A Manifestation of Model-Code Duality:

Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language

by

Omar Badreddin

PhD Thesis

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

Doctor of Philosophy (Computer Science[1])

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

[1] The Ph.D. program in Computer Science is a joint program with Carleton University, administered by the Ottawa Carleton Institute for Computer Science.

Acknowledgements

I wish to thank foremost my supervisor Dr. Timothy C. Lethbridge. Tim has been my supervisor throughout the PhD years and has provided guidance and deep insights that helped shape my understanding of the software engineering field.

A very special, and well-deserved, thank you to the following:

a) The Complexity Reduction in Software Engineering (CRUISE) research group. I have benefited from our weekly meetings and discussions. Particular thanks to Andrew Forward, Garzon Miguel, and Hamoud Ajman. My family and friends. Thank you to my mom, Sameha, for her unconditional support, my father, Bahy, for his reviews and input.

b) The Natural Sciences and Engineering Research Council of Canada (NSERC), IBM, and University of Ottawa for their collaboration and funding. These institutions have made available an environment through which I was able to conduct research while staying in touch with the industry.

c) Software professionals around the world. Sincere thanks to the individuals that participated in my research, published valuable references for my writings, as well as to those in the various news-groups about software engineering that I follow. Your knowledge and insight helped provide the necessary substance for my work.

Abstract

This thesis presents research to build and evaluate embedding of a textual form of state machines into high-level programming languages. The work entailed adding state machine syntax and code generation to the Umple model-oriented programming technology. The added concepts include states, transitions, actions, and composite states as found in the Unified Modeling Language (UML). This approach allows software developers to take advantage of the modeling abstractions in their textual environments, without sacrificing the value added of visual modeling.

Our efforts in developing state machines in Umple followed a test-driven approach to ensure high quality and usability of the technology. We have also developed a syntax-directed editor for Umple, similar to those available to other high-level programming languages. We conducted a grounded theory study of Umple users and used the findings iteratively to guide our experimental development. Finally, we conducted a controlled experiment to evaluate the effectiveness of our approach.

By enhancing the code to be almost as expressive as the model, we further support model-code duality; the notion that both model and code are two faces for the same coin. Systems can be and should be equally-well specified textually and diagrammatically. Such duality will benefit both modelers and coders alike. Our work suggests that code enhanced with state machine modeling abstractions is semantically equivalent to visual state machine models.

The flow of the thesis is as follows; the research hypothesis and questions are presented in "Chapter 1: Introduction". The background is explored in "Chapter 2: Background". "Chapter 3: Syntax and semantics of simple state machines" and "Chapter 4: Syntax and semantics of composite state machines" investigate simple and composite state machines in Umple, respectively. "Chapter 5: Implementation of composite state machines" presents the approach we adopt for the implementation of composite state machines that avoids explosion of the amount of generated code. From this point on, the thesis presents empirical work. A grounded theory study is presented in "Chapter 6: A Grounded theory study of Umple", followed by a controlled experiment in "Chapter 7: Experimentation". These two chapters constitute our validation and evaluation of Umple research. Related and future work is presented in "Chapter 8: Related work".

How to read this thesis

For readers who are not familiar with UML modeling, specifically, state machine models, we recommend a start-to-end reading of this document. However, readers familiar with UML modeling can choose to skip Chapter 1 and Chapter 2. Readers interested in empirical studies can focus on Chapter 6 and Chapter 7. Readers familiar with Umple and interested in the development and architecture work can read Chapter 3, Chapter 4, and Chapter 5.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The context for this thesis is a software development environment where code and model reside in the same artifact. It is an environment where the programming language is enhanced by modeling abstractions typically available to modelers in a visual environment. This approach effectively raises the abstraction level of today's modern high-level programming languages.

Our work is part of research efforts aiming at uniting code-centric and model-centric software engineering with the ultimate goal of enhancing modeling practices in the software engineering industry. We approach this goal by incorporating modeling abstractions in textual form that extends, or is similar to, a programming language. In particular, we investigate the incorporation of UML state machines to enhance the Umple language [3]. This approach is a manifestation of model-code duality. Model-code duality means that we consider both model and code to be a single entity with two representations. More specifically, we aim at demonstrating that graphical state machine modeling abstractions and their equivalent textual representations can be equally effective for designing and understanding systems.

Traditional development environments treat models and code as two separate entities. Such approaches induce software professionals to create, edit, and manage independently two separate artifacts; models and code. Forward and reverse engineering for code and model is therefore needed to keep the two artifacts in synch. On the other hand, if we treat models and code as a single entity, having two representations, we encourage the treatment of models and code as a single artifact (model-code duality). The need for model-to-code and code-to-model transformations are then eliminated or minimized.

A key research hypothesis we investigate is whether the core features of state machine diagrammatical modeling language can be effectively represented textually, in a high level programming-like syntax. Effective representation means that software professionals can comprehend, develop, and maintain software models textually in a manner suitable particularly to those who are accustomed to textual programming languages. In fact, developers and modelers will blend modeling and coding in the same development artifacts.

Our research approach is threefold; first, we investigate and evaluate how software engineers generate code from models, focusing on state machine models, and we use a grounded theory study to understand how Umple early adopters perceive textual modeling. Iteratively, we use the empirical research findings to drive the second part of this research, which is experimenting with state machine enhancements to the existing Umple research platform. And finally, we empirically evaluate our findings by means of conducting controlled experimentation.

The goals of our research activities are: 1) Understanding how the current tools handle code generation for state machines. 2) Empirical assessment of the use of textual modeling in software development. 3) Utilizing the findings of goals 1 and 2 to drive activities that aim at incorporating state machine modeling in a textual modeling environment to generate effective models and code. 4) Evaluating our approach.

1.1 Research Questions

Our research activities are guided by the following questions:

1.1.1 RQ1:

> *To what extent do software developers use state machines to model system behavior and specifications? What are the major factors behind that level of adoption?*

The origins of state machines can be traced back to the notion of "calculating machine", introduced by Charles Babbage in 1834 [4]. The mathematical model, since then, has been continuously improved and refined. We discuss the history and development of state machines in the section "History of State Machines" on page 26.

State machines are now a well-established modeling approach and are incorporated in the UML modeling specification. State machine models are supported in a significant number of software modeling tools and there is considerable support for automated code generation from state machines diagrams. However, our research findings indicate a low level of adoption of modeling notations in the software industry [5] and specifically for state machines [6]. Our personal observation of modelers, and our survey of capabilities of modeling tools, discussed later, indicate that adoption of state machines is particularly low. Reasons for the low adoption of state machines models may include:

1. State machine support by software modeling tools is poor.

   Other than in certain high-end real-time modeling tools, the available software modeling tools tend to have little support for state machine analysis and code generation; and some do not support basic modeling of state machines. In such a situation, lack of proper support in the available commercial and open source modeling tools will inevitably have a negative impact on the adoption of state machines.

2. Typical state machine diagrams are represented using a mixture of diagrammatical modeling elements and textual elements.

   Elements like states and transitions render themselves suitable for diagrammatical representations, while elements like actions and guard conditions are more suited for textual

representation. In most software modeling tools, developers have to switch from visual context to textual context to accomplish their modeling tasks.

3. There is little correspondence between state machine diagrams and the generated code.

   There are multiple design patterns for code generated from state machines. Our survey of generated code from a number of leading open source and commercial software modeling tools, discussed later, indicates the existence of several distinct design patterns with variations in the generated code that go beyond implementation specifics. This creates a wider gap between models and code that further induces developers to treat code and models as separate artifacts that need to be independently managed. This is discussed in "Chapter 2: Background".

4. Integration of state machine notation with other object oriented concepts tends to be poor.

   State machine notation is poorly integrated with other related UML modeling concepts. The overwhelming majority of tools support state machine notation in a standalone fashion; where the state machine diagrams do not integrate smoothly with other modeling notations such as class diagrams. For example, they do not generally support refinement of state machines over inheritance.

5. Awareness of state machines as a modeling notation is low among software developers.

   Software professionals may choose not to use state machines because they are not familiar with their concepts or applications. There is relatively little guidance on building applications that incorporates state machines.

1.1.2 RQ2:

Can the gap between state machine diagrams and code be minimized by incorporating core state machine abstractions in a high-level programming-like language?

Software modeling tools treat state models and code as two separate artifacts. Updates in the visual model have to be synchronized to the corresponding code, and vice versa. Software developers therefore need to make updates in both the visual model and the textual code, further complicating development tasks. A common scenario is for developers, at some point during the development process, to stop updating the visual model and rely only on editing the generated textual code, which renders models out of date and obsolete.

By incorporating state machine core concepts in a textual language that supports in-line native code, the model is maintained as long as the code is maintained.

1.2 Hypothesis and Approach

The following is the hypothesis we are investigating in this thesis:

H 1: *Software developers can comprehend software more effectively if state machine abstractions are embedded within the code.*

Existing software modeling and code generating tools imply that both the visual and textual contexts are in use, and are required, for system development using state machine diagrams. Our investigations focus on incorporating state machine concepts in a textual modeling language, and allow for in-line native code embedding. We anticipate significant reduction in the gap between state models and code, enabling developers to effectively treat both visual and textual code as a single entity. Our evaluation indicates that, for simple tasks, this approach improves comprehension when compared to a typical high level programming language. The evaluation is discussed in Chapter 7: Experimentation on page 151.

This hypothesis is investigated throughout our research activities.

1.3 Research Activities

We have conducted the following research activities and used the findings to address our research questions and verify our hypothesis.

I. Continuously explore how the Umple research platform is perceived by end users. Prior to our research, Umple already supported core class diagram features including associations and attributes. Understanding how users perceive textual modeling of class diagram elements helped guide our research activities and the implementation of state machine features in a way that is best suited towards developers' usability needs and cognitive patterns. We carry out this task by conducting a grounded theory study of Umple. Details of the study and findings are presented in "Chapter 6: A Grounded theory study of Umple".

II. Explore the design and implementation of state machine concepts in the Umple platform. Our understanding of the prevailing modeling practices and modeling tools has helped guide our research activities in adding state machine features to Umple. To accomplish this task, we explored existing related technologies and research.

III. Implement an interpretation of the latest UML state machine specifications and incorporate that implementation in the Umple platform. The implementation covers both simple and composite state machines.

IV. Evaluate our approach using a controlled experiment. Participants are presented with samples of models and code using a visual UML notation, a typical high level

programming language, and Umple. Participants are then asked a series of questions that aims at measuring their level of comprehension. The study suggests a positive added value of Umple technology.

V. To accomplish the empirical study, it is required to present participants with a compiler and environment that reflects Umple's vision, and that is of quality matching their expectations and helps participants focus on core research questions, rather than limitations in the platform. Towards that objective, we built a sophisticated textual editor.

1.4 Thesis contributions

The contributions of this research and the publications based on this thesis are presented in this section. The contributions are listed in order of importance. References to thesis sections discussing the contribution are given.

- Adding state machine abstractions in the Umple language

  The Umple technology now supports state machine abstractions. These abstractions are supported in the core Umple, and hence, are reflected in all Umple based tooling, such as the Umple online [3]. Implementing such abstractions required:

  - Defining new textual syntax to represent UML state machine modeling elements.
  - Integrating and extending the syntax into the Umple technology.

- Implementing semantics for state machine abstractions

  The semantics of the state machine abstractions are part of Umple core technology. The state machine abstractions are implemented by means of code generation of high level programming language. Umple distinguishes between two types of state machines; simple state machines (Chapter 3), and composite state machines (Chapter 4 & Chapter 5). This contribution entails the following sub-contributions:

  - Code generation for state machines that is similar to what software developers would write as implementation for a state machine model (Chapter 3 & Chapter 4).
  - A novel approach to implementing composite state machine semantics (Chapter 5).

- Investigation and analysis of the latest UML state machine specifications.

  We introduce a deep investigation of the UML state machine specifications exposing some of the undefined semantics of state machines. We also analyze areas of the specifications where there are two or more alternative interpretations. Umple's

implementation provides clarifications and a working solution to some of these ambiguities. (Chapter 4)

- Empirical evaluation of the Umple technology (Chapter 7).

  A controlled experiment has been designed and conducted to evaluate the effectiveness of the Umple technology. This experiment is the topic of Chapter 7: Experimentation.

- Open-sourcing the Umple technology

  Umple is now open for developers and contributors. Umple source is hosted in the Google code repository [7].

- Reporting on an application of Grounded Theory research methodology. We used a grounded theory study to learn about the community of Umple users and utilize their feedback to enhance our research direction and priorities. (Chapter 6)

1.4.1 Publications based on this thesis

All publications based on this thesis are presented in this section. The first author is the main author.

1. "Combining Experiments and Grounded Theory to Evaluate a Research Prototype: Lessons from the Umple Model-Oriented Programming Technology"

Omar Badreddin, Timothy C. Lethbridge. To appear in ICSE Workshop on User Evaluation for Software Engineering Researchers (USER), 2012.

2. "Model-Driven Rapid Prototyping with Umple"

Andrew Forward, Omar Badreddin, Timothy C. Lethbridge. In Software: Practice and Experience Journal, 2011.

3. "A study of applying a research prototype tool in industrial practice"

Omar Badreddin and Timothy C. Lethbridge. 2010. In Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10- Doctoral Symposium). ACM, New York, NY, USA, 353-356. http://dx.doi.org/10.1145/1882291.1882345

4. "Umple: A model-oriented programming language"

Omar Badreddin. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Doctoral Consortium - Volume 2, 2010, pp. 337--338. http://dx.doi.org/10.1145/1810295.1810381

5. "Teaching UML Using Umple: Applying Model-Oriented Programming in the Classroom"

   Timothy C. Lethbridge, Gunter Mussbacher, Andrew Forward, Omar Badreddin. In Proceedings of CSEE&T 2011, co-located with ICSE 2011, , pp. 421-428.

6. "Umplification: Refactoring to Incrementally Add Abstraction to a Program"

   Timothy C. Lethbridge, Andrew Forward, Omar Badreddin. In proceedings of the 17th Working Conference on Reverse Engineering http://dx.doi.org/10.1109/WCRE.2010.32. 2010, pp. 220-224.

7. "Umple: Towards Combining Model Driven with Prototype Driven System Development"

   Andrew Forward, Omar Badreddin and Timothy C. Lethbridge. In proceedings of the 21st IEEE International Symposium on Rapid System Prototyping http://dx.doi.org/10.1109/WCRE.2010.32. 2010.

8. "Challenges and opportunities in applying research prototypes and findings into industrial practice"

   Omar Badreddin, Tim Lethbridge, Hisham El-Shishiny, Margaret-Anne Storey, Andrew Forward. CASCON '10 Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research. ACM. doi:10.1145/1923947.1924021.

9. "Perceptions of Software Modeling: A Survey of Software Practitioners"

   Andrew Forward, Omar Badreddin, and Timothy C. Lethbridge. (2010) 5th Workshop From code centric to model centric: Evaluating the effectiveness of MDD (C2M:EEMDD), Paris, June 2010, http://www.esi.es/modelplex/c2m/papers.php

In addition, we have published the following technical report. A conference paper has been submitted and is being considered for publication.

10. "An Empirical Experiment of Comprehension on Textual and Visual Modeling Approaches".

    Omar Badreddin and Timothy C. Lethbridge. Technical report number TR-2011-03. Accessed 2011. http://www.eecs.uottawa.ca/eng/school/publications/techrep/2011/

1.5 Outline

Presented here is a short summary of each chapter.

Chapter 2: Background

This chapter presents background research, a brief introduction of Umple state machines, and a survey of state machine code generation approaches.

Covered in this chapter are existing technologies in state modeling and code generation approaches from state machines.

Chapter 3: Syntax and semantics of simple state machines

Our approach of representing state machines abstractions in Umple is presented in this chapter. The chapter also covers the design decisions and compromises that we undertook throughout the research study.

Chapter 4: Syntax and semantics of composite state machines

Nested and concurrent states concepts syntax and semantics are explored in this chapter. The chapter also explores aspects of the latest UML standard and how it relates to our approach.

Chapter 5: Implementation of composite state machines

A novel implementation of composite state machine semantics is presented in great detail in this chapter.

Chapter 6: Grounded theory study of Umple

We conducted a series of interviews with users of the existing Umple language, compiler and environment. We analyzed the interviews using the grounded theory approach and used the results as guidance to our research and experimental development.

Chapter 7: Experimentation

Experiment goals and objectives, metrics, design, results and analysis are presented in this chapter.

Chapter 8: Related Work

We present in this chapter selected on-going research activities that bear similarity to our research. We focus on highlighting aspects of the existing research that influenced our direction, and position our research with respect to existing work.

Chapter 9: Summary and conclusion

This summarizes our research activities and gives an outline of future research directions.

Chapter 2: Background

This chapter presents background research, a brief introduction of Umple state machines, and a survey of state machine code generation approaches.

2.1 History of State Machines

The mathematical foundations of state machines can be traced back to the Turing machines that were first described by Alan Turing in 1936 [8]. A Turing machine is composed of a tape, head, a table, and a state registry. The mathematical foundation of Turing state machines has been formalized in the Church Turing thesis that informally states that if an algorithm (a procedure that terminates) exists then there is an equivalent Turing machine.

The history of state machines is graphically summarized in the timeline in Figure 1.

