and pooled state machines have different semantics; thus, a user must not define a state machine to be queued and pooled at the same time. If the user writes both queued and pooled keywords on the same line before the definition of the state machine as in Listing 4.6, the Umple compiler will detect this error and raise an error message.

```
1  class X {
2    queued pooled sm {
3      s1 {
4        e1 -> s2;}
5      s2 {
6        e2 -> s1;}
7    }
8  }
```

Listing 4.6: An error message raised if a state machine is defined as queued and pooled at same time

4.5 Semantics of Umple Queued State Machine (QSM) and Pooled State Machine (PSM)

The Umple metamodel is built using Umple itself (Badreddin, 2010). Also, the Umple metamodel of a state machine is similar to the UML 2.4.1 meta-model; however, it has some elements that do not exist in the UML 2.4.1 meta-model, and only covers a subset of UML. In his Ph.D. thesis, Badreddin (2010) presents more details about the different components of the Umple metamodel.

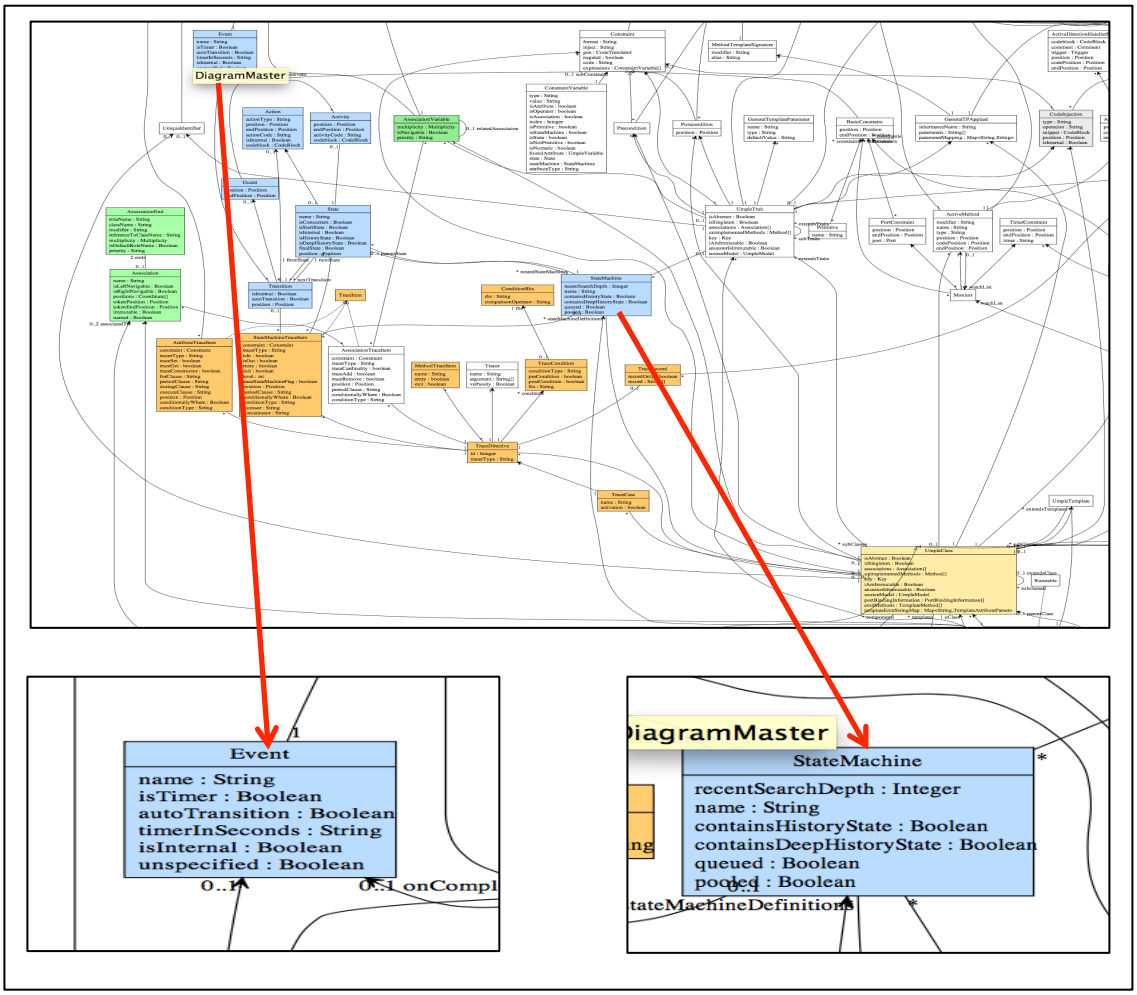


Figure 4.11: Umple state machine metamodel

Figure 4.11 illustrates the Umple metamodel for state machines, and as our work is specifically related to two classes of the Umple metamodel, we take a closer look at them to show the changes that we made.

The focus is on the 'StateMachine' and "Event' classes. Regarding the 'StateMachine' class, we have added two Boolean variables '*queued'* and '*pooled'* to be used to define a state machine as queued or pooled. The state machine can be defined to be queued, pooled or plain. We cannot define a state machine to be both queued and pooled at the same time because each concept has different semantics.

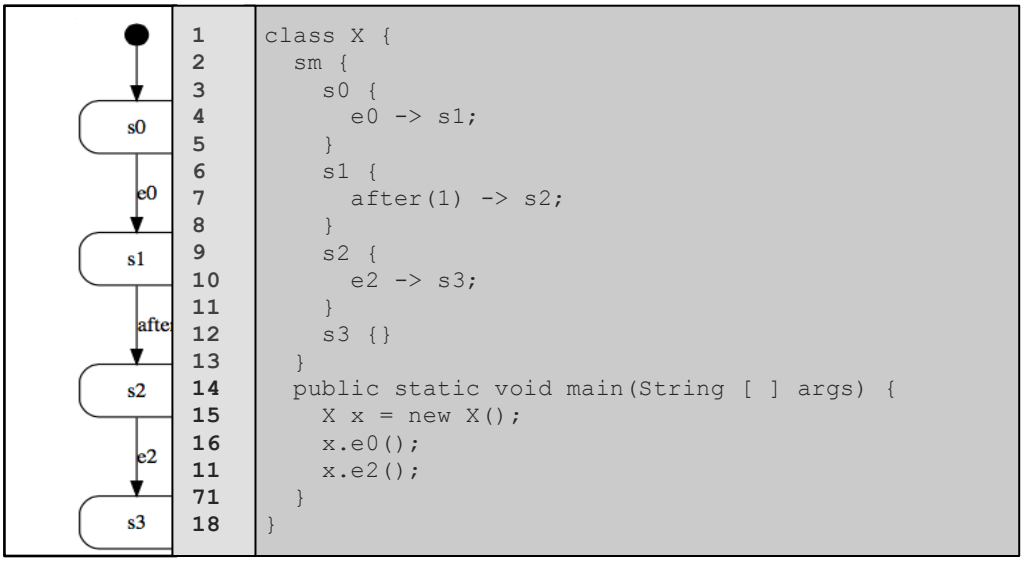
Moreover, we change the 'Event' class to have an 'unspecified' Boolean variable indicating it is a special event and it will be used to handle unspecified reception errors that may occur at various states.

4.6 Common Issues in Umple Queued State Machine (QSM) and Pooled State Machine (PSM)

4.6.1 Issue with Timed Transitions in Queued or Pooled State Machines

To illustrate the semantics of a timed transition in case of QSM and PSM, the examples in Figure 4.12 and Figure 4.13 are provided. The first example in Figure 4.12 shows the case when a state of the machine has only a timed transition. The other example in Figure 4.13 demonstrates the semantics of the state machine when a state has a timed transition and another regular event.

If the execution trace of the example in Figure 4.12 is assumed to be the sequence events <e0, e2>, then unspecified receptions will be detected as follows:




```
1   class X {
2     sm {
3       s0 {
4         e0 -> s1;
5       }
6       s1 {
7         after(1) -> s2;
8       }
9       s2 {
10        e2 -> s3;
11      }
12      s3 {}
13    }
14    public static void main(String [ ] args) {
15      X x = new X();
16      x.e0();
11      x.e2();
71    }
18  }
```

Figure 4.12: Specifying a timed transition in state (s1) without any other events

In a basic state machine: When event 'e0' occurs, the transition is fired leading the machine to state (s1). Then, event 'e2' occurs and it is ignored because the machine is now in the state (s1) and there is no transition to consume 'e2'. The event 'e2' is considered as unspecified event. When the machine enters state (s1), timer starts; it executes for a specific delay (one second). After one second, the timed event is processed: the timer stops, and the machine moves to state (s2) waiting for e2.

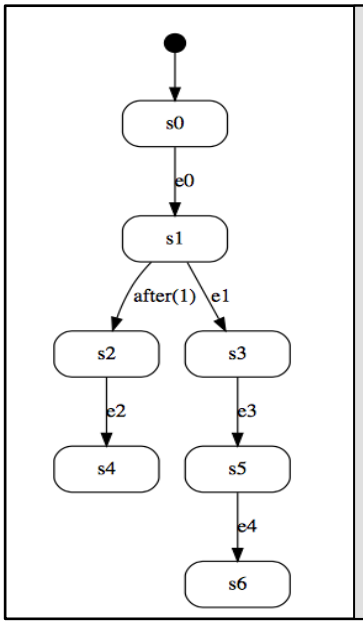
In a queued state machine (QSM): It processes the events in the same order as the basic state machine. When event 'e0' occurs, it triggers the transition causing the

machine to change to state (s1). The event ‘e2’ then occurs and it is added to the queue. It is then removed but it will be ignored and will not be processed because there is no transition to consume ‘e2’. When the machine enters state (s1), timer starts; it executes for a specific delay (one second). After one second, the timed event is added to the queue. It is then removed from the queue and processed: the timer stops, and the machine moves to state (s2).

The semantics of 'after()' transition is that if no event has yet caused a transition, then the state will 'timeout' and takes the specified transition. So even in the queued case, we want to process events, and some will be ignored.

In a pooled state machine (PSM): When the event ‘e0’ occurs, it triggers the transition leading the machine to state (s1). The event ‘e2’ occurs and it is added to the pool. There is no transition to consume ‘e2’; therefore, it is saved in the pool. When the machine enters state (s1), timer starts; it executes for a specific delay (one second). After one second, the timed event is added to the pool. It is then removed from the pool and processed: the timer stops, and the machine moves to state (s2). The machine is now in the state (s2) and it has a transition to consume the saved event ‘e2’. The event ‘e2’ is removed from the pool and is processed, leading the machine to state (s3).

The execution trace of the example in Figure 4.13 is assumed to be the sequence events < e0, e1, e3> as follows:




```
1   class X {
2     sm {
3       s0 {
4         e0 -> s1; }
5       s1 {
6         after(1) -> s2;
7         e1 -> s3; }
8       s2 {
9         e2 -> s4; }
10      s3 {
11        e3 -> s5; }
12      s4 {}
13      s5 {
14        e4 -> s6; }
15      s6 {}
16    }
117   public static void main(String [ ] args) {
118     X x = new X();
19      x.e0();
20      x.e1();
21      x.e3();
22    }
23  }
```

Figure 4.13: Specifying a timed transition in state (s1) with other event (e1)

In a basic state machine: When the event ‘e0’ occurs, the transition is triggered leading the machine to state (s1). The state (s1) has two transitions; timed transition and a state transition. As the machine enters the state (s1), the timer starts but the event ‘e1’ occurs, so the timer interrupts (it cancels) and the event e1 is processed, leading the machine to move to state (s3). The event ‘e3’ occurs and the machine is now in the state (s3). The transition is fired and the machine moves to the state (s5).
In a queued state machine (QSM): It processes the events in the same order as the basic state machine does.
In a pooled state machine (PSM): The event ‘e0’ occurs, it is added o the pool. It is then removed from the pool and processed, leading the machine to state (s1). The state (s1) has two transitions; timed transition and a state transition. When the machine enters the state (s1), the timer starts but the event ‘e1’ occurs. It is added to the pool. It is then removed from the pool and processed. The timer interrupts (it cancels) and the machine moves to state (s3). The event ‘e3’ occurs and the machine is now in the state (s3). It is added to the pool. It is then removed from the pool and the transition is fired. The machine moves to the state (s5).

We identified two problems occurred when implementing timed transitions for all types of machines.

The first problem is that defining timed transitions in nested state machines were not enabled in the original Umple state machine semantics. The original Java code generated from an Umple nested state machine did not work properly in case the substates have timed transitions. The code just had a method called " timeout" and it did not have any handling mechanism to deal with the timer; that is, the code is not generated properly.

The solution for this problem is to modify the Java code generated from the state machine to implementing the timer mechanism.

The second problem is that in case of PSM, if any event is saved in the pool because it is unspecified event, and the timed transition is triggered after a specific delay leading the machine to the target state where the event would be consumed, then this event would not be removed from the pool, and they would not be processed. To

clarify, the event-processor thread will not remove the saved events from the pool when the machine is in the state where the events can be consumed.

To solve this problem, we proposed three different solutions to allow for handling unspecified receptions that occur before the timed events are triggered, in order to overcome this problem.

First Solution: The first alternative is that when the machine enters a state that has a timed transition, the timed event would not be added to the queue/pool. This is the same semantics as an instantaneous transition in case of QSM and PSM.

This solution will not work in the case of PSM because the event-processor thread will not remove the saved events from the pool when there are transitions to consume them. In other words, the saved events will stay in the pool forever.

Second Solution: The second alternative is that the main thread sleeps and waits 'a little longer' time than the timer thread's time delay. Thus, the timer thread calls the timed event to add it to the queue/pool. The event-process thread then removes the timed event form the queue/pool and processes it. Then the main thread wakes up and resumes. This means that any other actions to be done by the main thread are delayed, and the events should supposedly occur in the right order. To make this change, we added " try {Thread.sleep((time*1000)+10);} catch (InterruptedException e) {}" to every ‘startTimeoutHandler' method. This solution seemed to solve the problem, but it still had some issues.

Disadvantage of this solution: In the main thread, we can add Thread.sleep((time*1000)+10) to delay creating an event until the timed transition is triggered. Firstly, we should never rely on sleeping a thread for a measured amount of time because this leads to a race condition. The extra time, for example (time*1000)+10), is not necessary 'enough' as we expect because thread may take longer time and then this will bring us back to the same problem. Secondly, we do not want to delay the main thread. There should not be any artificial delays in the system because it breaks performance. We do, in fact, want the main thread to be able to go on and trigger other events.

For these reasons, we choose the third solution for the implementation of timed transitions in QSM and PSM.

Third Solution: The third alternative is that in case of QSM and PSM, the timer thread calls the timed events to be injected into the queue/pool. This should work totally transparently as if any other thread had done it. In other words, when the timer thread(s) wake up, it calls the timed events. The timed events are then injected into the queue/pool. However, the timer thread(s) should be cancelled if one of the other events in the state with the timer is triggered (processed from the queue) first. This semantics is the same as the timers in SDL. However, SDL timers must be explicitly reset if they should become inactive (SDL-RT, 2013).

In a basic state machine, in normal operation:

- Thread 1 - initiates events and processes them
- Thread 2 - Possible timer thread(s) - each initiates an event at the end of the timeout period, processes it, and then cancels.

In a queued state machine (QSM) or pooled state machine (PSM):

- Thread 1 - initiates events and puts them on the queue
- Thread 2 - Event-processor thread - takes events from the queue/pool and processes them
- Thread 3 - Possible timer thread(s) - should put events on the queue/pool

In the case of basic state machines, there is normally only one thread to initiate the state machine events. If there is more than one thread, there can be undefined behavior due to non-thread-safe code. Also, there can be more than one timer thread sequentially but ideally only one at a time. Umple has not been designed yet to deal with several timeouts at the same time, which could be the case in concurrent states. Users of Umple should avoid this. However, the design of QSM and PSM allows for more than one thread of each type (thread 1, thread 2, and thread 3).

4.6.2 Issue with Instantaneous Transitions in Queued or Pooled State Machine

Instantaneous transitions (also called auto-transitions) in a basic state machine

in Umple are transitions that are processed upon entry to a state that has no events. Instantaneous transitions are supposed to transition automatically from one state to another state immediately after completing entry actions or upon completion of a do activity if the state has 'entry' action or 'do' activity, or simply transition to the other state immediately if there is no 'entry' action, 'do' activity or guards on it. It can have a transition action to be executed when the transition is fired or a guard that may block the transition from happening. To simplify certain logic, the instantaneous transition would be typically done in a state that has no other events. Also, the instantaneous transition has a priority to be fired if a state has other events unless it has a guard that prevents it from occurring.

It is an important design issue to decide what to do in each of the various cases of having instantaneous transitions in QSM and PSM: an instantaneous transition after 'do' activity, an instantaneous transition with guard, and an instantaneous transition with no guard.

Most of the time we use such transitions after a do activity, to initiate a transition after the activity completes. Usually there would be a guard. If there is no guard, there is no way any other event can even get in the queue/pool before the transition is taken.

There are two design options to deal with the case of instantaneous transitions in QSM and PSM. The first option is to allow the instantaneous transitions in case of QSM and PSM and give them a priority to be fired if there are other regular events in the same state, but they are not added to the queue/pool and, as a result, the event-processor thread does not check for them while taking the events from the queue/pool. This means that the instantaneous transitions of QSM or PSM are always taken unless guards preclude them. The second option is to disallow the instantaneous transitions in QSM and PSM.

We implemented the first option because if users explicitly code the instantaneous transitions, then they must have a reason for them. If the state machine is queued or pooled, the instantaneous transitions will not be added to the queue/pool; instead, they would be fired directly.

Chapter 5 Implementation of Queued and Pooled State Machines

5.1 Goals For Code Generation

The Umple QSM and PSM capability allows developers to focus on their models and to satisfy their development needs without editing the underlying generated Java code. The developers simply edit the Umple code to make any necessary changes. Our goal in this research has been to ensure that the generated Java code from Umple code is efficient and satisfies the queued/pooled state machines semantics. Also, we wanted to ensure that the generated code should be readable and understandable, so auditors and developers can easily verify it.

5.2 Example of Queued, and Pooled State Machines in Umple

Below is an example of Umple code containing queued and pooled state machines that we will be using in later discussion.

```
1  //Queued state machine
2  class X {
3   queued sm {
4      s1 {
5        e1 -> s2;
6      }
7      s2 {
8        e2 -> s3;
9      }
10     s3 {
11       e3 -> s1;
12     }
13   }
14   public static void
15 main(String[] args){
16      X x=new X();
17      x.e1();
18      x.e2();
19      x.e3();
20      x.e3();
21      x.e1();
22      x.e2();
23   }
24 }
```

```
1  //Pooled state machine
2  class X {
3   pooled sm {
4      s1 {
5        e1 -> s2;
6      }
7      s2 {
8        e2 -> s3;
9      }
10     s3 {
11       e3 -> s1;
12     }
13   }
14   public static void
15 main(String[] args){
16      X x=new X();
17      x.e1();
18      x.e2();
19      x.e3();
20      x.e3();
21      x.e1();
22      x.e2();
23   }
24 }
```

Listing 5.1: An Umple code example of basic, queued, and pooled state machines

The state machine written in Listing 5.1 has three states: s1, s2, and s3. Each state has an event: state s1 has the event e1 that transitions to s2, state s2 has the event e2 that transitions to s3, and state s3 has the event e3 that transitions to s1.

5.3 Comparisons of Generated Java Code Between Umple Basic, Queued, and Pooled State Machines

Taking the example illustrated in Listing 5.1, we present the different implementations of the three types of state machines. The Umple program starts executing at the main method, where the code is compiled and run with a series of events in order to highlight the differences among the original and proposed semantics. We call the state machine events over and over in a random order for the purpose of validation, and trace the results to see that the system is getting into the correct states at all times. The trace execution of above can be shown in Table 5.1:

Table 5.1: Execution trace of Umple code that is shown in Listing 5.1

	Basic State Machine	Queued State Machine	Pooled State Machine
initial	s1	s1	s1
e1	- s2 (e1 is called).	- s2 (e1 is added to the queue and then removed from the queue and called).	- s2 (e1 is added to the pool and then removed from the pool and called).
e2	- s3 (as above).	- s3 (as above).	- s3 (as above).
e3	- s1 (as above).	- s1 (as above).	- s1 (as above).
e3	- s1 (e3 is ignored because it is unspecified reception).	- s1 (e3 is added to the queue and then removed and ignored because it is an unspecified reception).	- s1 (e3 is unspecified. It is pooled but it will not be removed from the pool.)
e1	- s2 (e1 called).	- s2 (e1 is called and added to the queue and then it is removed from the queue and called).	- s2 (e1 is added to the pool and then removed from the pool and processed; e3 remains on the pool).
e2	- s3	- s3	- s3 (e2 is added, then removed and called) - s1 (e3 is then taken off the pool and called)

Table 5.1 illustrates the different semantics and implementation of basic, queued, and pooled state machines. We see how each state machine behaves when a series of events are triggered.

5.4 Simple QSM and PSM in Umple

The queued and pooled state machines can be defined to be simple state machines that contain a set of states that do not have substates. More details about the syntax and semantics of a simple state machine in Umple are provided in Chapter 4 and in (Badreddin, 2010).

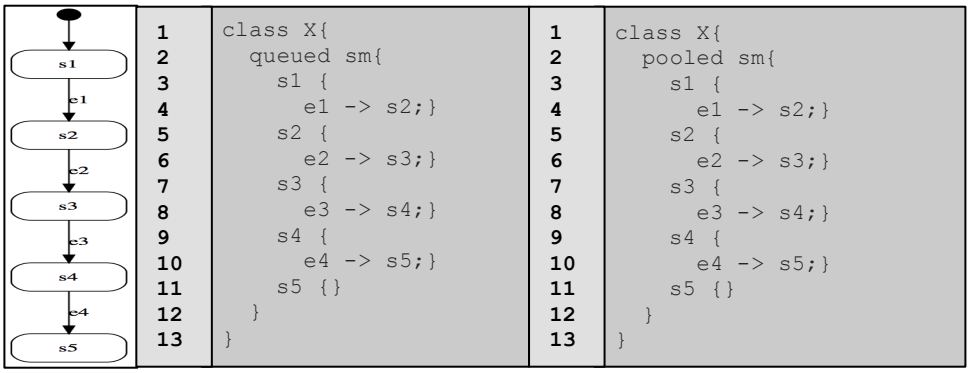
5.4.1 Examples of Simple QSM and PSM in Umple

In the following subsections, we show examples of defining simple state machines with queuing and pooling semantics in Umple. The first part shows the examples of queued and pooled state machines in case where their events do not have parameters, and the other part illustrates cases of queued and pooled state machines in which their events do have parameters. We give a brief description of both examples.

5.4.1.1 Case Where State Machine Events Have No Arguments

The state machine in Figure 5.1 has events with no parameters. There is no special notation in the diagram that indicate that the state machine is queued or pooled. The other team members will take care of adding the notation on the state machine diagram to indicate if it is queued, pooled, or plain. However, in the Umple code, the user has to specify that the state machine is queued or pooled by writing '*queued*' or '*pooled*' keyword, respectively, before the definition of the state machine.

In addition to having no event parameters, the queued and pooled state machines illustrated in Figure 5.1 are simple, meaning that they do not have substates. The state diagram has five states: s1, s2, s3, s4, and s5. Each state has one transition except the final state s5 that does not have any events. The initial state is s1. There are 4 events: e1, e2, e3 and e4.




```
1  class X{
2    queued sm{
3      s1 {
4        e1 -> s2;}
5      s2 {
6        e2 -> s3;}
7      s3 {
8        e3 -> s4;}
9      s4 {
10       e4 -> s5;}
11     s5 {}
12   }
13 }
```

```
1  class X{
2    pooled sm{
3      s1 {
4        e1 -> s2;}
5      s2 {
6        e2 -> s3;}
7      s3 {
8        e3 -> s4;}
9      s4 {
10       e4 -> s5;}
11     s5 {}
12   }
13 }
```

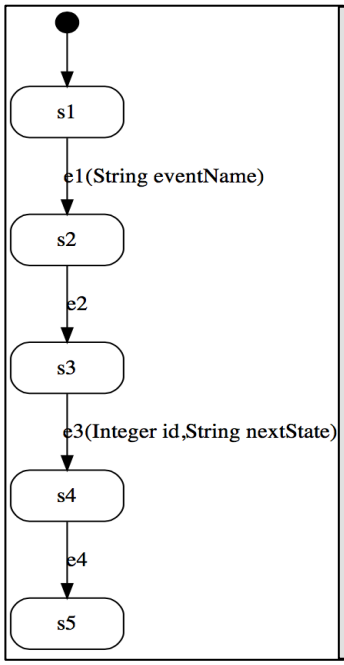
Figure 5.1: A simple state machine (events with no arguments) as shown in UmpleOnline

5.4.1.2 Case Where Some State Machine Events Have Arguments

The examples below are defined in Umple in the same way as defining the above example. The only difference is that the queued and pooled state machines' events in the below examples have parameters.

As shown in Figure 5.2, both events: e1 and e3 have parameters. When the Java code is generated from the Umple code of this diagram, the public event-handling methods that are generated from events e1 and e3 have arguments as follows:

```
Event e1 has the parameter: String eventName
Event e3 has the parameters: Integer id and String nextState.
```




```
1  class X{
2    pooled sm{
3      s1 {
4        e1 (String
5  eventName) -> s2;
6      }
7      s2 {
8        e2 -> s3;
9      }
10     s3 {
11       e3 (Integer
12 id, String
13 nextState) -> s4;
14     }
15     s4 {
16       e4 -> s5;
17     }
18     s5 {
19     }
20   }
21 }
```

```
1  class X{
2    queued sm{
3      s1 {
4        e1 (String
5  eventName) -> s2;
6      }
7      s2 {
8        e2 -> s3;
9      }
10     s3 {
11       e3 (Integer
12 id, String
13 nextState) -> s4;
14     }
15     s4 {
16       e4 -> s5;
17     }
18     s5 {
19     }
20   }
21 }
```

Figure 5.2: A simple queued state machine (events with arguments) as shown in UmpleOnline

5.4.2 Java Code Generation For Simple QSM and PSM

5.4.2.1 Case Where State Machine Events Have No Arguments

The Umple language has already enabled the users to generate code from Umple state machines in Java and C++. In our thesis, we work on generating the pieces of code in Java that support the queuing and pooling logics.