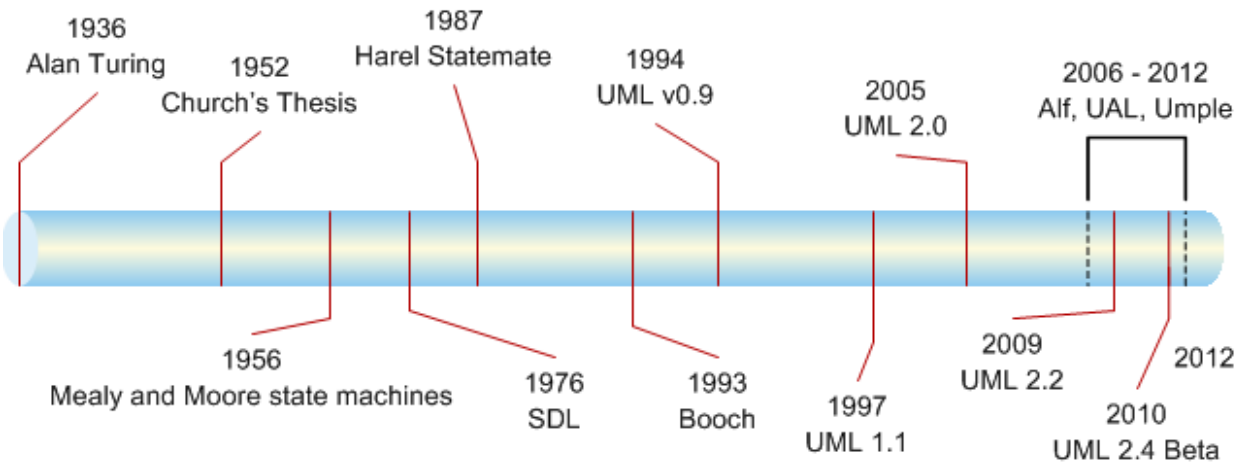


Figure 1: History of state machines

A Turing machine is a type of a state machine. At any point of time, the Turing machine is at one of a finite number of states. In modern terms, reading a character on the tape may, or may not, trigger a transition to a new, or the same, state. Any Turing machine can be effectively modeled using modern state machine diagrams. It is therefore that the Turing machine's mathematical model laid the grounds for more elaborate models that resembles today's notion of state machines, most notably are Mealy and Moore machines [9]. Mealy machines output depends on the current state and on the input (transition oriented state machine), while Moore machines' output depends only on the state (state-oriented state machine). Therefore, the same model implemented using Moore machines usually result in more states compared to the same model implemented using Mealy machines. For example, Wagner et al [10] present a

microwave implementation that results in a Moore machine with 7 states, compared to only 5 states using Mealy machine.

2.1.1 The Evolution of State Machines

A significant factor behind the development of the concept of state machines was the understanding of the practical significance of state machines. A prominent step towards that understanding is the work of Borger [11]. He realized that abstract state machines can solve some central problems that had faced the ISO Prolog standardization committee for years. After a number of unsuccessful attempts, a few engineers from IBM, Quintus, Bim, Interface, Siemens, demonstrated the benefits of state machines by highlighting the ability for supporting changing designs. State machines have also been significantly utilized in hardware design. Since the practical significance of state machines became widely accepted by researchers and practitioners, there have been a number of case studies and experiments that explore the full potential of this concept [12]. This takes us to the late 1980s and early 1990s that mark the origin of UML state machines diagram.

Specification and Description Language (SDL)

SDL has emerged from the communication domain and it is mainly used in the modeling of real time and communication systems [13]. SDL emerged from a study at the International Telecommunication Union (ITU) in 1968. The first SDL standard was produced in 1976. SDL has both graphical representation (SDL/GR) and a phrase or physical representation (SDL/PR) [14-16]. SDL is further discussed in "Comparing Umple and SDL" on page 175.

Harel Statecharts

Mealy and Moore machines suffered from a limitation; the machine was either in one state or in another state. The machine is never in two states at the same time. Harel [17] introduced the concept of an *and-state*. This allowed the state machine, or the statechart, to be decomposable into lower states, or sub-states, of a high level state. Those sub-states need not be sequential; Harel's proposed statecharts allows sub-states to be concurrent. In addition, Harel defined communication and synchronization methods in which these sub-states can communicate with each other. Douglass [18] has provided a well-defined enumeration of these communication and synchronization methods. In 1988, Harel presented StateMate [19], a working tool that encapsulates those concepts.

The Booch Method

Five years after the introduction of *StateMate*, a new enhanced method, based on Harel's statecharts, was introduced. Grady Booch developed an Object Modeling Language and methodology that became widely used in object-oriented modeling analysis and design [20]. Booch's focus was on states and events. Events could be defined within a state model, or could

be external to the system under design. The property "*StateKind*" determines whether the state is a normal state, or a special state (initial state, end state). The method also supported *stateRegion* that can be either sequential or concurrent. Events are attached to transitions that can have conditional expressions that are commonly called guards today.

The Object Modeling Technique

During the same period of time, another methodology was being developed by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen, named Object Modeling Technique (OMT) [21]. OMT supported a dynamic model that was primarily composed of states, transitions, and actions. The dynamic model captures control information without regard for what the operations act on or how they are implemented. It was conceptually very similar to the Booch's state machines.

The Unified Modeling Language (UML)

The development of UML began in 1994 when Booch and Rumbaugh began their work on unifying the methods. They were later joined by Ivar Jacobson, the author of OOSE (Object-Oriented Software Engineering) method. The three authors created UML v0.9 in October of 1996 [22].

Realizing the strategic importance of standardizing UML, a number of organizations joined forces to form the OMG (Object Management Group). This effort resulted in UML v1.0 in 1997. In the same year, the standard was enhanced and UML 1.1 was released. The current specification adopted by OMG today is UML 2.2 [23] that supports 13 different diagrams under three categories; structure, behavior and interactions diagrams. Our state machine implementation in Umple builds on the latest UML specifications, although we have not rigorously followed UML for pragmatic reasons, and because we want to be free to explore new ideas.

Current Developments

OMG, along with a number of industrial partners, is developing new standards that enhance UML executability; UML Action Language (UAL) and Action Language for Foundational UML (ALF). These two standards are at an early stage of development. We elaborate on UAL and ALF in Chapter 8: Related work.

2.2 Umple state machine example

We illustrate Umple state machine basic syntax by briefly introducing a state machine example. A much more complete demonstration of Umple state machine features is presented in Chapter 3: Syntax and semantics of simple state machines. Figure 2 illustrates a state machine of a car transmission system.

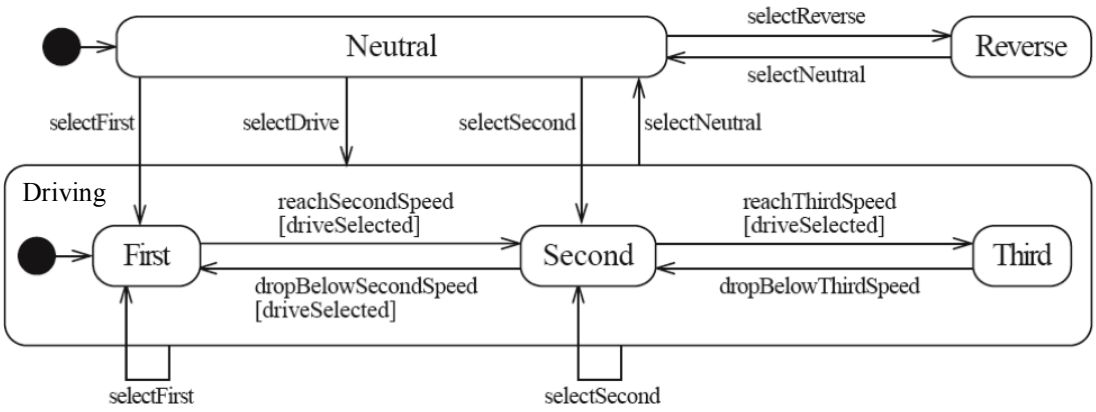


Figure 2: State machine of a car transmission

As shown in Figure 2, the car transmission system is comprised of a two-level nested state machine. The transmission starts in *Neutral* state. While in *Neutral* state, the state machine responds to four events; namely, *selectFirst*, *selectDrive*, *selectSecond*, and *selectReverse* events. Each event triggers a transition to a new state. For example, the transition *selectSecond* triggers a transition to *Second* state.

While in Second state, the transmission system responds to two events; *reachThirdSpeed* and *dropBelowSecondSpeed* that trigger transitions to *Third* state and *First* state respectively. Transition to *Third* state and *First* State are guarded. The guard *driveSelected* has to evaluate to true for the transition to take effect. If the guard *driveSelected* evaluates to false, the transition is inhibited.

Next, we illustrate how this state machine is represented in Umple.

```
1   class Car {
2     transmission   {
3       Neutral {
4         selectreverse -> Reverse;
5         selectSecond -> Second;
6         selectDrive -> Driving;
7         selectFirst -> First;
8       }
9
10      Reverse {
11         selectNeutral -> Neutral;
12      }
13      Driving {
14         selectNeutral -> Neutral;
15         selectSecond -> Second;
16         selectFirst -> First;
17
18        First {
19           reachSecondSpeed [driveSelected] -> Second;
20         }
21
22        Second {
23           dropBelowSecondSpeed [driveSelected] -> Second;
24           reachThirdSpeed [driveSelected]-> Third;
25        }
26
27        Third {
28           dropBelowThirdSpeed -> Second;
29
30  } } } }
```

Listing 1: Umple state machine syntax

Listing 1 illustrates Umple state machine syntax for the state machine illustrated in Figure 2.

Line 1: declares a class named Car.

Line 2: a class attribute named transmission. Because there is no declared type, Umple defaults the attribute in Java to be an *Enum* and in Php to be a string.

Line 3 to line 8: declares a state *Neutral*. The state is an initial state (Umple sets the first state defined to be the start state), and has 4 unguarded transitions to *Reverse*, *Second*, *Driving*, and *First* states.

Line 13 to line 30: Creates a state *Driving* that contains several nested substates; *First*, *Second*, and *Third*. Some of the transitions between *First*, *Second* and *Third* states are guarded transitions.

Line 19: defines a transition from state *First* to state *Second*. The transition is triggered by the event *reachSecondSpeed*. This is a guarded transition. The event *reachSecondSpeed* will not trigger the transition unless the value of the guard *driveSelected* evaluates to true. Umple users have two ways to declare a guarded transition. The transition can either be written as

event [Guard] -> StateName

or alternatively, the transition can be written as

[Guard] event -> StateName

Transition may have optional actions. The syntax for the transition with action is as follows:

EventName / ActionName -> StateName;

All actions have to be preceded by the character "/". The guard can be placed anywhere before the transition characters "->".

The next section presents a survey and an investigation of existing state machine code generation approaches.

2.3 Code Generation from State Machines

In this section, we give a survey of existing approaches to code generation approaches from state machine models. This survey guided our decision-making process with regard to generating executable artifacts from Umple models. We present Umple code generation and our decision points in section "State Machine Design Decisions" on page 60.

Different design approaches for code generation from state machines have been presented in the literature [24]. Adamczyk brings together a number of implementation approaches for state machines, and evaluates them based on the flexibility of the implementation, problem domain and user expectations. Adamczyk presents an implementation of a traffic light state system and analyzes the ease with which the system can be maintained. Briefly, the implementation patterns discussed in this work are grouped based on the state machine element concerned; state, event, transition and action. For example, the work presents three ways to implement a state: enumerated values, methods and classes. An action on one extreme can be a single statement, or a complex computation that can be encapsulated within a dedicated class. Adamczyk classified action implementation under three categories, unstructured code, methods, and classes. Similarly, transitions can be implemented using tables, state-driven transitions, and classes.

A similar study, [1], investigates extensions to the state pattern formulated as advice to developers implementing state machine behavior.

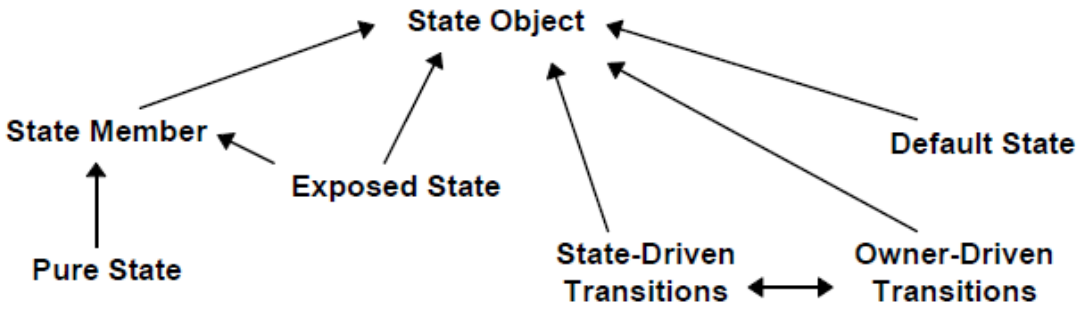


Figure 3: Extensions to state pattern [1]

As shown in Figure 3, the state object represents the core of the state pattern. Here, the state is encapsulated as an object. In the pattern of State-Driven Transitions, the state object is responsible for the handling of the transition. On the other hand, the Owner-Driven Transitions pattern represents the case where the owning object is responsible for the implementation of the transition. The State Member pattern deals with whether data members should be placed in the owning object or in the State Object. The Pure State is a pattern where the state object has nothing but a state-specific behavior.

Dyson explains how different patterns suit different types of state machines. For example, if the developer is faced with a large number of state objects, he can use the pure state pattern to cut down on the number of objects required.

We identify state machine patterns by investigating existing tools that support state machine code generation. We achieve this by conducting a survey and analyzing the code generation from state machines as exhibited in the existing open source and commercial tools. Our findings indicate the existence of distinct design characteristics for state machine code generation. We identify and group the design approaches under three categories; the *in-class pattern*, the *multiple-class pattern*, and the *extended multiple-class pattern*. Of those three design approaches, none is an all-time winner, as each alternative is more attractive under certain circumstances. In this section, we present the three main design approaches for code generation from state machines, as well as variations of those approaches. While laying out the design alternatives, we make reference to the latest commercial and open source tools and the design each has adopted.

While executing state machines and automated code generation have been reported in the literature for some time now, a surprising number of state-of-the-art commercial and open source tools do not support state machine code generation. According to Gartner's reports [25] IBM Rational Software Architect (RSA), as of 2007, is the top leading commercial object-oriented analysis and design tool. For open source tools, Gartner in another report [26] puts *ArgoUML* as the most active UML modeling tool, and *StarUML* as the most active open source tool that supports UML 2.0. *RSA*, *ArgoUML*, and *StarUML* support code generation from Class diagrams, but provide only limited support for code generation from state diagrams.

There are a number of other tools that support code generation from state machines. *Telelogic Tau* [27], Mentor Graphics' *BridgePoint* [28], *Borland Together for Eclipse* [29], *RSA RealTime*, and *SmartState* [30] are some leading commercial tools that support automated code generation from state machines. On the open source side, *FSMGenerator* [31], *Concurrent Hierarchical State Machine* (CHSM) [32], HUGO [33], and *FSM Framework* offer that support.

For the modeling tools that support code generation from state machines, we identified significant variations in the design approach followed by the existing modeling tools (summarized in Table 2 and Table 3 on page 40). Even for the tools that adopt the same design pattern, each follows a variant of it. This wide variation can be attributed to one or more of the following factors:

1. State machine elements may or may not be first-class object-oriented elements, which gives flexibility in the implementation of those elements. For example, states can be implemented as simple data attributes, or instances of classes.
2. The existence of a number of design approaches and the lack of comprehensive understanding of which design approach is most effective.
3. Certain application domains or platforms bring their own considerations, for example, embedded applications or performance-sensitive systems have specific needs.

In order to move towards a comprehensive understanding of state machine code generation design alternatives, we present the three main design approaches, and their variations as exhibited in the literature and the existing tool implementations. Our analysis evaluates candidates of those design approaches and aims at laying the foundations for more uniformity in state machine code generation.

2.3.1 Design Approaches

In this section, we present the three design approaches, and their variations. We make reference to commercial and open source tool implementations whenever possible. We illustrate the alternative design approaches by referring to the simple state machine in Figure 4.

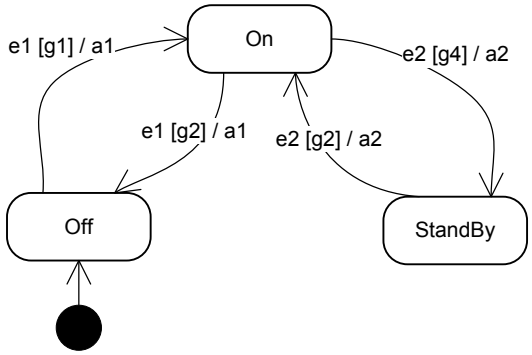


Figure 4: An Example State machine

The state machine in Figure 4 represents simple functionality present in a device. The device can be in one of three states, *On*, *Off*, and *StandBy*. Each state has an entry and exit action. Each transition has a guard condition and a corresponding transition action as illustrated. Event "*e1*" triggers the transition from state *Off* to *On*, and the same event triggers the transition from state *On* to *Off*, depending on which state is active. Similarly, event "*e2*" triggers the transition from state *On* to state *StandBy*, and the same event will trigger the transition from state *StandBy* to state *On*. For simplicity, we assume guard conditions to be a Boolean variable, and all entry and exit actions to be a function call. Events are triggered by an action external to the system.

This example is intentionally simple, as it does not include nested states, concurrent states, joins, or forks. The simplicity in this example enables us to consistently implement the example and generate code from a wide range of available commercial and open source tools. We employ two additional, and more complex, examples in our assessment and analysis (illustrated in Figure 9: Nested example and Figure 10: Concurrent example).

2.3.2 In-class pattern

In this design approach, the whole state machine behavior is implemented in a single class. The single class includes code to implement the core state machine behavior, typically by means of nested switch statement, if statements, or a transition table. The class includes implementations for functions representing all entry, exit, and transition actions, as well as guard implementations.

Variations. The core behavioral logic is implemented as a switch statement, or by implementing a state transition table as in Mentor graphics *BridgePoint*, or a nested if statement as in *Telelogic Tau*. The use of the deprecated *goto* statement is reported in the literature [1], however, there is no evidence that *goto* statements are implemented in any of the existing open source or commercial modeling tools. *Goto* statements suffer from weak readability and maintainability of the generated code, but may provide for a faster execution time.

Transitions in the implementation code are comprised of statements to call exit actions, checking for guard conditions, transition actions, deactivating old state and activating the new state. Such code is typically embedded within the switch statement or nested if statements. *Telelogic Tau* groups such code in a single method "*leave()"* that would execute all statements for affected transitions which results in a more modular and readable generated code. Umple adopts a similar implementation of the *leave* method.

Another variation of the single-class approach is the use of a code library that implements specific functions that are called by the state machine's single class. For example, the function to execute a transition from one state to the other can be implemented in a separate library. This approach is adopted in *Telelogic Tau*.

2.3.3 Multiple-class pattern

In this approach, a separate class is generated for each state that inherits from a State superclass. The superclass defines entry and exit actions that are then implemented in each state class. This approach is adopted in *HUGO* [33, 34], *Telelogic Rhapsody* [27], and SmartState. This design approach is similar to the State design pattern presented in [35] . For the example presented in Figure 4, adopting this design approach results in the class diagram shown in Figure 5.

Variations. This design pattern allows for a larger number of variations than the in-class pattern. In addition to the variations related to the implementation of the state machine behavior, there are variations related to when objects are created and destroyed. Objects representing states can be created only when that state is active. Another alternative is for all state objects to be created as soon as the state machine becomes active. Yet another alternative, implemented in *Telelogic Rhapsody*, is the use of an additional helper class that implements the state machine behavior.

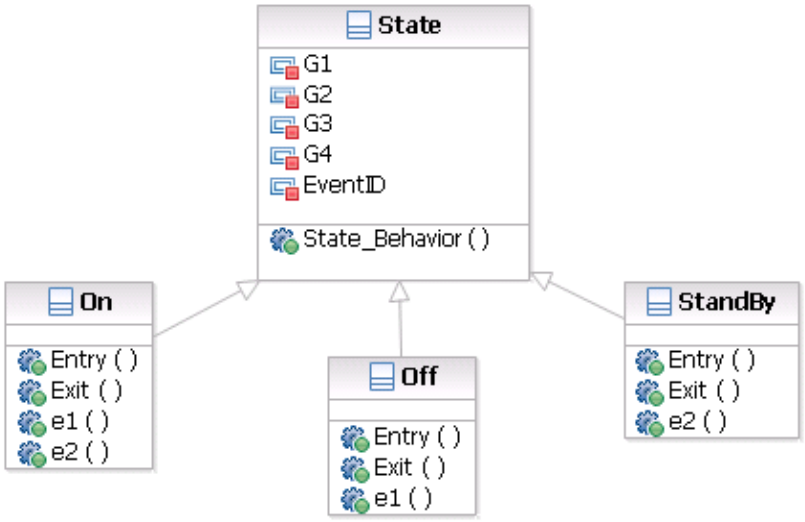


Figure 5: Multiple-class design pattern

2.3.4 Extended multiple-class pattern

In this approach, object orientation is taken even further, with separate classes used to implement actions and, in some variations, guards [35-37]. This design pattern provides for some centralization of state machine elements. In this approach, all actions, entry actions, exit actions, and guards are grouped together in dedicated classes. Following this design pattern to implement our example results in the class diagram in Figure 6.

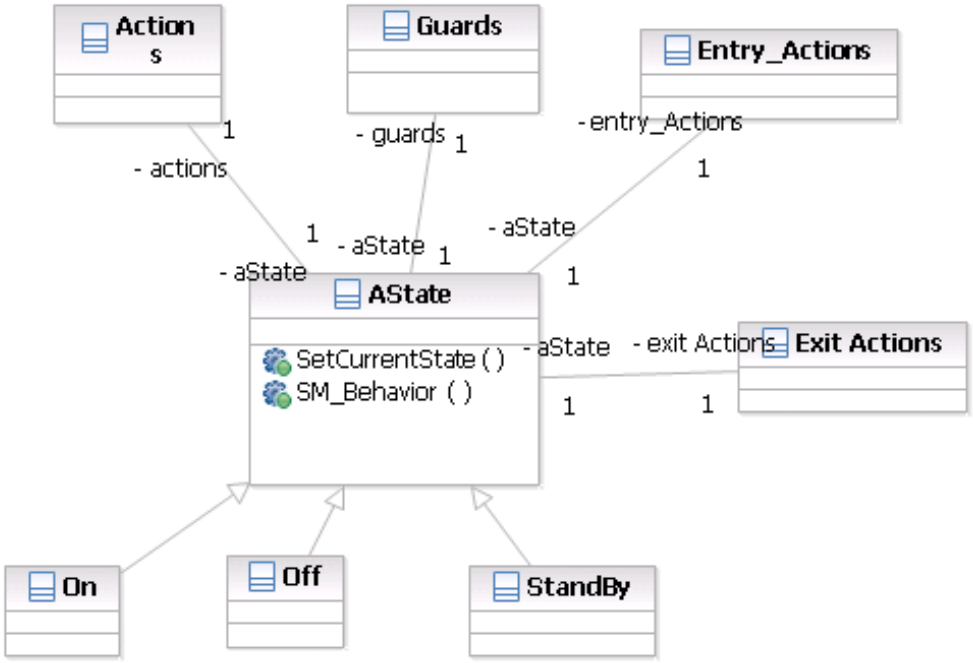


Figure 6: Extended multiple-class design approach

Variations. Variations for this design pattern are related to what elements of state machines are grouped together in a dedicated class. For example, [38] Tomura implements a separate abstract class for Entry Action, Do Action, Exit Action, and Guard condition. Other variations may implement transitions in a separate class. When a dedicated class implements the state machine transitions, the transition object will include statements to call any exit action from the current active state, deactivate current active state, check any guard condition, call any transition action, call any entry action into the new current state, and update the current state. This design approach is referred to as Owner-Driven transition [1]. If transitions are not grouped into a transition class, then state objects are responsible for transition from one state to the other. This design approach is referred to as State-Driven transition.

2.3.5 Alternatives within design patterns

In addition to the different design approaches, there are implementation specifics that can be adopted within any design pattern previously presented. Those implementation specifics relate to how states are represented and stored, as well as how actions, and guards are realized.

How states are represented

In the case of the in-class approach, states are represented by attributes. Those attributes can be strings, or constant values, or simply integer attributes as implemented in *SmartState*.

In the case of the multiple-class and extended multiple-class design approaches, states are represented as instances of classes, with one class per state. The current active state is tracked as a reference to the current state object. *Telelogic Rhapsody* creates objects for all states up front, which stay active in memory as long as the state machine is executing. Since in many systems, there is likely to be a number of states machines active at the same time, upfront object creation has performance significance especially since object creation can be expensive, particularly in embedded systems, or systems with a large number of states. Gurp and Bosch [36] recommend the Flyweight pattern [35] that allows objects to be shared among multiple contexts.

How guards are realized

There are three ways to implement guard conditions. The simplest way is the use of a Boolean attribute or a Boolean expression to represent the guard condition. Alternatively, guard conditions can be implemented as Boolean functions.

Semantically, guards prevent transitions in response to an event whenever the guard value is true. This behavior can be semantically equivalent to implementing “ignore events” for each guarded transition. An ignore event is a new event that is triggered when the original event is triggered and the Boolean expression is true. Semantically, this is equivalent to assigning a guard

condition and original event to the transition. This guard implementation approach is adopted in Mentor Graphics *BridgePoint* tool. We illustrate this design alternative further in Figure 7.

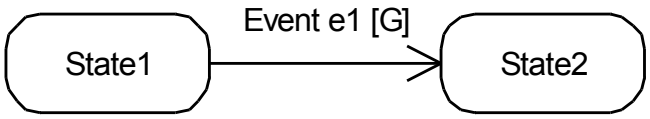


Figure 7: ignore event

Let's assume that when in State1 the state machine responds to Event e1 by triggering a transition to State2. In such a situation, the state machine implementation will check for the value of the guard 'G' before executing the transition to State2. If the value of 'G' is false, the transition is inhibited.

The *'ignore events'* design alternative does not use a guard condition. However, to implement the same behavior of the state machine in Figure 7, we delete the transition shown and replace it with a transition with a new event (say e2) that is triggered whenever e1 occurs and G is true.

How actions are represented

There is a general convention in tools and in the literature to implement actions as functions. However *where* the actions are implemented has a significant impact on complexity, maintenance, and performance, as we discuss later on when we assess the design patterns.

Actions can be implemented in each state class, as in the multiple-class pattern. Or, as in the extended-multiple-class pattern, actions can be grouped together in a designated action class. There are two arguments for grouping actions in the same class. The same arguments apply for variations that group guards, transition functions, and entry and exit actions in the same class. The first argument for grouping actions is to facilitate reuse and maintenance. The second argument is to maintain the separation between the state machine behavior (represented by actions) and structure (represented by the class hierarchy).

On the other hand, in object-oriented best practices, classes should include functions that manipulate the behavior of the instances of that class. Grouping all actions in a single class breaks this convention. We summarize the impact of grouping actions in a single class in Table 1. We further discuss the three design patterns and analyze their complexity, maintainability, and performance in the next section.

Table 1: Variations of implementation of Actions

	Grouping actions in the same class	Distribute actions on state classes
Applicable in design	In-class and extended-multiple class design approaches.	Multiple-class design approach.
Standard object oriented design principles	Breaks object oriented design principle of distributing responsibilities so that each object implements functions that manipulates its own data.[2]	In accordance with object oriented principles.
Reuse of actions among multiple states and state machines	Enhances reuse and facilitates maintenance by grouping all actions in the same entity.	Actions are distributed on the classes of each state, making reuse less intuitive and harder to implement.
Number of objects	State machine implementation results in an overhead of one additional action object.	No additional objects created for actions.
Actions representation	State machine actions have first class representation in the generated code.	Actions have no first class representation in the generated code (implicit representation).
Performance	There is evidence of performance degradation.	Distributing actions seems to reduce the computational overhead.

Summary of tools design approaches

Table 2 summarizes a number of leading tools and the designs they incorporate. Out of the six commercial tools and four open source tools we examined, three adopt the in-class design pattern, three adopt the multiple-class pattern, and one adopts the extended-multiple-class pattern. Three tools had little to no code generation support for state machines. Table 3 presents design variations related to the state machine core behavioral implementation, representation of states, and implementation of guard conditions.

[2] In the case of the in-class design approach, the whole state machine is considered to be a single entity, and is therefore implemented in a single class. In this case, there is no violation of standard object oriented design principles, however, the power of object orientation is not harnessed.

Table 2: Tool design approaches

		Design Pattern			Little to no support
		In-Class	Multiple-Class	Extended-Multiple-Class	
Commercial	**Telelogic Rhapsody**		**X**		
	Mentor graphics BridgePoint			**X**	
	Telelogic Tau	**X**			
	SmartState		**X**		
	RSA and RSM				**X**
	RSA RealTime	**X**			
Open Source	**StarUML**				**X**
	ArgoUML				**X**
	HUGO		**X**		
	FSMGenerator	**X**			
	FSM Framework [36]		**X**		

Table 3: Design variations implementation

	If Statement	Switch Statement	Table Driven	Representation of states	Guard Conditions
FSM framework			X		
BridgePoint			X		Ignore actions
Tau	X				User defined expressions
SmartState		X		Integer attribute	
Rhapsody		X		Upfront State object creation	
FSMGenerator	X			String attribute	
RSA RealTime		X		String Attribute[3]	

Discussion of the three design approaches

Figure 8 illustrates the different design approaches and their variations. Some of the variations apply to all three design alternatives, like "core behavioral logic" and "Guard implementation". Other variations are applicable only to a subset of the design alternatives, such as "object creation", which is only applicable to multiple-class and extended-multiple class design alternatives.

[3] RSA RealTime also creates an Index for states to identify parent state in the case of nested states.

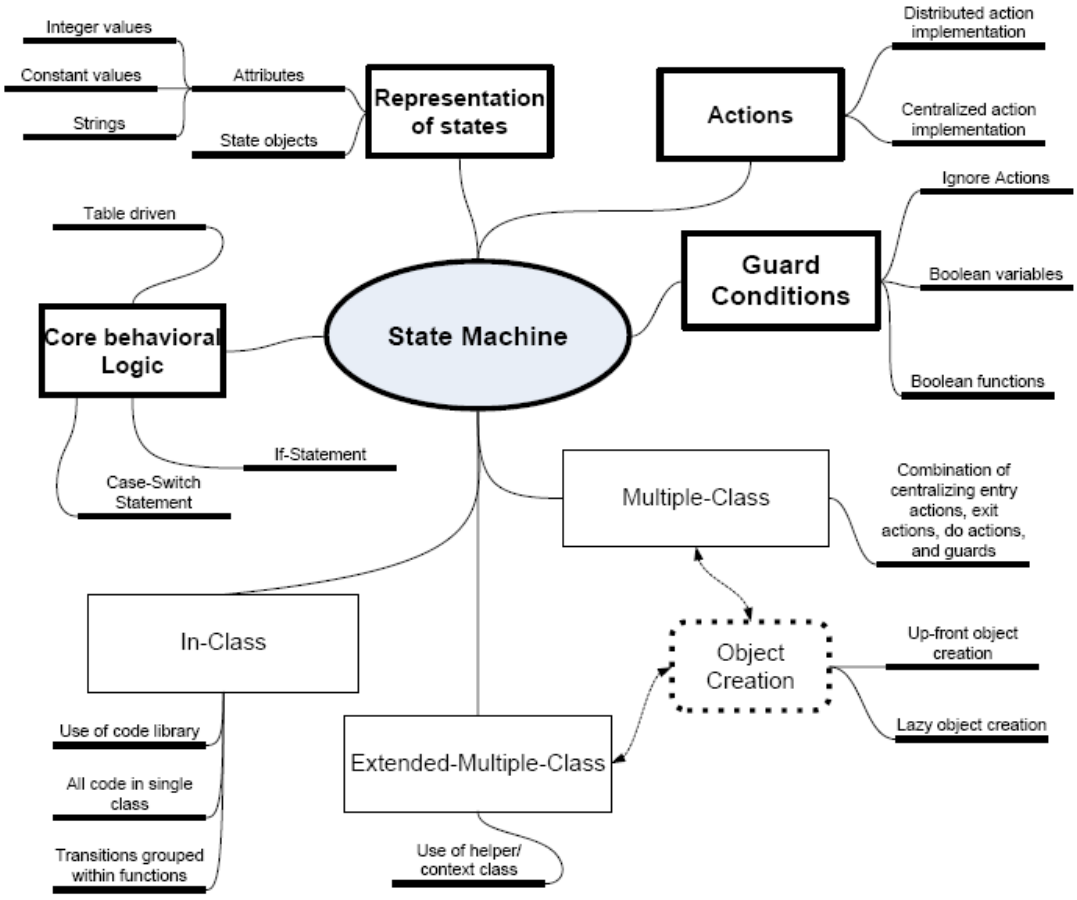


Figure 8: summary of design approaches and variations

The three design approaches, and the design variations, illustrate the gap between the model and the executable code that exists in state machine modeling. This gap induces the modeling tools, as well as developers, to manage the modeling artifact and the executable code as two separate entities. Making changes to one artifact will inevitably require some kind of synchronization; otherwise the two artifacts quickly become out of synch.

The in-class approach results in a smaller number of classes, although the number of lines of code inside that single class may be large, particularly if the state machine has many states or actions, or responds to a large number of events. At the other extreme, the extended multiple-class approach is assumed to provide for better reusability and maintenance, since all events, actions, and guards are grouped in their respective classes. Gurp [36] argues that the maintenance and evolution of the generated code from state machines, when actions are scattered, is very complex. By grouping actions in a dedicated class, maintenance tasks become less complex. To illustrate the complexity of the generated code, we applied a candidate of each design approach to three state diagrams with varying complexity. The simple example is illustrated in Figure 4. The nested example, illustrated in Figure 9, is comprised of four states, with two nested states.

The state machine in Figure 10 implements two concurrent states and is comprised of 5 states, as well as join and merge elements.

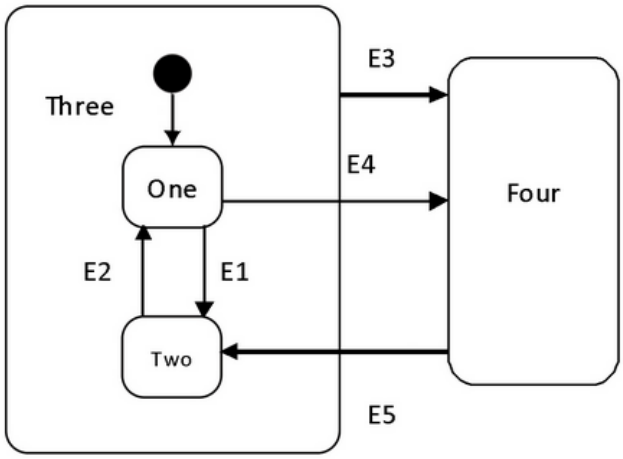


Figure 9: Nested example

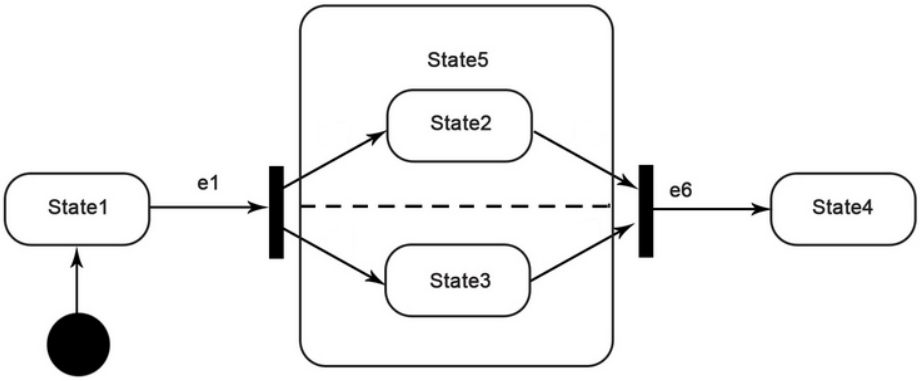


Figure 10: Concurrent example

Table 4 presents a comparison between the three design approaches and their corresponding generated lines of code and number of classes.

Table 4: Design approach comparison

	#of States	Least number of Lines of Code[4]			Least Number of classes[5]		
		In-Class	**Multiple-Class**	**Extended-Multiple Class**	**In-Class**	**Multiple Class**	**Extended-Multiple Class**
Simple	3	78	114	106	1	4	6
Nested	4	115	147	142	1	5	7
Concurrent	4	140	182	180	1	8	10[6]

The number of lines of code for the in-class design pattern is consistently the smallest, while there is no significant difference in the number of lines of code for the multiple-class and extended multiple-class design patterns. Table 5 presents a summary of the generated code from *Telelogic Tau* and Mentor Graphics *BridgePoint*. Tau implements a variation of the in-class design pattern where they make use of a *superclass* to implement some of the functionality of state machine behavior. The reported number of lines of code corresponds only to the in-class lines of code. On the other hand, *BridgePoint* implements a variation of the extended-multiple class design pattern, with generated classes for actions, events definitions, the state-events matrix that implements state machine behavior, as well as a header file for each class. *BridgePoint* does not support Java code generation; the reported numbers are based on a generated C code. There is no support for guard conditions; therefore, all guards are ignored when generating the code using *BridgePoint*.

[4] Measured by the number of Java code lines to implement the state machine behavior.

[5] Measured by the least number of classes in each design approach. In some variations of a specific design approach, the number of generated classes may be larger.