The generated Java code the example shown in Figure 5.1 (the case of the QSM with events lacking parameters) is shown in Appendix B.3.

5.4.2.2 Case Where Some State Machine Events Have Arguments

Java code generated from the example in Figure 5.2 is given in Appendix B.4. The difference can be seen in the pieces of code that are related to the event-handling methods, message-accepting methods, and the 'run' method. More information about the parameters of queued or pooled state machine events is presented in section 5.8.1.

We display the parts of the code that are influenced by using the queued or pooled state machine events' parameters because we want to highlight the difference when generating Java code for the events whether they have parameters or not. First, the Java code generated from events e1 and e3 in Figure 5.2 results in generating the event-handling methods that have the same arguments as the state machine events:

```
1   public boolean _e1(String eventName)
2   {
3     boolean wasEventProcessed = false;
4
5     Sm aSm = sm;
6     switch (aSm)
7     {
8       case s1:
9         setSm(Sm.s2);
10        wasEventProcessed = true;
11        break;
12      default:
13        // Other states do respond to this event
14    }
15    return wasEventProcessed;
16  }
17 }
```

Second, the message-accepting methods generated from these two events are:

```
1   public void e1 (String eventName)
2   {
3     Vector v = new Vector(1);
4     v.add(0, eventName);
5     queue.put(new Message(MessageType.e1_M, v));
6   }
7   public void e2 ()
8   {
9     queue.put(new Message(MessageType.e2_M, null));
10  }
11  public void e3 (Integer id,String nextState)
12  {
13    Vector v = new Vector(2);
14    v.add(0, id);
15    v.add(1, nextState);
16    queue.put(new Message(MessageType.e3_M, v));
17  }
```

Finally, the run method looks a bit different because each message can have one or more parameters based on the number of arguments of the event. A piece of code generated for such purpose is shown below:

```
1   @Override
2     public void run ()
3     {
4       boolean status=false;
5       while (true)
6       {
7         Message m = queue.getNext();
8
9         switch (m.type)
10        {
11          case e1_M:
12            status = _e1((String) m.param.elementAt(0));
13            break;
14          case e2_M:
15            status = _e2();
16            break;
17          case e3_M:
18            status = _e3((Integer) m.param.elementAt(0), (String)
19  m.param.elementAt(1));
20            break;
21          case e4_M:
22            status = _e4();
23            break;
24          default:
25        }
26        if(!status)
27        {
28          // Error message is written or  exception is raised
29        }
30      }
31    }
```

In Chapter 4, we provide more details about the generated Java code of QSMs and PSMs.

5.5 Composite QSM and PSM in Umple

As we discussed above, state machines can be categorized as simple and composite. Composite state machines have nested and/or concurrent states. Events that need to cause effects in every substate of the outer state do not have to be repeated in each of the substates (CRuiSE, 2013).

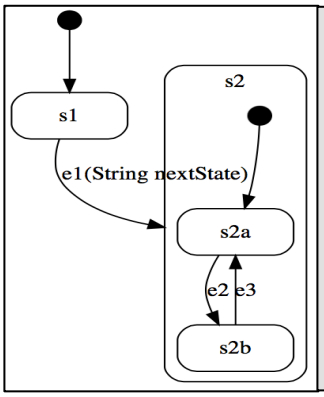
In Umple, we defined multiple concurrent blocks to become active when in a specific state by using a state machine with two substates separated by the || symbol which indicates that both blocks will run concurrently. In Umple, the position is taken if an event is processed in any one of the concurrent substates (or state machines) then it is consumed. Therefore, in a pooled and queued cases, this means that generated code needs to recognize the possibility of being in multiple states at once, and to process an event from the queue/pool if it is received by any of the current states.

In the following subsections, we provide an example that illustrates nesting abstractly in a queued/pooled state machine. The other example shows the case of concurrency for a queued/pooled state machine. We then give samples of the generated Umple code of both examples.

5.5.1 Examples of Composite QSM and PSM in Umple

5.5.1.1 Example of Umple QSM and PSM with Nested States

Figure 5.3 shows that the state machine has two states s1 and s2. The state s2 has two substates; s2a and s2b. Events e1 takes the system into state s2. Event e2 then takes the system into substate s2b and event e3 takes the system into substate s2a.

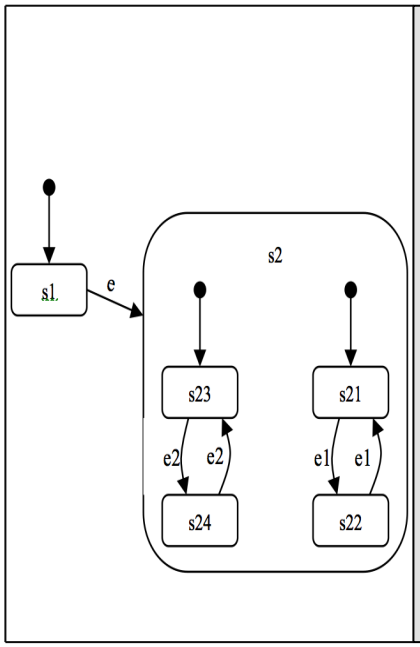



```
1  class X {
2    queued/pooled sm {
3       s1 {
4          e1 (String nextState) -> s2;
5       }
6       s2 {
7          s2a {
8            e2 -> s2b;
9          }
10         s2b {
11           e3 -> s2a;
12         }
13      }
14   }
15 }
```

Figure 5.3: A state diagram for a queued/pooled state machine with nested states as shown in UmpleOnline

5.5.1.2 Example of Umple QSM and PSM with Concurrent States

Figure 5.4 illustrates the case of having concurrent states for the queued or pooled state machine. The state machine has two states: s1 and s2. The state s2 has four concurrent substates: s21, s22, s23 and s24.




```
1  class X {
2    queued/pooled sm {
3      s1 {
4       e-> s2;
5      }
6      s2 {
7        s21 {
8          e1 -> s22;
9        }
10       s22 {
11         e1 -> s21;
12       }
13       ||
14       s23 {
15         e2 -> s24;
16       }
17       s24 {
18         e2 -> s23;
19       }
20     }
21   }
22 }
23
24
```

Figure 5.4: A state diagram for a queued/pooled state machine with concurrent states as shown in UmpleOnline

The Umple code shown in Figure 5.4 illustrates that the queued/pooled state machine defined here has concurrent substates that will run concurrently when the event e1 and e2 are called.

5.5.2 Java Code Generated From Umple Composite QSM and PSM Examples

We aim to show that generated Java code for both cases mentioned above will have the same style of the generated Java code described in Chapter 4.

By having a closer look at specific aspects of the Java code generated from the examples in Figure 5.4, we are concerned with the generation of enumerations of the messages types for both states and substates: In a QSM, it looks like the following:

```
//enumeration type of messages accepted by X
enum MessageType { e_M, e1_M, e2_M }
```

In a PSM, it looks like the following:

```
//enumeration type of messages accepted by X
enum MessageType { e_M, e1_M, e2_M, null_M }
```

It shows that because the state s2 has no events, there is an extra message 'null_M' declared as an enumerated value.

In addition, we declare a set of messages for each state using a HashMap:

```
1   // Map for a X pooled state machine that allows querying which
2   events are possible in each map
3   public static final Map<Object, HashSet<MessageType>>
4   stateMessageMap = new HashMap<Object, HashSet<MessageType>>();
5   static {
6     stateMessageMap.put(Sm.s1,new
7   HashSet<MessageType>(Arrays.asList(MessageType.e_M)));
8     stateMessageMap.put(Sm.s2,new
9   HashSet<MessageType>(Arrays.asList(MessageType.null_M)));
10    stateMessageMap.put(SmS21.Null,new
11  HashSet<MessageType>(Arrays.asList(MessageType.null_M)));
12    stateMessageMap.put(SmS21.s21,new
13  HashSet<MessageType>(Arrays.asList(MessageType.e1_M)));
14    stateMessageMap.put(SmS21.s22,new
15  HashSet<MessageType>(Arrays.asList(MessageType.e1_M)));
16    stateMessageMap.put(SmS23.Null,new
17  HashSet<MessageType>(Arrays.asList(MessageType.null_M)));
18    stateMessageMap.put(SmS23.s23,new
19  HashSet<MessageType>(Arrays.asList(MessageType.e2_M)));
20    stateMessageMap.put(SmS23.s24,new
21  HashSet<MessageType>(Arrays.asList(MessageType.e2_M)));
22  }
```

As shown above, each state has a list of messages, except for the state s2 that does not have an event. It has a message instead 'null_M.'

The message-accepting methods generated for the events defined in the state and substates are shown below:

```
1   //-----------------------------
2   //messages accepted
3   //-----------------------------
4     public void e (){
5       pool.put(new Message(MessageType.e_M, null));
6     }
7     public void e1 (){
8       pool.put(new Message(MessageType.e1_M, null));
9     }
10    public void e2 (){
11      pool.put(new Message(MessageType.e2_M, null));
12    }
```

The run method will be generated to take all events defined in states and substates of the queue/pool. All the events defined in the substates are treated in the same way as the events defined in the outer states.

5.6 Code Generation Templates of Queued and Pooled State Machines

We use Java Emitter Templates (JET) technology in Umple in order to determine what the generated code should look like. Then, we compile the JET templates into Java code which in turn generates the code in different base languages, resulting in an instance of the Umple metamodel (Badreddin, 2010).

For each base programming language supported by Umple, there is a set of JET templates. All JET templates for Java code can be found at: http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2FUmpleToJava%2Ftemplates

In Table 5.2, we provide a summary of new key templates added to the JET files in order to support the generation of Java code from the queued and pooled state machines.

Table 5.2: Key Jet templates for generation of queued and pooled state machines

Template name (*.JET)	Function
queued_state_machine_queuedEvent.jet	This is a new template added to create state machine events handling methods in the case of queued or pooled state machines. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_queuedEvent.jet
queued_state_machine_removalThread_run.jet	This new template is added to handle the generated code for the run method of the event-processor thread. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_removalThread_run.jet
queued_state_machine_inner_class.jet	This new file is added to handle the code for the Message and MessageQueue/MessagePool inner classes. These classes will be generated if the queued or pooled state machine is defined. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_inner_class.jet

5.7 Test-Driven Development of QSM and PSM in Umple

We follow an agile approach that includes test-driven development (TDD) to extend the original semantics of Umple state machines in order to apply the queuing and pooling semantics for the state machines' events.

To follow TDD, we first write examples of state machines in Umple in which we use these new features. These examples are categorized into simple and composite queued/pooled state machines. Then we generate Java code from those models. After that, we modify the generated code by adding all parts we need such as inner classes and so on. Finally, we end up with the final version of Java code for QSM and PSM. Now, we follow TDD to do all these modification in Umple by extending the syntax and semantics of Umple state machine and enabling the generator to generate Java code as we expect.

The following subsections provide an overview of each level of the testing process of the Umple compiler.

5.7.1 Parser Testing

The parsing testing is the first level of the testing process, in which the parser verifies that the Umple code is parsed and tokenized correctly (Badreddin, 2010). As illustrated in Figure 5.5, Forward (2010) summarized the steps of the process of testing the Umple parser.

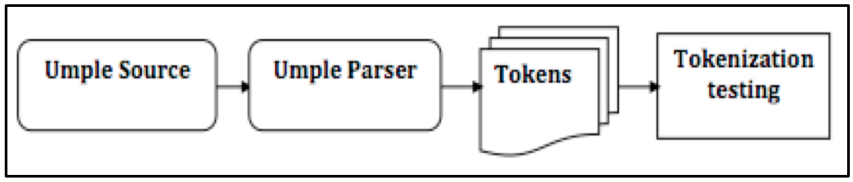


Figure 5.5: Process of the Umple parsing test

For the purpose of supporting the '*queued'* and '*pooled'* features in Umple, we first modify the Umple state machine grammar to accommodate these change by adding the keywords *'queued'* and *'pooled*.' To do that, we change a file called 'umple_state_machines.grammar' in '*cruise.umple*' project to add the keywords '*queued'* and *'pooled'* to the following Umple state machine grammar rules: *stateMachineDefinition* and *inlineStateMachine*. We have to add the '*queued'* and

'pooled' keywords to the *stateMachineDefinition* rule that allows for declaring a state machine independently of the class. Also, we add the *queued* and *pooled* keywords to the *inlineStateMachine* rule that allows for declaring a state machine inside a class.

After that, we need to add separate test cases to ensure that the modifications we made for both QSM and PSM are parsed correctly into required tokens. The reason for this is because we will use these tokens to populate the Umple metamodel related to QSM and PSM. The Umple code to be parsed for QSM is shown in Listing 5.2 as:

```
1   class X
2   {
3     queued sm
4     {
5       s1 {
6         e1 -> s2;
7       }
8       s2 {
9         e2 -> s1;
10      }
11    }
12  }
```

Listing 5.2: Umple code for a queued state machine

To parse the above Umple code, we need to add a test to the '*UmpleParserStateMachineTest.java',* which is the key file for parsing Umple state machine elements:

```
1   public void queuedStateMachine()
2   {
3   assertParse("108_queuedStateMachine.ump","[classDefinition][name:Que
4   uedSM][stateMachine][inlineStateMachine][queued:queued][name:sm][sta
5   te][stateName:State1][transition][event:e1][stateName:State2][state]
6   [stateName:State2][transition][event:e2][stateName:State1]");
7     UmpleClass c = model.getUmpleClass("QueuedSM");
8     StateMachine sm = c.getStateMachine(0);
9     Assert.assertEquals(true,sm.isQueued());
10    Assert.assertEquals("sm", sm.getName());
11    Assert.assertEquals(2, sm.numberOfStates());
12    State state1 = sm.getState(0);
13    Assert.assertEquals("State1", state1.getName());
14    Assert.assertEquals(1, state1.numberOfTransitions());
15    Transition t1 = state1.getTransition(0);
16    Event event1 = t1.getEvent();
17    Assert.assertEquals("e1", event1.getName());
18    State state2 = sm.getState(1);
19    Assert.assertEquals("State2", state2.getName());
20    Assert.assertEquals(1, state2.numberOfTransitions());
21    Transition t2 = state2.getTransition(0);
22    Event event2 = t2.getEvent();
23    Assert.assertEquals("e2", event2.getName());
24  }
```

Listing 5.3: Parser test for a queued state machine

When this test runs, the parser analyzes the above Umple text and generates the tokens.

Currently, there are about 13 test cases that are related to testing the parsing for QSM and PSM, each covering different issues. You can find those test cases at:

http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2Fcruise.umple%2Ftest%2Fcruise%2Fumple%2Fcompiler

5.7.2 Metamodel Testing

At this level, we verify that the metamodel of Umple is being populated correctly with different Umple constructs. It ensures that Umple maintains an accurate internal representation for the input Umple model; ensuring that the input model, after being correctly parsed, populates the correct elements into an instance of the metamodel (Badreddin, 2010).

Regarding QSM and PSM, we adapt the Umple state machine metamodel to detect the keywords '*queued*' and '*pooled*' by adding the following code to the Metamodel in the key file '*StateMachine.ump*':

```
1   Boolean queued = false;
2   Boolean pooled = false;
```

Listing 5.4: Umple metamodel (StateMachine class)

Then, we change the parser to set these variables to 'true' when the '*queued*' or '*pooled*' variable is encountered. The file '*UmpleInternalParser_CodeStateMachine.ump*' is responsible for processing state machine statements when they are parsed. It analyzes and populates the required Umple element in the metamodel, which in this case are the keywords *'queued'* and '*pooled*.' Below are the pieces of code added to the method called '*populateStateMachine*' in the '*UmpleInternalParser_CodeStateMachine.ump*':

```
1   if (stateMachineToken.is("queued"))
2   {
3     sm.setQueued(true);
4   }
5   …
6   if (stateMachineToken.is("pooled"))
7   {
8     sm.setPooled(true);
9   }
```

Also, we add these pieces of code to the 'analyzedReferencedStateMachine' method called in UmpleInternalParser_CodeStateMachine.ump, which helps analyzing the state machine tokens, and when queued or pooled is detected, it will set the queued or pooled variable to true. This is because we can define a state machine inside a class, and also can define a standalone state machine.

After that, we add JUnit tests to '*StateMachineTest.java'* to check that the keywords *queued* and *pooled* are populated correctly.

```
1  if (stateMachineToken.is("queued")){
2    sm.setQueued(true);
3  }
4  …
5  if (stateMachineToken.is("pooled")){
6    sm.setPooled(true);
7  }
```

Also, we modify the tests we have added to '*UmpleParserStateMachineTest.java'* to parse the keywords '*queued'* and '*pooled'* by adding the following code:

```
1  Assert.assertEquals(true,sm.isQueued());
```

```
1  Assert.assertEquals(true,sm.isPooled());
```

In addition to documenting how the system behaves under normal conditions, it is important to document how it behaves in abnormal scenarios (Forward, 2010). One must document the situation when preconditions are not satisfied. For instance, a test case in Listing 5.5 was written to detect the situation when a QSM has no events which results in a warning notification indicating that there are no events to be queued.

```
1  @Test
2  public void queuedStateMachine_pooledStateMachine_noEvents(){
3    assertHasWarning("106_queuedStateMachine_noEvents.ump", 0, 56, new
4  Position("106_queuedStateMachine_noEvents.ump", 2, 2, 16));
5    assertHasWarning("106_queued_nestedSM_noEvents.ump", 0, 56, new
6  Position("106_queued_nestedSM_noEvents.ump", 3, 2, 24));}
```

Listing 5.5: Queued state machine with no events metamodel test

Error and Warning messages are emitted by using 'setFailedPosition' and 'assertHasWarning' methods that take as arguments the message number, the location in the parse results and any positional parameters to substitute into the error or warning message (Lethbridge, Forward & Badreddin, 2012).

5.7.3 Template or Code Generation Testing

The input to the code generator is a populated instance of the metamodel, and the output is the base language such as Java or another base language (Almaghthawi, 2013). The third level of the Umple testing process is template testing, otherwise known as code generation testing. This level of testing is used to check that the generated code in the target base language, such as Java, C++, PHP, or Ruby, matches exactly what we expect. Creating this kind of tests makes sure that the generated code is syntactically correct according to the base language's syntax. Umple currently supports various base languages such as Java, C++, PHP, and Ruby. There is a specific test suite for each target language to verify the correctness of the language syntax (Badreddin, 2010).

The expected target base language we are concerned with is Java; the other languages are outside of our scope. In order to generate the expected Java code for QSM and PSM, we have to deal with different aspects of that generated code; therefore, we go over several steps. The proposed code that realizes the semantics of queued and pooled state machines is explained in Chapter 4.

These template Jet files are found in the '*UmpleToJava'* directory. Java Emitter templates (Jet) are used to specify how Java code or code in any another language should be generated. '*UmpleToJava'* is used to generate Java code that is specified using these Jet files. These templates are compiled in Eclipse by Jet, generating Java source.

The tables show all templates that are modified or changed to reach this goal, and also the modifications that have been done are listed in Appendix C.1.

In addition, we also made the required change to '*Generator_CodeJava.ump'* which is the core code used to generate the Java code for Umple; it is located at: http://code.google.com/p/umple/source/browse/trunk/cruise.umple/src/Generator_CodeJava.ump

In order to generate the Java library for Thread and Timer in case of *queued* or *pooled* state machines and to avoid any duplication that may occur, we add the following piece of code to the '*Generator_CodeJava.ump'* file.

```
1   if (hasTimedEvents)
2   {
3     if(!foundQueued && !foundPooled) {
4       aClass.addDepend(new Depend("java.util.*"));
5     }
6   }
7   for (StateMachine sm : aClass.getStateMachines())
8   {
9     if (sm.isQueued() || sm.isPooled())  {
10      genClass.addMultiLookup("import", "java.util.*");
11      genClass.addMultiLookup("import", "java.lang.Thread");
12    }
13  }
```

Also, we write pieces of code for generating a list of event message types for both queued and pooled state machines.

After doing all these changes, now we should test them to see if the target generated code in Java for queued or pooled state machines matches what we expect. In particular, the generated code should be syntactically correct.

For this purpose, we add the following JUnit tests to StateMachineTest.java:

```
1   @Test
2   public void queuedStateMachine()
3   {
4   assertUmpleTemplateFor("queuedStateMachine.ump",languagePath +
5   "/queuedStateMachine."+ languagePath +".txt","Course");
6   }
7   @Test
8   public void pooledStateMachine()
9   {
10  assertUmpleTemplateFor("pooledStateMachine.ump",languagePath +
11  "/pooledStateMachine."+ languagePath +".txt","Course");
12  }
```

Listing 5.6: Template tests for queued and pooled state machines

These tests check that each file of the target base language generated from the Umple code conforms to the syntax of the expected base language. As mentioned, we are working with Java, so other languages are outside of our scope.

For instance, the expected generated code for example model for QSM depicted in Chapter 4 if the target language is Java is listed in Appendix B.1.

All templates for java can be found at:
http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2FUmpleToJava%2Ftemplates

There are about 42 code-generation test cases that are written to ensure that code generations from QSM and PSM are as we expect. They can be found at:

http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2Fcruise.umple%2Ftest%2Fcruise%2Fumple%2Fstatemachine%2Fimplementation%2Fjava

5.7.4 Language-Oriented Semantic or Testbed Testing

This is the final step of the Umple testing process; called the testbed testing or semantic testing. This level is performed by doing more extensive tests to ensure that the compiled code behaves properly and to verify its semantics (Badreddin, 2010). In other words, these tests verify that the generated code can compile under the language compiler and execute properly. The reason for making these tests is to ensure the generated code works properly. In order to add this level of testing, Umple currently provides sample applications for each base languages supported by Umple, by creating an independent testing project for each language referred in Umple as a 'testbed' to validate the semantics of systems built using Umple.

To do semantic testing of pooled and queued state machines, we add small programs to the source folder of the testbed project in Java that implement the new features: '*queued'* and *'pooled*.'

We create Umple files called '*TestHarnessQueuedStateMachine.ump'* and '*TestHarnessPooledStateMachine.ump'* that contain multiple tests for queued and pooled state machines.

The Umple example seen in Listing 5.2 demonstrates a simple generated system test of a queued state machine. This system is fed as an input to Umple, and the test is performed on the generated system.

After compiling this Umple program, we get the generated code in Java illustrated in Appendix B.1.

Then we add the JUnit tests to test the behavior in '*QueuedStateMachineTest.java*' in the test folder '*cruise.queued.statemachine.test*.' It shows that our program, after compilation, behaves properly as we expect.

The testing of the above example can be performed by feeding the system with a number of events that are added to a queue, and then they are taken off the queue one by one and processed to transition to the expected states.

For example, the above system can be fed the following events:

```
e1
e2
e2
e1
```

This sequence of events is tested by the test written in JUnit4 syntax listed in Appendix D.1.. The '@ test' means that we conducted two tests: one to check that the number of messages values in 'MessageType" is equivalent to the number of events of the state machine, and the other test to check that the process of the events is done as we expect.

In the semantic tests of queued state machines, we have different aspects of generated Java code that need to be checked to ensure that the machine behaves properly.

First, we check that the number of messages in the message types is equal to the number of events in the state machine without duplication and without counting the auto-transition and '*unspecified*' events. Also, we test the processing of the events to make sure that each event is processed as it is taken off the queue, the events are processed in FIFO order, and also to check that there are no events left in the queue.

If the tests pass, we obtain more confidence that the generated system behaves as expected. But if it fails, then we have to investigate the cause of failure and resolve it following the steps used to detect and resolve any defect that may be uncovered in the Umple compiler as described in Forward (2010).

We also add some semantic tests for the behavior of systems that have pooled state machines. An example of one of these semantic tests is shown in Appendix D.2.

In the semantic tests for pooled state machines, we also test different aspects of generated Java code which helps us make sure that our code behaves in the correct way semantically. First, we check that the size of messages in message types is equal to the

number of events in the state machine without duplication and without counting the auto-transition (as in the case of queued state machines, except that we could have 'null_M' message types if one or more states or substates do not have any event). In addition, we compare the number of states or substates to the number of keys in the stateMessageMap. We also check that every state of the state machine is put in the stateMessageMap. Moreover, we check that every state and substate in the state machine has its own set of messages. Also, we test the processing of the events to make sure that each event is processed as it is taken off the queue, and the events are processed in the correct order. We also test the case where there is unspecified reception by checking that the event is saved in the pool until it becomes ready to be taken and consumed. We also check the number of events that are left in the pool by the end of the process.

There are about 32 semantic tests that can be found at:

http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2Ftestbed%2Ftest%2Fcruise%2Fqueued%2Fstatemachine%2Ftest

http://code.google.com/p/umple/source/browse/#svn%2Ftrunk%2Ftestbed%2Ftest%2Fcruise%2Fpooled%2Fstatemachine%2Ftest

5.8 Various Issues Related to Java Code Generation of Simple and Composite QSM and PSM

There are number of different design issues related to queued and pooled state machines' generated code in Java. We study and investigate the various alternatives to resolve them and then we make a decision to select the best one among them according to some criteria we have established to compare between them.

In the following subsections, we discuss a list of design issues, discuss the possible solutions of these issues depending on some criteria, and finally explain the reasons for choosing a specific solution.

5.8.1 Queued or Pooled State Machine's Events with Arguments

A state machine's event can have parameters with any valid data types. The

generated code from the event of state machine should have these parameters. In fact, the values of the parameters can be used in guard conditions and/or transition actions.

The semantics of event arguments was specified and implemented in Umple state machine events. However, we noticed that Umple originally did not parse the arguments to events correctly which led to 'bad' code being allowed to be passed through as event parameters. Therefore, we needed to make Umple parse type and name of the parameter, with multiple parameters separated by commas. This semantics is allowed for basic, queued, and pooled state machines.

In the case of a QSM, we need to be able to parse an event's parameters, so they can be added to the vector in a message method where the event is added to the queue. Thus, QSM can work with events with no parameters as well as events with parameters. For instance, we have the following Umple code for the QSM:

```
1   class X
2   {
3     queued sm{
4       s1 {
5         e1 (String name) -> s2;
6       }
7       s2
8       {
9         e2 (Integer id, String name) -> s1;
10      }
11    }
12  }
```

Listing 5.7: Example of the queued state machine events with parameters

The piece of Java generated code for the above Umple code related to message methods shows the event e1 has one parameter called *name*, and the other event e2 has two parameters: one Integer called *id*, and the other is string called *name*.

```
1   //------------------------------
2   //messages accepted
3   //------------------------------
4   public void e1(String name {
5      Vector v = new Vector(1);
6      v.add(0, name);
7      queue.put(new Message(MessType.e1_M, v));
8   }
9
10  public void e2(Integer id, String name){
11     Vector v = new Vector(2);
12     v.add(0, id);
13     v.add(1, name);
14     queue.put(new Message(MessType.e1_M, v));
15  }
```

If the events arguments are written in bad formatting, then a warning message should be raised to indicate that there is a mistake in state machine syntax, and it could not be processed and then the state machine will be considered as extra code that means it will not be generated. For example, if an event argument is declared as following:

```
1   class X{
2     sm{
3       s1 {
4         e (S tring name) -> s2;
5       }
6       s2
7       {
8         e1 (Integer id) -> s1;
9       }
10    }
11  }
```

Warning message because this argument is written in bad formatting

The warning message will indicate that there is a mistake in line 4.

Also, an error message should be raised when an event parameter is inconsistent with the previous declaration of the same event.

```
1   class X{
2     sm{
3       s1 {
4         e (String name) -> s2;
5       }
6       s2
7       {
8         e (Integer id) -> s1;
9       }
10    }
11  }
```

The error message will indicate that there is an error in line 9.

To do changes on event arguments, we went through the following steps:

1. Adding a new 1->* association in the Umple metamodel from Event class to MethodParameter class representing the set of parameters called 'params.'

In ‘StateMachine.ump’, we wrote the following code:

```
1   // The event parameters.
2   1 -> 0..* MethodParameter params;
```

2. Modifying the State Machine grammar and passing code to parse method parameters more precisely; instead of populating the 'args' attribute as a String,

which was the original semantics, populating links of the new association. In 'umple_state_machines.grammar', we changed it to the following:

```
eventDefinition- : [[afterEveryEvent]] | [[afterEvent]] | [~event] (
[[parameterList]] )?
```

3. Getting rid of the 'args' attribute, and adding manually a method 'getArgs' that will return the same string as it did before (by scanning the params), so we do not need to change the code elsewhere in the system that existed before, such as in the diagram generator, etc.

```
public String getArgs(){
  String args="";
  String paramName="";
  String paramType="";
  Sring aSingleParameter="";
  String isList="";
  String parameters = "";
  String finalParams= "";
  if(this.hasParams() == true){
    for (MethodParameter aEventParam : this.getParams()){
      paramName = aEventParam.getName();
      paramType = aEventParam.getType();
      isList = aEventParam.getIsList() ? " [] " : " ";
      aSingleParameter = paramType + isList + paramName;
      parameters += aSingleParameter + ",";
    }
    finalParams = parameters.substring(0, parameters.length()-1);
    args=finalParams;
  }return args;
}
```

Also, we change UmpleInternalParser_CodeStateMachine.ump to allow for parsing the tokens of event parameters. And adding a parser test case (100_eventWithArgument.ump) to test if Umple parsing tokens correctly and another test case to raise an error message with inconsistent event arguments:

```
class LightFixture {
  Integer brightness = 0;
  bulb{
    Off{
      turnDimmer(Integer lightval) /{setBrightness(lightval)} -> Off;
         flipSwitch -> Dimmed;
       }
       Dimmed{
       entry[dimmer > 99] -> On;
         flipSwitch -> Off;
       turnDimmer(Integer lightval) /{setBrightness(lightval)} -> Dimmed;
       }
       On{
         flipSwitch -> Off;
       turnDimmer(Integer lightval) /{setBrightness(lightval)} -> Dimmed;
       }
  }
}
```

Listing 5.8: ' eventWithArgument.ump' Umple test

```
1   @Test
2   public void eventWithArgument()
3   {
4   assertParse("100_eventWithArgument.ump","[classDefinition][name:Ligh
5   tFixture][attribute][type:Integer][name:brightness][value:0][stateMa
6   chine][inlineStateMachine][name:bulb][state][stateName:Off][transiti
7   on][event:turnDimmer][parameterList][parameter][type:Integer][name:l
8   ightval][action][code:setBrightness(lightval)][stateName:Off][transi
9   tion][event:flipSwitch][stateName:Dimmed][state][stateName:Dimmed][t
10  ransition][event:entry][guard][numExpr][constraintName][name:dimmer]
11  [moreOp:>][number:99][stateName:On][transition][event:flipSwitch][st
12  ateName:Off][transition][event:turnDimmer][parameterList][parameter]
13  [type:Integer][name:lightval][action][code:setBrightness(lightval)][
14  stateName:Dimmed][state][stateName:On][transition][event:flipSwitch]
15  [stateName:Off][transition][event:turnDimmer][parameterList][paramet
16  er][type:Integer][name:lightval][action][code:setBrightness(lightval
17  )][stateName:Dimmed]");
18    UmpleClass c = model.getUmpleClass("LightFixture");
19    StateMachine sm = c.getStateMachine(0);
20    State off = sm.getState(0);
21    Event turnDimmer = off.getTransition(0).getEvent();
22    Event flipSwitch = off.getTransition(1).getEvent();
23    Assert.assertEquals("Integer lightval", turnDimmer.getArgs());
24    Assert.assertEquals("", flipSwitch.getArgs());
25    }
```

Listing 5.9: Parser test for generating tokens

```
1   @Test
2     public void eventsWithInconsistentArguments(){
3
4   assertFailedParse("100_eventWithInconsistentArguments.ump", new
5   Position("100_eventWithInconsistentArguments.ump", 13,8,248), 51);
6     }
```

Listing 5.10: Parser test

Since Jet files related to state machine events have been already changed to enable having parameters for state machine's events, we still need to change Jet files related to QSM and PSM events to do the required change which are:

- queued_state_machine_inner_class.jet
- queued_state_machine_queuedEvent.jet
- queued_state_machine_removalThread_run.jet

Also, we add a set of test cases to check that actual generated code matches expected generated code. In StateMachineTest.java, a set of test cases is:

- eventWithArguments.ump
- twoEventsWithArguments.ump

In StateMachineTest.java, set of QSM events' parameters test cases are:

- queuedStateMachine_withParameters.ump
- queuedStateMachine_withParameters_1.ump

In StateMachineTest.java, a QSM events' parameters test case is:

- pooledStateMachine_withParameters.ump

5.8.2 Timed Transitions in QSM and PSM

In subsection 4.6.1, we give an explanation of the semantics of a timed transition specified in Umple state machine whether it is basic, queued, or pooled. We also identify two problems of specifying the timed transition in a nested state machine and in a pooled state machine.

The first problem, as mentioned in Chapter 4, arises because improper generation of Java code from the nested state machine that have timed transitions specified in its substates.

First, we do the required changes on the following Jet files; so that, the timer mechanism is generated properly in Java, and the generated code from the state machine is compiled and runs.

- members_AllHelpers.jet: to generate helper variables if a state machine has nested states.
- state_machine_Event_StartStopTimer_NestedStates.jet: to allow for generating start and stop methods for timers for the case of nested state machines.
- state_machine_timedEvent_All.jet: this is a generic file that calls the other jet files related to the timer.
- state_machine_timedEvent_Handler.jet: This is generic jet file that calls the jet files related to the run method of the timer thread.
- state_machine_timedEvent_run_NestedStates.jet: to generate the run method of the timer thread for the case of a nested state machine.
- Generator_CodeJava.ump: to generate a library "java.util.*" for the state machine if it is not queued or pooled.

Second, we write a set of test cases in the ‘StateMachineTest.java’ file for generated Java code to make sure that the actual generated code matches syntactically the expected code.

The second problem is that in case of PSM, if any event is saved in the pool because it is an unspecified event, and the timed transition is triggered after a specific delay leading the machine to the target state where the event would be consumed, then this event would not be removed from the pool, and they would not be processed. In other words, the event-processor thread does not work properly to get the saved event from the pool when there is a transition to consume it.

We proposed three different solutions to solve this problem, and we choose the third solution as the best alternative to fix this problem. Therefore, we make some changes on the generated Java code to apply this solution.

Details about how we make these changes and the parts that were affected by these changes are as follows. In the case of queued and pooled state machines, we created a message in the ‘MessageType' enumeration with name of a timed event followed with ‘_M.' In the case of pooled state machine, we added this message to the set of the states that have the timed transitions. In the case of queued and pooled state machines, we added the '_' underscore symbol before the name of the timed events methods. Once the state machine is in the state that has the timed transition, the timer starts and the timer thread wakes up, and it injects the timed event into the queue/pool. When the timed event is taken off the queue/pool to be processed. The timer stops after a specific delay or when the message is inserted and then transitions to the next state. If the state that has the timed transition contains other events, and one of the events is called, the timer is cancelled, and a timed event is not processed.

This requires the following changes on Jet files to generate Java code in a proper way:

1. Member of state machine jet file: we add a timed event message to the list (queue/pool state machine).
2. Event of state machine jet file: we add '_' symbol before the timed event name.

3. Event of Queued/Pooled state machine jet file: we add a method to inject the event in the queue/pool.
4. We call this method in the run() of timer class.
5. We add code to the event-processor thread to allow for taking timed event message off the queue/pool.

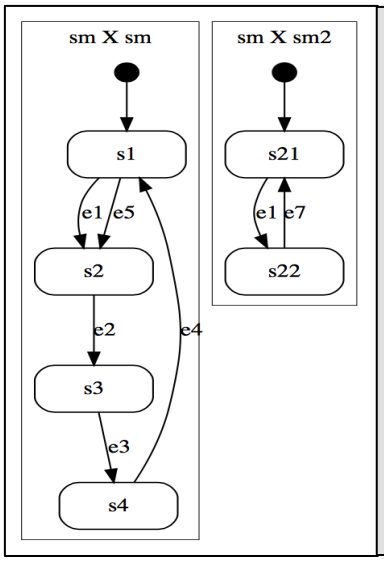
Then, we created four test cases in the 'StateMachineTest.java' directory to ensure that the Java code is generated correctly.

5.8.3 Single Event Causing Multiple Transitions in Nested or Concurrent States of Queued or Pooled State Machines

If a QSM or PSM has an event that is defined in one or more nested or concurrent states, then in generated Java code, this event would not be duplicated in the enumerations of the message types, public message accepting methods, or run method.

5.8.4 Single Event Causing Multiple Transitions in Multiple Queued or Pooled State Machines in the Same Class

Figure 5.6 shows state diagrams for two queued state machines defined in one class. The event e1 is defined in both state machines.




```
1  class X{
2     queued sm{
3      s1 {
4        e1 ->s2;
5        e5 ->s2;}
6      s2 {
7        e2 ->s3;}
8      s3 {
9        e3 ->s4;}
10     s4 {
11       e4 ->s1;}
12    }
13    queued/pooled sm2
14 {
15     s21 {
16       e1 -> s22;}
17     s22 {
18       e7 -> s21;}
19    }
```

```
1  class X{
2   pooled sm{
3      s1 {
4        e1 ->s2;
5        e5 ->s2;}
6      s2 {
7        e2 ->s3;}
8      s3 {
9        e3 ->s4;}
10     s4 {
11       e4 ->s1;}
12    }
13    queued/pooled sm2
14 {
15     s21 {
16       e1 -> s22;}
17     s22 {
18       e7 -> s21;}
19    }
```

Figure 5.6: State diagrams show that the same events are defined in multiple queued/pooled state machines as shown in UmpleOnline

As shown in Figure 5.6, when event e1 is called, it will take the system into state s2 in the first state machine and state s22 in the second state machine.

In UML, when the same event causes multiple transitions to be fired from the same state, the guards should be mutually exclusive. That is to say; multiple transitions from the same target state can have the same event trigger, if and only if the guard conditions on these transitions do not overlap. At the time the event occurs, a guard condition will be evaluated just once for the transition.

This semantic is the same for the basic, queued, and pooled state machines in Umple when the same events triggers transitions in the same state of the machine.

Also, in UML, multiple transitions can be triggered within one UML state machine if it has orthogonal regions. The firing order of these transitions is not determined by the UML standards.

The semantic of the basic version of Umple state machine enables defining the same event in two state machines defined in the same class. In the example above in Figure 5.6, the event 'e1' is defined in machine 'sm' and 'sm2'. These two machines are considered to be a single machine with two orthogonal regions 'sm' and 'sm2'. Therefore, if the event occurs and both machines are in the target states, the event will be processed concurrently on both machines, leading each machine to transition to the destination states. However, if only one machine is in the target state and the event occurs, the event will be processed only on the machine that is in the current state, and it will not be processed on the other machine. This also is the same semantic for queued and pooled state machines.

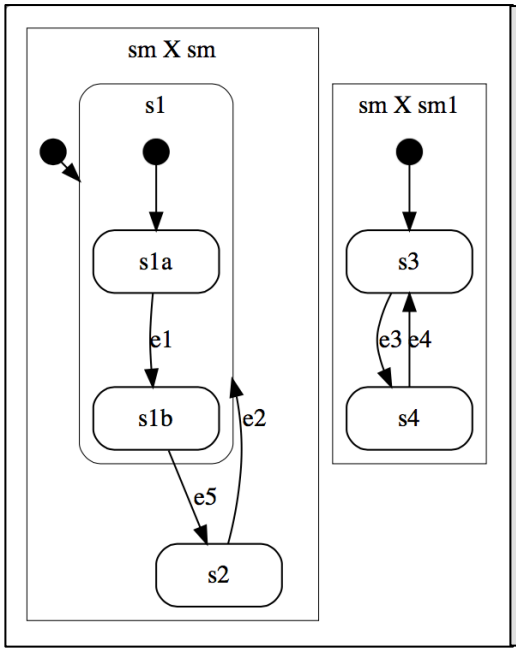
If we have this case, then the event that is specified in one or more queued or pooled state machines, or in one or more substates should not be duplicated in the enumerations of message types, in the message-accepting methods, or in the run method.

For example, the Java code generated from the above Umple code will have an enumeration of the messages types for both state machines as follows:

```
enum MessageType { e1_M, e2_M, e3_M, e4_M, e5_M, e7_M }
```

5.8.5 Multiple Queued State Machines (QSMs) or Multiple Pooled State Machines (PSMs) in the Same Class

Figure 5.7 shows a state diagram for two queued/pooled state machines defined in one class; one of the queued/pooled state machines has nested states (substates).




```
1   class X{
2     queued/pooled sm{
3       s1{
4         s1a{
5           e1 -> s1b;
6         }
7         s1b{
8           e5 -> s2;
9         }
10      }
11      s2{
12        e2 ->s1;
13      }
14    }
15    queued/pooled sm1{
16      s3{
17        e3 -> s4;
18      }
19      s4{
20        e4 -> s3;
21      }
22    }
23  }
```

Figure 5.7: A state diagram for multiple queued state machines in one class as shown in UmpleOnline

The multiple QSMs or PSMs in one class are equivalent to a single QSM or PSM state machine with concurrent regions. We have one queue/pool in which incoming messages are added. Then, these messages are removed form the queue/pool and processed.

In order to have generated Java code of Umple code with a class that has two or more queued/pooled state machines, we have two design decisions.

The first design choice is summarized in the following points:

1. Creating an enumeration of message types for each QSM/PSM such as:
   `MessageType and MessageType_1, … etc.`
2. Declaring and initializing a MessageQueue/MessagePool instance for each queued/pooled state machine such as:
   `queue and queue_1, … etc.`

```
pool and pool_1, … etc.
```

3. Defining an overloaded constructor in Message class for each queued/pooled machine where its parameters are MessageType_1 t and Vector p and so on.
4. Events methods of the second queued/pooled state machine to add incoming messages to the queue/pool have been defined as:

```
public void event ()
{ queue_1.put(new Message(MessageType_1.event_M, null)); }
public void event ()
{ pool_1.put(new Message(MessageType_1.event_M, null)); }
```

5. In the run() method the content of a while loop looks different.

```
1   public void run ()
2   {
3     boolean status=false;
4     while (true)
5     {
6       Message m = queue/pool.getNext();
7       switch (m.type)
8       {
9          //cases of first queued/pooled SM's MessageType variables
10      }
11      Message m_1 = queue/pool_1.getNext();
12      switch (m_1.type_1)
13      {
14         //cases of second queued/pooled SM's MessageType variables
15      }
16      if(!status)
17      {
18         // Error message is written or  exception is raised
19      }
20    }
21  }
```

This design has some issues. There is a single run method for the class, and it retrieves messages from both state machines in a fixed order. However, we do not know in which order messages may arrive for the two state machines. Each state machine needs its own thread fetching the messages. The second issue is that not all messages come from the environment of the class. One should also consider that one of the state machines generates an output that goes to the other state machine. The third issue is that there could be potentially 100 state machines in a complex system with lots of nesting and concurrency. The problem might occur because of generating these state machines with many arguments that cannot be handled by this design.

However, it is clearly the case that 'several state machines within one class' is

exactly equivalent to a single state machine with concurrent hierarchical states at the top level, each realizing one of the simple state machines. That has been the Umple semantics all along. The single hierarchical state machine would only need one queue.

Therefore we adopt this second design choice and resolve the above issues by making the following changes to our generated code:

1. Avoiding starting multiple 'removal' threads.
2. Only declaring one message queue/pool for all state machines
3. Having one enumeration of message types for all events of all state machines.
4. Having message type lists for the top-level states, and for the bottom level.
5. In the inner class, only declaring type for all state machines.
6. In the messages accepted section, everything gets put on a single queue/pool.
7. In the run method, there would be just one switch statement not more that one.
8. Avoiding duplicated methods such as run and the event handling methods.
9. Avoiding duplicate inner classes.
10. Generating messages-accepting methods for all events of all state machines.
11. Having "_" prefix for all SMs' event methods.
12. Avoiding duplication of the constructor body.
13. Avoiding duplication of "implements runnable."
14. Declaring a static Map<Object, MessageType> stateMessageMap and initializing it statically. In the Map, we put pairs mapping every possible state Enum value in every possible state machine to the relevant list of messages.
15. Simplifying 'getNextProcessableMessage' with one 'if' statement per machine.

5.8.6 One or More (QSMs) with One or More Basic and/or (PSMs) in the Same Class

We decided that it should not be allowed to have more than one type of state machine in one class except for the eventless state machine because each type of state machine has different semantics. A queued state machine has a different semantic from a pooled state machine and basic state machine in Umple. It is not possible to have one queue for all types of state machines when they are defined in the same class for several reasons. First, the basic state machine does not have an event-queue handler mechanism. Second, the queued and pooled state machines have different strategies to

handle and remove the events from the queue.

Listing 5.11 below presents a class with three types of state machines: queued state machine (sm), pooled state machine (sm1), and basic state machine (sm2). The result of defining two or more types machines in one class is to generate an error notification indicating there is a mistake that needs to be fixed.

```
1   class X{
2     queued sm{
3       s1{
4         e1 -> s1;}
5     }
6     pooled sm1{
7       s3{
8         e3 -> s4;}
9       s4{
10        e4 -> s3;}
11    }
12    sm2 {
13      s5 {
14        e5 -> s5;
15      }
16    }
17  }
```

Listing 5.11: Example of defining basic, queued, and pooled state machines in one class

For example, the error that is raised from compiling the above Umple code is:

```
Error on line 20: class X must have no queued state machine,
pooled state machine, and regular state machine in the same
class. More information (58)
```

5.8.7 QSM or PSM with at Least One State But with No Events (Raising a Warning Message)

If we have a queued or pooled state machine defined in Umple that has a set of states, but does not have any regular events or timed events, then a warning message will be raised to indicate to the user that there is no event to be queued/pooled as following:

```
Warning on line 2: Queued/Pooled State machine (name of the
sate machine) has no events to be queued/pooled. More
information (56)
```

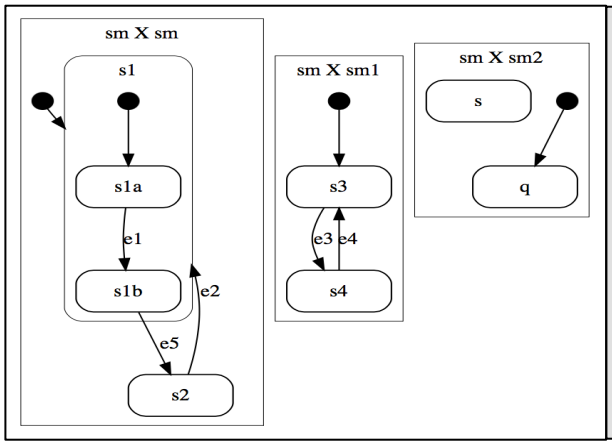
5.8.8 One or More Eventless State Machines with One or More QSMs or One or More PSMs in the Same Class

The eventless state machine is a construct in Umple that has states, but has no events to cause change of states. The eventless state machine can be used to generate a simple 'enum' in any language (enumerated type). It also can be used in the early stages of setting up a state machine (a programmer can call 'set' methods to set a state manually), before one has decided on the events yet.

If the eventless state machine is defined with a basic, queued, or pooled state machine within one class, then the eventless state machine can be ignored because it is not really a state machine. It is just an enumeration type. Therefore, the Java code would be generated normally, and no error or warning notifications will be raised.

If the eventless state machine is defined with both QSM and PSM in one class, then an error will be raised because it is not possible to have both QSM and PSM in the same class.

Figure 5.8 shows state diagrams for three state machines; sm, sm1, and sm2. The third state diagram shown in Figure 5.8 (sm2) has two states but no events.




```
1   class X {
2     queued sm {
3       s1{
4         s1a{ e1 -> s1b; }
5         s1b{ e5 -> s2; }
6       }
7       s2{ e2 -> s1; }
8     }
9     queued sm1 {
10      s3{ e3 -> s4; }
11      s4{ e4 -> s3; }
12    }
13    sm2 {
14      q {}
15      s {}
16    }
17  }
```

Figure 5.8: State diagrams for two queued state machines (sm and sm1) and one eventless state machine (sm2) as shown in UmpleOnline

The state machine (sm2) is an eventless one because it has states but it does not have any events. Therefore, it is allowed to define this kind of the state machine with queued state machines in one class. The user can define the eventless state machine before or after the declaration and definition of the queued or pooled state machine(s).

5.9 Unspecified Reception Handler Mechanism in Umple

A special event called '*unspecified*' is specified in any state where the user wants to detect unspecified receptions when events arrive, and the current state does not respond to them. This special transition can be placed in any state, or in any substate at any level of nesting. It has an event with string arguments: state and event. The event of this transition '*unspecified*' is generated automatically from a state machine as a special method that is called and handled at the end of every event method, in the 'default' section of the switch statement. When unspecified reception is encountered in the state, this will match any event not handled. If the event method is called and the current state does not respond to it, then this special event method '*unspecified*' is called instead.

The semantics of this special transition is the same as any other transition in a state machine, which means that it could perform any kind of actions; it could transition to some other states, and it can be guarded.

The Umple code for the state machine SM shown in Figure 4.8 is as following:

```
1  class X{
2    SM {
3      s1 {
4        a -> s2;
5        unspecified -> s1;
6      }
7      s2 {
8        b -> s3;
9      }
10     s3 {
11       c -> s1;
12       unspecified -> s3;
13     }
14   }
15   public static void main (String[] args) {
16     X x=new X();
17     x.a();
18     x.b();
19     x.c();
20     x.c();
21     x.a();
22     x.b();
23   }
}
```

The generated code of 'unspecified' transition of the state machine in Figure 4.8 is shown below. The 'unspecified' event method will be called at the end of the event methods (a) and (c). Specifically, it is put in the default section, and it will not be called

at the end of the event method (b) because the user does not want to handle the unspecified reception for this event.

A set of test cases to test the implementation of using 'unspecified' transition for handling unspecified receptions in case of basic and queued state machines are shown in Table 5.3.

Table 5.3: Different test cases of unspecified reception handler mechanism for the basic and queued state machines

Basic state Machine	Queued state Machine
Class with a simple state machine where at least one state has '*unspecified'* event	Class with a queued state machine with simple states where at least one state has '*unspecified'* event
Class with a composite (nested or concurrent) state machine where at least one state has '*unspecified'* event	Class with a queued state machine with composite (nested or concurrent) states where at least one state has '*unspecified'* event
Two classes where each class has a basic state machine communicating with each other. Each of them has simple states where at least on of them has '*unspecified'* event.	Two classes where each class has a queued state machine communicating with each other. Each of them has simple states where at least on of them has '*unspecified'* event.
Two classes where each class has a composite (nested or concurrent) state machine communicating with each other. Each of them has states where at least on of them has '*unspecified'* event.	Two classes where each class has a queued state machine with composite (nested or concurrent) states communicating with each other. Each of them has states where at least on of them has '*unspecified'* event.
Class has multiple basic state machines either simple or composite where at least one state in each state machine has '*unspecified'* event.	Class has multiple queued state machines either simple or composite where at least one state in each state machine has '*unspecified'* event.

Because the '*unspecified'* is a special transition of Umple state machine and '*unspecified'* is a keyword, when the '*unspecified'* event method is called by another event handling method, it would not be added to the queue in case of QSM. All other regular events will be added to the queue when they are created. To clarify, when processing the first event at the head of queue, if it would not be handled then the unspecified logic takes place, and the *'unspecified'* event method is called to handle the problem immediately.

The parts of generated code that show how the unspecified reception handler is called in the event 'a' for a basic and queued state machine is in Table 5.4.

Table 5.4: 'unspecified' event method is called in 'default' section of event handling methods

Event	Basic state Machine	Queued State Machine

a	`public boolean a()`
	`{`
	`  boolean wasEventProcessed = false;`
	`  SM aSM = sM;`
	`  switch (aSM)`
	`  {`
	`  case s1:`
	`  setSM(SM.s2);`
	`  wasEventProcessed = true;`
	`  break;`
	`  default:`
	`  // Other states do respond to this event`
	`  wasEventProcessed = unspecified(getSM().toStrin g(), "a");`
	`  }`
	`  return wasEventProcessed;`
	` }
		`public boolean _a()`
		`{`
		`  boolean wasEventProcessed = false;`
		`  SM aSM = sM;`
		`  switch (aSM)`
		`  {`
		`  case s1:`
		`  setSM(SM.s2);`
		`  wasEventProcessed = true;`
		`  break;`
		`  default:`
		`  // Other states do respond to this event`
		`  wasEventProcessed = unspecified(getSM().toString( ), "a");`
		`  }`
		`  return wasEventProcessed;`
		` }

The pseudocode of the generated Java code for the unspecified reception mechanism used in this example is shown in Appendix E.1.