[6] Variations in this design approach may result in a larger number of classes generated.

Table 5: Generated code from commercial tools

Lines of Code	#of States	Telelogic	MentorGraphics
		Tau	BridgePoint
Simple	**3**	100	150[7]
Nested	**4**	95[8]	N/A[7]
Concurrent	**7**	N/A[9]	N/A[10]

Assessment of the three design approaches based on Complexity, maintainability, and performance

The fundamental question of which design approach is 'better' is not easy to resolve, as evident by the diverse approaches and variations adopted by the commercial and open source tools available today. We base our assessment on three factors; complexity, evolution, and performance. Similar factors have been adopted for evaluating automated code generation [39].

Complexity

We measure complexity of the generated code by Lines of Code (LOC), number of generated classes, and the separation of structure and behavior of the state machine. LOC, despite its apparent simplicity, is arguably the most effective measure for complexity [40]. The number of generated classes increases the complexity of the generated code. Separation of structure and behavior is the main benefit of the extended multiple-class design approach. As with cohesion [41], the separation of structure and behavior results in systems that are less complex.

LOC analysis. As illustrated in

Table 4, the in-class design pattern consistently resulted in a smaller number of LOC. On average, code generated with the in-class design pattern is 74% smaller than the code generated with the multiple-class design pattern and 77% smaller than the code generated with the extended multiple-class design pattern.

[7] This number ignores code in header files, as well as code comments.

[8] Telelogic does not support transitions into an inner state. The reported lines of code hence implement a model that is semantically different.

[9] Telelogic supported orthogonal nested states.

[10] BridgePoint does not support concurrent states.

Number of generated classes.

Table 6 illustrates the generated classes for the three examples.

Table 6: Number of classes for different design approaches

#of Classes	#of States	Design				
		In-Class	Multiple-class		Extended-Multiple class	
			Minimum	Up to	Minimum	Up to
Simple	**3**	1	4	5	6	11
Nested	**4**	1	5	6	7	12
Concurrent	**7**	1	8	9	10	15

The in-class design pattern always results in the same number of classes for the three examples. For the multiple-class pattern, the number of classes is equal to the number of states, but can have an additional helper class. For the extended multiple-class approach, the total number of classes is between three and eight more than the number of states. This variation depends on whether there are separate classes for entry, exit, and do actions, as well as guards, and transitions.

Separation of structure and behavior. The correspondence between the structure of the state machine and the generated code is more evident in the multiple-class and the extended-multiple class design patterns; this is because states are represented as classes. While in the case of the in-class design pattern, this correspondence is less evident. From the state machine behavior perspective, actions are distributed on all states in the multiple-class design pattern, while they are grouped in a single class in the extended-multiple class design pattern.

However, since actions are implemented as functions, they are cohesively grouped together within a single class in the case of in-class design pattern. A developer trying to understand the behavior of the system will know exactly where to look for actions within the single class implementation.

Maintainability

Maintainability, or evolution, from the perspective of this analysis, is the ease with which the code of a state machine generated system can be maintained and modified. Ideally, evolution and maintenance tasks should be performed on the state machine model and the code regenerated. However, in many cases models are either lost, or they are not updated and quickly become obsolete, then maintenance tasks are performed on the generated code itself. We measure maintainability by the complexity of adding a new transition to a new state, and

measuring how much code needs to be edited, and where. Programmatically, to add a new transition to a new state, the following micro tasks are required:

1. Edit core state machine behavior (whether it is switch, nested if statement, or table driven)
2. Create Entry, do, and exit actions
3. Create transition from existing state to the new state.
4. Create transition from new state to existing states, if any.

For the in-class design pattern, the developer will accomplish all micro tasks by editing the same single class. On the other extreme, in the extended-multiple class design pattern, the developer needs to edit the core state machine class and the one or more action classes.

Performance

We implemented the simple and the nested examples using the three design approaches; in-class, multiple-class, and extended-multiple class. The core state machine behavior was implemented by using a nested switch statement, and all guards were implemented as Boolean variables. In the case of in-class design pattern, states are represented by an integer variable. For the multiple-class and the extended multiple-class patterns, the current state is identified by a reference to the current state object. The code for the three design patterns was manually written in Java.

We evaluate the performance of the same 1 million state transitions, taking readings every 100th transition. The sequence of events was randomly generated. Each event is assigned equal probability of occurrence so that the number of occurrences of each event is probabilistically equal. Because the number of events is vast (1 million) in comparison to the number of states (3 states in the simple example), each state was entered and exited at least once. All guards were implemented on each transition, but were assigned a fixed true value. Since we are not evaluating different guard implementations, assigned fixed true value to guard conditions ensures that the performance analysis results reflect the design pattern of the state machine implementation.

The concurrent example incorporates concurrent states that have implementation specifics beyond the scope of analysis of the design approach, and we therefore exclude it from the performance analysis. Our findings are summarized in Figure 11. The multiple-class design pattern results in the best performance, only slightly better than the in-class design pattern. While the extended multiple-class design pattern implementation exhibited the worst performance. Because the extended-multiple-class design pattern implements separate objects for guards and transitions that have to be referenced in response to each event, this results in an additional computational overhead and hence relatively lower performance.

Figure 11 illustrates the results of our performance analysis for the three design approaches. The y-axis represents time, and the x-axis represents the number of transitions. We note that our results did not give a straight line. We believe this is due to memory exhaustion or some similar operating system phenomenon.

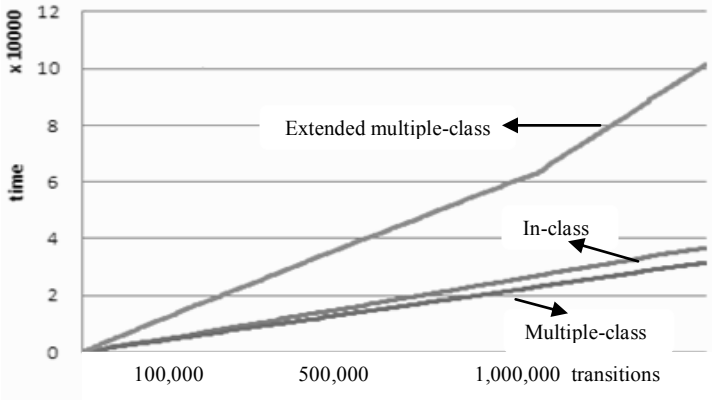


Figure 11: Performance analysis of the three design approaches

Our results are in accordance with the performance analysis reported in [41]. In their work, they analyzed a variant of the multiple class design pattern (that they named state pattern), and a variant of the extended multiple-class design pattern (their proposed framework). Their performance analysis concludes that the state pattern is more efficient if a lot of small transitions take place, as was the case in our performance analysis. They also conclude, however, that this difference becomes negligible if the actions on the transitions become more computationally intensive.

2.4 Summary

The history and evolution of state machines was briefly surveyed in this chapter. We also introduced Umple`s modeling approach for simple state machines. We presented a survey of state machines code generation where we identified three code generation patterns:

1. In-class pattern, where the entire state machine code is generated within a single class.
2. Multiple class pattern, where each state is generated in a separate class.
3. The extended multiple class pattern is where additional state machine elements are implemented in a separate class. For example, in this pattern, all state machine actions can be implemented in a separate class.

We have drawn these patterns from studying existing modeling tools, both commercial and open source. We made an assessment of each code generation pattern. This work laid down the

foundation for our experimental development of state machines in the Umple platform. As we demonstrate in the next chapters, we have chosen the In-class code generation pattern for Umple. Reasons for this choice are discussed in the next chapter in section “Design decisions” on page 62.

Chapter 3: Syntax and semantics of simple state machines

State machines in UML have two types of states, simple states and composite states. Simple states are the focus of this chapter. Composite states, the topic of the two next chapters, can be nested states (substates) or concurrent states.

In this chapter, we present the incorporation of state machine features into Umple. We present Umple state machine syntax and features related to simple states. We also present the language grammar, meta-model, and various design decisions we made.

This chapter focuses on the following aspects of state machines:

1. Designating an attribute for control by a state machine.
2. Creation of an arbitrary number of states and transitions.
3. Support of guard conditions on transitions.
4. Support of transition actions.
5. Support of automated code generation for Java and PHP.
6. Support of inline implementation of guards and actions.
7. Support for reusable state machines.
8. Support for timer based events.

The next section introduces an example showing Umple state machine features.

3.1 State Machines in Umple: The Basics

An attribute in Umple can be declared to be a state machine. This means that its value is determined by various events that may occur. When an attribute is controlled by a state machine, Umple does not generate a public setter for that attribute since updates to that attribute will be controlled by the state machine itself.

States: Any string or integer attribute can have an unbounded number of states. Listing 2 illustrates an example.

```
class CourseSection {
  status {
    Planned {//state contents, events, transitions and actions }

    Closed {//state contents, events, transitions and actions }
  }
}
```

Listing 2: Attribute controlled by a state machine in Umple

This defines the string attribute *status* to be controlled by a state machine. This state machine has two states, *Planned* state and *Closed* state.

Umple by default makes the first state to be the start state. In our example, *Planned* state is the start state. Any state that does not have any outgoing transitions is considered an *End* state.

We now can define the state machine behavior by adding events, guards, transitions, and actions.

Events: From a state machine perspective, events occur outside of the system; the system only reacts to those events. Umple, therefore, implements event-handling functions. These event-handling functions execute steps to check the current state of the state machine, and call any entry and exit functions, and executing the transition action, if such an action exists.

Because Umple supports native code, the developer can write any function that could trigger any Umple event. This is a powerful feature in Umple because it gives the developer the ability to call Umple events at any time. However, it is the developer's responsibility to make sure that the event function does not have any side effects.

Before we show an example of Umple event syntax, we first introduce *transitions*.

Transitions: Umple supports syntax for state transitions. Umple also supports reflexive transitions, where the new state is the same as the start state.

The next example adds a transition to our state machine.

```
status {
  Planned {
    registerStudent -> Closed;
  }

  Closed { }
}
```

The example above defines an event *registerStudent* that triggers a transition to the state *Closed*. Umple events are implemented as functions that return a Boolean value. If the event results in triggering a transition, true is returned, otherwise, false is returned.

Guards: Guards may prevent a transition from occurring. If the guard evaluates to true, the transition is triggered, otherwise, the transition is inhibited.

Umple uses the square brackets [ ] which is the same as the UML syntax for defining guards. The following code shows the addition of a guard to a transition.

```
[authorized] registerStudent -> Closed;
```

This guard means that only if the value of authorized is true, that the transition is triggered. Note that authorized has to be a Boolean variable, a Boolean expression, or a Boolean function. The guard syntax could also be written as:

```
[authorized == true] registerStudent -> Closed;
```

The code inside the square brackets has to match the native language code. So, if the user's intention is to generate Php, the user has to use Php syntax, and if the user's intention is to generate Java, the user has to follow the syntax for Java.

Umple also supports any function call within the square brackets, as long as the function returns a Boolean value. This enables developers to create guards once, and reuse them in as many transitions as they wish.

Umple also supports guards to appear syntactically after the event. This can enhance readability and usability when there are many transitions and the developer wants the person reading the code to more readily notice the names of events. For example, the transition above can be written as:

```
registerStudent [authorized] -> Closed;
```

Actions: Umple supports the three types of state machine actions, transition actions, entry actions, and exit actions.

A transition action is an action that is associated with a state machine transition. An entry action is an action that is executed upon transiting into a state. Similarly, an exit action is an action that is executed upon transiting out of a state.

The following shows a transition action.

```
registerStudent /{sendNotification();} -> Closed
```

This transition reads as follows: when the event *registerStudent* occurs, execute the action *sendNotification( )* and transit to state *Closed*. In this example, the transition action is a function call.

Umple also supports actions to be any native code, or block of code. For example, the following transition when triggered prints "*transition*" on the console:

```
registerStudent /{System.out.println("transition");} -> Closed;
```

As with guards, allowing actions to be any function call means actions can be reused across transitions, state machines and classes. In addition, the same action can be reused as entry or exit actions. An Umple user can create a method to call multiple actions and/or events. This approach enhances the usability of the language by grouping together a number of actions and events within the same method.

The following is an example of an entry and exit action for the state *Closed*.

```
Closed {

  entry /{ System.out.println("entry action");};

  exit /{ System.out.println("exit action");};

}
```

This creates one entry and one exit action for the state *Closed*. This means, whenever we transit into *Closed*, the entry action is executed, and whenever we transit out of *Closed*, the exit action is executed. Similar to transition actions, entry and exit actions can also be function calls, and can be reused in the same way. In addition, it is possible to have more than one entry action or exit action associated with the same state.

Do Activities: Actions take a negligible amount of time to execute. Do activities, on the other hand, represent a longer-running computation while in a state. In languages such as Java that support it, a thread will be started to execute the do activity. This allows the state machine to 'stay live' and be able to respond to other events, even while the do activity is running. A transition out of a state terminates the do activity.

The following is an example of a do activity in the *Closed* state.

```
Closed {

  do {doThisContinuouslyWhileClosed();}

}
```

3.2 Grammar defining the syntax of Umple state machines

The grammar to parse state machine elements has to be embedded within the grammar that parses classes, attributes and associations. This is because the parsing process has to recognize the tokens for class and state machines at the same time. The grammar is published as part of the Umple Google Code project [7] and can be found in the following directory: `svn/trunk/cruise.umple/src/umple_state_machines.grammar`

R1	**classContent** : [[comment]] |...| [[stateMachine]] | [[extraCode]]
R2	**associationClassContent**:[[comment]]|...|[[stateMachine]]|[[extraCode]]
R3	**stateMachineDefinition** : *statemachine* [**name**] { [[state]]* }
R4	**stateMachine** : [[enum]] | [[inlineStateMachine]]
R5	**inlineStateMachine** : [**name**] { ([[comment]] | [[state]])* }
R6	**enum** : [**name**] { } | [**name**] { [**stateName**] (, [**stateName**])* }
R7	**state** : [**stateName**] { ([[comment]] | [=changeType:-|*]? [[stateEntity]])* }
R8	**stateEntity-** : [=-||] | [[transition]] | [[entryOrExitAction]] | [[activity]] | [[state]]
R9	**transition** : [[guard]] [[eventDefinition]] -> [[action]]? [**stateName**] ; | [[eventDefinition]] [[guard]]? -> [[action]]? [**stateName**] ; | [[activity]] -> [**stateName**]
R10	**eventDefinition-** : [[afterEveryEvent]] | [[afterEvent]] | [**event**]
R11	**afterEveryEvent-** : *afterEvery* -([**timer**] -)
R12	**afterEvent-** : *after* -([**timer**] -)
R13	**action** : / { [**actionCode] }
R14	**entryOrExitAction** : [=type:**entry**|**exit**] / { [**actionCode] }
R15	**activity** : *do* { [**activityCode] }
R16	**guard** : [[**guardCode]]

Listing 3: Umple state machine grammar

3.2.1 Overview of the notation

The grammar notation that Umple uses is slightly different than the standard EBNF notation. This is because the Umple language is unique in the way it supports the embedding of arbitrary native languages. At the time of writing, Umple supported Java, Ruby, and Php. Additional language support is underway. This means that an Umple user can choose to embed a wide variety of native code within Umple. The grammar and the parser therefore need a mechanism to be able to identify blocks of code and accept them as is without parsing. However, the grammar notation developed for Umple resembles as much as possible the EBNF. The following discussion clarifies Umple grammar notation.

Managing rule names

The Umple grammar introduces the minus character ("-") which is a special control character that controls whether the rule name is added to the tokenization string or not. The minus character is useful when a rule acts as a place holder to help modularize the grammar. By adding the minus character to the end of the rule name, it removes the rule name from the tokenization string. For example, in rule R8 in Listing 3, the rule name *stateEntity* is not added to the tokenization string. This helps keep the tokenization string for states relatively short and simpler for testing and debugging.

Non-terminals

The Umple grammar supports two types of non terminals, simple non-terminals, and rule-based non-terminals. A simple non-terminals is a sequence of characters that is non-whitespace and is delimited by the next symbol as defined in the grammar.

inlineStateMachine : [name] { ([[state]])* }

In this example, name is a non-terminal followed by a curly bracket, a space, or a new line character.

The rule-based non-terminal notation uses double square brackets. In the example above, state is a rule-based non-terminals, which is defined in R7 in Listing 3.

Managing code blocks

As we explained, the tokenization process must be able to '*skip-over'* code blocks without any strict parsing rules. This special need for Umple is the main reason why Umple grammar does not use Antlr [42]. The grammar notation supports two methods to accomplish this task.

```
entryOrExitAction : [=type:entry|exit] / { [*actionCode] }
```

This rule defines that an entry or exit action is defined by the terminal entry or the terminal exit, followed by the terminal "/", followed by a curly bracket. The ***actionCode*** will match everything until a new line character is reached. This is very useful for Umple because it means that the action code can be specified in any target language, and allows the grammar to stay unchanged as new languages are added.

Note that this means that an action code must be in one line. This is an undesired limitation. Umple grammar therefore supports the following notation.

```
entryOrExitAction : [=type:entry|exit] / { [**actionCode] }
```

When the action code is preceded by two stars "**", the rule will match everything, including a new line character until the next character sequence is matched. This means that action can span multiple lines with no limitation on the sequence of action code itself.

The following explains Umple grammar rules for parsing state machines that are in Listing 3.

R1 and R2 define that Umple classes and Umple association classes can have state machines as attributes.

R3 defines a state machine by the keyword *statemachine* followed by a name followed by a number of states between curly brackets. This is used to declare a state machine independently of a class.

R4 defines two types of state machines in Umple that can be embedded in classes; *enum*, and *inline state machine.*

R5 and R6: Inline state machines are defined as a name followed by a number of states (R5). *Enum* state machines (R6) are empty state machines, or state machines with only states (with no transitions or actions). These are logically equivalent to an enumerated data type. The only way to change the state is to set the state using an assignment statement.

R7 and R8 define a state. Notice that a state contains state entities, which themselves can be states. This supports the implementation of nested and concurrent states discussed in Chapter 4: Syntax and semantics of composite state machines.

R9 defines Umple state machine transitions.

R10, R11 and R12: Umple defines three event types; *afterEvery* event, and *after* event, and the generic event. The first two are timed events, causing a transition to be taken after a certain amount of time has lapsed. The main difference between *afterEvery* and the *after* events is that the timer automatically resets itself and starts counting again. While in the case of *after* event, it is a simple timer that triggers the event after a specific amount of time.