If we use the same example in Figure 4.8 to define PSM and show the effect of using the 'unspecified' transition, we notice that this mechanism has no effect on the PSM. As a result, the '*unspecified*' events defined in this example are not longer treated as special events, and they are generated as any regular event defined in this example. The reason behind this is because that the pooled state machine is enhanced for two purposes: first is to allow the state machine events to be pooled, and second is to define policy of handling events that avoid unspecified receptions. If '*unspecified*' event is specified, a warning message would be raised. The warning message indicates the number of the line where '*unspecified*' event is specified and the message states that: *"(unspecified) must not be used in combination with Pooled State machine 'name of the state machine' - it is treated such as other regular events - it is pooled. More information (62)."*

5.9.1 Test-Driven Development (TDD) For Adopting the Unspecified Reception Handler Mechanism in Umple

The main changes to be done to allow for unspecified reception handling mechanism can be shown at:

http://code.google.com/p/umple/source/detail?r=3571

The following subsections show the different levels of testing we went through which shows that we use the TDD approach for developing this mechanism in Umple.

5.9.1.1 Modifying Umple State Machine Grammar to Have 'unspecified' Keyword as a Special Event in a State Machine (Parser Testing)

The Umple state machine grammar was first changed to allow for the '*unspecified'* keyword. The basic and queued state machines can now have this special event to detect the problem of unspecified reception.

```
1  transition : ( [[eventDefinition]] [[guard]] | [[guard]]
2  [[eventDefinition]] | [=unspecified]? [[guard]] |
3  [[eventDefinition]])? ( [[action]] -> | -> [[action]] | -> )
4  [stateName] ;
```

We add a test case called '100_stateMachine_UnspecifiedReception.ump ' to the test suite to check if we can now specify '*unspecified'* event in the Umple basic and queued state machines.

```
1   class UnSpecifiedReceptionTestCaseOne {
2     sm {
3       s1 {
4         e1 -> s2;
5         unspecified /{printError();}-> error1;
6       }
7       s2 {
8         e2 -> s1;
9         unspecified -> error2;
10      }
11      error1 {
12        -> s1;
13      }
14
15      error2 {
16        -> s2;
17      }
18    }
19  }
```

Listing 5.12: Umple code for parser test

5.9.1.2 Modifying Umple State Machine Semantics to Recognize 'unspecified' Keyword (Metamodel Testing)

We adapt the Umple metamodel to allow for '*unspecified'* keyword by changing 'StateMachine.ump' to add the following piece of code to Event class.

```
1
2
Boolean unspecified = false;
```

We also Modify 'UmpleInternalParser_CodeStateMachine.ump' to allow for parsing the keyword '*unspecified'* by adding the following piece of code to the methods of analyzing the state machine's transitions and stand-alone transitions:

```
1
2
3
4
if(event.getName().equals("unspecified"))
{
  event.setUnspecified(true);
}
```

We then add a test case to the test suite 'UmpleParserStateMachineTest.java' to show that the Umple metamodel is populated correctly for '*unspecified'* event of a state machine.

5.9.1.3 Modifying Jet Templates to Generate Java Code From Unspecified Reception Handler Mechanism (Template or Code Generation Testing)

In the generated Java code, each state that has the '*unspecified'* event, should call the '*unspecified'* event method at the end of its events methods in the default section. The '*unspecified'* event method in the generated code would have the state and event arguments as strings, and because these arguments would always be the same wherever the 'unspecified' event method is called, and then the arguments would not be derived from arguments in the Umple code. The result of calling the '*unspecified'* event method in all cases would be assigned to the 'wasEventProcessed' variable.

We make a change on a set of Jet files to enable Umple to generate Java code for a state machine that has '*unspecified'* events. Below, we list the Jet files that are affected by these changes.

- queued_state_machine_queuedEvent.jet

- queued_state_machine_removalThread_run.jet
- state_machine_Event.jet

Also, we modify 'Generator_CodeJava.ump' to deal with some issues related to an unspecified event. In addition, a bunch of test cases is added to ensure that Java generated code is syntactically correct which means that the actual generated code matches the expected generated code.

We then write a list of test cases that covers the basic and queued state machines and the case of nested state machines where '*unspecified*' event is specified.

5.9.1.4 Adding Semantic Tests For Java Code Generation of Unspecified Reception Handler Mechanism (Semantic Testing)

Finally, we add some semantic tests to ensure that generated programs behave as expected, which means that they are semantically correct. For example, we have a generated system model illustrated in Listing 5.13. We use the queued state machine to model this system, and we use 'unspecified' events to handle the unspecified reception problem when it occurs at a specific state (idle) and substate (validating). Therefore, when the state machine is in idle state, and the event is called that does not match one of the events that state has, the '*unspecified*' event will be called, and the state machine will transition to state (error1) which has instantaneous transition to move to (idle) state. In addition, a substate (validating) has an 'unspecified' event that will be called if an event is called and does not match 'validated' event.

```
1   class AutomatedTellerMachine
2   {
3     String[] logs;
4
5     queued sm
6     {
7       idle
8       {
9         cardInserted -> active;
10        maintain -> maintenance;
11        unspecified -> error1;
12      }
13      maintenance
14      {
15        isMaintained -> idle;
16      }
```

```
1       active
2       {
3         entry /{addLog("Card is read");}
4         exit /{addLog("Card is ejected");}
5         validating
6         {
7           validated -> selecting;
8           unspecified -> error2;
9         }
10        selecting
11        {
12          select -> processing;
13        }
14        processing
15        {
16          selectAnotherTransiction -> selecting;
17          finish -> printing;
18        }
19        printing
20        {
21          receiptPrinted -> idle;
22        }
23        cancel -> idle;
24      }
25      error1
26      {
27        ->idle;
28      }
29      error2
30      {
31        ->validating;
32      }
33    }
34  }
```

Listing 5.13: Umple example used as semantic test

The following events are considered as inputs to feed the above system:

```
cardInserted
validated
select
finish
receiptPrinted
selectAnotherTransiction
maintain
isMaintained
cancel
cardInserted
select
validated
select
selectAnotherTransiction
select
finish
receiptPrinted
finish
```

If we trace the execution of this system, we will figure out that there are four events that are considered to be unspecified reception.

When the event ‘cardInserted' is called, the state machine transitions to ‘active' state, and when the event ‘validate' is called, it will transition to ‘selecting' state. Then the event ‘select' is triggered which causes the transition to move to the ‘processing' state. The event ‘finish' is called which enable the state machine to transition to ‘printing' state. The event ‘receiptPrinted' is then called causing the state machine to move to the ‘idle' state. After that, the event ‘selectAnotherTransition' is called, which is considered as unspecified reception in the current state ‘idle.' It will be removed from the queue, and then it will not be processed and the unspecified reception method is called which handles this error by transitioning to ‘error1' state that in turn allows the state machine to transition to ‘idle' state. The semantic test that can make a comprehensive test of the processing of the events showing unspecified reception mechanism in QSM is written in Appendix D.3.

Chapter 6 Test Cases and a Real-Time Case Study

6.1 Test Cases For Different Features and Issues of Queued and Pooled State Machines

Tables 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 below illustrate the main queuing and pooling features that are supported in Umple for different flavors of state machines with given links.

As shown in Table 6.1, a simple state machine that has a set of simple states can now be defined as a queued or pooled state machine. This type of a state machine can have any number of events or timed events to be added to the queue/pool when they are triggered. Then they are taken off the queue/pool and processed when the machine is in current corresponding state.

In addition, Umple supports composite state machines that are developed by Badreddin (2010) to have nested and/or concurrent states as shown in Table 6.1. Currently, it is possible to have queued and pooled nested and/or concurrent state machines as shown in Table 6.1, Table 6.2 and Table 6.3.

Besides, Table 6.4 has a set of test cases to check the case when multiple QSMs or PSMS are defined in the same class, and when multiple QSMs and PSMs are defined in the same class.

Table 6.5 has a set of test cases to check queued and pooled state machine's elements: transitions, actions, entry/exit actions, and guards. There are also test cases to check when events have parameters and when they have no parameters. Furthermore, tests cases are written to show the use of unspecified mechanism for QSM and the potential outputs.

We also define a list of test cases in Table 6.6 to check the different types of transitions in case of QSM and PSM such as state transitions, instantaneous transitions, and timed transitions.

In addition, we write a set of test cases for various issues that are covered during this thesis (Chapter 5) such as eventless state machine with QSM or PSM as shown in Table 6.7 (refer to section 5.8.8 for more information about eventless state machines),

QSM and Basic SM in the same class (Table 6.8), PSM and Basic SM in the same class (Table 6.8), two or more QSMs/PSMs communicate with each other in the same class/in different classes (Table 6.8), and QSM or PSM with states that do not have events (Table 6.8).

6.1.1 Tests For Different Types of Umple State Machines

Table 6.1: Umple test cases for both flavors of Umple state machines

Types of State Machine			
• Basic (Simple) State Machine			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachine.ump
• Composite State Machine			
– Nested State Machines			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedWithNestedStateMachines.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/testPooledwithNestedStates.ump
– Concurrent State Machines			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedWithConcurrentStateMachines.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachineWithConcurrentStates_autoTransition.ump

6.1.2 Nested State Machines

Table 6.2: Umple test cases for QSM and PSM in case of nested state machines

Nested State Machine Cases			
Case 1:	Nested States with transitions		
Case 2:	Nested States with auto-transitions		
Case 3:	Nested States with timed transitions		
Case 4:	Nested States with entry/exit actions		
Case 5:	Nested States with transition actions		
Case 6:	Nested States with guards		
Case 7:	Nested States with do activities		
Case 8:	Same events' names in one or more Nested States		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedWithNestingStatesATM.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/testPooledwithNestedStates_4.ump

6.1.3 Concurrent States

Table 6.3: Umple test cases for QSM and PSM in case of concurrent states

Concurrent State Machine Cases			
Case 1:	Concurrent Substates with transitions		
Case 2:	Concurrent Substates with auto-transitions		
Case 3:	Concurrent Substates with timed transitions		
Case 4:	Concurrent Substates with entry/exit actions		
Case 5:	Concurrent Substates with transition actions		
Case 6:	Concurrent Substates with guards		
Case 7:	Concurrent Substates with do activities		
Case 8:	Same events' names in one or more Concurrent Substates		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedWithConcurrensStatesCourseAttempt.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachineWithConcurrentStates_autoTransition.ump

6.1.4 Tests For Multiple State Machines in the Same Class

Table 6.4: Umple test cases for case of multiple queued /pooled state machines in the same class

Cases of Multiple State Machines in The Same Class		
Case 1: Multiple QSMs in the same class Case 2: Same events' names in one or more QSMs		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multipleQSM.ump	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multipleQSM_sameEvents.ump
Case 3: Multiple PSMs in the same class Case 4: Same events' names in one or more PSMs		
PSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multiplePooledStateMachine.ump	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multiplePooledStateMachines_sameEvents.ump
Case 5: One or more QSMs and one or more PSMs in the same class. (Error Message is generated)		
http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidQSMandPooledSMinSameClass.ump	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidQSMandPooledSMinSameClass.ump	

6.1.5 Umple State Machine's Elements

Table 6.5: Umple test cases for different elements of Umple state machine

Elements of State Machine			
• Transition, Action, Entry/Exit Action, Guard			
QSM	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessQueuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessPooledStateMachine.ump
• Event:			
– Event without Parameters			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachine.ump
– Event with Parameters, Same Events with same Parameters			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedStateMachine_withParameters.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachine_withParameters.ump
– Same Events with different Parameters (Error message is generated)			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/100_eventWithInconsistentArguments.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/100_eventWithInconsistentArguments.ump
– Unspecified event (Warning message is generated in case of PSM)			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/stateMachine_unSpecifiedReception_QSM.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalid_PooledStateMachine_UnspecifiedReception.ump

6.1.6 Umple State Machine Transitions

Table 6.6: Umple test cases for different types of Umple state machine transitions

Types of State Machine's Transition			
• State Transition:			
Case 1: event -> next state Case 2: event/ action -> next state Case 3: event [guard] -> next state Case 4: event [guard] / action -> next state			
QSM	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessQueuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessPooledStateMachine.ump
• Instantaneous Transition:			
Case 1: Auto-transition to next state Case 2: Auto-transition with Transition Action to next state Case 3: Auto-transition with Guard to next state Case 4: Auto-transition with Guard and Transition Action to next state			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedStateMachine_autoTransition.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachine_autoTransition.ump
• Timed Transition:			
Case 1: after/afterEvery (Time in Seconds) -> next state Case 2: after/afterEvery (Time in Seconds) / action -> next state Case 3: after/afterEvery (Time in Seconds) [guard] -> next state Case 4: after/afterEvery (Time in Seconds) [guard] / action -> next state			
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/queuedStateMachine_timedEvents.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/pooledStateMachine_timedEvents.ump

6.1.7 Eventless State Machines

Table 6.7: Umple test cases for Umple eventless state machines

Eventless State Machine			
Case 1:	One or more Eventless State Machines *after* one or more Queued/Pooled State Machines.		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_valid_PooledSM_with_EmptyRegularSM.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_valid_QSM_with_EmptyRegularSM.ump
Case 2:	One or more Eventless State Machines *before* one or more Queued/Pooled State Machines.		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/eventlessStateMachine_QueuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/eventlessStateMachine_PooledStateMachine.ump
Case 3:	One or more Eventless State Machines before one or more Queued/Pooled State Machines, and one or more Eventless State Machines after one or more Queued/Pooled State Machines		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multipleQSM_EventlessStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/multiplePooledStateMachine_EventlessStateMachine.ump

6.1.8 Common Issues

Table 6.8: Umple test cases for other common issues in QSM and PSM

Issues			
Case 1:	Queued/Pooled State Machines and Regular State Machines in the same class (Error Message is generated)		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidQSMandRegularSMinSameClass.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidRegularSMandPooledSMinSameClass.ump
Case 2:	PSM, QSM, and Regular State Machine in the same class (Error Message is generated)		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidQSMandPooledSMandRegularSMinSameClass.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_invalidQSMandPooledSMandRegularSMinSameClass.ump
Case 3:	Two or more Queued/Pooled State Machines communicate with each other • In the same class • In separate classes		
QSM	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessQueuedStateMachine.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/testbed/src/TestHarnessPooledStateMachine.ump
Case 4:	Queued/Pooled State Machine with states that do not have events (Warning Message is generated)		
QSM	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_queuedStateMachine_noEvents.ump	*PSM*	http://code.google.com/p/umple/source/browse/trunk/cruise.umple/test/cruise/umple/compiler/106_pooledStateMachine_noEvents.ump

6.2 Evaluation of Umple Queued State Machines (QSM)– Case Study

In order to validate our models, we built a set of software systems using Umple:

- Agents communicating
- TCP/IP simulation
- Coffee machine
- Security light
- Elevator controller system

In our thesis, we focus on elevator controller system that shows the usefulness of modeling realistic systems in Umple using queued state machines (QSM).

We have based our work on resources (Chen, 2005; Horne, Subburaj, & Urban, 2012; Zhang & Mackworth, 1993) that demonstrate the requirements of the elevator controller systems.

An elevator system is comprised of several components besides the elevator controller that are necessary to operate the elevator on a daily basis. In order to enable passengers to use the elevator system in an efficient way, the elevator controller is used to control most of the other elevator system's components.

This system is composed of number of elevator cars that service a number of floors where passengers move up/down the building.

Essentially, the users of the elevators controller are the passengers and other components of the elevator system that interact with it. Those components must use the same format of protocol for signals that the controller uses in order to make communication among them possible.

The elevator controller's main function is to interact with and control the following other components of the elevator system:

Buttons: They are categorized into two types of Buttons:

Summon/Hall Buttons: On the button panel outside of the elevator, the summon buttons are located. There are two summon buttons up and down on each floor, except for the ground floor where there is only one up button and the top floor where there is only one down button. The elevator controller interacts with the summon buttons by receiving pressed/released signals that indicate the floor number from where the

summon button is pressed and the requested direction either up or down, and sending light on/off signals to indicate the status of the summon button that is pressed/released.

Floor Request Button: The floor request buttons are located on a control panel that is on the interior of each elevator cab. The number of the floor request buttons in each elevator cab is according to the number of floors in a particular building starting from ground floor to the top floor. The elevator controller that controls the elevator cabs in a building that has number of floors interacts with floor request buttons by receiving pressed signals that indicate the elevator cab from where the button is pressed and the desired floor number passenger wants to go to, and sending light on/off signals to indicate the status of the floor request button that is pressed.

Open Door Button: The open door button is located on a control panel. Passenger can press open door button to open the elevator doors or to keep pressing the button to keep the elevator doors open when the elevator cab is parked at a floor. The elevator controller interacts with this button by receiving pressed/released signal that indicates the elevator cab from where the button is pressed/released.

Displays (Indicators): These are:

Floor Number Display/Indicator: Inside of each elevator cab, there is a floor number display that indicates to its passengers which floor the elevator cab is currently on. The elevator controller interacts with the floor number display by sending a signal to inform the controller which floor number to display.

Direction Display/Indicator: Interior of each elevator cab, there is a direction display that indicates to passengers in the elevator cab the current direction of the elevator cab, which is either up or down. The elevator controller interacts with the direction display by sending a signal to inform the controller which direction to display.

Elevator Engine (Motor): The elevator engine is in charge of moving an elevator cab in either up or down direction between floors. The elevator controller interacts with the elevator engine by sending a signal to the controller to determine in what direction the engine should be going in and at what speed it should move the elevator cab where a stop signal is constructed by setting the signal's speed parameter to zero.

Door Timer: For each elevator door, there is a door timer used to set a mount of time

to wait before the elevator door is closed.

Elevator doors or simply doors: Refer to the inner door on the elevator cab. The Door Opening device on elevator cab always opens and closes the outer door (floor door) of the shaft at the same time as the inner door of the elevator cab. Therefore, there is no need to refer to the inner and outer doors separately.

Our goal is to use queued state machine models in Umple to simulate the behavior of the elevator controller are by enabling it to handle input signals from other components of the elevator systems and respond accordingly with output signals. The QSM helps in logging and processing requests of passengers and then to moving the elevator cars between floors in response to those requests.

The class diagram modeled in UmpleOnline that shows the main structure of the elevator controller system as depicted is shown in Figure 6.1. Note that in a real elevator system, there would be no 'Passenger' class, since passengers are not tracked individually. However such a class is added here to allow simulation.

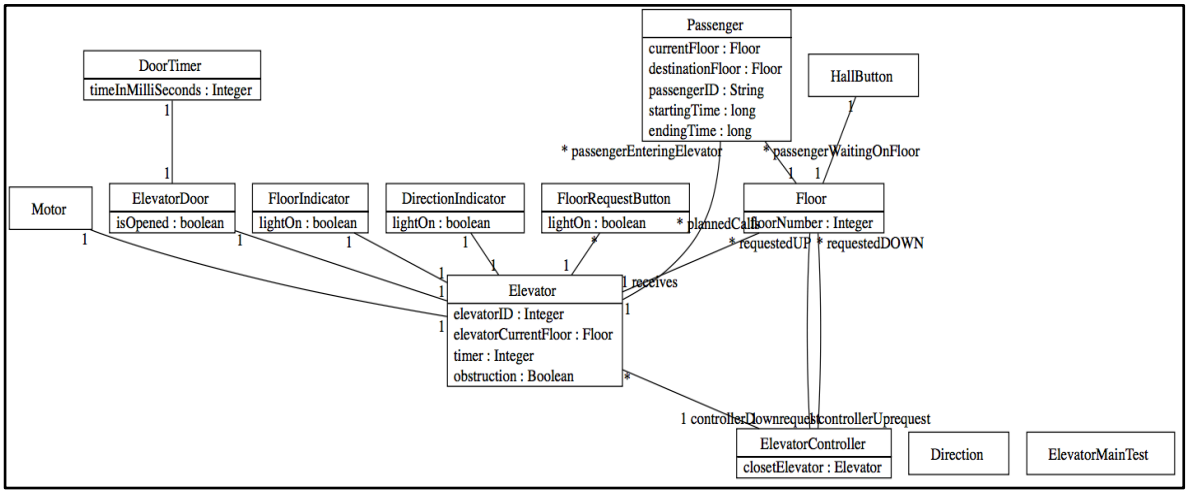


Figure 6.1: Class diagram of an elevator controller system as shown in UmpleOnline

The state machine diagrams shown in Figure 6.2 are modeled in UmpleOnline to represent the behavior of each component of elevators controller system. The main state machine described here is the elevator state machine depicted in Figure 6.3, which describes the actual behavior of the elevator system.

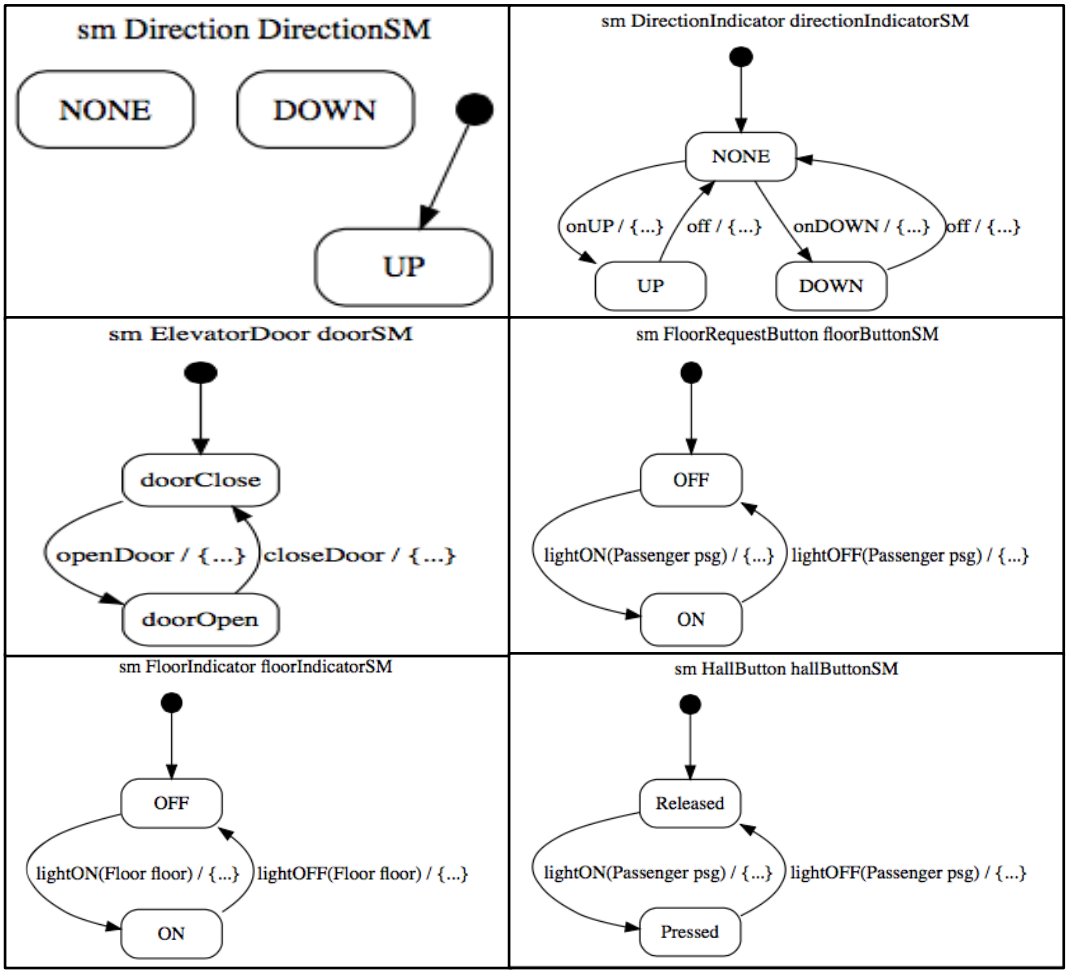


Figure 6.2: State machine diagrams as shown in UmpleOnline for: Floor button, Hall button, Floor indicator, Direction indicator, elevator door, and eventless machine for direction of the elevator

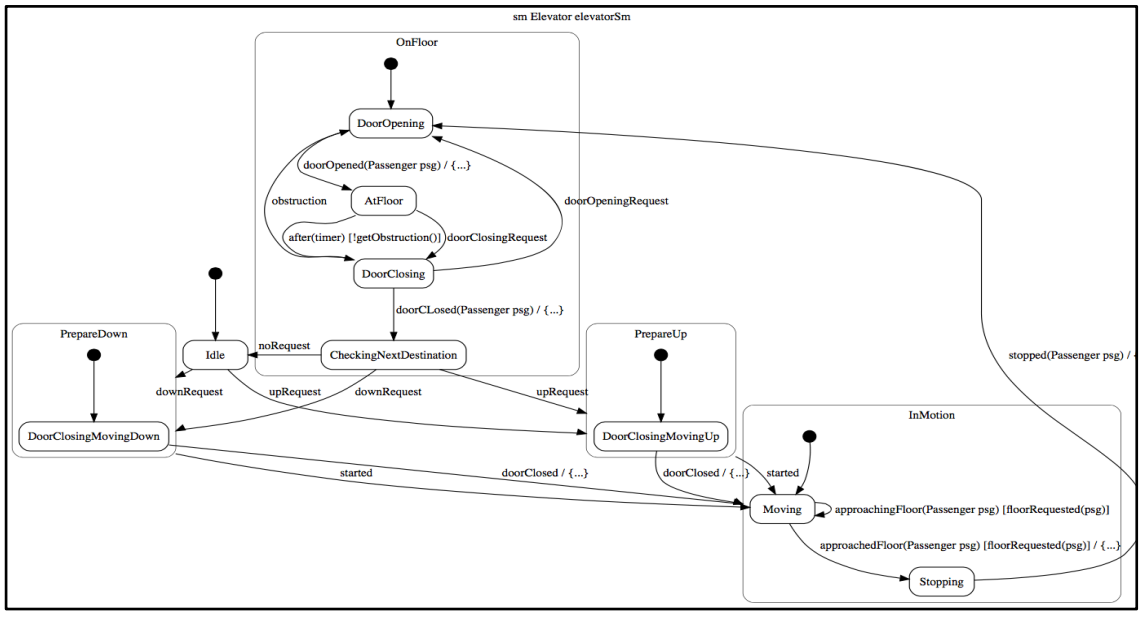


Figure 6.3: State machine for elevator behavior as shown in UmpleOnline

Five of the above state machine diagrams are modeled using queued state machine as follows:

- ***elevatorDoorSM***: to demonstrate the behavior of elevator door.
- ***hallButtonSM***: to demonstrate the process of requesting hall buttons of the elevator.
- ***floorButtonSM***: to demonstrate the process of requesting floor buttons.
- ***floorIndicatorSM***: to demonstrate the response to the passenger requests by displaying the number of floor which the elevator at.
- ***directionIndicatorSM***: to demonstrate the process of showing the direction where the elevator is going.

The elevator state machine diagram is modeled using pooled state machine as shown in Figure 6.3.

The Umple code that represents the elevator controller is listed in Appendix F.1, which can be complied and run to simulate the behavior of the system.

The simulation of different components of the elevator controller system using queued and pooled state machines brings two advantages to its implementation. Each component is represented by a state machine that is attached to a class. Using queued and pooled state machines enables those components to communicate with each other by passing messages. Those messages are injected into the queue in order to be processed; that is, those components are communicating asynchronously. Second, we use a pooled state machine to model the elevator to show that unexpected events that might occur will be handled.

To highlight these advantages, we use basic state machines to model the components of the elevator controller system when two passengers arrive and request the elevators. We have modeled the behaviors of three elevators in a building with five floors. We then compare the output of this simulation with the output of simulating the elevator controller system using QSM and PSM.

Figure 6.4 (a) shows the output of the implementation of the elevator controller system using basic state machines. It shows that some messages are missed which results in incomplete implementation.

However, modeling the elevator controller system using QSM and PSM in Figure 6.4 (b) allow for handling the messages that are passed between the components of the systems. It shows a complete implementation of the system (i.e. simulation of the behavior of elevators when two passengers arrive and press the hall buttons).

```
Simulation Output (using Basic SM):

[Passenger STEPHEN] arrives on [Fl 3] wants to go to [Fl 5]
[Passenger ALIAA] arrives on [Fl 5] wants to go to [Fl 2]
[Passenger STEPHEN] presses Fl [5  UP request]
[Passenger ALIAA] presses Fl [2  DOWN request]
[Fl 3] UP request light ON

[Fl 5] UP request light ON

CONTROLLER selects Elevator  [3] ON FLOOR [4] to serve Passenger [ALIAA]
Elevator  [3] DOWN Direction Indicator
[Elevator 3] starts moving DOWN
[Fl 4] signal light ON  for [Elevator 3]
[Fl 4] signal light OFF for [Elevator 3]
[Elevator 3] [Fl 5] destinations [Fl 5] moving
[Fl 5] signal light ON  for [Elevator 3]
[Fl 5] signal light OFF for [Elevator 3]
Elevator  [3] arrives
[Fl 5] UP request light OFF
CONTROLLER selects Elevator  [3] ON FLOOR [4] to serve Passenger [STEPHEN]
[Elevator 3] elevator door is opened
Elevator  [3] EMPTY Direction Indicator
```

```
Simulation Output (using QSM and PSM):

[Passenger ALIAA] arrives on [Fl 5] wants to go to [Fl 2]
[Passenger STEPHEN] arrives on [Fl 3] wants to go to [Fl 5]
[Passenger ALIAA] presses Fl [2  DOWN request]
[Passenger STEPHEN] presses Fl [5  UP request]
[Fl 3] UP request light ON
[Fl 5] UP request light ON
CONTROLLER selects Elevator  [3] ON FLOOR [4] to serve Passenger [STEPHEN]
CONTROLLER selects Elevator  [3] ON FLOOR [4] to serve Passenger [ALIAA]
[Elevator 3] starts moving UP
Elevator  [3] UP Direction Indicator
[Elevator 3] [Fl 5] destinations [Fl 5] moving
Elevator  [3] arrives
[Fl 5] UP request light OFF
[Fl 4] signal light ON  for [Elevator 3]
[Fl 4] signal light OFF for [Elevator 3]
[Fl 5] signal light ON  for [Elevator 3]
[Fl 5] signal light OFF for [Elevator 3]
[Elevator 3] elevator door is opened
Elevator  [3] EMPTY Direction Indicator
[Elevator 3] elevator door is closed
[Passenger ALIAA] Trip took : [3005]
[Elevator 3] starts moving UP
Elevator  [3] UP Direction Indicator
[Fl 5] signal light ON  for [Elevator 3]
[Elevator 3] [Fl 4] destinations [Fl 3] moving
[Fl 5] signal light OFF for [Elevator 3]
[Elevator 3] [Fl 3] destinations [Fl 3] moving
Elevator  [3] arrives
[Fl 3] signal light ON  for [Elevator 3]
[Fl 3] UP request light OFF
[Fl 3] signal light OFF for [Elevator 3]
Elevator  [3] EMPTY Direction Indicator
[Elevator 3] elevator door is opened
[Passenger STEPHEN] Trip took : [6008]
[Elevator 3] elevator door is closed
```

Figure 6.4: Simulation Outputs of Elevator Controller System Example

Chapter 7 Conclusion

The main goal of our thesis was to extend Umple state machines to implement new semantics, allowing for queuing and pooling of incoming events. Also, we implemented a mechanism to handle unspecified reception by writing a keyword 'unspecified' in a state or states of a state machine in Umple where we want to detect and handle unexpected events.

We followed the agile approach that allows for the test-driven development (TDD) to add these enhancements to state machines. In order to do that, we changed Umple state machine syntax, its metamodel, and the Java code generation.

The core of our work was presented in Chapters 4 and 5, where we showed the designs of queued and pooled state machines and their implementation. We also discussed the mechanisms of handling unspecified receptions, and we showed how we implemented them in Umple.

In Chapter 6, we provided a list of test cases used to evaluate the correctness and quality of the implementations of QSM and PSM. Also, we gave an example of a real case study modeled in Umple using queued state machines.

More importantly, we gave answers to the research question presented in the first chapter. We implemented the queued state machine in Umple taking into consideration the types of messages that can be passed between queued state machines such as allowing state machine events to have parameters.

To conclude, we summarize here the contributions that we achieved by the end of our thesis:

- We enabled support for a FIFO event queue in Umple. An event that is triggered is added to the queue and then it is taken off the queue and processed in a separate thread. This extended semantics of Umple state machine can be used to enhance the communication between components by allowing signals and asynchronous communication semantics. This

semantics is used if the keyword ‘queued’ precedes a state machine definition in Umple.

- We also provided an implementation for pooled semantics that is used if the keyword ‘pooled’ precedes a state machine definition in Umple. This type of a state machine is implemented in the same way as a queued state machine, but it has an additional mechanism where the unspecified reception can be detected and handled by leaving events at the head of the queue to be processed later, and instead processing the first matching event that is further back in the queue.
- We added an additional mechanism for dealing with unspecified reception in Umple for queued and basic state machines. A pseudo-event name ‘unspecified’ can be indicated on a transition in any state that will match any event not handled by the state. Such a transition can have all the capabilities of other events, including a guard, and an action.

Umple has evolved continually, and many extensions have been added to Umple state machines. Therefore, we have a plan to achieve some work in the future that can be summarized as follows:

Multithreading: we used ‘implements Runnable’ to create a thread for the class generated from a queued state machine to allow for multithreading environment. In the future, we want to implement a new design alternative, which may provide more flexibility and usability. We can define an inner class that extends Thread where we define the run method and start it in the constructor of this class. For example:

```
1   private static class EventProcessorThread extends Thread{
2     X controller;
3     String removalName;
4
5     public EventProcessorThread (X aController, String aRemovalName){
6       controller = aController;
7       removalName = aRemovalName;
8       start();
9     }
10    @Override
11    public void run (){
12      while (true) {
13        if ("removalThread".equals(removalName)){
14          controller.removalProcessor();
15        }
16      }
17    }
18  }
```

Deadlocks: this design error is described in our thesis in Chapter 3. Deadlock should be handled by avoiding, detecting, preventing, or ignoring it. There are some techniques that can be used to detect this error at the specification level such as model checking and reachability analysis. After detecting deadlocks, methods can be used to recover and correct them such as process termination in which one or more processes involved in the situation can be terminated. In addition, we need to do more research on different techniques and methods used to capture and recover from this design error in the models. We also need to do more investigations and tests for a system with multiple Umple state machines to see if mechanisms for detecting and recovering deadlocks can be implemented effectively.

Livelock: this is another design error that may occur when two processes or subsystems keep cycling through a limited set of states, but cannot get out of that cycle. We aim to do more research on this topic and figure out if we can implement a solution in Umple state machine to handle this design error.

Number of messages in the pool in case of PSM: we have two more issues to deal with in the future. The first issue is when the number of messages in the pool becomes too big with pool communication. The other issue is when some messages are kept in the pool and they are not processed because there are no receivers to consume them.

Code generation from queued and pooled state machines in other languages: we enable generating Java code from QSM and PSM, others on our team are working to generate C++ code from these new types of state machines.

Validating our proposed models with more Umple test cases and additional case studies: we aim to write larger and complicated test cases in the future to check that the new features added to Umple state machine are behaving correctly and generating high-quality code. Also, we have a plan to use QSM and PSM to model systems such as business cases, communication protocols. Doing this would further show the qualities and usefulness of Umple queued and pooled state machines.

Visulalization of queued and pooled state machine in UmpleOnline: one of our goals is to visualize the extentions of Umple state machines (QSM and PSM) on the graphical formats for the corresponding Umple code written in UmpleOnline. This is future work to be handled be other team members in CRuiSE group.

References

Aljamaan, H. I., Lethbridge, T. C., Badreddin, O. B., Guest, G., & Forward, A. (2014). *Specifying Trace Directives for UML Attributes and State Machines.* Modelsward, 2014, Portugal.

Almaghthawi, S. E. A. (2013). *Umple C++ Code Generator.* (Masters Thesis), University of Ottawa, Canada.

Ambler, W. S. (2012). Introduction to Test Driven Development (TDD). *In AgileData.org: Techniques for Disciplined Agile Database Development*. Retrieved April 25, 2013 from http://www.agiledata.org/essays/tdd.html

Babich, F., & Deotto, L. (2002). Formal methods for specification and analysis of communication protocols. *Communications Surveys & Tutorials, IEEE, 4*(1), 2-20.

Badreddin, O. (2010). *Umple: a model-oriented programming language.* Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2 (pp. 337-338), Cape Town. Doi: 10.1145/1810295.1810381.

Badreddin, O. (2012). *A manifestation of model-code duality: facilitating the representation of state machines in the umple model-oriented programming language.* (PhD Thesis), University of Ottawa, Canada.

Badreddin, O. B., Lethbridge, T. C., Forward, A., Elaasar, M., Aljamaan, H. I., & Garzón, M. (2014). *Enhanced Code Generation from UML Composite State Machines.* Second International Conference on Model-Driven Engineering and Software Development, Modelsward, 2014, Lisbon, Portugal.

Bakal, MR., Althouse, J., Verma, P. (2012, August14). Continuous integration in agile Development: How agile methods, continuous integration, and test-driven enhance design and development of complex systems. Worldwide Offering Manager, Electronics Industry, IBM.

Beck, K. (2003). *Test-driven development: by example*. Boston, MA, USA: Addison-Wesley Professional.

Beydeda, S., Book, M., & Gruhn, V. (2005). *Model-driven software development* (Vol. 15): Springer.

Bochmann, G. V. (1978). Finite state description of communication protocols. *Computer Networks (1976), 2*(4), 361-372.

Bochmann, G. V., & Sunshine, C. A. (1980). Formal methods in communication protocol design. *Communications, IEEE Transactions on, 28*(4), 624-631.

Bochmann, G. V. (1990). Specifications of a simplified transport protocol using different formal description techniques. *Computer Networks and ISDN Systems*,*18*(5), 335-377.

Bochmann, G. V. (1990). Protocol specification for OSI. *Computer Networks and ISDN Systems*, *18*(3), 167-184.

Bochmann, G. V. (2008). *System modeling using state machines*. Personal Collection of Software Construction. University of Ottawa, Ottawa, Canada.

Bochmann, G. V. (2013). *Model-Based Design and Verification of Distributed Real-Time Systems*. Personal Collection of Introduction to state-oriented modeling. University of Ottawa, Ottawa, Canada.

Bollig, B. (2006). Communicating Finite-State Machines. *Formal Models of Communicating Systems: Languages, Automata, and Monadic Second-Order Logic*, 117-150.

Boydens, J., Cordemans, P., & Steegmans, E. (2010, May). Test-driven development of embedded software. *In Proceedings of the Fourth European Conference on the Use of Modern Information and Communication Technologies* (pp. 117-128).

Brand, D., & Zafiropulo, P. (1983). On communicating finite-state machines. *Journal of the ACM (JACM), 30*(2), 323-342.

Budkowski, S., & Dembinski, P. (1987). An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN systems, 14*(1), 3-23.

Cantu, M. (2008). Essential Pascal. CreateSpace.

Chen, Q. (2005). *Software Requirements Specifications (SRS) for a Low-Rise Building Elevator System*. Retrived from https://cs.uwaterloo.ca/~dberry/ATRE/ElevatorSRSs/FinalSRSes/SRS-Qian-Chen.pdf.

CRuiSE. "Umple Online,", accessed 2013, http://try.umple.org/.

Dingsøyr, T., Dybå, T., & Brede Moe, N. (2010). Agile Software Development. Agile Software Development: Current Research and Future Directions, ISBN 978-3-642-12574-4. Springer-Verlag Berlin Heidelberg, 2010, 1.

Forward, A. (2010). *The convergence of modeling and programming: facilitating the representation of attributes and associations in the Umple model-oriented programming language.* (Ph.D. Thesis), University of Ottawa, University of Ottawa, Canada.

Forward, A., Badreddin, O., & Lethbridge, T. C. (2010). *Umple: Towards combining model driven with prototype driven system development.* Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on.

Forward, A., Badreddin, O., Lethbridge, T. C., & Solano, J. (2012). Model-driven rapid prototyping with Umple. *Software: Practice and Experience, 42*(7), 781-797.

Forward, A. & Lethbridge, T. C. "Umple Language", accessed 2014, http://www.umple.org

Girault, A., Lee, B., & Lee, E. A. (1999). Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *18*(6), 742-760.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, *8*(3), 231-274.

Hart, T., & Levin, M. (1962). "The New Compiler", AIM-39 CSAIL Digital Archive – Artificial Intelligence Laboratory Series.

He, S. (2013). *Concurrency and Multi-threading - Basic concepts* [PDF document]. Personal Collection of Networks/Concurrency. School of Computer Science, University of Birmingham, Edgbaston, Birmingham. Retrieved from http://www.cs.bham.ac.uk/~szh/teaching/ssc/lecturenotes/Concurrency/Lecture1_BasicConcepts.pdf

Hogrefe, D. (2013). *SDL-88 Tutorial*. SDL Forum Society. Retrieved from http://www.sdl-forum.org/sdl88tutorial/index.html

Holzmann, G. J. (1991). Finite State Machine. In *Design and validation of computer protocols* (pp. 163-187). New Jersey: Prentice Hall.

Horne, B. D., Subburaj, V. H., & Urban, J. E. (2012). Formal Specification For Real-Time Object Oriented Systems With UML Design. *International Journal of Digital Information and Wireless Communications (IJDIWC)*, *2*(4), 53-65.

Janzen, D. S., & Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. *Computer Science and Software Engineering*, 33.

Karai, V. (2009, April 08). Agile Test Driven Development [PowerPoint slides]. Retrieved from http://www.slideshare.net/vkarai/agile-test-driven-development-1265878

Klemm, R. P. (August 1996). *Systems of communicating finite state machines as a distributed alternative to finite state machines.* (Ph.D Thesis), The Pennsylvania State University.

Lethbridge, T. C. (2013, May 3). Umple Toolkit for Model-Oriented Programming. Webinar presented by the NECSIS project (Network on Engineering Complex Software Intensive Systems for Automotive Systems).

Lethbridge, T. C. (2013). Key Properties for Comparing Modeling Languages and Tools: Usability, Completeness and Scalability. *Proceedings from the 4th International Comparing Modeling Approaches Workshop 2013.* Miami, Florida, USA.

Lethbridge, T. C., Forward, A., & Badreddin, O. (2010). Umplification: Refactoring to incrementally add abstraction to a program. 17th Working Conference on Reverse Engineering (WCRE), 2010

Lethbridge, T. C., Mussbacher, G., Forward, A., & Badreddin, O. (2011). *Teaching UML using umple: Applying model-oriented programming in the classroom.* Conference on Software Engineering Education and Training (CSEE&T), 2011

Lethbridge, T. C., Forward, A. & Badreddin, O. (2012). Umple Google Code Project. Retrieved May 28, 2012, from http://code.google.com/p/umple

Lilius, J., & Paltor, I. P. (1999). The semantics of UML state machines. Turku Centre for Computer science.

McConnell, T. (2014, April 22). Test-driven development and Umple [PowerPoint slides]. Retrieved from http://www.slideshare.net/tylerjdmcconnell/testdriven-development-and-umple

Mealy, G. H. (1955). A Method to Synthesizing Sequential Circuits. Bell System Technical Journal. pp. 1045–1079.

Moore E. F. (1956). Gedanken-experiments on Sequential Machines. Automata Studies, Annals of Mathematical Studies, 34, 129–153. Princeton University Press, Princeton, N.J.

OMG (2013, September). "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.5

Pragmadev: Real time developer studio (RTDS). (2014). Retrieved from http://www.pragmadev.com/

Parviainen, P., Takalo, J., Teppola, S., & Tihinen, M. (2009). Model-Driven Development Processes and practices. *VTT Technical Research Centre, Tech. Rep*.

Patrick, N. (2006, March). Test driven development methodology. Retrieved February 4, 2013 from http://www.pnexpert.com/files/Test_Driven_Development.pdf

Rapp, C. W. The State Machine Compiler (SMC). Retrieved September, 2014 from http://smc.sourceforge.net.

Rouse, M. (2008). Continuous Integration (CI). In TechTarget. Retrieved from http://searchsoftwarequality.techtarget.com/definition/continuous-integration

Saunders C. M., Coulson N. R., and Folse J. L. Applications of Artificial Intelligence to Argicultural and Natural Resource Management Responsibilities in a Global Context, in Allen Kent and James G. Williams, eds.,*Encyclopedia of Computer Science and Technology, vol.* 25, supplement 10 (New York: Marcel Dekker, 1992), pp. 76–78.

SDL-RT. (2013, April). *Specification & Description Language – real time*. Retrieved from http://www.sdl-rt.org/standard/V2.2/html/SDL-RTa4.html

SDL Forum Society. (2013). *Towards SDL-2010*. Accessed 2014 from http://www.sdl-forum.org/ftp/pub/SDL-2010/index.htm.

Selic, B., & Rumbaugh, J. (1999). Mapping SDL to UML. *A Rational Software white paper*.

Singh, C. (2013). Difference between ArrayList and Vector In java. In Java Collections. Retrieved from http://beginnersbook.com/2013/12/difference-between-arraylist-and-vector-in-java/

Singh, C. (2013). Difference between ArrayList and HashMap In java. In Java Collections. Retrieved from http://beginnersbook.com/2013/12/difference-between-arraylist-and-hashmap-in-java/

Sparx. (2014). Enterprise Architect. Retrieved from http://www.sparxsystems.com

Uijen, J. (2009). *Learning Models of Communication Protocols using Abstraction Techniques*. (Masters Thesis). Uppsala University.

Umple User Manual. (2015). Retrieved May 28, 2012, from http://manual.umple.org

Van der Schoot, J. J. (1999). *Improving state exploration techniques for the automatic verification of concurrent systems* (PhD Thesis), University of Ottawa, Canada.

Venkataram, P., & Manvi, S. S. (2005). *Communication protocol Engineering*: PHI Learning Pvt. Ltd.

Venkataram, P. (2014). Introduction to Basics of Communication Protocol [PowerPoint slides]. Department of Electrical Communication Engineering, Indian Institute of Science, Bangalore, India. Retrieved from http://pet.ece.iisc.ernet.in/course/E2223/Cha1.pdf

Venkataram, P. (2014). *Protocol Validation* [PDF document]. Personal Collection of Communication Protocols. Dept. of Electrical Communication Engineering, Indian Institute of Science, Bangalore. Retrieved from Lecture Notes Online Web site http://pet.ece.iisc.ernet.in/course/E2223/

Vogel, L. (2013). *Java concurrency (multi-threading).* Retrieved from http://www.vogella.com/tutorials/JavaConcurrency/article.html

Völter, M., Stahl, T., Bettin, J., Haase, A., & Helsen, S. (2013). *Model-driven software development: technology, engineering, management*: John Wiley & Sons. Chichester, UK.

Zhang, Y., & Mackworth, A. K. (1993). *Design and analysis of embedded real-time systems: An elevator case study*, Tech. Rep. (TR 93-4). University of British Columbia, Vancouver, B.C. Retrieved from https://www.cs.ubc.ca/~mack/Publications/TR93-ZM.pdf.