R13 and R14: Umple supports three types of actions; transition action, entry action, and exit actions.

R15: This defines do activities, that start a long-running and interruptible thread to perform some lengthy computation, for example.

R16: Guards, similar to state machine actions, can be defined in any native language.

Notice that the grammar is agnostic about composite state machines. Concurrent states and nested state machines are handled at the meta-model level. This is discussed in greater details in Chapter 5: Implementation of composite state machines.

3.3 Umple state machine meta-model

The Umple state machine metal-model is similar to the UML 2.2 meta-model. There are elements that are in our meta-model that are not in the UML 2.2 meta-model specifications [23]. We introduce our meta-model first, and then discuss the similarity and differences with the UML 2.2 specifications.

We built the state machine meta-model using Umple itself. Figure 12 illustrates the Umple state machine Meta model visually, and using Umple syntax.

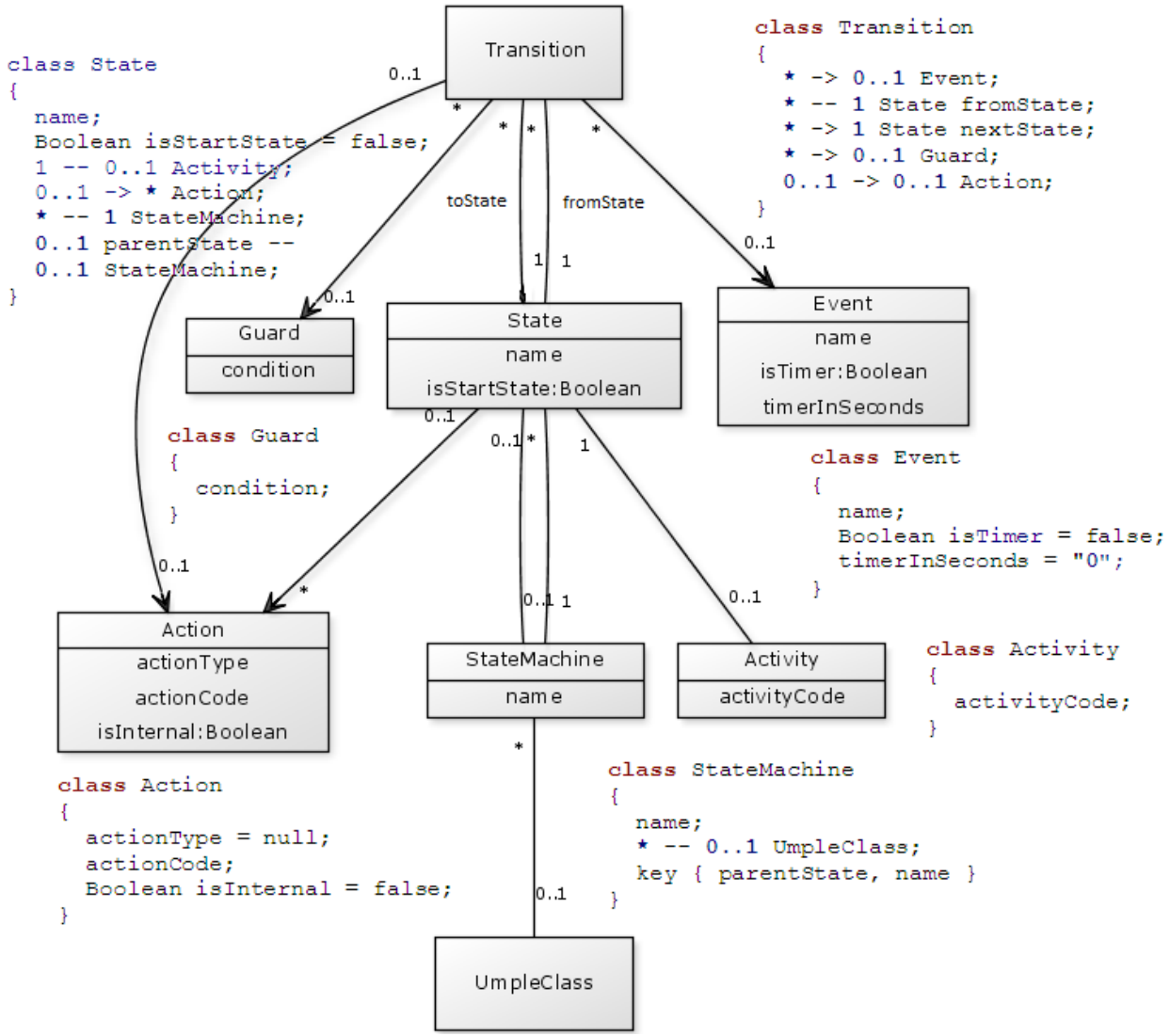


Figure 12: Umple meta-model

As shown in the meta-model, *UmpleClass* can be associated with many state machines. This is because each Umple class can have multiple, or no, state machines. A state machine, however, may, or may not, be associated with an *UmpleClass*. This is because Umple supports standalone state machines; state machines that are not yet associated with any Umple Class. Umple uses a novel approach for handling composite states (nested, concurrent, forks, joins, history and deep history states) that requires only minimal meta-model dependency. This approach is the topic of the next chapter. Because Umple supports reusing of actions and guards; guards have a 1-to-many relationship with transitions. Similarly, a state can be associated with many Actions.

The differences in attributes with UML 2.2 are summarized in Table 7. Some of the differences are because UML 2.2 includes specifications for the visual layout of the diagram. For example, UML 2.2 specifies *connection point, final state*, and *PseudoStateKind*, which are related to the visual layout of the state machine diagram. Umple supports regular events, and timed events; therefore, we have additional attribute for the events, while UML 2.2 imports events specifications from *UML:: CommonBehaviours:: Communications :: Trigger*.

Table 7: Comparison between Umple and UML 2.2 state machine meta-models

	Umple state machine meta-model	**UML 2.2 state machine meta-model**
State	string: name *boolean*: *isStartState*	*boolean*: *isComposite*, *isOrthogonal*, *isSimple*, *isSubmachineState*
Transition	No attributes	kind: internal, local, external
Pseudostate	Umple handles some *pseudostates* differently (refer to Chapter 4: Syntax and semantics of composite state machines and Chapter 5: Implementation of composite state machines). Entry and exit Points are not supported.	initial, *deepHistory*, *shallowHistory*, join, fork, junction, choice, *entryPoint*, *exitPoint*, terminate
event	string: name *boolean*: *isTimer* float: *timerInSeconds*	UML:: *CommonBehaviours*:: Communications :: Trigger
Action	string: *ActionType* string: *actionCode* *boolean*: *isInternal*	UML :: *CommonBehaviors* :: *BasicBehaviors* :: Behavior
Statemachine	string: name	No attributes
Activity	string: *activityCode*	UML :: *CommonBehaviors* :: *BasicBehaviors* :: Behavior
Guard	string: condition	UML::Classes:: Kernel :: Constrain
Other elements	No meta-model representation. Refer to Chapter 4: Syntax and semantics of composite state machines and Chapter 5: Implementation of composite state machines for more information on how Umple handles composite state machines.	Region, Vertex, ConnectionPointReference, FinalState

3.4 State Machine Design Decisions

In this section, we state our motivating goals and present the major design decisions we made during the building of state machines in Umple.

3.4.1 Umple state machine goals

Our objective is to create a straightforward syntax that can enable developers to quickly, efficiently, and sufficiently create executable state machines. Umple should provide a simple syntax to create and define state machine elements. We have the following syntax, design, and generated code related goals:

Goal 1: Minimal use of reserved words.

We should avoid the introduction of new reserved words as much as possible. Reducing the number of reserved words reduces the complexity of the language and makes it easier to learn. Wherever we do introduce reserved words, we should consider using reserved words that are used for the same purpose in other languages.

Table 8 summarizes the keywords and symbols used in Umple.

Table 8: Umple state machine keywords

entry/	An element of a state. Designates an entry action.
exit/	An element of a state. Designates an exit action.
do	An element of a state. Designates a do activity.
Final	A special state when reached indicates that a state machine is completed.
[ ]	A symbol for guard conditions.
->	A symbol for transition to a next state.
||	A symbol for a concurrent region.
{}	Curley brackets used for actions code.

By using concise syntax grammar, we were able to eliminate the need to use keywords for the following state machine elements (Table 9):

Table 9: Minimizing the number of keywords

State machine	An attribute name followed by a bracket is identified as a state machine.
Start state	The first state is the start state.
End state	Any state without outgoing transitions is considered an end state.
Action code	Action code is native language code between two curly brackets.
Transition action	Transition action code follows a '/'.
Guard code	A native language code that must evaluate to a Boolean value and is placed between two square brackets.
Nested states	Nested states use the syntax of nested curly brackets.

Goal 2: Umple syntax should be concise.

Developers should be able to create and specify state machine elements in a concise manner. Concise syntax contributes to enhanced readability, comprehensibility, and reduced complexity.

Goal 3: Umple syntax should be easily extensible

Whenever possible, Umple syntax should allow for additional functionality with minimal disturbance to the syntax, and underlying tokenization and parsing processes.

Goal 4: Umple syntax should look and feel like high level programming languages.

Developers who are already accustomed to writing code should find Umple familiar and easy to learn.

Goal 5: Umple syntax should eliminate the need to edit underlying generated code.

Umple supports native code for all types of actions and guard conditions. The syntax should enable developers to satisfy their development needs without requiring the editing or inspecting the generated code. This is similar to how software developers do not generally inspect the code generated by the high level programming languages compilers.

Goal 6: Umple generated code should be efficient.

By efficient we mean that the code should satisfy the state machine semantic behavior, while having comparable performance levels to the best code written by hand.

Goal 7: Umple generated code should look like code written by hand.

The generated code should be as readable as the best state machine code written by hand. The main reason for this is so that programmers can easily verify it. Note that there is no contradiction between eliminating the need to edit the code (Goal 5) and this goal. We aim at

making Umple's generated code easy to understand and verify, and at the same time, users can edit Umple code itself to make any necessary changes.

Goal 8: Umple should exploit textual modeling potential.

Textual modeling, we claim, allows us to create state machines models in a unique and powerful way. For example, Umple should maximize reuse of state machine models. It ought also to be possible to, for example, merge several state machines or compose them from textual files containing various components of a state machine. The appearance of multiple actions for the same state should also be supported, with the compiler simply combining them.

3.4.2 Design decisions

This section presents the design decisions we have made. We present the design alternatives, the decisions made, and align our decisions to our stated goals.

Decision Point 1: State machine design pattern

Contributes to goal 5 and goal 7

In section "Code Generation from State Machines" on page 31, we presented our survey of existing design alternatives for state machine code generation as exhibited in the state-of-the-art commercial, open source, and research prototype tools. Our Umple state machine implementation adopts a variant of the In-Class design pattern. The In-Class design pattern has the following properties that contribute to our Umple goals:

1. Number of lines of Code.

   The In-Class design decision results on average in a smaller number of lines of code.

2. Performance considerations.

   The In-Class design pattern performance analysis results in performance that is significantly better than the extended multiple-class design pattern, and only negligibly worse than the multiple-class design pattern.

3. Number of generated classes.

   The In-Class design pattern always generates a single class. Comparison of the three design pattern is summarized in Table 6 on page 45. In typical systems that are comprised of a number of classes, having more classes generated for a state machine implementation results in generated code that is less intuitive, and confuses the developer since classes that represent real system entities become mixed with state machine implementation classes.

4. State machine for attributes.

   Because Umple supports state machines for attributes that are already within an Umple class, it is more convenient to generate the state machine code within the same Umple generated class. The simplicity can be even more significant when there are multiple attributes in a given class, each with its own state machine controlling it.

There are factors that may result in other design patterns being more attractive. For example, our Umple syntax and meta-model, as well as the parsing and tokenization mechanisms, support reusable actions and reusable guards. Implementing reusable state machine elements may be easier if another design pattern is adopted. For example, generating a dedicated class for all actions may make it easier for developers to locate actions and reuse them. This is particularly true for a state machine diagrams with a large number of actions. In addition, having more classes means objects that are created are smaller in size, which could mean enhanced run-time performance.

The mitigation of such compromises brings about the following alternatives:

Alternative 1: Always implement the In-Class design pattern regardless of the state machine characteristics (size, reusable actions and/or guards, etc). This is the alternative that Umple currently adopts.

Pros: The generated native code always looks the same regardless of the state machine characteristics. In situations where developers need to inspect the generated code, the code will look more familiar and predictable. The Umple platform is hence less complex, as we always generate the code using similar templates.

Cons: Less flexibility, as the user cannot override the chosen design pattern.

Alternative 2: dynamically apply a design pattern based on the state machine characteristics.

This alternative implies that the characteristics of the state machine itself (i.e, number of states and transitions) determine the design pattern used for code generation.

Pros: The generated code is customized to the type of state machine under implementation. The size of the generated code may be well balanced on a number of classes if the state machine was large in size.

Cons: The generated code is more complex, and the number of classes is larger in the case of multiple-class pattern and extended multiple-class patterns. The generated code pattern is more complex. Developers, particularly who need to validate the generated code, will be faced with a number of different code patterns.

Alternative 3: Allow the developer to choose, or control, the type of design pattern to be adopted.

This alternative implies that Umple user would be able to include a directive to control which design pattern to be used for code generation.

Pros: maximum flexibility is given to the user to choose which design alternative to adopt.

Cons: This alternative shifts the burden to the developer to decide on the most appropriate design alternative. This also increases the complexity in the language, and the underlying Umple platform.

Decision Point 2: Handling of events

Contributes to goal 6 and goal 7

A state machine responds to the occurrence of events that are typically, but not always, outside of the context of the state machine itself. The events that Umple state machine responds to are implemented as public functions that can be called by any component of the system. The functions return a Boolean value; true if the event has resulted in transition, and false otherwise. This implementation results in maximum flexibility, as those public functions can then be easily encapsulated into functions that can implement additional event types.

Decision Point 3: Core state machine behavior

Contributes to goals 5, 6, and 7

Unlike most code generated from the surveyed modeling tools, and even though we envision Umple users to never edit the generated code, Umple generates code that resembles hand-written code. We distributed the core state machine behavior for each event handler function. The event handler function uses a switch statement on the current active states, and determines the appropriate behavior.

Each transition requires the following steps:

- Check for guard conditions
- Execute exit action(s). There may be multiple exit actions for nested states.
- Execute transition action.
- Execute entry action(s). There may be multiple entry actions for nested states.

To hide such details, we encapsulated these actions within a function, similar to the approach adopted in the *Telelogic tau* modeling tool [43].

Decision Point 4: Implementation of composite state machines

Contributes to goals 1, 2, 3, 4, 7, and 8.

Composite state machines are state machines with nested states or concurrent regions. Umple supports nested states without introducing additional keywords. Umple uses the syntax of nested curly brackets to define nested states. For concurrent regions, Umple uses the symbol ||.

For implementation of the code generation for composite states, Umple uses a novel methodology. Traditional code generation from composite state machines results in generated code that is exponentially large, harder to read, understand and maintain.

The syntax, semantics, and code generation for composite state machines are the topic of the next two chapters.

3.5 State machine reuse and mixins

Contributes to goal 8.

Umple supports an unbounded number of state machines in every class, each of which can be defined independently. The same event in Umple can trigger transitions in one or more state machines. Simple functions defining guards and actions can be reused across a number of state machines, or across classes and components, and again the definitions of these can be defined independently, allowing mixing in of different sets to explore different requirements.

The following simple example illustrates a simple traffic control system, where the pedestrian light is dependent on, or controlled by, another state machine controlling the car traffic. For conciseness, we illustrate only partial models.

```
class trafficLightSystem {
  carTraffic {
    Red {
      entry / {goingRed();}
      after(redTimer)[!emergency] -> Yellow;
      emergencyNotice -> AllRed;
    }
  pedestrianTraffic
    DontWalk {
      goingRed [!emergency] -> Walk;
      emergencyNotice -> DontWalk;
} }
```

In this example, the event *emergencyNotice* triggers a *transition* in two separate state machines in the same class. Similarly, the guard *emergency* is used in two transitions in two state machines. The example also shows how an action in one state machine, *goingRed*( ), can function as an event and trigger a transition in an another state machine.

We have so far presented one aspect of reuse and mixins in Umple, where more than one state machine can reuse elements and behave interdependently. We now illustrate another aspect, where complete state machines are reused and customized.

A traffic light's basic operation is timer-based transitions from three states, Red, Green, and Yellow. This simple and basic model can initially be implemented as a stand-alone state machine, and later incorporated into various classes:

For simplicity, we continue to present partial models.

```
Statemachine coreTrafficController {
  Red {
    After(redTimer) -> Green;
    After(greenTimer) -> Yellow;
    After(yellowTimer) -> Red;
} }
```

In systems where a basic traffic light is desired, the previous standalone state machine can be referenced as follows :

```
class TrafficLightController {
  simpleController as coreTrafficController;
}
```

This example creates a state machine called *simpleController* that behaves identically to the *coreTrafficController* state machine.

Some traffic lights may have additional states, like flashing red, or flashing yellow, that are not part of the basic traffic light behavior. Let's call this type of traffic light *FrFy* for short. Adding such a feature can be accomplished as follows:

```
Class TrafficLightController {
  FrFy as coreTrafficController {
    Red {
    + midnightHour -> FlashingRed; }

    FlashingRed {
     morningHour -> Red;
} }
```

The previous example illustrates a scenario of adding to a basic state machine. The next example illustrates removing an existing element of a state machine.

Let's assume now we are modeling a traffic light for a high way entrance, and that the light is either Red or Green. We call this traffic light *H-way* for short.

```
class TrafficLightController {
  H-way as coreTrafficController {
    - After(greenTimer) -> Yellow;
} }
```

This example illustrates a scenario where a transition is removed from the model.

The process of modeling controllers may reveal a number of reusable state machines. These reusable state machines can then be refined and used as we described above. The outline view of the Umple editor (discussed in Section 3.7.1 Umple textual on page 70) facilitates the discovery of such reusable state machines.