Appendix A

A.1 TCP/IP Simulation Model (Umple)

```
1   // UML state machine and simulation of a the TCP-IP protocol
2
3   namespace tcp_simulation;
4
5   class Tcp {
6     depend java.util.LinkedList;
7     depend java.util.Queue;
8     depend java.lang.Thread;
9     depend java.io.*;
10    depend java.lang.InterruptedException;
11    depend java.io.IOException;
12
13    // Tcp flags
14    const String SYN="SYN";
15    const String ACK="ACK";
16    const String FIN="FIN";
17    const String RST="RST";
18    const String SYNACK="SYNAck";
19
20    // Queues for adding and removing Tcp flags
21    //LinkedList messages= new LinkedList();
22    Queue_Tcp q=new Queue_Tcp();
23    BufferedReader in=null;
24    lazy String sentence;
25
26    connection{
27      Closed{
28        passiveOpen->Listen;
29        activeOpen ->/{ sendSyn();}  SYN_Sent;
30      }
31      Listen{
32        syn ->/{ sendSynAck();} SYN_Received;
33        close -> Closed;
34      }
35      SYN_Received{
36        ack -> Established;
37      }
38      SYN_Sent{
39        synAck->/{ sendAck();} Established;
40      }
41      Established{
42        fin ->/{ sendAck();} Close_Wait;
43        activeClose->/{ sendFin();} FIN_Wait_1;
44        data->/{sendData();} Established;
45      }
46      Close_Wait{
47        passiveClose ->/{ sendFin();} Last_ACK;
48      }
49      Last_ACK{
50        ack -> Closed;
51      }
52      FIN_Wait_1{
53        ack ->FIN_Wait_2;
54        fin ->/{ sendAck();} Closing;
55        finAck ->/{ sendAck();} Timed_Wait;
56      }
```

```
57      FIN_Wait_2{
58        fin ->/{ sendAck();} Timed_Wait;
59      }
60      Closing{
61        ack -> Timed_Wait;
62      }
63      Timed_Wait{
64        timeOut->Closed;
65      }
66    }
67
68    public synchronized void sendSyn(){
69      try{
70        q.putMessage(SYN);
71      }catch (InterruptedException e) {}
72    }
73
74    public synchronized void sendSynAck() {
75       try{
76         q.putMessage(SYNACK);
77       }catch (InterruptedException e) {}
78    }
79
80    public synchronized void sendAck() {
81      try{
82        q.putMessage(ACK);
83      }catch (InterruptedException e) {}
84    }
85
86     public synchronized void sendData() {
87       in=new BufferedReader(new InputStreamReader(System.in));
88       String sn;
89       try{
90         sn=in.readLine();
91         try{
92           q.putMessage(sn);
93         }catch (InterruptedException e) {}
94       } catch (IOException ioe){}
95     }
96      public synchronized void sendFin() {
97        try{
98          q.putMessage(FIN);
99        }catch (InterruptedException e) {}
100     }
101 }
102 //Queue_1 class that have acquire and release methods used to send
103 //signals between the threads
104 class Queue_1{
105   depend java.util.concurrent.Semaphore;
106   depend java.lang.InterruptedException;
107
108   Semaphore semC = new Semaphore(0);
109   Semaphore semS = new Semaphore(1);
110   public void acquire_semC() {
111     try {
112       semC.acquire();
113     } catch (InterruptedException e) {
114       System.out.println("InterruptedException caught");
115     }
116   }
117   public void release_semC(){
118     semC.release();
119   }
```

```
120   public void acquire_semS() {
121     try {
122       semS.acquire();
123     } catch (InterruptedException e) {
124       System.out.println("InterruptedException caught");
125     }
126   }
127
128   public void release_semS(){
129    semS.release();
130   }
131 }
132
133 class Queue_Tcp{
134   depend java.lang.Thread;
135   depend java.lang.InterruptedException;
136   depend java.util.*;
137   //LinkedList messages;
138   LinkedList<String> messages = new LinkedList<String>();
139
140   public synchronized void putMessage(String var) throws
141   InterruptedException {
142     messages.add(var);
143     notify();
144     //String msg = (String)messages.getFirst();
145     String msg = messages.element();
146     System.out.println("send message: " +msg);
147   }
148
149   // Called by Receiver thread
150   public synchronized String getMessage() throws InterruptedException {
151     while ( messages.size() == 0 )
152       wait();
153     String message =messages.remove();
154     return (message);
155   }
156   public Boolean isEmptyMessage() {
157     if(messages.size() == 0)
158       return true;
159     return false;
160   }
161 }
162
163 external Thread{
164 }
165
166 class Receiver{
167   depend java.lang.Thread;
168   depend java.lang.InterruptedException;
169   isA Thread;
170   Queue_Tcp q;
171
172   public void run() {
173     try {
174       while ( !q.isEmptyMessage() ) {
175         String message = q.getMessage();
176         System.out.println("Got message: " + message);
177         sleep( 2000 );
178     }
179     }catch( InterruptedException e ) { }
180   }
181 }
```

```
182 class MySocket{
183
184   String host;
185   Integer port;
186
187   //MySocket needs a constructor to create the tcp
188   Tcp tcp=new Tcp();
189
190   // use a semaphore to synchronize sending signals between the threads
191   Queue_1 q=new Queue_1();
192
193   public void connect(){
194
195     //Connects client to the server
196     try{
197       tcp.activeOpen();
198       q.acquire_semS();
199     }catch(Exception e){}
200     q.release_semS();
201     tcp.synAck();
202   }
203
204   /**void receive(){
205     *//receive data from the server
206     *tcp.Data_R();
207   }**/
208
209   public void send(){
210      //send data to client
211      tcp.data();
212    }
213
214   public void close(){
215     //Closes the socket
216     tcp.activeClose();
217     //Closes the serverSocket
218     tcp.passiveClose();
219   }
220 }
221
223 class MyServerSocket{
224
225    Integer port;
226    Tcp tcp=new Tcp();
227    MySocket clientsocket=null;
228    MyServerSocket serverSocket=null;
229
230    // use a semaphore to synchronize sending signals between the threads
231    Queue_1 q=new Queue_1();
232
233    public void accept(){
234
235      //Listens for a connection to be made to this socket and accepts it
236      tcp.passiveOpen();
237
238      try{
239        tcp.syn();
240        q.acquire_semC();
241      }catch(Exception e){}
242      q.release_semC();
243      tcp.ack();
244    }
245 }
```

```
246 //create client class to implement MySocket class
247 class Tcp_Client
248 {
249   depend java.io.IOException;
250   depend java.lang.Thread;
251   depend java.lang.InterruptedException;
252   depend java.util.LinkedList;
253   isA Thread;
254   MySocket clientSocket=null;
255   boolean readyToStop= false;
256   Queue_Tcp q=new Queue_Tcp();
257   Receiver receive=new Receiver(q);
258   String host="localhost";
259   Integer port=999;
260
261   public void run() {
262     //LinkedList messages= new LinkedList();
263     //q=new Queue_Tcp();
264     //receive=new Receiver(q);
265     try
266     {
267       // connects client with the server
268       clientSocket= new MySocket(getHost(), getPort());
269       if(clientSocket != null)
270       {clientSocket.connect();}
271       while(!q.isEmptyMessage()){
272         receive.start();
273       }
274       System.out.println("Client: Connected");
275     } catch (Exception ex){
276       try
277       {
278         //Close the socket
279         if (clientSocket != null)
280           clientSocket.close();
281       }
282       finally
283       {
284         clientSocket = null;
285       }
286       System.out.println("Client: Closed");
287     }
288     if(clientSocket == null){
289       // closes the socket
290       setReadyToStop(true);
291       try
292       {
293         //Close the socket
294         if (clientSocket != null)
295         clientSocket.close();
296       }
297       finally
298       {
299         clientSocket = null;
300       }
301     }
302   }
303 }
304 //Creat Server class to implement the MyServerSocket class
305 class Tcp_Server
306 {
307   depend java.io.IOException;
308   depend java.lang.Thread;
```

```
  depend java.lang.InterruptedException;
  depend java.util.LinkedList;
  isA Thread;
  MyServerSocket serverSocket = null;
  MySocket clientSocket = null;
  Boolean isListening=false;
  Queue_Tcp q=new Queue_Tcp();
  Receiver receive=new Receiver(q);
  Integer port=999;
  Boolean readyToStop = true;

  public void run() {
    //LinkedList messages= new LinkedList();
    //q=new Queue_Tcp();
    //receive=new Receiver(q);
    //Begins waiting for a new client.
    if (!getIsListening())
    {
      if (serverSocket == null)
      {
        serverSocket = new MyServerSocket(getPort());
      }
    }
    //listens to the client and accept the connection
    setReadyToStop(false);
    serverStarted();
    try
    {
      // waits for a new client connection, accepts it
      while(!getReadyToStop())
      {
        try
        {
          receive=new Receiver(new Queue_Tcp());
          if(serverSocket != null){
            serverSocket.accept();}
          while(!q.isEmptyMessage()){
            receive.start();
          }
          //Sends a message to client connected to the server
          if (clientSocket == null)
            System.out.println("socket does not exist");
           try
           {
            clientSocket.send();
           } catch (Exception ex) {}
        }catch (Exception ex){}
       }
    } catch (Exception ex){}
    finally
    {
      setReadyToStop(true);
      serverStopped();
      //Closes all connection to the server
      try
      {
        // Close the sSocket and the connection with client
        if (serverSocket == null)
        {
          stopListening();
        }else if (serverSocket != null || clientSocket != null)
          try
          {
```

```
372             clientSocket.close();
373           } catch(Exception ex) {}
374       } catch(Exception ex) {}
375       finally
376       {
377         clientSocket = null;
378         serverSocket = null;
379       }
380     }
381   }
382
383   protected void serverStarted() {
384     System.out.println("Server: Started");
385   }
386
387   protected void serverStopped() {
388     System.out.println("Server: Stopped");
389   }
390
391   //Causes the server to stop accepting new connections
392   public void stopListening(){
393     setReadyToStop(true);
394   }
395
396 }
397
398 // mainTest class
399 class MainTest{
400   depend java.lang.Thread;
401
402   public static void main(String[] args) {
403     Tcp_Server server= new Tcp_Server();
404     server.start();
405     Tcp_Client client= new Tcp_Client();
406     client.start();
407   }
408 }
```

Appendix B

B.1 Simple Queued State Machine (QSM) (Java)

```
1   public class X implements Runnable
2   {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6     //X State Machines
7     enum Sm { s1, s2 }
8     private Sm sm;
9     //enumeration type of messages accepted by X
10    enum MessageType { e1_M, e2_M }
11    MessageQueue queue;
12    Thread removal;
13
14    //------------------------
15    // CONSTRUCTOR
16    //------------------------
17    public X()
18    {
19      setSm(Sm.s1);
20      queue = new MessageQueue();
21      removal=new Thread(this);
22      //start the thread of X
23      removal.start();
24    }
25
26    //------------------------
27    // INTERFACE
28    //------------------------
29    public String getSmFullName()
30    {
31      String answer = sm.toString();
32      return answer;
33    }
34    public Sm getSm()
35    {
36      return sm;
37    }
38
39    public boolean _e1()
40    {
41      boolean wasEventProcessed = false;
42
43      Sm aSm = sm;
44      switch (aSm)
45      {
46        case s1:
47          setSm(Sm.s2);
48          wasEventProcessed = true;
49          break;
50         default:
51          // Other states do respond to this event
52      }
53      return wasEventProcessed;
54    }
55
56    public boolean _e2()
57    {
```

```
58      boolean wasEventProcessed = false;
59      Sm aSm = sm;
60      switch (aSm)
61      {
62        case s2:
63          setSm(Sm.s1);
64          wasEventProcessed = true;
65          break;
66        default:
67          // Other states do respond to this event
68      }
69
70      return wasEventProcessed;
71    }
72
73    private void setSm(Sm aSm)
74    {
75      sm = aSm;
76    }
77    public void delete()
78    {}
79
80    protected class Message
81    {
82      MessageType type;
83
84      //Message parameters
85      Vector<Object> param;
86
87      public Message(MessageType t, Vector<Object> p)
88      {
89        type = t;
90        param = p;
91      }
92
93      @Override
94      public String toString()
95      {
96        return type + "," + param;
97      }
98    }
99
100   protected class MessageQueue {
101     Queue<Message> messages = new LinkedList<Message>();
102
103     public synchronized void put(Message m)
104     {
105       messages.add(m);
106       notify();
107     }
108     public synchronized Message getNext()
109     {
110       try {
111         while (messages.isEmpty())
112         {
113           wait();
114         }
115       } catch (InterruptedException e) { e.printStackTrace(); }
116       //The element to be removed
117       Message m = messages.remove();
118       return (m);
119     }
120   }
```

```
121   //------------------------------
122   //messages accepted
123   //------------------------------
124   public void e1 ()
125   {
126     queue.put(new Message(MessageType.e1_M, null));
127   }
128   public void e2 ()
129   {
130     queue.put(new Message(MessageType.e2_M, null));
131   }
132
133   @Override
134   public void run ()
135   {
136     boolean status=false;
137     while (true)
138     {
139       Message m = queue.getNext();
140
141       switch (m.type)
142       {
143         case e1_M:
144           status = _e1();
145           break;
146         case e2_M:
147           status = _e2();
148           break;
149         default:
150       }
151       if(!status)
152       {
153         // Error message is written or  exception is raised
154       }
155     }
156   }
157   // line 13 "model.ump"
158    public static  void main(String [] args){
159     X x = new X();
160     x.e1();
161     x.e2();
162     x.e2();
163     x.e1();
164   }
165 }
```

B. 2 Simple Pooled State Machine (PSM) (Java)

```
1   public class X implements Runnable
2   {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6     //X State Machines
7     enum Sm { s1, s2 }
8     private Sm sm;
9     MessagePool pool;
10    Thread removal;
11    //enumeration type of messages accepted by X
12    enum MessageType { e1_M, e2_M }
13    // Map for a X pooled state machine that allows querying which events
14    //are possible in each map
15    public static final Map<Object, HashSet<MessageType>> stateMessageMap =
16    new HashMap<Object, HashSet<MessageType>>();
17    static {
18      stateMessageMap.put(Sm.s1,new
19      HashSet<MessageType>(Arrays.asList(MessageType.e1_M)));
20      stateMessageMap.put(Sm.s2,new
21      HashSet<MessageType>(Arrays.asList(MessageType.e2_M)));
22    }
23
24    //------------------------
25    // CONSTRUCTOR
26    //------------------------
27    public X()
28    {
29      setSm(Sm.s1);
30      pool = new MessagePool();
31      removal=new Thread(this);
32      //start the thread of X
33      removal.start();
34    }
35    protected class Message
36    {
37      MessageType type;
38      //Message parameters
39      Vector<Object> param;
40      public Message(MessageType t, Vector<Object> p)
41      {
42        type = t;
43        param = p;
44      }
45      @Override
46      public String toString()
47      {
48        return type + "," + param;
49      }
50    }
51    public boolean _e1()
52    {
53      boolean wasEventProcessed = false;
54      Sm aSm = sm;
55      switch (aSm)
56      {
57        case s1:
58          setSm(Sm.s2);
59          wasEventProcessed = true;
60          break;
61        default:
```

```
62        // Other states do respond to this event
63      }
64      return wasEventProcessed;
65    }
66
67    public boolean _e2()
68    {
69      boolean wasEventProcessed = false;
70      Sm aSm = sm;
71      switch (aSm)
72      {
73        case s2:
74          setSm(Sm.s1);
75          wasEventProcessed = true;
76          break;
77        default:
78          // Other states do respond to this event
79      }
80      return wasEventProcessed;
81    }
82
83    private void setSm(Sm aSm)
84    {
85      sm = aSm;
86    }
87
88    public void delete()
89    {}
90
91    //------------------------------
92    //messages accepted
93    //------------------------------
94    public void e1 ()
95    {
96      pool.put(new Message(MessageType.e1_M, null));
97    }
98
99    public void e2 ()
100   {
101     pool.put(new Message(MessageType.e2_M, null));
102   }
103   protected class MessagePool {
104     Queue<Message> messages = new LinkedList<Message>();
105     public synchronized void put(Message m)
106     {
107       messages.add(m);
108       notify();
109     }
110
111     public synchronized Message getNext()
112     {
113       Message message=null;
114
115       try {
116         message=getNextProcessableMessage();
117         while (message==null)
118         {
119           wait();
120           message=getNextProcessableMessage();
121         }
122       } catch (InterruptedException e) { e.printStackTrace(); }
123       // return the message
124       return (message);
125     }
```

```
126     public Message getNextProcessableMessage()
127     {
128       // Iterate through messages and remove the first message that
129       // matches one of the Messages list
130       // otherwise return null
131       for (Message msg: messages)
132       {
133         if(stateMessageMap.get(getSm()).contains(msg.type))
134         {
135           //The element to be removed
136           messages.remove(msg);
137           return (msg);
138         }
139       }
140       return null;
141     }
142   }
143   //------------------------
144   // INTERFACE
145   //------------------------
146   public String getSmFullName() {
147     String answer = sm.toString();
148     return answer;
149   }
150
151   public Sm getSm() {
152     return sm;
153   }
154   @Override
155   public void run ()
156   {
157     boolean status=false;
158     while (true)
159     {
160       Message m = pool.getNext();
161
162       switch (m.type)
163       {
164         case e1_M:
165           status = _e1();
166           break;
167         case e2_M:
168           status = _e2();
169           break;
170         default:
171       }
172       if(!status)
173       {
174         // Error message is written or  exception is raised
175       }
176     }
177   }
178    public static  void main(String [] args){
179     Thread.currentThread().setUncaughtExceptionHandler(new
180     UmpleExceptionHandler());
181     Thread.setDefaultUncaughtExceptionHandler(new
182     UmpleExceptionHandler());
183     X x = new X();
183     x.e1();
184     x.e2();
185     x.e2();
186     x.e1();
187   }
```

B.3 Simple Queued State Machine (Events With No Parameters) (Java)

```
1   /*PLEASE DO NOT EDIT THIS CODE*/
2   /*This code was generated using the UMPLE 1.20.1.4254 modeling
3    *language!
4    */
5   import java.util.*;
6   import java.lang.Thread;
7
8   // line 2 "model.ump"
9   // line 21 "model.ump"
10  public class X implements Runnable{
11    //------------------------
12    // MEMBER VARIABLES
13    //------------------------
14    //X State Machines
15    enum Sm { s1, s2, s3, s4, s5 }
16    private Sm sm;
17    //enumeration type of messages accepted by X
18    enum MessageType { e1_M, e2_M, e3_M, e4_M }
19    MessageQueue queue;
20    Thread removal;
21    //------------------------
22    // CONSTRUCTOR
23    //------------------------
24    public X(){
25      setSm(Sm.s1);
26      queue = new MessageQueue();
27      removal=new Thread(this);
28      //start the thread of X
29      removal.start();
30    }
31    //------------------------
32    // INTERFACE
33    //------------------------
34    public String getSmFullName()
35    {
36      String answer = sm.toString();
37      return answer;
38    }
39    public Sm getSm()
40    {
41      return sm;
42    }
43    public boolean _e1() {
44      boolean wasEventProcessed = false;
45      Sm aSm = sm;
46      switch (aSm)
47      {
48       case s1:
49          setSm(Sm.s2);
50          wasEventProcessed = true;
51         break;
52       default:
53          // Other states do respond to this event
54      }
55      return wasEventProcessed;
56    }
57
58    public boolean _e2(){
59      boolean wasEventProcessed = false;
60      Sm aSm = sm;
```

```
61      switch (aSm)
62      {
63        case s2:
64          setSm(Sm.s3);
65          wasEventProcessed = true;
66          break;
67        default:
68          // Other states do respond to this event
69      }
70      return wasEventProcessed;
71    }
72    public boolean _e3(){
73      boolean wasEventProcessed = false;
74      Sm aSm = sm;
75      switch (aSm)
76      {
77        case s3:
78          setSm(Sm.s4);
79          wasEventProcessed = true;
80          break;
81        default:
82          // Other states do respond to this event
83      }
84      return wasEventProcessed;
85    }
86
87    public boolean _e4(){
88      boolean wasEventProcessed = false;
89      Sm aSm = sm;
90      switch (aSm)
91      {
92        case s4:
93          setSm(Sm.s5);
94          wasEventProcessed = true;
95          break;
96        default:
97          // Other states do respond to this event
98      }
99       return wasEventProcessed;
100   }
101   private void setSm(Sm aSm){
102     sm = aSm;
103   }
104   public void delete()
105   {}
106
107   protected class Message
108   {
109     MessageType type;
110     //Message parameters
111     Vector<Object> param;
112     public Message(MessageType t, Vector<Object> p)
113     {
114       type = t;
115       param = p;
116     }
117     @Override
118     public String toString()
119     {
120       return type + "," + param;
121     }
122   }
```

```
123   protected class MessageQueue {
124     Queue<Message> messages = new LinkedList<Message>();
125
126     public synchronized void put(Message m)
127     {
128       messages.add(m);
129       notify();
130     }
131     public synchronized Message getNext() {
132       try {
133         while (messages.isEmpty())
134         {
135           wait();
136         }
137       } catch (InterruptedException e) { e.printStackTrace(); }
138       //The element to be removed
139       Message m = messages.remove();
140       return (m);
141     }
142   }
143   //--------------------------
144   //messages accepted
145   //--------------------------
146   public void e1 (){
147     queue.put(new Message(MessageType.e1_M, null));
148   }
149   public void e2 (){
150     queue.put(new Message(MessageType.e2_M, null));
151   }
152   public void e3 (){
153     queue.put(new Message(MessageType.e3_M, null));
154   }
155   public void e4 (){
156     queue.put(new Message(MessageType.e4_M, null));
157   }
158   @Override
159   public void run (){
160     boolean status=false;
161     while (true) {
162       Message m = queue.getNext();
163       switch (m.type)
164       {
165         case e1_M:
166           status = _e1();
167           break;
168         case e2_M:
169           status = _e2();
170           break;
171         case e3_M:
172           status = _e3();
173           break;
174         case e4_M:
175           status = _e4();
176           break;
177         default:
178       }
179       if(!status)
180       {
181         // Error message is written or exception is raised
182       }
183     }
184   }
185 }
```

B.4 Simple Queued State Machine (Events With Parameters) (Java)

```
1   /*PLEASE DO NOT EDIT THIS CODE*/
2   /*This code was generated using the UMPLE 1.20.2.4413 modeling
3   language!*/
4   import java.util.*;
5   import java.lang.Thread;
6
7   // line 2 "model.ump"
8   // line 21 "model.ump"
9   public class X implements Runnable
10  {
11    //------------------------
12    // MEMBER VARIABLES
13    //------------------------
14    //X State Machines
15    enum Sm { s1, s2, s3, s4, s5 }
16    private Sm sm;
17    //enumeration type of messages accepted by X
18    enum MessageType { e1_M, e2_M, e3_M, e4_M }
19    MessageQueue queue;
20    Thread removal;
21    //------------------------
22    // CONSTRUCTOR
23    //------------------------
24    public X()
25    {
26      setSm(Sm.s1);
27      queue = new MessageQueue();
28      removal=new Thread(this);
29      //start the thread of X
30      removal.start();
31    }
32    //------------------------
33    // INTERFACE
34    //------------------------
35    public String getSmFullName(){
36      String answer = sm.toString();
37      return answer;
38    }
39    public Sm getSm(){
40      return sm;
41    }
42    public boolean _e1(Integer i)
43    {
44      boolean wasEventProcessed = false;
45      Sm aSm = sm;
46      switch (aSm)
47      {
48        case s1:
49          setSm(Sm.s2);
50          wasEventProcessed = true;
51          breake;
52        default:
53          // Other states do respond to this event
54      }
54      return wasEventProcessed;
55    }
56    public boolean _e2(){
57      boolean wasEventProcessed = false;
58      Sm aSm = sm;
59      switch (aSm)
```

```
58      {
59        case s2:
60          setSm(Sm.s3);
61          wasEventProcessed = true;
62          break;
63        default:
64          // Other states do respond to this event
65      }
66      return wasEventProcessed;
67    }
68    public boolean _e3(String name){
69      boolean wasEventProcessed = false;
70      Sm aSm = sm;
71      switch (aSm)
72      {
73        case s3:
74          setSm(Sm.s4);
75          wasEventProcessed = true;
76          break;
77        default:
78          // Other states do respond to this event
79      }
80      return wasEventProcessed;
81    }
82    public boolean _e4()
83    {
84      boolean wasEventProcessed = false;
85      Sm aSm = sm;
86      switch (aSm)
87      {
88        case s4:
89          setSm(Sm.s5);
90          wasEventProcessed = true;
91          break;
92        default:
93          // Other states do respond to this event
94      }
95      return wasEventProcessed;
96    }
97    private void setSm(Sm aSm)
98    {
99      sm = aSm;
100   }
101   public void delete()
102   {}
103   protected class Message
104   {
105     MessageType type;
106     //Message parameters
107     Vector<Object> param;
108     public Message(MessageType t, Vector<Object> p)
109     {
110       type = t;
111       param = p;
112     }
113     @Override
114     public String toString()
115     {
116       return type + "," + param;
117     }
118   }
119 protected class MessageQueue {
120     Queue<Message> messages = new LinkedList<Message>();
```

```
121      public synchronized void put(Message m)
122     {
123       messages.add(m);
124       notify();
125     }
126     public synchronized Message getNext(){
127       try {
128         while (messages.isEmpty()){
129           wait();
130         }
131       } catch (InterruptedException e) { e.printStackTrace(); }
132       //The element to be removed
133       Message m = messages.remove();
134       return (m);
135     }
136   }
137   //--------------------------
138   //messages accepted
139   //--------------------------
140   public void e1 (Integer i){
141     Vector v = new Vector(1);
142     v.add(0, i);
143     queue.put(new Message(MessageType.e1_M, v));
144   }
145   public void e2 (){
146     queue.put(new Message(MessageType.e2_M, null));
147   }
148   public void e3 (String name){
149     Vector v = new Vector(1);
150     v.add(0, name);
151     queue.put(new Message(MessageType.e3_M, v));
152   }
153   public void e4 (){
154     queue.put(new Message(MessageType.e4_M, null));
155   }
156   @Override
157   public void run (){
158     boolean status=false;
159     while (true) {
160       Message m = queue.getNext();
161       switch (m.type){
162         case e1_M:
163           status = _e1((Integer) m.param.elementAt(0));
164           break;
165         case e2_M:
166           status = _e2();
167           break;
168         case e3_M:
169           status = _e3((String) m.param.elementAt(0));
170           break;
171         case e4_M:
172           status = _e4();
173           break;
174         default:
175       }
176       if(!status){
177         // Error message is written or  exception is raised}
178       }
179     }
180   }
181 }
```

Appendix C

C.1 Key Code Generation Jet Templates Files For QSM and PSM

Template name (*.JET)	Function

JavaClassGenerator.jumjet	It is a generic file used to generate JavaClassGenerator.Java that is the translator that will be used by the compiler later. We adapt this file to allow the class generated from Umple code to implement Runnable interface.
	`for (StateMachine smq : uClass.getStateMachines()){`
	`  if (smq.isQueued()){`
	`  append(stringBuffer," implements Runnable");`
	`  break;`
	`  }`
	`  else if(smq.isPooled()){`
	`  append(stringBuffer," implements Runnable");`
	`  break;`
	`  }`
	`}`
	Also, we add some templates related to the generated code of queued and pooled state machines to this generic file by looping over all state machines, and when a queued or pooled state machine is detected, these template files are called.
	`for (StateMachine smq : uClass.getStateMachines()){`
	`  if (smq.isPooled())`
	`  {%>`
	`  <%@ include file="queued_state_machine_inner_class.jet" %>`
	`  <%@ include file="queued_state_machine_queuedEvent.jet" %>`
	`  <%@ include file="queued_state_machine_removalThread_run.jet" %>`
	`  <% break;`
	`  }`
	`  if (smq.isQueued())`
	`  {%>`
	`  <%@ include file="queued_state_machine_inner_class.jet"  %>`
	`  <%@ include file="queued_state_machine_queuedEvent.jet" %>`
	`  <%@ include file="queued_state_machine_removalThread_run.jet" %>`
	`  <% break;`
	`  }`
	`}%>`
	For queued and pooled state machines, the template calls:
	`queued_state_machine_inner_class`
	`queued_state_machine_queuedEvent`
	`queued_state_machine_removalThread_run`

Template name (*.JET)	Function

constructor_Declare Default.jet	We make here the required changes to allow for generating the constructor of the class with a queued or pooled state machine. The constructor will have the initialization of the queue where the events are queued or the pool where the event are added into the pool, as well as the initialization and run of the queuing or pooling removal thread. It also includes calling start(0) method to run the thread.
	This can be done by looping over all state machines and when the queued or pooled state machine is detected, then make the required changes.
	`if(foundQueued == true)`
	`{`
	`  append(stringBuffer,"\n    queue = new MessageQueue();");`
	`  append(stringBuffer,"\n    removal=new Thread(this);");`
	`  append(stringBuffer,"\n    //start the thread of {0}", uClass.getName());`
	`  append(stringBuffer,"\n    removal.start();");`
	`}`
	`if(foundPooled == true)`
	`{`
	`  append(stringBuffer,"\n    pool = new MessagePool();");`
	`  append(stringBuffer,"\n    removal=new Thread(this);");`
	`  append(stringBuffer,"\n    //start the thread of {0}", uClass.getName());`
	`  append(stringBuffer,"\n    removal.start();");`
	`}`

members_AllStateMachines	The change we make here is to loop over all state machines, and when a queued or pooled state machine is detected, there will be two attributes to be added to the generated code which are:
	MessageQueue queue/MessagePool pool: to define the queue/pool in which messages will be queued/pooled.
	Thread removal: that is the queuing/pooling removal processor that removes the events from the queue/pool if there is one.
	For a queued state machine:
	`  append(stringBuffer,"\n  MessageQueue queue;");`
	`  append(stringBuffer,"\n  Thread removal;");`
	For a pooled state machine:
	`  append(stringBuffer,"\n  MessagePool pool;");`

Template name (*.JET)	Function

	`append(stringBuffer,"\n  Thread removal;");`
	
	Also, the enumeration of message types will be defined which has messages of all events of the state machine.
	
	`append(stringBuffer,"\n  //enumeration type of messages accepted by {0}", uClass.getName());`
	
	`append(stringBuffer, "\n  enum MessageType { {0} }", gen.translate("listEventsForQSM",uClass));`
	
	In the case of pooled state machine, the enumerated values of messages types will contain 'null_M' message type if there is a state or substate that does not have any events.
	
	The reason for this is that we will also create a set of message types for each state and substate if using Map.
	
	`{`
	`append(stringBuffer,"\n  //enumeration type of messages accepted by {0}", uClass.getName());`
	
	`append(stringBuffer, "\n  enum MessageType { {0} }", gen.translate("listEventsForPooledSM",uClass));`
	`}`
	
	`if(foundPooled == true)`
	`{`
	`append(stringBuffer,"\n");`
	`append(stringBuffer,"\n  // Map for a {0} pooled state machine that allows querying which events are possible in each map", uClass.getName());`
	`append(stringBuffer,"\n");`
	`append(stringBuffer,"\n  public static final Map<Object, HashSet<MessageType>> stateMessageMap = new HashMap<Object, HashSet<MessageType>>();");`
	`append(stringBuffer,"\n  static {");`
	`}`
	
	`for(StateMachine sm : uClass.getStateMachines())`
	`{`
	`if(sm.isPooled())`
	`{`
	`append(stringBuffer,"\n {0}",gen.translate("listMessageTypesStates",sm));`
	`}`
	`}`

Template name (*.JET)	Function
state_machine_Event	What we do here is to add the _ underscore symbol before the event name for each event handling method, including regular and timed events of a state machine but not the unspecified event. Also, we allow for events to have any number of arguments. If the event is defined to be an 'unspecified' event, then it will have two arguments: state and event as strings. `<%= scope %> boolean <%for (StateMachine sm : uClass.getStateMachines()){if((sm.isQueued() && e.getIsInternal() == false && e.isAutoTransition() == false && !e.isUnspecified()) || (sm.isPooled() && e.getIsInternal() == false && e.isAutoTransition() == false && !e.isUnspecified())){append(stringBuffer,"_");}break;}%><%=gen.translate("eventMethod",e)%>(<%=(e.getArgs()==null?"":e.getArgs())%><%if(e.isUnspecified()){append(stringBuffer,"String state, String event");}%>)`
queued_state_machine_queuedEvent.jet	This is a new template added to create state machine events handling methods in the case of queued or pooled state machines. When these event handling methods are called, they add events to the queue/pool including the event message types and any parameters they have. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_queuedEvent.jet
queued_state_machine_removalThread_run.jet	This new template is added to handle the generated code for the run method of the queuing and pooling removal thread. It contains the run method which when it starts, has a while loop to remove the events from the queue/pool and processes it, then it will check for the next event in the queue/pool and if there is any, it will remove it and process it and so on. If there are no events in the queue/pool, it will wait for an event to be added. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_removalThread_run.jet

Template name (*.JET)	Function
queued_state_machine_inner_class.jet	This a new file is added to handle the code for the Message and MessagQueue/MessagPool inner classes. These classes will be generated for queued or pooled state machines. The Message class is used for representing any message. The MessageQueue class has put and getNext methods that add and remove the events from the queue. The MessagePool class has a different getNext method which works in a different way while taking the events off the pool. You can find this template at location: http://code.google.com/p/umple/source/browse/trunk/UmpleToJava/templates/queued_state_machine_inner_class.jet

Appendix D

D.1 Semantic Test For a Queued State Machine

```
1  package cruise.queued.statemachine.test;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5  import static org.junit.Assert.*;
6  import static org.hamcrest.CoreMatchers.*;
7
8  public class QueuedStateMachineTest{
9    @Test
10   public void numberOfMessagesInMessageType()
11   {
12     // compare the number of messages in MessageType is equal to the
13     // number of events in State Machine except unspecified events and
14     // auto-transition
15     Assert.assertEquals(2, X.MessageType.values().length);
16     Assert.assertEquals(true,
17     QueuedSM.MessageType.valueOf("e1_M").equals(X.MessageType.e1_M));
18     Assert.assertEquals(true,
19     QueuedSM.MessageType.valueOf("e2_M").equals(X.MessageType.e2_M));
20   }
21   @Test
22   public void processEvents() throws InterruptedException
23   {
24     X qsm = new X();
25     int numChecks;
26     //initial state is s1
27     Assert.assertEquals(X.Sm.s1, qsm.getSm());
28     //e1 is triggered: e1 is queued
29     qsm.e1();
30     //e1 is dequeued and processed: transition to s2
31     numChecks=200; // we will check for a second
32     while(!qsm.getSm().equals(X.Sm.s2) && numChecks>0) {
33       Thread.sleep(5);
34       numChecks--;
35     }
36     AssertThat(numChecks, not(equalTo(0)));
37     Assert.assertEquals(X.Sm.s2, qsm.getSm());
38     // check if there is a message saved in the queue
39     noMessageIsSaved(qsm);
40
41     //e2 is triggered: e2 is queued
42     qsm.e2();
43     //e2 is dequeued and processed: transition to s2
44     numChecks=200; // we will check for a second
45     while(numChecks>0 && qsm.getSm().equals(X.Sm.s2)) {
46       if(!qsm.queue.messages.isEmpty()){
47         Thread.sleep(5);
48         numChecks--;
49       }
50       else
51       {
52         Assert.assertEquals(X.Sm.s2, qsm.getSm());
53         Assert.assertEquals(true, qsm.queue.messages.isEmpty());
54         break;
55       }
56     }
```

```
57      AssertThat(numChecks, not(equalTo(0)));
58      Assert.assertEquals(X.Sm.s2, qsm.getSm());
59      // check if there is a message saved in the queue
60      noMessageIsSaved(qsm);
61      //e2 is triggered: e2 is queued
62      qsm.e2();
63      //e2 is dequeued and processed: transition to s2
64      numChecks=200; // we will check for a second
65      while(numChecks>0 && qsm.getSm().equals(QueuedSM.Sm.s2)) {
66        if(!qsm.queue.messages.isEmpty()){
67          Thread.sleep(5);
68          numChecks--;
69        }
70        else
71        {
72          Assert.assertEquals(X.Sm.s2, qsm.getSm());
73          Assert.assertEquals(true, qsm.queue.messages.isEmpty());
74          break;
75        }
76      }
77      AssertThat(numChecks, not(equalTo(0)));
78      Assert.assertEquals(X.Sm.s2, qsm.getSm());
79      // check if there is a message saved in the queue
80      noMessageIsSaved(qsm);
81      //e1 is triggered: e1 is queued
82      qsm.e1();
83      //e1 is dequeued and ignored (not processed: case of unspecified
84      //reception)
85      numChecks=200; // we will check for a second
86      while(numChecks>0 && qsm.getSm().equals(X.Sm.s2))    {
87        if(!qsm.queue.messages.isEmpty()){
88          Thread.sleep(5);
89          numChecks--;
90        }
91        else
92        {
93          Assert.assertEquals(X.Sm.s2, qsm.getSm());
94          Assert.assertEquals(true, qsm.queue.messages.isEmpty());
95          break;
96        }
97      }
98      AssertThat(numChecks, not(equalTo(0)));
99      Assert.assertEquals(X.Sm.s2, qsm.getSm());
100     // check if there is a message saved in the queue
101     noMessageIsSaved(qsm);
102
103     //check that there is no events left in the queue
104     Assert.assertEquals(0, qsm.queue.messages.size());
105   }
106   public void noMessageIsSaved(QueuedSM qsm)
107   {
108     if(!qsm.queue.messages.isEmpty())
109     {
110        Assert.assertEquals(false, qsm.queue.messages.isEmpty());
111     }
112     else
113       Assert.assertEquals(0, qsm.queue.messages.size());
114   }
115 }
```

D.2 Semantic Test For a Pooled State Machine

```
1   package cruise.pooled.statemachine.test;
2
3   import org.junit.Assert;
4   import org.junit.Tes;
5   import static org.junit.Assert.*;
6   import static org.hamcrest.CoreMatchers.*;
7
8   public class PooledStateMachineTest
9   {
10    @Test
11    public void numberOfMessagesInMessageType()
12    {
13      // compare the number of messages in MessageType is equal to the
14      // number of events in State Machine except and auto-transition
15      Assert.assertEquals(2, X.MessageType.values().length);
16      Assert.assertEquals(true,
17      X.MessageType.valueOf("e1_M").equals(X.MessageType.e1_M));
18      Assert.assertEquals(true,
19      PooledSM.MessageType.valueOf("e2_M").equals(X.MessageType.e2_M));
20    }
21    @Test
22    public void numberOfKeysInstateMessageMap()
23    {
24      // compare the number of states is equal to the number of keys in
25      // stateMessageMap
26      Assert.assertEquals(X.Sm.values().length,
27      X.stateMessageMap.keySet().size());
28      Assert.assertEquals(2, X.stateMessageMap.keySet().size());
29    }
30    @Test
31    public void eachStateOfStateMachineAddedTostateMessageMap()
32    {
33      //check that every state of SM it is put in stateMessageMap
34      Assert.assertEquals(true, X.stateMessageMap.containsKey(X.Sm.s1));
35      Assert.assertEquals(true, X.stateMessageMap.containsKey(X.Sm.s2));
36    }
37    @Test
38    public void everyStateHasListOfMessages()
39    {
40      //check that every state has its set Of messages
41      Assert.assertEquals(1, X.stateMessageMap.get(X.Sm.s1).size());
42      Assert.assertEquals(1, X.stateMessageMap.get(X.Sm.s2).size());
43      Assert.assertEquals(true,X.stateMessageMap.get(X.Sm.s1)
44      .containsAll(X.stateMessageMap.get(X.Sm.s1)));
45      Assert.assertEquals(true,PooledSM.stateMessageMap.get(X.Sm.s2)
46      .containsAll(X.stateMessageMap.get(X.Sm.s2)));
47    }
48
49    @Test
50    public void sizeOfstateMessageMap()
51    {
52      //size of stateMessageMap which contains (state, list of
53      //MessageTypes)
54      Assert.assertEquals(2, X.stateMessageMap.size());
55    }
56    @Test
57    public void processEvents() throws InterruptedException
58    {
59      X psm = new X();
```

```
60      int numChecks;
61      // check initial state is s1
62      Assert.assertEquals(X.Sm.s1, psm.getSm());
63
64      // e1 is triggered: e1 is added to the message pool
65      psm.e1();
66      numChecks=200; // we will check for a second
67      // e1 is removed from the pool and is processed: transition to s2
68      while(!psm.pool.messages.isEmpty() && numChecks>0) {
69        if(!psm.getSm().equals(X.Sm.s1)){
70          Thread.sleep(5);
71          numChecks--;
72        }
73      }
74      AssertThat(numChecks, not(equalTo(0)));
75      Assert.assertEquals(X.Sm.s2, psm.getSm());
76      //there is no message saved in the pool
77      Assert.assertEquals(0, psm.pool.messages.size());
78
79      // e2 is triggered: e2 is added to the message pool
80      psm.e2();
81      numChecks=200;
82      // e2 is removed from the pool and is processed: transition to s1
83      while(numChecks>0) {
84        Thread.sleep(5);
85        numChecks--;
86        if(psm.pool.messages.isEmpty())
87        {
88          Assert.assertEquals(X.Sm.s1, psm.getStatus());
89          Assert.assertEquals(true, psm.pool.messages.isEmpty());
90          break;
91        }
92      }
93      AssertThat(numChecks, not(equalTo(0)));
94      Assert.assertEquals(X.Sm.s1, psm.getSm());
95      //there is no message saved
96      Assert.assertEquals(0, psm.pool.messages.size());
97
98      // e2 is triggered: e2 is added to the message pool
99      psm.register();
100     numChecks=200; // we will check for a second
101     // e2 is pooled at the head of pool: it is not processed
102     while(numChecks>0) {
103       Thread.sleep(5);
104       numChecks--;
105       if(psm.pool.messages.size() ==1)
106       {
107         Assert.assertEquals(X.Sm.s1, psm.getSm());
108         Assert.assertEquals(false, psm.pool.messages.isEmpty());
109         if(psm.pool.messages.size() == 1 &&
110         psm.pool.messages.element().type.equals(X.MessageType.e2_M)) {
111           Assert.assertEquals(1, psm.pool.messages.size());
112           break;
113         }
114       }
115     }
116     AssertThat(numChecks, not(equalTo(0)));
117     Assert.assertEquals(X.Sm.s1, psm.getSm());
118     // check that the message 'e2_M' is pooled
119     for (X.Message msg: psm.pool.messages)
120     {
121       if(msg.type.equals(X.MessageType.e2_M))
122       {
```

```
123             Assert.assertEquals(X.MessageType.e2_M, msg.type);
124           }
125         }
126         //Now, there are one messages saved at the head of the pool
127         Assert.assertEquals(1, psm.pool.messages.size());
128
129         // e1 is triggered: e1 is added to the message pool(queue)
130         psm.e1();
131         numChecks=200; // we will check for a second
132         // e1 is removed from the pool and is processed: transition to s2
133         // e2 is removed and it is processed: transition to s1
134         while(!psm.pool.messages.isEmpty() && numChecks>0) {
135           if(!psm.getStatus().equals(X.Sm.s1))
136           {
137             Thread.sleep(5);
138             numChecks--;
139           }
140         }
141         AssertThat(numChecks, not(equalTo(0)));
142         Assert.assertEquals(X.Sm.s1, psm.getSm());
143         //there is no message saved: e2 is removed and processed
144         Assert.assertEquals(0, psm.pool.messages.size());
145 }
```

D.3 Semantics Test For QSM Using Unspecified Reception Mechanism

```
1  package cruise.queued.statemachine.test;
2
3  import static org.hamcrest.CoreMatchers.equalTo;
4  import static org.hamcrest.CoreMatchers.not;
5  import static org.junit.Assert.assertThat;
6  import org.junit.Assert;
7  import org.junit.Test;
8
9  public class QueuedStateMachineTest_UnspecifiedReception
10 {
11   @Test
12   public void numberOfMessagesInMessageType()
13   {
14     // compare the number of messages in MessageType is equal to the
15     // number of events in State Machine except timed events and auto-
16     // transition
17     Assert.assertEquals(9,
18     AutomatedTellerMachine.MessageType.values().length);
19     Assert.assertEquals(true,
20     AutomatedTellerMachine.MessageType.valueOf("cardInserted_M").equals(A
21     utomatedTellerMachine.MessageType.cardInserted_M));
22     Assert.assertEquals(true,
23    AutomatedTellerMachine.MessageType.valueOf("maintain_M").equals(Automa
24    tedTellerMachine.MessageType.maintain_M));
25     Assert.assertEquals(true,
26    AutomatedTellerMachine.MessageType.valueOf("isMaintained_M").equals(Au
27    tomatedTellerMachine.MessageType.isMaintained_M));
28     Assert.assertEquals(true,
29    AutomatedTellerMachine.MessageType.valueOf("cancel_M").equals(Automate
30    dTellerMachine.MessageType.cancel_M));
31     Assert.assertEquals(true,
32    AutomatedTellerMachine.MessageType.valueOf("validated_M").equals(Autom
33    atedTellerMachine.MessageType.validated_M));
34     Assert.assertEquals(true,
35    AutomatedTellerMachine.MessageType.valueOf("select_M").equals(Automate
36    dTellerMachine.MessageType.select_M));
37     Assert.assertEquals(true,
38    AutomatedTellerMachine.MessageType.valueOf("selectAnotherTransiction_M
39    ").equals(AutomatedTellerMachine.MessageType.selectAnotherTransiction_
40    M));
41     Assert.assertEquals(true,
42    AutomatedTellerMachine.MessageType.valueOf("finish_M").equals(Automate
43    dTellerMachine.MessageType.finish_M));
44     Assert.assertEquals(true,
45    AutomatedTellerMachine.MessageType.valueOf("receiptPrinted_M").equals(
46    AutomatedTellerMachine.MessageType.receiptPrinted_M));
47   }
48   @Test
49   public void processEvents() throws InterruptedException
50   {
51     AutomatedTellerMachine qsm = new AutomatedTellerMachine();
52     int numChecks;
53     //initial state is idle
54     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
55     //cardInserted is triggered: cardInserted is queued
56     qsm.cardInserted();
57     //cardInserted is dequeued and processed: transition to active
58     numChecks=200; // we will check for a second
59     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.active) &&
60     numChecks>0) {
```

```
61        Thread.sleep(5);
62        numChecks--;
63      }
64
65      assertThat(numChecks, not(equalTo(0)));
66      Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
67      Assert.assertEquals("Card is read", qsm.getLog(0));
68      // check if there is a message saved in the queue
69      Assert.assertEquals(0, qsm.queue.messages.size());
70      //validated is triggered: validated is queued
71      qsm.validated();
72      //validated is dequeued and processed: transition to selecting
73      numChecks=200;
74      while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.select
75      ing) && numChecks>0) {
76        Thread.sleep(5);
77        numChecks--;
78      }
79      assertThat(numChecks, not(equalTo(0)));
79      Assert.assertEquals(AutomatedTellerMachine.SmActive.selecting,
80      qsm.getSmActive());
81      // check if there is a message saved in the queue
82      Assert.assertEquals(0, qsm.queue.messages.size())
83      //select is triggered: select is queued
84      qsm.select();
85      //select is dequeued and processed: transition to processing
86      numChecks=200; // we will check for a second
87      while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.proces
88      sing) && numChecks>0) {
89        Thread.sleep(5);
90        numChecks--;
91      }
92      assertThat(numChecks, not(equalTo(0)));
93      Assert.assertEquals(AutomatedTellerMachine.SmActive.processing,
94      qsm.getSmActive());
95      // check if there is a message saved in the queue
96      Assert.assertEquals(0, qsm.queue.messages.size());
97      //finish is triggered: finish is queued
98      qsm.finish();
99      //finish is dequeued and processed: transition to printing
100     numChecks=200; // we will check for a second
101     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.printi
102     ng) && numChecks>0) {
103       Thread.sleep(5);
104       numChecks--;
105     }
106     assertThat(numChecks, not(equalTo(0)));
107     Assert.assertEquals(AutomatedTellerMachine.SmActive.printing,
108     qsm.getSmActive());
109     // check if there is a message saved in the queue
110     Assert.assertEquals(0, qsm.queue.messages.size());
111     //receiptPrinted is triggered: receiptPrinted is queued
112     qsm.receiptPrinted();
113     //receiptPrinted is dequeued and processed: transition to idle
114     numChecks=200; // we will check for a second
115     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.idle) &&
116     numChecks>0) {
117       Thread.sleep(5);
118       numChecks--;}
119     assertThat(numChecks, not(equalTo(0)));
120     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
121     //check if there is a message saved in the queue
122     Assert.assertEquals(0, qsm.queue.messages.size());
```

```
123     //selectAnotherTransiction is triggered: selectAnotherTransiction is
124     //queued
125     qsm.selectAnotherTransiction();
126     //selectAnotherTransiction is dequeued: it is unspecified,
127     //unspecified method is called to handle this error
128     //transition to error1
129     //auto-transition to idle
130     numChecks=200; // we will check for a second
131     while(numChecks>0 &&
132     qsm.getSm().equals(AutomatedTellerMachine.Sm.idle)) {
133        if(!qsm.queue.messages.isEmpty()){
134          Thread.sleep(5);
135          numChecks--;
136        }else{
137           Assert.assertEquals(AutomatedTellerMachine.Sm.idle,
138           qsm.getSm());
139           Assert.assertEquals(true, qsm.queue.messages.isEmpty());
140            break;}
141         }
142     assertThat(numChecks, not(equalTo(0)));
143     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
144     // check if there is a message saved in the queue
145     Assert.assertEquals(0, qsm.queue.messages.size());
146     //maintain is triggered: maintain is queued
147     qsm.maintain();
148     //maintain is dequeued and processed: transition to maintenance
149     numChecks=200; // we will check for a second
150     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.maintenance) &&
151     numChecks>0) {
152        Thread.sleep(5);
153        numChecks--;
154     }
155     assertThat(numChecks, not(equalTo(0)));
156     Assert.assertEquals(AutomatedTellerMachine.Sm.maintenance,
157     qsm.getSm());
158     // check if there is a message saved in the queue
159     Assert.assertEquals(0, qsm.queue.messages.size());
160     //isMaintained is triggered: isMaintained is queued
161     qsm.isMaintained();
162     //isMaintained is dequeued and processed: transition to idle
163     numChecks=200; // we will check for a second
164     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.idle) &&
165     numChecks>0) {
166        Thread.sleep(5);
167        numChecks--;
168     }
169     assertThat(numChecks, not(equalTo(0)));
170     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
171     // check if there is a message saved in the queue
172     Assert.assertEquals(0, qsm.queue.messages.size());
173     //cancel is triggered: cancel is queued
174     qsm.cancel();
175     //cancel is dequeued: it is unspecified, unspecified method is called
176     //to handle this error
177     //transition to error1
178     //auto-transition to idle
179     numChecks=200; // we will check for a second
180     while(numChecks>0 &&
181     qsm.getSm().equals(AutomatedTellerMachine.Sm.idle)) {
182        if(!qsm.queue.messages.isEmpty()){
183          Thread.sleep(5);
184          numChecks--;
185        }else{
```

```
186          Assert.assertEquals(AutomatedTellerMachine.Sm.idle,
187          qsm.getSm());
188           Assert.assertEquals(true, qsm.queue.messages.isEmpty());
189         break;}
190      }
191      assertThat(numChecks, not(equalTo(0)));
192      Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
193      // check if there is a message saved in the queue
194      Assert.assertEquals(0, qsm.queue.messages.size());
195      //cardInserted is triggered: cardInserted is queued
196      qsm.cardInserted();
197      //cardInserted is dequeued and processed: transition to active
198     numChecks=200; // we will check for a second
199     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.active) &&
200     numChecks>0) {
201       Thread.sleep(5);
202       numChecks--;
203      }
204      assertThat(numChecks, not(equalTo(0)));
205      Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
206      // check if there is a message saved in the queue
207      Assert.assertEquals(0, qsm.queue.messages.size());
208      //select is triggered: select is queued
209      qsm.select();
210      Thread.sleep(10);
211      //select is dequeued: it is unspecified, unspecified method is called
212      //to handle this error
213      //transition to error2
214      //auto-transition to validating
215      numChecks=200; // we will check for a second
216     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.valida
217     ting) && numChecks>0) {
218        Thread.sleep(5);
219        numChecks--;}
220      assertThat(numChecks, not(equalTo(0)));
221      Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
222     Assert.assertEquals(AutomatedTellerMachine.SmActive.validating,
223     qsm.getSmActive());
224      // check if there is a message saved in the queue
225      Assert.assertEquals(0, qsm.queue.messages.size());
226      //validated is triggered: validated is queued
227      qsm.validated();
228      //validated is dequeued and processed: transition to selecting
229      numChecks=200; // we will check for a second
230     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.select
231     ing) && numChecks>0)
232     { Thread.sleep(5);
233       numChecks--;}
234      assertThat(numChecks, not(equalTo(0)));
235      Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
236      Assert.assertEquals(AutomatedTellerMachine.SmActive.selecting,
237      qsm.getSmActive());
238      // check if there is a message saved in the queue
239      Assert.assertEquals(0, qsm.queue.messages.size());
240      //select is triggered: select is queued
241      qsm.select();//select is dequeued and processed: transition to
242                   //processing
243      numChecks=200; // we will check for a second
244     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.proces
245     sing) &&numChecks>0)
246       {Thread.sleep(5);
247        numChecks--;
248       }
```

```
249     assertThat(numChecks, not(equalTo(0)));
250     Assert.assertEquals(AutomatedTellerMachine.SmActive.processing,
251     qsm.getSmActive());
252     Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
253     // check if there is a message saved in the queue
254     Assert.assertEquals(0, qsm.queue.messages.size());
255     //selectAnotherTransiction is triggered: selectAnotherTransiction is
256     //queued
257     qsm.selectAnotherTransiction();
258     //selectAnotherTransiction is dequeued and processed: transition to
259     //processing
260     numChecks=200; // we will check for a second
261    while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.select
262    ing) && numChecks>0)
263      {Thread.sleep(5);
264       numChecks--;}
265     assertThat(numChecks, not(equalTo(0)));
266     Assert.assertEquals(AutomatedTellerMachine.SmActive.selecting,
267     qsm.getSmActive());
268     Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
269     // check if there is a message saved in the queue
270     Assert.assertEquals(0, qsm.queue.messages.size());
271     //select is triggered: select is queued
272     qsm.select();
273     //select is dequeued and processed: transition to processing
274     numChecks=200; // we will check for a second
275     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.proces
276     sing) && numChecks>0) {
277       Thread.sleep(5);
278       numChecks--;}
279     assertThat(numChecks, not(equalTo(0)));
280     Assert.assertEquals(AutomatedTellerMachine.SmActive.processing,
281     qsm.getSmActive());
282     Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
283     // check if there is a message saved in the queue
284     Assert.assertEquals(0, qsm.queue.messages.size());
285     //finish is triggered: finish is queued
286     qsm.finish();
287    //finish is dequeued and processed: transition to printing
288    numChecks=200; // we will check for a second
289     while(!qsm.getSmActive().equals(AutomatedTellerMachine.SmActive.printi
290     ng) && numChecks>0) {
291        Thread.sleep(5);
292        numChecks--;}
293     assertThat(numChecks, not(equalTo(0)));
294     Assert.assertEquals(AutomatedTellerMachine.SmActive.printing,
295     qsm.getSmActive());
296     Assert.assertEquals(AutomatedTellerMachine.Sm.active, qsm.getSm());
297     // check if there is a message saved in the queue
298     Assert.assertEquals(0, qsm.queue.messages.size());
299     //receiptPrinted is triggered: receiptPrinted is queued
300     qsm.receiptPrinted();
301     //receiptPrinted is dequeued and processed: transition to idle
302     numChecks=200; // we will check for a second
303     while(!qsm.getSm().equals(AutomatedTellerMachine.Sm.idle) &&
304     numChecks>0) {
305       Thread.sleep(5);
306       numChecks--;
307     }
308     assertThat(numChecks, not(equalTo(0)));
309     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
310     // check if there is a message saved in the queue
311     Assert.assertEquals(0, qsm.queue.messages.size());
```

```
312     //finish is triggered: finish is queued
313     qsm.finish();
314     //finish is dequeued: it is unspecified, unspecified method is called
315     //to handle this error
316     //transition to error1
317     //auto-transition to idle
318     numChecks=200; // we will check for a second
319     while(numChecks>0 &&
320     qsm.getSm().equals(AutomatedTellerMachine.Sm.idle)) {
321       if(!qsm.queue.messages.isEmpty()){
322         Thread.sleep(5);
323          numChecks--;
324       } else
325       {
326         Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
327        Assert.assertEquals(true, qsm.queue.messages.isEmpty());
328        break;
329       }
330     }
331     assertThat(numChecks, not(equalTo(0)));
332     Assert.assertEquals(AutomatedTellerMachine.Sm.idle, qsm.getSm());
333     // check if there is a message saved in the queue
334     Assert.assertEquals(0, qsm.queue.messages.size());
335     //check that there is no events left in the queue
336     Assert.assertEquals(0, qsm.queue.messages.size());
337   }
338 }
```