3.6 State machine timers

Umple state machines support two types of timers. *After* timers and *afterEvery* timers. The *after* timer fires an event to trigger a transition after a specified amount of time. On the other hand, *afterEvery* timer fires an event on a specified intervals to trigger a transition. The following is an example describing timers in Umple.

```
class Timer {
  boolean G = true;

  status {
    S1 {
      after(5) -> S2;
    }
    S2 {
      afterEvery(5) [G] -> S1;
    }
  }
}
```

In this example, while the state machine status is in S1, and after 5 seconds, a transition to S2 is triggered. This timer expires only once, and if for any reason a transition does not occur (if there is a guard that evaluates to false), the timer is not restarted.

In the same example, while in S2, and after every 5 seconds, a transition to S1 is triggered, subject to the guard. This timer is restarted automatically every 5 seconds. The concept is that the state machine will keep trying until the guard becomes true.

The implementation of this timer behavior uses the *timerTask* in Java. For this example, Umple defines two helper variables as follows:

```
//Helper Variables

  private TimedEventHandler timeoutS1ToS2Handler;
  private TimedEventHandler timeoutS2ToS1Handler;
```

The event handling method is similar to any normal transition. The event name given to this transition timeout <name of the source state> <name of the destination state>.

```
public boolean timeoutS1ToS2()
  {
    boolean wasEventProcessed = false;

    Status aStatus = status;
    switch (aStatus)
    {
      case S1:
        exitStatus();
        setStatus(Status.S2);
        wasEventProcessed = true;
        break;
    }
```

Since states may have other outgoing transitions, it is required to stop timers whenever we exit states with active timers. The following method is called whenever state S1 or S2 is exited.

```
private void exitStatus()
  {
    switch(status)
    {
      case S1:
        stopTimeoutS1ToS2Handler();
        break;
      case S2:
        stopTimeoutS2ToS1Handler();
        break;
    }
  }
```

Similarly, any transition into either S1 or S2 should start the timer.

```
private void setStatus(Status aStatus)
  {
    status = aStatus;

    // entry actions and do activities
    switch(status)
    {
      case S1:
        startTimeoutS1ToS2Handler();
        break;
      case S2:
        startTimeoutS2ToS1Handler();
        break;
    }
  }
```

3.7 Umple textual editor and automated update site

Contributes to goals 4, 5, and 8

In order to enhance Umple adoption and increase the pool of available participants for our grounded theory study, we need to enhance Umple editors. The challenge is that the Umple system and language are under continuous development and modifications. The approach for the textual editor has to accommodate this aspect of Umple. An Umple textual editor has to be tightly related, and at the same time loosely coupled, with the underlying Umple components. This allows us to quickly refactor changes in the Umple language and bring them to the editor, and at the same time, not depend on the editor for any change in Umple.

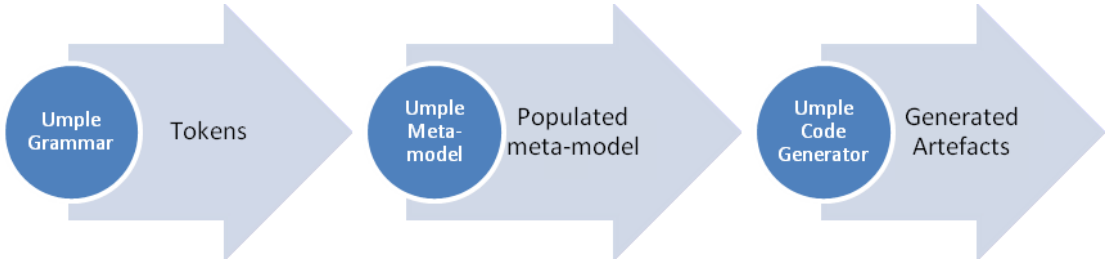


Figure 13: Umple high-level system components

Figure 13 illustrates Umple components that relates to the editors. As we show in the next two sections, the Umple textual editor relies solely on Umple Grammar and Umple Meta-model respectively.

3.7.1 Umple textual editor

We have built an Umple textual editor based on Xtext technology [44]. Xtext is a language development platform that supports the development of general purpose programming languages and domain specific languages. We have identified Xtext to be a suitable approach to implement an Eclipse-based Umple textual editor for the following reasons:

1. Xtext is open source.
2. The Xtext based editor becomes tightly related to Umple grammar. This means that to reflect any change in the Umple grammar requires only straightforward changes to the corresponding Xtext Umple grammar. As future work it is planned to be able to generate one from the other.
3. We can easily extend the editor to limit side effects, where the developer may gain access to aspects of the generated code that he is not supposed to; for example, a transition action that may update the value of the state machine.
4. Most importantly, Xtext is built on standard technologies, like Java and Antlr [42]. Building on standard technologies simplifies maintenance.

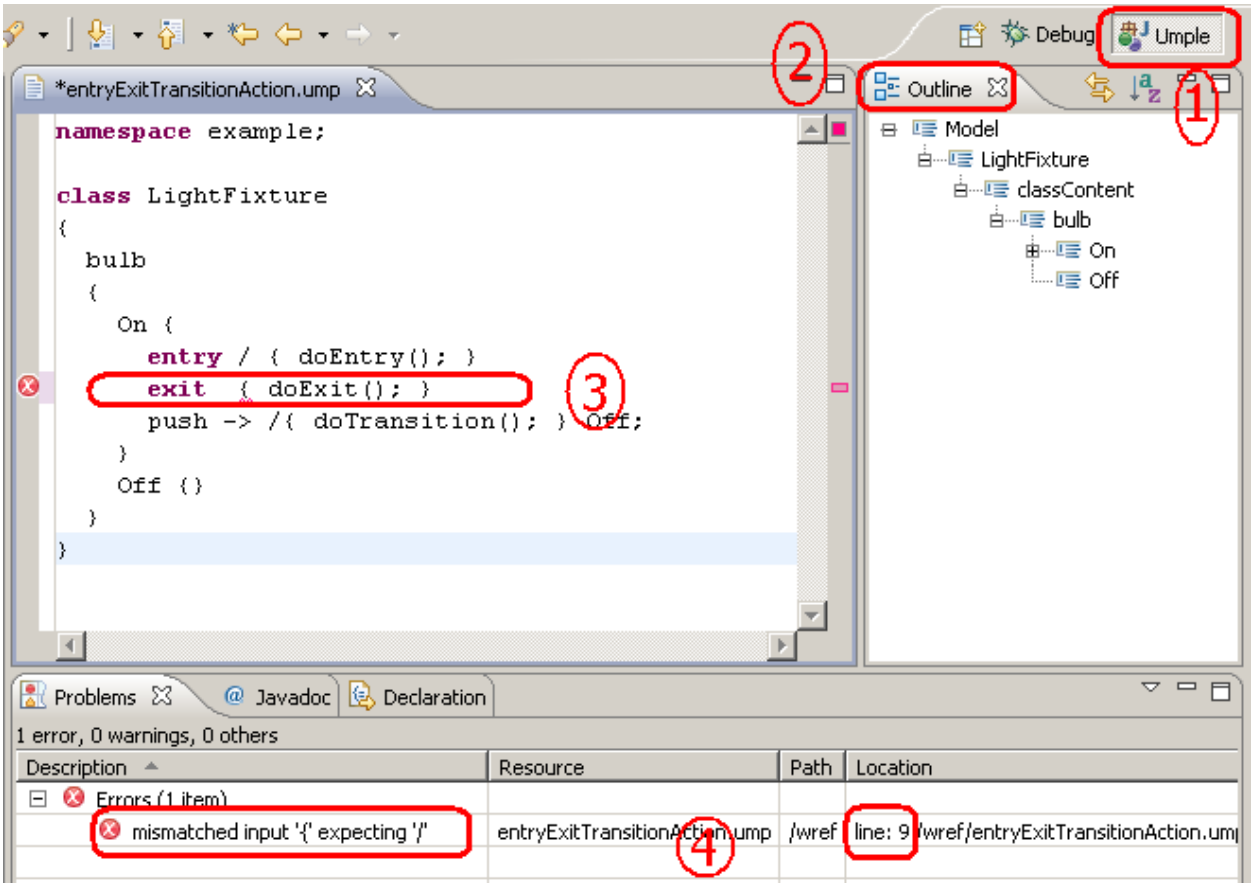


Figure 14: Umple textual Editor

Figure 14 illustrates some of the features in the Umple textual editor.

1. Umple perspective. Clicking on the Umple perspective opens the layout that is most appropriate for Umple development.

2. Outline view shows the elements of the Umple model organized in a hierarchy. The elements shown correspond to both Class and state machine modeling elements. Optionally, the developer can choose to show the native code hierarchy.

3. Sophisticated error recovery in the Umple textual editor. In this case, the model is missing the forward slash before the action code. The editor identifies the syntactic error, and quickly recovers and continues parsing at *best guess* in the next token.

4. Error messages with expected tokens.

3.7.2 Automated update site

As the number of Umple contributors and users are expanding, we implemented a mechanism whereby users running Umple plug-in get notified automatically whenever there is a newer

Umple release. We achieved this by implementing an automated update site. When a new version is released, the Umple Eclipse plug-in notices the server version is newer than the local version, and prompts the user to automatically download and install the newer version.

3.8 Summary

In this chapter, we introduced the syntax and semantics of simple state machines in Umple. We introduced the Umple grammar, and the meta-model. We compared Umple's meta-model to the latest UML state machine meta-model. We discussed the major design decisions we took, such as the code generation pattern used and the approach to represent state machine model elements textually in Umple. Our decisions were largely driven by a number of goals, which themselves were derived from the vision for the Umple technology. Umple design goals are:

1. Minimize the use of reserved words.
2. Keep the syntax concise and extendable.
3. Umple syntax should look and feel like high level programming languages.
4. Eliminate the need to edit the generated code.
5. The generated code should be efficient.
6. The generated code should look like code written by hand.
7. Exploit textual modeling potential.

We demonstrated how Umple supports reuse and mixing in of state machines. Finally, we presented the Umple textual editor similar to editors available to other high level programming languages, like auto-complete, code-assist, outline and error views. The Umple update site enables Eclipse users to update their Umple compiler whenever a newer release is available.

Chapter 4: Syntax and semantics of composite state machines

The objective of this chapter is to explore the complexities brought about by UML composite states and to outline the syntax and semantics of nesting and concurrency concepts. We highlight some of the outstanding issues and demonstrate Umple's approach in handling such issues. We use UML 2.4 beta II specifications [23] as our reference (the latest published at the time of writing). However, and as we demonstrate in this chapter, Umple is not just another implementation for UML specifications. Umple does deviate from the standard when we find objective justifications. Such deviations are not uncommon, many modeling and code generation tools adopt different code generation styles, and occasionally, their own implementation flavor of the semantics.

In addition, we explore the undefined semantics of UML composite state machines, and show how some of such semantics can be unambiguously defined in Umple. UML specifications do not specify code generation patterns. Umple, in this area, draws from related work, and existing modeling tools in weighing the options. Umple's approach in handing code generation from composite state machine is novel. The approach avoids explosion of the generated code and maximizes reuse of simple state machine semantics.

It is the topic of the next chapter to illustrate how such semantics are implemented in code generation. The next chapter presents a modified flattening approach for code generation, and demonstrates how the semantics issues discussed in this chapter are implemented.

This chapter is a deep investigation of the UML specifications that relate to composite states. We assume the reader is well familiar with the basic semantics of state machine presented in Chapter 3: Syntax and semantics of simple state machines.

4.1 Syntax of Composite state machines

Encapsulation of state machines enables the modeling of complex behavior concisely. Every composite state machine can be flattened in one or more simple state machines. The real power in composite state machines is conciseness. Our objective therefore is to enable the textual modeling of composite state machines in a way that maintains or enhances on this conciseness. "The concept of hierarchical state machine is a true blessing only if it is easy enough to implement in a main stream programming language" [45]. The grammar for simple state machines was presented in the previous chapter (Chapter 3: Syntax and semantics of simple state machines). For the purpose of this chapter, we only present the grammar for composite state

machines. We start by presenting the syntax for nested state machines, and then present the syntax for concurrent state machines.

Nesting of state machines is defined recursively. As shown in grammar rules R7 and R8, a state has a state entity. A state entity may itself contain a state. This enables the syntax to define unlimited levels of nesting of states.

Concurrency is defined using the symbol ||. When a state entity is ||, Umple understands that the next state to be defined is concurrent.

R7	**state :** [stateName] { ([[**comment**]] | [=changeType:-|*]? [[**stateEntity**]])* }
R8	**stateEntity- :** [=-||] | [[**transition**]] | [[**entryOrExitAction**]] | [[**activity**]] | [[**state**]]

Umple uses nested brackets to represent nesting levels. The example below defines *stateA2* to be a substate of *stateA1*, which is itself is a substate of *stateA*.

```
stateA {
  stateA1 {
    stateA2 {
    }
  }
}
```

The following shows concurrent states.

```
state A {
  stateB { }
  ||
  stateC { }
}
```

In UML terminology, *stateB* and *StateC* are two concurrent regions of state A.

More examples are presented in the next chapter.

4.2 Semantics of composite states machines

The UML 2.4 Beta II specifications [23] leave significant room for undefined semantics (known unknowns). More interestingly are the unstated undefined semantics (unknown unknowns). As we tread over the semantics of composite states, we carefully expose these two aspects of UML state machines and show how Umple addresses them.

We start by exploring composite state semantics by using the example in Figure 15 as a playground to lay out our analysis of the semantics. The example is comprised of seven states, one nesting level, two concurrent regions, and 11 transitions. For simplicity's sake, the example does not include any actions, guards, or activities, but our analysis can easily extend to include such elements.

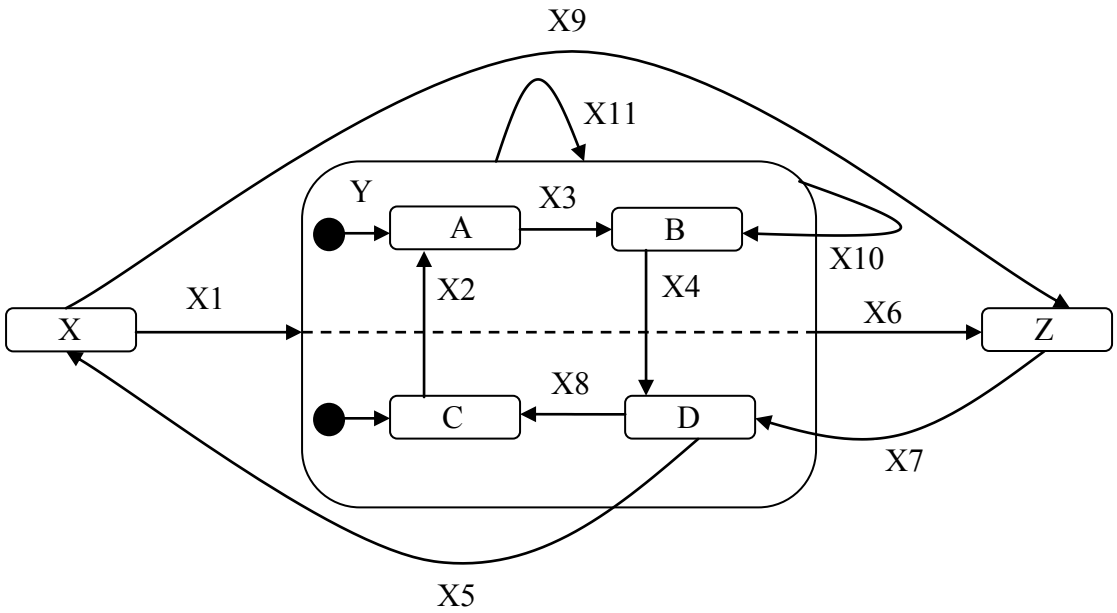


Figure 15: Exploring the semantics of state machines

We now analyze the transitions in Figure 15 one by one in more depth.

X1:
The state machine exits the source simple state 'X' and enters the destination composite state 'Y'. Instantaneously, the state machine is in state 'A' and state 'C'.

Using UML specification terminology, this is a '*default entry'* into the '*containing'* composite state Y (page 570 in the UML specifications).

X2:
This is an invalid transition. The state 'Y' is a concurrent state, and the two regions must at all times stay active.

We determined that this is an invalid transition, despite the fact that UML specifications can be interpreted in a way to make such transitions valid; a transition that crosses the boundaries of

concurrent regions forces exits of all regions and re-entry of all regions (page 591 in the UML specifications).

X3:
This is a simple transition from state 'A' to state 'B'.

X4:
This is an invalid transition, similar to transition X2.

X5:
The state machine exits the source state 'D' and also exits both regions of state 'Y', and enters the simple destination state X.

X6:
This exits state Y, and any substates, and enter the simple destination state 'Z'. The UML specifications call this a '*higher level transition'*.

According to the UML specifications, a higher level transition with a target outside a composite state forces exits of all substates and regions. But if the target is within the composite state, then no exits are forced. But what if the target is in a different region (like X2 and X4 above)? UML does not rule out the validity of such transitions as discussed earlier.

X5 and X6, despite their apparent similarity, bear significant semantics differences. X5 can only be triggered while the state machine is state 'D', while X6 can be triggered while at any state combinations of the inner states of 'Y'.

Another issue with X6 is the question of which region is exited first? Imagine each of the two regions has exit action A and exit action B. Which exit action is called first? UML specifications specify that exiting the regions has to occur first before updating the state machine active state (Page 571 in the UML specifications). But if the state machine is being executed in a single-thread environment, the need to define which region is exited first becomes necessary.

Due to the linear nature of text, Umple will exit the region whose definition comes first in the linear text. If the developer would like another behavior, he can simply alter the sequence in the Umple source. This is one aspect where the linear nature of text clears potential ambiguity in the visual model.

X7:
Enter state 'D', and instantaneously, enter state 'A'. In UML 2.4 terminology, this is a transition to a *direct substate*. The UML specifications calls this *explicit entry* (page 570 in the UML

specifications), as opposed to *implicit entry* in the case of X1, where the transition into state A and state C are implicit.

X8:
Similar to X3, this triggers a transition from 'D' to 'C'.

X9:
This is a simple transition between two simple states within a composite state machine.

X10:
This transition triggers exiting all inner states of 'Y', exiting the state 'Y' itself, and then entering state B. Instantaneously, the state machine also enters state 'C'.

This is an undefined semantics (under specification) in UML 2.4 Beta II specification. The specifications do not mention the semantics of this transition.

X11:
This exits all inner states of 'Y', exits 'Y' itself, and then enters states 'A' and 'C'.
Similar to transition X10, this is an undefined transition in UML 2.4 specifications.

4.3 Final States

"A final state is a special kind of state signifying that the enclosing region is completed" (UML 2.4 Beta II specifications page 547). When all regions in a state machine reach a final state, then it means that the entire state machine is *completed*.

According to the UML specifications [23] (page 547 on version 2.3), a final state has the following constraints: 1. No outgoing transitions; 2. has no regions; 3. has no reference to a sub machine; 4. has no entry behavior; 5. has no exit behavior; 6. has no do activity behavior. Umple interprets a completion of a state machine to mean deletion of the object. An Umple class can contain multiple state machines. A completion of any state machine in the class will delete the entire object.

Similarly, in a composite state machine, completion of a region implies the completion of the entire state machine, and object deletion follows. The UML specifications state that completion of a region does not mean the completion of the entire state machine. This is an area where Umple semantics differs from UML specification. We made the decision to delete the object when a region is completed for the following reasons:

1. This makes the behavior of completion in the case of multiple state machines in the same class work the same as a state machine with concurrent regions.

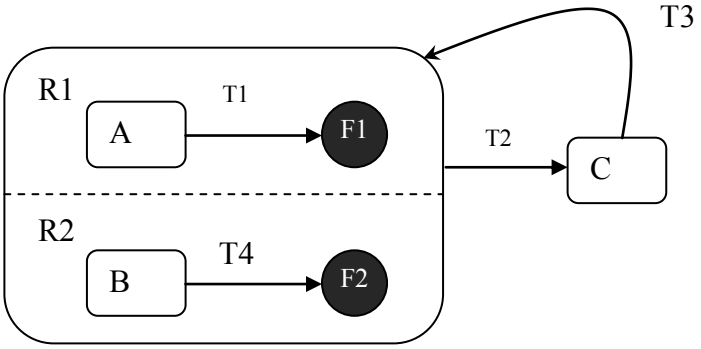
2. Supporting the UML alternative requires the introduction of the notion of *partial completion,* which adds complexity that, we submit, will usually not be needed. Partial completion is the concept that one region has completed, while one or more of its concurrent regions have not yet completed. It would have been necessary to track which regions have reached partial completeness, so that if all of them reach partial completeness then the state machine as a whole can become complete, and the object can be deleted. But there are many other complexities: For example, if a transition is taken out of a state with concurrent regions, one or more of which are partially complete, then the partial completeness status would need to be cancelled. But upon returning to 'history' this would need to be restored.

3. Any behavior supported in the UML preferred semantics can be supported by using Umple's End states in the following manner: Imagine there is the intent to transition to final when End states *s1* and *s2* in two concurrent regions are *both* reached. The entry action in such end states can set variable *end1*, *and2* to true and trigger event *s1s2final*. Then there can be a transition *s1s2final* to 'final' from the surrounding state machine, guarded by [end1 && end2].

We illustrate Umple syntax and semantics of final states in the following three cases.

4.3.1 Case 1: Final states in regions

Figure 16 illustrates a composite state machine with two regions. Each region has a *Final* state. Note that the keyword *Final* is case sensitive.



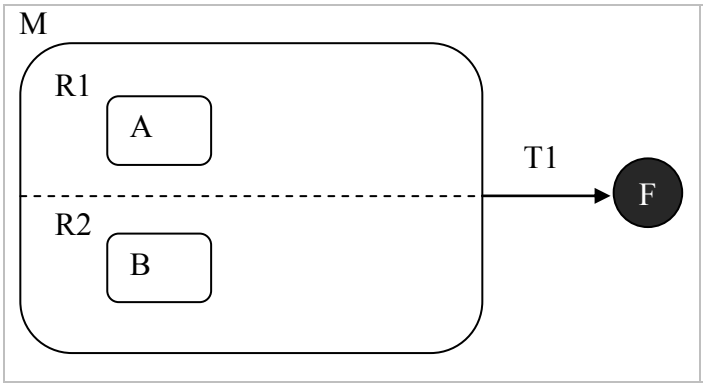

```
class FinalState {
  stateMachine {
    M {
      T2 -> C;
      R1 {
        A {
          T1 -> Final;
        } }
      ||
      R2 {
        B {
          T4 -> Final;
        }
      }
    }
    C {T3 -> M}
  }
}
```

Figure 16: Final states in regions

When T1 occurs, the state machine becomes complete and Umple deletes the object. The UML alternate interpretation would mean that the state machine would have been partially complete since R2 would have been still active. In that case, the state machine would still have been able to respond to T4 and T2.

4.3.2 Case 2: Transition from a composite state to a simple Final state

Figure 17 illustrates a transition from a composite state to a *Final* state.



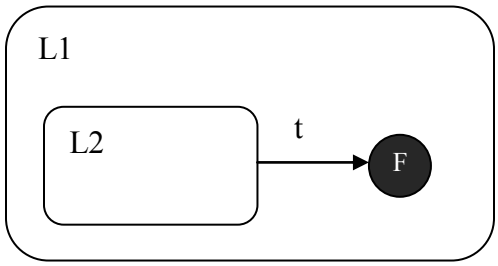

```
class FinalState {
  stateMachine {
    M {
      T1 -> Final;
      R1 {
        A {}
      }
      ||
      R2 {
        B {   }
} } } }
```

Figure 17: Transition from a composite state to a Final state

When T1 occurs, Both R1 and R2 becomes instantaneously inactive. The state machine reaches a *Final* state and the state machine becomes *completed*. Object deletion follows.

4.3.3 Case 3: Final state in nested configuration

Figure 18 illustrates final states in a nested configuration.




```
class FinalState {
  stateMachine {
    L1 {
      L2 {
        t -> Final;
      }
    }
  }
}
```

Figure 18: Final state in nested configuration

When the state machine is in L2 *state*, and *T* occurs, the transition to the Final state takes place and the whole state machine becomes *completed*.
In such a nested configuration, the exit action of L2 and L1 is called prior to object deletion.

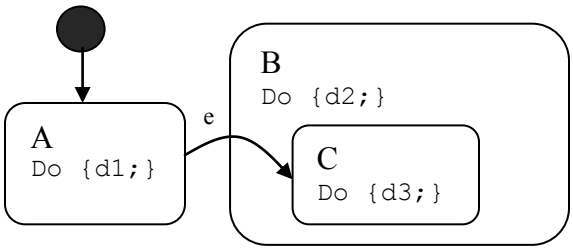
4.4 Do Activities

UML specifies the do activity to be the execution of a behavior that takes place while in a specific state. The execution of the thread representing the do activity starts when the state is entered following the execution of the entry action of that state, if such an action exists. If the state is exited before the do activity is completed, the do activity is aborted prior to its completion.

In Umple, any state can have an associated do activity. We demonstrate the behavior of do activities in Umple using three cases; 1) Nested configuration, 2) Concurrent configuration, 3) A configuration where a single event triggering more than one transition in two separate state machines within the same class.

4.4.1 Case 1: Do activity in nested configuration

This case demonstrates nested states with two do activities at two different levels.



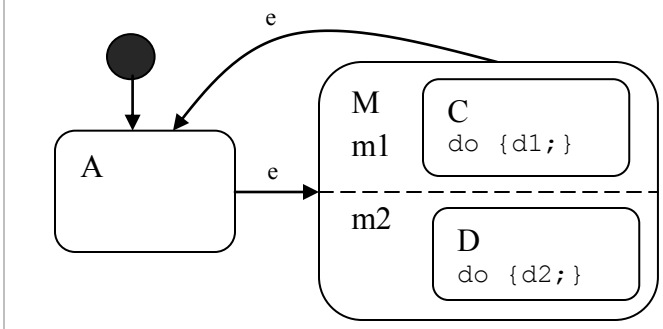

```
class DoActivity {
  stateMachine {
    A {
      e -> C ;
      do {d1;}
    }
    B {
      do {d2;}
      C { do {d3;} }
    }
  }
}
```

Figure 19: Case 1: Do activity in nested configuration

The state machine starts in state A. At this state, Umple creates and executes the thread d1. When the event e occurs, the thread d1 is stopped and the transition to the inner state C takes place. Upon this transition, Umple creates two threads, one for d2 and one for d3.

4.4.2 Case 2: Do activities in concurrent configuration

This case demonstrates a concurrent state machine with two do activities executing in parallel.




```
class DoActivity {
  stateMachine {
    A {
      e -> M ;
    }
    M {
      e -> A ;
      m1 {
        C { do {d1;} }
      }
      ||
      m2 {
        D { do {d2;} }
} } } }
```

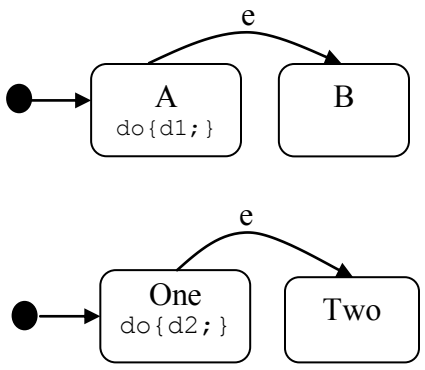
Figure 20: Case 2: Do activities in concurrent configuration

In this case, the state machine starts in state A. When the event e occurs, the transition to the concurrent state M takes place, and an implicit transition into both C and D is fired. This is because in both regions, m1 and m2, C and D are the start states by default. Once the state machine enters C and D, both threads d1 and d2 start executing.

If e occurs again while the concurrent state M is active, a higher-level transition to state A takes place, exiting both states C and D. Upon this transition, both threads d1 and d2 are stopped.

4.4.3 Case 3: Do activities in Multiple state machines within the same class

Umple's support for multiple state machines in the same class enhances the simplicity with which a developer can model parallel behavior. Using concurrent state machines can be simulated by using multiple state machines in the same class. The example below demonstrates this use case.




```
class DoActivity {
  stateMachineOne {
    A {
      e -> B ;
      do { d1; }
    }
    B { }
  }
  stateMachineTwo {
    One {
      e -> Two ;
      do { d2; }
    }
    Two { }
} }
```

Figure 21: Case 3: Do activities in Multiple state machines within the same class

This Umple class contains two state machines, *stateMachineOne* and *stateMachineTwo*. The event e triggers two transitions in the two separate state machines from state A to state B in the first state machine, and from state One to state Two in the second state machine. These two transitions result in the stopping of the two do activities, d1 and d2.

4.5 Outstanding issues

Our investigation of the latest UML composite states specifications uncovered a number of outstanding issues. Some of these issues are known and stated in the UML specifications, others are not mentioned.

4.5.1 A higher level transition to composite states with regions without start state

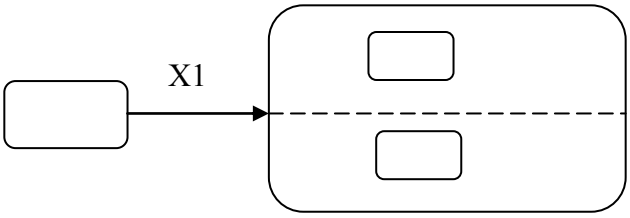


Figure 22: A higher level transition to a composite state

Consider the higher-level transition X1 in Figure 22. What is the semantics of such transition?

The UML specifications discuss two alternate interpretations (page 566 in the UML specifications). One interpretation is that such a model is invalid. The second interpretation is that this is a valid model, and that the state machine enters the composite states, but does not enter any of the substates. The UML specifications do not prefer either interpretation.

However, this model becomes more problematic if one of the two regions happened to have a start state. If such is a valid model, then what is the resulting state?

Umple resolves such ambiguities by implicitly making the first state the start state. Hence, the transition X1 implicitly enters the start state in the two regions.

4.5.2 Conflicting transitions

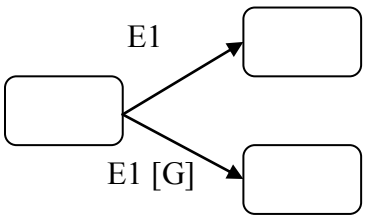


Figure 23: Conflicting transitions

A conflicting-transitions situation occurs when the same event fires two different transitions (pages 581, 582, and 583 in the UML specifications). This can occur in unguarded transitions, or in guarded transitions when the guard value is true (Figure 23). Conflicting transitions result in a non-deterministic state machine.

The UML specification states that the state machine in such situation can choose a subset of those transitions to fire; however, the sequence of the firing is not straightforward. Some of the conflicting transitions are resolved by complex algorithms. For example, the innermost transition always has a priority. But what if you have regions, which one has a priority then?

Umple's linear nature resolves such ambiguity. The transition that comes first in the linear text is always chosen first. This approach makes the state machine deterministic.

4.5.3 Forks and Joins with actions and guards

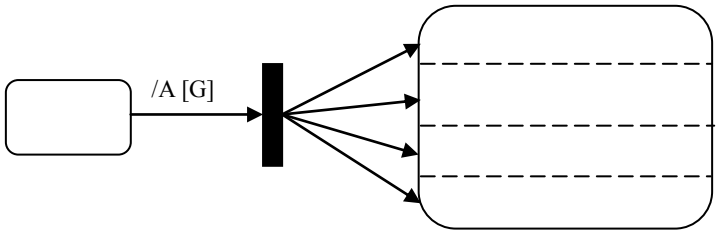


Figure 24: Fork with actions and guards

UML state machine forks and joins cannot have guards or actions associated with them. Figure 24 is therefore an invalid UML model (Constraint 1 on page 589 in the UML specifications). Umple's forks and joins can have guards and actions. The guard functions in a way identical to a guard on a simple transition (i.e. if the guard evaluates to false, none of the forks transitions takes place). A fork action is executed before the transition takes place, and a join action is executed after the exit actions of all substates and regions are executed.

4.5.4 *Partial Forks and Joins*

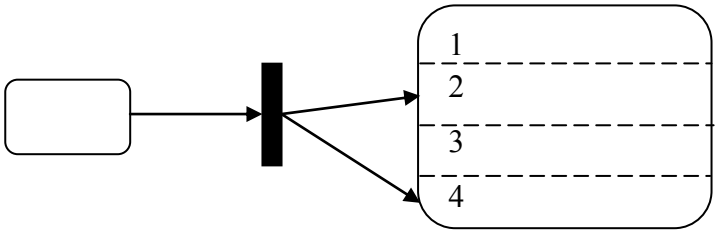


Figure 25: Partial fork

UML specifies that regions 2 and 4 are entered explicitly (Figure 25). The remaining two regions (region 1 and 3) are entered implicitly. UML does not specify any semantic difference between the explicit and implicit entries (page 571 in the UML specifications). Semantically, this is identical to a higher level transition to the boundary of the composite state machine (i.e., identical to X1 in Figure 15). There is a semantic difference only if the transition is pointing to an inner state in region 2 and 4 that is not the start state. In situations where there is a transition to two or more different inner states, with none being a default start state, support for partial transitions makes semantic sense. Umple, however, does not currently support such a case.

4.5.5 *Event processing in concurrent states*

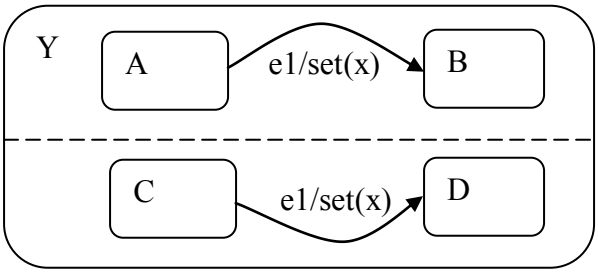


Figure 26: Event processing in concurrent regions

The same event cannot trigger two transitions; except if the transitions are in two separate regions. Consider the event 'e1' in Figure 26. There is a need to unambiguously determine the firing sequence of the event e1.

The UML specifications include a transition selection algorithm (TSA) that resolves most, but not all, conflicting transitions (we refer to semantics section on page 581 and transition selection algorithm on page 583). The TSA assigns priorities to transitions based on their relative nesting; the highest priority is given to the inner most state in the active state (in Figure 26, state Y is the

active state). This algorithm works well for transitions that are at different nesting levels, but does not address the transitions similar to those in Figure 26. Umple gives higher priority to the region defined first in the linear nature of its textual notation.

4.6 Large State Machine Example

We have so far presented relatively simple state machine examples. In "Chapter 7: Experimentation" we also experiment with relatively small models. Here, we illustrate how larger and more complex state machines can be effectively represented textually in Umple.

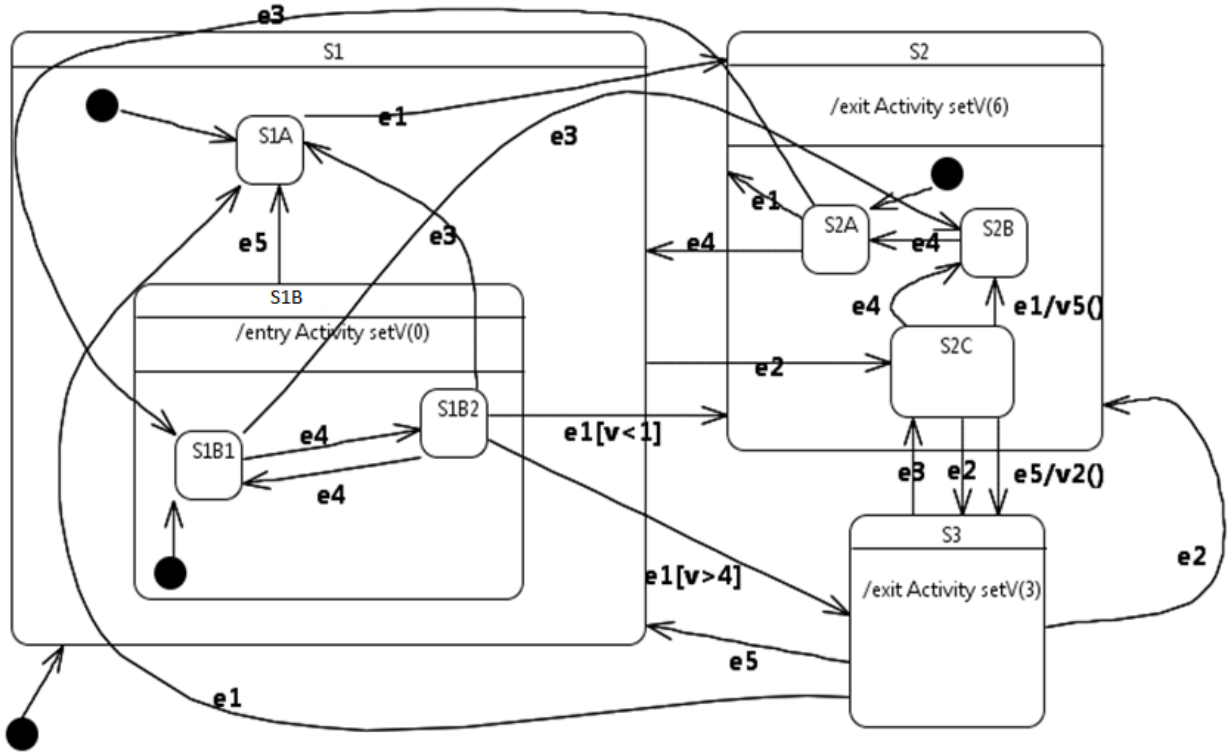


Figure 27: Complex state machine model

Consider the example in Figure 27 when more action code and guard conditions are inserted in the visual representation, and model elements have more expressive naming. The image can quickly become too cumbersome to maintain. Another consideration is model refinements and edits; as the model grows, additional model modifications entail increasing effort to adjust the model layout and spacing.