Appendix E

E.1 Pseudocode of the Generated Java Code For Unspecified Reception Mechanism For QSM and Basic State Machine

```
1  public class X {
2    // Constructor and member variables are generated regularly
3    // Event method handlings
4    // Event a: in switch state we check the current state, if it is s1,
5    // then the transition is fired and the state machine transitions to
6    // s12. Otherwise, it is ignored
7    c(){
8      switch (current state) {
9        case current state (s1):
10         move to the next state (s2)
11         break;
12       default:
13         // Other states do respond to this event
14         // We don't call 'unspecified' method because it is not specified
15         // in the state that has a event }
16   }
17   // Event b: in switch state we check the current state, if it is s2,
18   // then the transition is fired and the state machine transitions to
19   // s3. Otherwise, the pseudo-event 'unspecified' method is called
20   b(){
21     switch (current state) {
22       case current state (s2):
23         move to the next state (s3)
24         break;
25       default:
26         // Other states do respond to this event
27         Call unspecified(current state name, event method name);
28     }
29   }
30   // Event c: in switch state we check the current state, if it is s3,
31   // then the transition is fired and the state machine transitions to
32   // s1. Otherwise, it is ignored
33   c(){
34     switch (current state) {
35       case current state (s3):
36         move to the next state (s1)
37         break;
38       default:
39         // Other states do respond to this event
40         // We don't call 'unspecified' method because it is not specified
41         // in the state that has c event
42     }
43   }
44   // Pseudo-Event specification 'unspecified' method
45   unspecified(String state, String event){
46     switch (current state){
47       case (current state where 'unspecified' is defined):
48         // execute transition action if there is one
49         // move to the next state
50         break;
51       default:
52         // Other states do respond to this event
53     }
54   }
55 }
```

Appendix F

F.1 Umple Code For Elevator Controller System

```
1  /**
2   * This elevator system example contains a queued state machine
3   * that has[[[[ number of state machines in different classes]]]]]
4   * The queued state machine in44 ElevatorController class communicate
5   * with other queued state machines in other classes(HallButton,
6   * FloorRequestButton, FloorIndicator and DirectionIndicator) and with a
7   * simple state machine of ElevatorDoor class.
8   * It is complete example consisting of:
9   * (a) receiving a request from a passenger by pressing a UP/DOWN hall
10  * button
11  * (b) Controller then check which elevator is near form passenger to
12  * serve them among multiple elevators
13  * (c) After choosing the closest elevator, the elevator then moves to
14  * the floor where the request comes from
15  * (d) then once the passenger enters the elevator, he/she press the
16  * floor button where he/she wants to go to
17  * (e) elevator then moves to the target floor requested by passenger
18  **/
19 namespace elevatorSystem2;
20 /**
21  * Elevator controller that coordinates and controls the operation of the
22  * components in the elevator system
23  */
24 class ElevatorController{
25   singleton;
26   1 -- * Elevator;
27   1 controllerUprequest -- * Floor requestedUP;
28   1 controllerDownrequest -- * Floor requestedDOWN;
29   Elevator closetElevator = null;
30   //Choosing the nearest elevator for the passenger
31   void closetElevator(Passenger psg, Direction.DirectionSM
32   passengerDirection){
33     List<Integer> checkdistanceBetweenFloors = new ArrayList<Integer>();
34     int diff=0;
35     System.out.println("");
36     for(int i=0; i < this.getElevators().size(); i++) {
37 diff=Math.abs((this.getElevator(i).getElevatorCurrentFloor().getFloorNumb
38 er()) - (psg.getCurrentFloor().getFloorNumber()));
39       checkdistanceBetweenFloors.add(diff);
40     }
41     int small=checkdistanceBetweenFloors.get(0);
42     for(int i=1; i < checkdistanceBetweenFloors.size(); i++) {
43       if(checkdistanceBetweenFloors.get(i) < small) {
44         small=checkdistanceBetweenFloors.get(i);
45       }
46     }
47     Elevator closetElevator = this.getElevator(0);
48     int distance=0;
49     for(int x=0; x < this.getElevators().size(); x++) {
50       distance=Math.abs((this.getElevator(x).getElevatorCurrentFloor().
51       getFloorNumber()) - (psg.getCurrentFloor().getFloorNumber()));
52       //Check if requestedUP is not empty, then check which elevator is
53       //near you
54       if(this.hasRequestedUP()) {
55          if(distance == small) {
56            //Choose closet Elevator to serve a passenger UP
57            closetElevator = this.getElevator(x);
```

```
58            if(closetElevator.getElevatorCurrentFloor().getFloorNumber() <
59            psg.getCurrentFloor().getFloorNumber()){
60              closetElevator.upRequest();
61            }
62            if(closetElevator.getElevatorCurrentFloor().getFloorNumber() >
63            psg.getCurrentFloor().getFloorNumber()){
64              closetElevator.downRequest();
65            }
66          }
67        }
68        //Check if requestedDOWN is not empty, then check which elevator is
69        //near you
70        if(this.hasRequestedDOWN()){
71           if(distance == small) {
72             //Choose closet Elevator to serve a passenger DOWN
73             closetElevator = this.getElevator(x);
74             if(closetElevator.getElevatorCurrentFloor().getFloorNumber() <
75             psg.getCurrentFloor().getFloorNumber()){
76               closetElevator.upRequest();
77             }
78             if(closetElevator.getElevatorCurrentFloor().getFloorNumber() >
79             psg.getCurrentFloor().getFloorNumber()){
80               closetElevator.downRequest();
81             }
82           }
83        }
84        if(!this.hasRequestedUP() && !this.hasRequestedDOWN()) {
85           //No Requests
86           this.getElevator(x).noRequest();
87        }
88      }
89      System.out.println("CONTROLLER selects Elevator
90      ["+closetElevator.getElevatorID()+"] ON FLOOR ]
91      ["+closetElevator.getElevatorCurrentFloor().getFloorNumber()+"] to
92      serve Passenger ["+psg.getPassengerID()+"]");
93      closetElevator.doorClosed();
94      closetElevator.approachedFloor(psg);
95      closetElevator.stopped(psg);
96      closetElevator.doorOpened(psg);
97      closetElevator.doorCLosed(psg);
98    }
99  }
100 namespace elevatorSystem2;
101 /**
102  * Class represents an Elevator.
103  * Elevator has passengers, and a current floor number.
104  * elevator opens and closes doors to allow passengers getting in.
105  */
106 class Elevator{
107   1 -- 1 Motor;
108   1 -- 1 ElevatorDoor;
109   1 -- 1 FloorIndicator;
110   1 -- 1 DirectionIndicator;
111   1 -- * FloorRequestButton;
112   1 -- * Passenger passengerEnteringElevator;
113   1 receives -- * Floor plannedCalls;
114   //Elevator id
115   Integer elevatorID;
116
117   //Elevator's current floor.
118   Floor elevatorCurrentFloor;
119   Integer timer = 3;
120   Boolean obstruction = false;
```

```
121   //Elevator state machine
122   pooled elevatorSm {
123     Idle {
124       upRequest -> PrepareUp;
125       downRequest -> PrepareDown;
126     }
127     PrepareUp {
128       DoorClosingMovingUp {
129         doorClosed -> / {this.getDirectionIndicator().onUP();
130         this.getMotor().goUp();} InMotion;
131       }
132       started -> Moving;
133     }
134     PrepareDown {
135       DoorClosingMovingDown {
136         doorClosed  -> / {this.getDirectionIndicator().onDOWN();
137         this.getMotor().goDown();} InMotion;
138       }
139       started -> Moving;
140     }
141     InMotion {
142        entry/{this.getFloorIndicator().lightON(this.getElevatorCurrentFloo
143        r());}
144       Moving {
145         approachingFloor(Passenger psg) [!floorRequested(psg)] -> Moving;
146         approachedFloor(Passenger psg)  [floorRequested(psg)] -> /
147         {this.getFloorIndicator().lightOFF(this.getElevatorCurrentFloor()
148         ); this.getMotor().stop();} Stopping;
149       }
150       Stopping {
151         stopped(Passenger psg) / {
152         psg.getCurrentFloor().getHallButton().lightOFF(psg);
153         this.getElevatorDoor().openDoor();
154         this.getDirectionIndicator().off();} -> DoorOpening;
155       }
156     }
157     OnFloor {
158       DoorOpening {
159          doorOpened(Passenger psg)
160          /{psg.getCurrentFloor().removePassengerWaitingOnFloor(psg);}->
161          AtFloor;
162       }
163       AtFloor {
164         //after(this.getElevatorDoor().getDoorTimer().timeIsUp())
165         //[!obstruction] -> DoorClosing;
166         after(timer) [!obstruction] -> DoorClosing;
167         doorClosingRequest -> DoorClosing;
168       }
169       DoorClosing {
170         doorCLosed(Passenger psg) /{
171           this.getElevatorDoor().closeDoor();
172           if(ElevatorController.getInstance().hasRequestedUP()){
173              ElevatorController.getInstance().removeRequestedUP(psg.getCu
174              rrentFloor());
175           }
176           if(ElevatorController.getInstance().hasRequestedDOWN(){
177              ElevatorController.getInstance().removeRequestedDOWN(psg.get
178              CurrentFloor());}
179           psg.stopClock();
180         } -> CheckingNextDestination;
181         obstruction -> DoorOpening;
182         doorOpeningRequest -> DoorOpening;
183       }
```

```
184       CheckingNextDestination {
185         upRequest -> PrepareUp;
186         downRequest -> PrepareDown;
187         noRequest -> Idle;
188       }
189     }
190   }
191   public Elevator(int aElevatorID, Floor aElevatorCurrentFloor,
192   ElevatorController aElevatorController)
193   {
194     elevatorID = aElevatorID;
195     elevatorCurrentFloor = aElevatorCurrentFloor;
196     timer = 3;
197     obstruction = false;
198     motor = new Motor(this);
199     elevatorDoor = new ElevatorDoor(timer, this);
200     floorIndicator = new FloorIndicator(this);
201     directionIndicator = new DirectionIndicator(this);
202     floorRequestButtons = new ArrayList<FloorRequestButton>();
203     passengerEnteringElevator = new ArrayList<Passenger>();
204     plannedCalls = new ArrayList<Floor>();
205     boolean didAddElevatorController =
206     setElevatorController(aElevatorController);
207     if (!didAddElevatorController){
208       throw new RuntimeException("Unable to create elevator due to
209       elevatorController");
210     }
211     setElevatorSmPrepareUp(ElevatorSmPrepareUp.Null);
212     setElevatorSmPrepareDown(ElevatorSmPrepareDown.Null);
213     setElevatorSmInMotion(ElevatorSmInMotion.Null);
214     setElevatorSmOnFloor(ElevatorSmOnFloor.Null);
215     setElevatorSm(ElevatorSm.Idle);
216     pool = new MessagePool();
217     removal=new Thread(this);
218     //start the thread of Elevator
219     removal.start();
220   }
221   //floorRequested
222   Boolean floorRequested(Passenger psg) {
223     this.getFloorIndicator().lightOFF(this.getElevatorCurrentFloor());
224     Boolean floorRequested = false;
225     if(psg.getCurrentFloor().getFloorNumber() ==
226     this.getElevatorCurrentFloor().getFloorNumber()){
227       floorRequested = true;
228     }
229     if(psg.getCurrentFloor().getFloorNumber() !=
230     this.getElevatorCurrentFloor().getFloorNumber()){
231       if(this.getElevatorCurrentFloor().getFloorNumber() <
232       psg.getCurrentFloor().getFloorNumber()){
233         int x = psg.getCurrentFloor().getFloorNumber();
234         for(int i = this.getElevatorCurrentFloor().getFloorNumber(); i <
235         x; i++){
236           if(this.getElevatorCurrentFloor().getFloorNumber() != x) {
237             setElevatorCurrentFloor(new
238             Floor(this.getElevatorCurrentFloor().getFloorNumber() + 1));
239             System.out.println("[Elevator "+this.getElevatorID()+"] [Fl
240             "+this.getElevatorCurrentFloor().getFloorNumber()+"]
241             destinations [Fl "+x+"] moving");}
242           if(this.getElevatorCurrentFloor().getFloorNumber() == x){
243       this.getFloorIndicator().lightON(this.getElevatorCurrentFloor());
244             floorRequested = true;
245           }
246         }
```

```
      if(this.getElevatorCurrentFloor().getFloorNumber() >
      psg.getCurrentFloor().getFloorNumber()) {
        int x = psg.getCurrentFloor().getFloorNumber();
        for(int i = this.getElevatorCurrentFloor().getFloorNumber(); i >
        x; i--){
          if(this.getElevatorCurrentFloor().getFloorNumber() != x) {
            setElevatorCurrentFloor(new
            Floor(this.getElevatorCurrentFloor().getFloorNumber() - 1));
            System.out.println("[Elevator "+this.getElevatorID()+"] [Fl
            "+this.getElevatorCurrentFloor().getFloorNumber()+"]
            destinations [Fl "+x+"] moving");
          }
          if(this.getElevatorCurrentFloor().getFloorNumber() == x){
            this.getFloorIndicator().lightON(this.getElevatorCurrentFloor
            ());
            floorRequested = true;
          }
        }
      }
    }
    return floorRequested;
  }
}
namespace elevatorSystem2;
/**
 * Direction enumerations
 **/
class Direction {
  //Direction Enumeration
  DirectionSM {
    UP {}
    DOWN {}
    NONE {}
  }
}
namespace elevatorSystem2;
/**
 * A building floor
 */
class Floor{
  1 -- * Passenger passengerWaitingOnFloor;
  1 -- 1 HallButton;
  Integer floorNumber;
  public Floor(int aFloorNumber)
  {
    floorNumber = aFloorNumber;
    passengerWaitingOnFloor = new ArrayList<Passenger>();
    hallButton = new HallButton(this);
  }
}
namespace elevatorSystem2;
/**
 * A passenger requesting and travelling on an elevator
 */
class Passenger{
  Floor currentFloor;      //A passenger's current floor
  Floor destinationFloor;  //A passenger's destination floor
  String passengerID;      //Passenger id which could be a name
  long startingTime;       //Start time of trip for each passenger
  long endingTime;         //End time of trip for each passenger
  active {
    System.out.println("[Passenger "+getPassengerID()+"] arrives on [Fl
    "+this.getCurrentFloor().getFloorNumber()+"] wants to go to [Fl
```

```
310     "+this.getDestinationFloor().getFloorNumber()+"]");
311     //Passenger is waiting On Floor
312     getCurrentFloor().addPassengerWaitingOnFloor(this);
313     //Passenger is pressing Hall button
314     if (this.getCurrentFloor().getFloorNumber() <
315     this.getDestinationFloor().getFloorNumber()) {
316       System.out.println("[Passenger "+getPassengerID()+"] presses Fl
317       ["+this.getDestinationFloor().getFloorNumber()+"
318       "+this.passengerDirectionUP()+" request]");
319       this.getCurrentFloor().getHallButton().pressHallButton(this,
320       this.passengerDirectionUP());
321     }
322     else if (this.getCurrentFloor().getFloorNumber() >
323     this.getDestinationFloor().getFloorNumber()) {
324       System.out.println("[Passenger "+getPassengerID()+"] presses Fl
325       ["+this.getDestinationFloor().getFloorNumber()+"
326       "+this.passengerDirectionDOWN()+" request]");
327       this.getCurrentFloor().getHallButton().pressHallButton(this,
328       this.passengerDirectionDOWN());
329     }
330     startClock();
331   }
332   public Passenger(Floor aCurrentFloor, Floor aDestinationFloor, String
333   aPassengerID) {
334     currentFloor = aCurrentFloor;
335     destinationFloor = aDestinationFloor;
336     passengerID = aPassengerID;
337     setStateMachine1TopLevel(StateMachine1TopLevel.Null);
338     setStateMachine1(StateMachine1.topLevel);
339   }
340   // A direction UP when a passenger presses UP hall button
341   Direction.DirectionSM passengerDirectionUP() {
342     return Direction.DirectionSM.UP;
343   }
344   // A direction DOWN when a passenger presses DOWN hall button
345   Direction.DirectionSM passengerDirectionDOWN() {
346     return Direction.DirectionSM.DOWN;
347   }
348   //Start a timer once a passenger requests an elevator
349   void startClock() {
350     startingTime = System.currentTimeMillis();
351   }
352   //Stop a timer once a passenger arrives to a destination floor and then
353   //get the total time he/she spent
354   void stopClock(){
355     endingTime = System.currentTimeMillis();
356     System.out.print("[Passenger "+this.getPassengerID()+"] Trip took :
357    [" + this.getTripTimeInMilliseconds() + "] \n");
358   }
359   //Get time in Seconds a passenger spends from requesting an elevator
360   //till arriving to a destination floor
361   long getTripTimeInMilliseconds() {
362     return ((endingTime- startingTime));
363   }
364 }
365 namespace elevatorSystem2;
366 /**
367  * An electric motor for each elevator.
368  * When directed by an Elevator Controller, the motor moves, or stops the
369  * elevator.
370  */
371 class Motor{
372   depend java.util.*;
```

```
273   //Motor is moving an elevator UP
274   void goUp(){
275     System.out.println("[Elevator "+getElevator().getElevatorID()+"]
276     starts moving UP");
277   }
278   //Motor is moving an elevator DOWN
279   void goDown(){
280     System.out.println("[Elevator "+getElevator().getElevatorID()+"]
281     starts moving DOWN");
282   }
283   //Motor stops an elevator at a destination floor of a waiting passenger
284   void stop(){
285     System.out.println("Elevator  ["+getElevator().getElevatorID()+"]
286     arrives");
287   }
288 }
289 namespace elevatorSystem2;
290 /**
291  * Elevator's Door which is opened or closed
292  */
293 class ElevatorDoor{
294   1 -- 1 DoorTimer;
295   boolean isOpened=false;
296   queued doorSM {
297     doorClose {
298       openDoor /{this.openElevatorDoor();} -> doorOpen;
299     }
300     doorOpen {
301       closeDoor /{this.closeElevatorDoor();} -> doorClose;
302     }
303   }
304   public ElevatorDoor(int aTimeInMilliSecondsForDoorTimer, Elevator
305   aElevator) {
306     isOpened = false;
307     doorTimer = new DoorTimer(aTimeInMilliSecondsForDoorTimer, this);
308     if (aElevator == null || aElevator.getElevatorDoor() != null) {
309       throw new RuntimeException("Unable to create ElevatorDoor due to
310       aElevator");
311     }
312     elevator = aElevator;
313     setDoorSM(DoorSM.doorClose);
314     queue = new MessageQueue();
315     removal=new Thread(this);
316     //start the thread of ElevatorDoor
317     removal.start();
318   }
319   //An elevator door is opened and a timer is started
320   void openElevatorDoor() {
321     isOpened=true;
322     System.out.println("[Elevator "+getElevator().getElevatorID()+"]
323     elevator door is opened");
324   }
325   //An elevator door is closed and a timer is up
326   void closeElevatorDoor(){
327     isOpened=false;
328     System.out.println("[Elevator "+getElevator().getElevatorID()+"]
329 elevator door is closed");
330   }
331 }
332 namespace elevatorSystem2;
333 /**
334  * Determine a timer for an Elevator Door
335  */
```

```
436 class DoorTimer{
437   Integer timeInMilliSeconds;
438   //Set Timer for an Elevator Door once it is opened
439   public int timeIsUp(){
440     timeInMilliSeconds=5;
441     return timeInMilliSeconds;
442   }
443 }
444 namespace elevatorSystem2;
445 /**
446  * UP and DOWN push Buttons (Hall Buttons)
447  */
448 class HallButton{
449   queued hallButtonSM{
450     Released {
451       lightON(Passenger psg) /{this.lightingOn(psg);} -> Pressed;
452     }
453 
454     Pressed {
455       lightOFF(Passenger psg) /{this.lightingOff(psg);} -> Released;
456     }
457   }
458   //Light a hall button on
459   void lightingOn(Passenger psg){
460     if(psg.passengerDirectionUP().equals(Direction.DirectionSM.UP)){
461       System.out.println("[Fl "+psg.getCurrentFloor().getFloorNumber()+"]
462       "+psg.passengerDirectionUP()+" request light ON");
463     }
464     if(psg.passengerDirectionUP().equals(Direction.DirectionSM.DOWN)) {
465       System.out.println("[Fl "+psg.getCurrentFloor().getFloorNumber()+"]
466       "+psg.passengerDirectionUP()+" request light ON");
467     }
468   }
469   //Light a hall button off
470   void lightingOff(Passenger psg){
471     if(psg.passengerDirectionUP().equals(Direction.DirectionSM.UP)) {
472       System.out.println("[Fl "+psg.getCurrentFloor().getFloorNumber()+"]
473       "+psg.passengerDirectionUP()+" request light OFF");
474     }
475     if(psg.passengerDirectionUP().equals(Direction.DirectionSM.DOWN)) {
476       System.out.println("[Fl "+psg.getCurrentFloor().getFloorNumber()+"]
477       "+psg.passengerDirectionDOWN()+" request light OFF");
478     }
479   }
480   //Passenger presses a hall button to request an elevator
481   void pressHallButton(Passenger psg, Direction.DirectionSM
482   passengerDirection){
483     this.lightON(psg);
484     //Add requestedUp Floor To a list AND requestedDown Floor to a list
485     if(passengerDirection.equals(Direction.DirectionSM.UP)){
486       if (psg.getCurrentFloor().getFloorNumber() <
487       psg.getDestinationFloor().getFloorNumber()){
488       ElevatorController.getInstance().addRequestedUP(psg.getCurrentFloor(
489       ));
490       }
491     }
492     if (passengerDirection.equals(Direction.DirectionSM.DOWN)) {
493       if (psg.getCurrentFloor().getFloorNumber() >
494       psg.getDestinationFloor().getFloorNumber()) {
495         ElevatorController.getInstance().addRequestedDOWN(psg.getCurrentFl
496         oor());
497       }
498     }
```

```
499     ElevatorController.getInstance().closetElevator(psg,
500     passengerDirection);
501   }
502 }
503 namespace elevatorSystem2;
504 /**
505  * Floor buttons allow a passenger to select a floor wants to go to
506  */
507 class FloorRequestButton{
508   boolean lightOn=false;
509   queued floorButtonSM{
510     OFF {
511       lightON(Passenger psg) /{this.lightingOn(psg);}-> ON;
512     }
513 
514     ON {
515       lightOFF(Passenger psg) /{this.lightingOff(psg);}-> OFF;
516     }
517   }
518   //Light a selected destination floor button on
519   void lightingOn(Passenger psg){
520     System.out.println("Elevator  ["+getElevator().getElevatorID()+"] :
521     Passenger ["+psg.getPassengerID()+"] : FLOOR REQUEST BUTTON LIGHT
522     ON");
523     this.lightOn=true;
524   }
525   //Light a selected destination floor button off
526   void lightingOff(Passenger psg){
527     System.out.println("Elevator  ["+getElevator().getElevatorID()+"] :
528     Passenger ["+psg.getPassengerID()+"] : FLOOR REQUEST BUTTON LIGHT
529     OFF");
530     this.lightOn=false;
531   }
532 }
533 namespace elevatorSystem2;
534 /**
535  * Floor indicators inside an elevator on top of the elevator door.
536  * It indicates to a passenger inside of the elevator the current
537  * location (floor number) of the elevator
538  */
539 class FloorIndicator{
540   boolean lightOn=false;
541   queued floorIndicatorSM{
542     OFF {
543       lightON(Floor floor) /{this.lightingOn(floor);}-> ON;
544     }
545     ON {
546       lightOFF(Floor floor) /{this.lightingOff(floor);}-> OFF;
547     }
548   }
549   //Light a floor number button on
550   void lightingOn(Floor floor){
551     this.lightOn=true;
552     System.out.println("[Fl "+ floor.getFloorNumber()+"] signal light ON
553     for [Elevator "+getElevator().getElevatorID()+"] ");
554   }
555   //Light a floor number button off
556   void lightingOff(Floor floor){
557     this.lightOn=false;
558     System.out.println("[Fl "+ floor.getFloorNumber()+"] signal light OFF
559     for [Elevator "+getElevator().getElevatorID()+"] ");
560   }
562 }
```

```
563 namespace elevatorSystem2;
564 /**
565  * Light Indicators (Direction Indicators) inside an Elevator indicate
566  * the current direction the elevator
567  */
568 class DirectionIndicator {
569   boolean lightOn=false;
570   queued directionIndicatorSM {
571     NONE {
572       onUP /{this.lightingOnUP();}-> UP;
573       onDOWN /{this.lightingOnDOWN();}-> DOWN;
574     }
575     UP {
576       off /{this.lightingOff();}-> NONE;
577     }
578     DOWN {
579       off /{this.lightingOff();}-> NONE;
580     }
581   }
582   //Light a direction indicator UP – on
583   void lightingOnUP(){
584     this.lightOn=true;
585     System.out.println("Elevator  ["+getElevator().getElevatorID()+"] UP
586     Direction Indicator");
587   }
588   //Light a direction indicator DOWN – on
589   void lightingOnDOWN(){
590     this.lightOn=true;
591     System.out.println("Elevator  ["+getElevator().getElevatorID()+"]
592     DOWN Direction Indicator");
593   }
594   //Light a direction indicator off
595   void lightingOff(){
596     this.lightOn=false;
597     System.out.println("Elevator  ["+getElevator().getElevatorID()+"]
598     EMPTY Direction Indicator");
599   }
600 }
601 namespace elevatorSystem2;
602 class ElevatorMainTest{
603   depend java.util.*;
604   public static void main(String[] args) {
605     //Initiating Floors
606     Integer numberOfFloors = 5;
607     ArrayList<Floor> floors = new ArrayList<Floor>();
608     for (int i = 0; i < numberOfFloors; i++) {
609       Floor floor = new Floor(i+1);
610       floors.add(i, floor);
611     }
612     //Initiating Elevators
613     Elevator elevator1 = new Elevator(1, new Floor(1),
614     ElevatorController.getInstance());
615     Elevator elevator2 = new Elevator(2, new Floor(2),
616     ElevatorController.getInstance());
617     Elevator elevator3 = new Elevator(3, new Floor(4),
618     ElevatorController.getInstance());
619     //Passengers:
620     Passenger passenger1=new Passenger(floors.get(2), floors.get(4),
621     "STEPHEN");
622     Passenger passenger2=new Passenger(floors.get(4), floors.get(1),
623     "ALIAA");
624    }
625 }
```