Software engineers spend a considerable amount of their time modifying and maintaining models [46]. One would expect that model maintenance to grow at a higher-than-linear rate as model size increases. This is because increasing effort is needed to rearrange and position increasing

numbers of model elements. We claim that Umple handles modifications more effectively especially for larger models. Some aspects of this claim is investigated in “Chapter 7: Experimentation”.

The equivalent model is illustrated in Listing 5 below using Umple notation.

```
class StateMachineTest {

  Integer v = 0;
  status {
    S1 {
      e2 -> S2C;
      S1A {
        e1 -> S2;
      }
      S1B {
        entry /{setV(0);}
        e5 -> S1A;
        S1B1 {
          e3 -> S2B;
          e4 -> S1B2;
        }
        S1B2 {
          e1 [v>4] -> S3;
          e1 [v<1] -> S2;
          e3 -> S1A;
          e4 -> S1B1;
        }
      }
    }
    S2 {
      exit / {setV(6);}
      S2A {
        e3-> S1B1;
        e1-> S2;
        e4 -> S1;
      }
      S2B {
        e4 -> S2A;
      }
      S2C {
        e1 -> / {setV(5);} S2B;
        e2 -> S3;
        e5 -> / {setV(2);} S3;
        e4 -> S2B;
      }
    }
    S3 {
      exit / {setV(3);}
      e1 -> S1A;
      e2 -> S2;
      e3 -> S2C;
      e5 -> S1;
} }
```

Listing 4 : Complex state machine model

In the visual representation of this model, we found it difficult to use fully expressive event names and we had to minimize the use of actions and guards to keep elements from overlapping. Using Umple notation, it was relatively more effective to use full naming, actions and guards. Textual features such as refactoring and 'find-and-replace' were handy in implementing such changes.

4.7 Test Driven Development

The Umple platform and tools are developed using a Test Driven approach [47] which provides for confidence that new development in Umple does not result in regression defects. The test Driven Development (TDD) approach adopted in the development of Umple is well explained in Forward's thesis [2]. In this section, we briefly describe the process, giving examples specific to the Umple state machine features. We also demonstrate how the TDD approach was instrumental in the development of the composite state machines.

4.7.1 Umple Testing Process

The Umple compiler starts by parsing the input Umple code into tokens. The tokens are then used to populate the meta-model, which is in turn used to drive a number of code generation templates to generate the target language code. The generated system can then itself be tested to make sure that Umple models generate code that behaves as expected. This testing process is summarized in Figure 28 below.

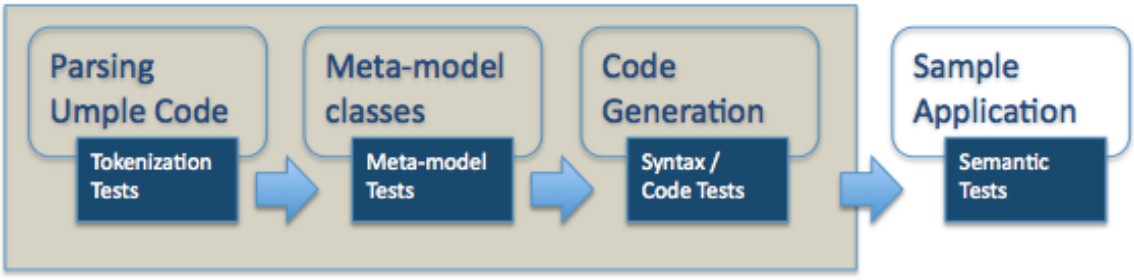


Figure 28: Testing Process [2]

4.7.2 Parsing Umple code into tokens

Consider the following simple state machine.

```
class LightFixture
{
  bulb
  {
    On {
      push -> On;
    }
  }
}
```

The parser analyzes the input text and identifies tokens. Parsing this simple state machine generates the following tokens:

```
[classDefinition][name:LightFixture][stateMachine]
[inlineStateMachine][name:bulb][state][stateName:On]
[transition][event:push][stateName:On]
```

There are 80 test cases covering the parsing for state machines ranging from very simple state machines to a larger more complex composite states. The complete listing of test cases is published as part of the Google code project and can be found at the following location:

`http://code.google.com/p/umple/source/browse/#svn/trunk/cruise.umple/test/cruise/umple/compiler/`

4.7.3 Meta-model tests

The objective of this group of test cases is to ensure that Umple maintains an accurate internal representation for the input model. The meta-model is tested to verify that an input model, after being correctly parsed, populates the right elements into an instance of the meta-model. For the simple example, we test the following are populated correctly:

- State machine name
- The number of states within a state machine.
- The first state name (start state name).
- The number of transitions.
- Events names.

Listing 5 illustrates the JUnit code that tests these aspects of the model.

```
UmpleClass c = model.getUmpleClass("LightFixture");
StateMachine sm = c.getStateMachine(0);
Assert.assertEquals("bulb", sm.getName());

Assert.assertEquals(1, sm.numberOfStates());
State state = sm.getState(0);
Assert.assertEquals("On", state.getName());

Assert.assertEquals(1, state.numberOfTransitions());
Transition t1 = state.getTransition(0);
Assert.assertEquals("push", t1.getEvent().getName());
```

Listing 5: Meta-model test

The number of meta-model test cases is similar to the number of the parser test cases. This is because for each model tested from a parsing perspective, is also tested from a meta-model population perspective.

4.7.4 Code generation tests

For each target language, we test to make sure that the generated code matches exactly our expectation. This can be done by writing by hand the expected generated code, and then testing to make sure that what is actually generated matches our expectations. For our sample model, if the target language is Java, the expected generated code is shown in Listing 6.

```
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE *Umple Version* modeling language!*/

public class LightFixture
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //LightFixture State Machines
  enum Bulb { On }
  private Bulb bulb;

  //------------------------
  // CONSTRUCTOR
  //------------------------

  public LightFixture()
  {
    setBulb(Bulb.On);
  }
  //------------------------
  // INTERFACE
  //------------------------
  public String getBulbFullName()
  {
    String answer = bulb.toString();
    return answer;
  }
  public Bulb getBulb()
  {
    return bulb;
  }
  public boolean push()
  {
    boolean wasEventProcessed = false;

    Bulb aBulb = bulb;
    switch (aBulb)
    {
      case On:
        setBulb(Bulb.On);
        wasEventProcessed = true;
        break;
    }
    return wasEventProcessed;
  }
  private void setBulb(Bulb aBulb)
  {
    bulb = aBulb;
  }

  public void delete()
  {}

}
```

Listing 6: Generated Java code

There are about 98 code generation test cases. All test cases follow the same pattern; test the expected or desired output to the real output and make sure both are identical. These test cases are published as part of the Umple Google Code project and can be found at the following location:

`http://code.google.com/p/umple/source/browse/#svn/trunk/cruise.umple/test/cruise/umple/statemachine/implementation/`

4.7.5 Generated-systems tests

The final stage of testing involves testing the behavior of systems generated by Umple. For example, we feed the compiler the model shown in Figure 27, and then test the generated system using a sequence of events, and make sure that the resulting state is the expected state.

Composite state machines are built by means of reusing the implementation of simple state machines (see Chapter 5: Implementation of composite state machines). This testing approach has enabled us to efficiently build the composite state machines in Umple with few regression defects.

Listing 7 illustrates a sample generated system test. The system shown below is fed as an input to Umple and tested is performed on the generated system

```
class GarageDoor
{
   status {
      Open {
        buttonOrObstacle -> Closing;  }

     Closing {
          buttonOrObstacle -> Opening;
          reachBottom -> Closed;
      }

     Closed {
       buttonOrObstacle -> Opening; }

     Opening {
          buttonOrObstacle -> HalfOpen;
          reachTop -> Open;
      }

     HalfOpen { buttonOrObstacle -> Opening; }
  }
}
```

Listing 7: Sample generated system test

This example is for a simple garage door system. The testing of the generated system is performed by feeding the system with a number of events, and checking whether the system is in the expected state or not. For example, the system can be fed the following events.

```
buttonOrObstacle

reachedButtom

buttonOrObstacle

reachTop
```

After these events, the expected state is Open. If the test case succeeds, we have more confidence that Umple generated systems work as expected. If such a test case fails, then we investigate where the failure took place.

4.8 Summary

Concurrent and nested state machines are the main topic of this chapter. We first introduced Umple syntax for these. We then presented the semantics of Umple's composite state machines, which were drawn to a large extent from the UML specifications. We also highlighted some of the outstanding issues that exist in the latest UML specifications and demonstrated when such inconsistencies occur. In some cases, Umple deviated from the UML specifications when there were convincing reasons. In other cases, Umple ironed out some of the undefined semantics.

Composite state machines tend to be larger and more complex than simple machines. We demonstrated how Umple's textual representation can effectively represent large and complex state machine models. We also demonstrated the test-driven development approach adopted in Umple.

Chapter 5: Implementation of composite state machines

This chapter focuses on code generation of composite state machine in Umple. As we explained in Chapter 4: Syntax and semantics of composite state machines, the code generation of composite state machine in Umple is novel. Umple uses a flattening approach termed Compress-Flatten Code Generation (CFCG). In this chapter, we demonstrate this approach, and compare it to other code generation approaches for composite state machines.

5.1 Convention

Throughout this chapter, we adopt a convention to help illustrate the CFCG process. In the following sections, we illustrate how Umple flattens and generates the implementation code. Umple's meta-model (see Figure 12: Umple meta-model on page 57) is unaware of composite states. The CFCG process therefore adds additional state machine elements to the meta-model to simulate the behavior of composite state machines, without adding additional complexity for the code generation templates. This approach allows us to make significant reuse of the implementation of simple state machines. For example, a region in Umple is defined internally as a full state machine. This approach is explained in detail in this chapter.

Clearly, this approach is not language specific. However, we use Java as a representative language. To distinguish between Java and Umple in this chapter, Java code will always appear in grey boxes.

The code generation implementation approach presented in this chapter aims at generating code for all possible, and valid, state and transition combinations, while maintaining the relatively concise size of the generated code. This approach is termed Compress-Flatten Code Generation (CFCG) summarized in Figure 29.

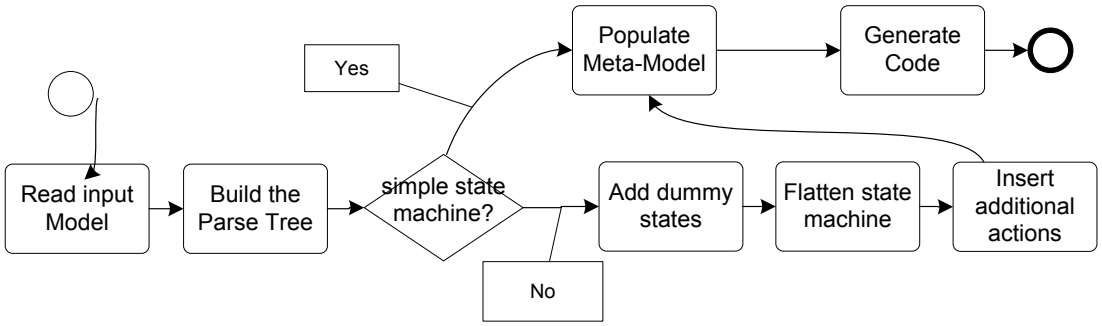


Figure 29: CFCG Process

The CFCG process works as follows. Umple reads the input model, and builds the parse tree. The process distinguishes between two types of state machine models; a state machine that has at least one nesting level, or one concurrent region, is considered a complex state machine. The rationale is that such state machines require additional processing (compression and flattening) in order to generate concise state machine code. The path for simple state machines is discussed in "Chapter 3: Syntax and semantics of simple state machines". Here, we limit our discussion on aspects of the CFCG process related to nested and concurrent states.

5.2 Composite state cases

We demonstrate the code generation of composite states by demonstrating a number of cases. A case is a composite state pattern. For example, a transition from an outer state to an inner state in a nested states environment is one case. Each of the following cases demonstrates one specific aspect of a composite state machine. For each case, we show 5 items as follows:

1. The top left quadrant shows the input model visually.
2. The top right quadrant shows the input Umple model.
3. The lower left quadrant shows the flatted state machines visually.
4. The lower right quadrant shows the algorithm adopted for code generation.
5. The bottom shows an excerpt of the generated code.

Note that for each case, only an excerpt of the generated code is presented. This is because the analysis of each case focuses on a specific aspect of code generation. Therefore, some questions may be left unanswered for some cases and should be cleared in the cases to follow.

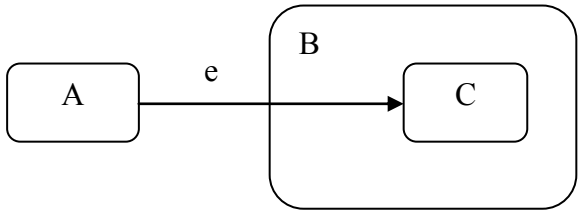
We use Java for the code generation language. But arguments in this chapter can easily be extended to any high level programming language.

For simplicity, the models illustrated in this chapter ignore all types of actions, guards, and do activities. The analysis, however, does address these model elements. Later in this chapter, we present expanded examples that include all types of actions.

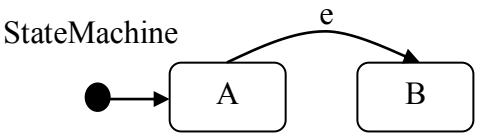
5.2.1 Case 1: Transition to an inner state

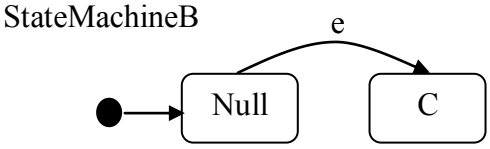
The first case we address is a transition to an inner state. In our example, the state machine starts in state A. When the event 'e' occurs, the transition to the inner state C takes place. This is equivalent to transition from state A to B, and then from state B to state C.

Any exit action(s) from state A are called first, then transition actions, followed by any entry actions into B, and finally, entry actions into C.




```
class ToInnerState {

  stateMachine {
    A {
      e -> C;
    }
    B {
      C {}
} } }
```





1. Flatten by generating *stateMachine* and *StateMachineB*.

2. Set *stateMachine* to A (the start state)

3. Set *stateMachineB* to Null (state B is not active)

4. When event e occurs:

- If state A is active, set *stateMachineB* to *stateMachineB.C*
- Return true to indicate the event was processed.

```
// Flattened state machines
enum StateMachine { A, B }
enum StateMachineB { Null, C }

// Construction
public ToInnerState(){
  setStateMachine(StateMachine.A);
  if (stateMachineB == null) { setStateMachineB(StateMachineB.Null); }
}

// Event prcoessing
public boolean e(){
  boolean wasEventProcessed = false;
  switch (stateMachine) {
    case A:
      setStateMachineB(StateMachineB.C);
      wasEventProcessed = true;
      break;
  }
}
```

Figure 30: Transition to an inner state

As shown in the abstracts of the generated code, Umple internally creates two state machines, the first state machine has two states, A and B. The second state machine is Null and C.

Upon construction the first state machine is set to state A, and the second state machine is updated to state Null. As a matter of fact, the state Null is used to indicate that *stateMachineB* is not active; i.e., the higher level state machine is in some other state than B (here it is in state A).

As with simple state machines, the event handler is generated as a public method. This method updates the state machine state by calling a private method *setStateMachineB( ).* This method encapsulates calls to any actions and do activities. This encapsulation is very important to our code generation approach for two reasons:

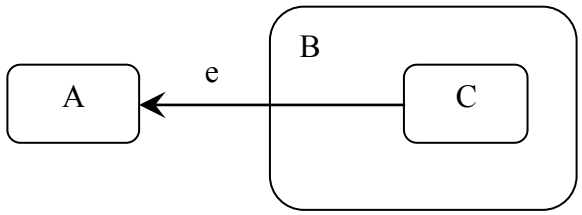
1. It makes all event processing methods relatively small in size; they become easier to read and understand.

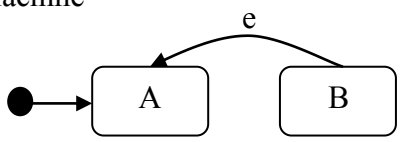
2. It simplifies the code generation patterns. All event processing methods look very similar, and can therefore use the same code generation template.

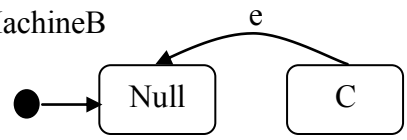
This state machine method is very simple: it encapsulates all method calls when transitioning from some state to another state. But also, this method allows for arbitrary complexity in the state machines the modeler can create; there are an unlimited number of combinations of source and destination states. For this reason, we will ignore the complexity of this method while we are discussing these code generation cases. The specifics of the code generation for this method are discussed in section 5.3 in this chapter.

5.2.2 Case 2: Transition from an inner state

This case is similar to the previous case except that the transition originates from an inner state to an outer state.

<table>
<tr>
<td> 
</td>
<td><pre>
class FromInnerState {

  stateMachine {
    A { }
    B {
      C {
        e -> A;
      }
} } }
</pre></td>
</tr>
<tr>
<td>stateMachine
 

StateMachineB
 
</td>
<td>1. Flatten by generating stateMachine and StateMachineB.

2. Set stateMachine to A (the start state)

3. Set stateMachineB to Null (state B is not active)

4. When event e occurs:

• If state C is active, set stateMachineB to Null.

• Set stateMachine to A.

• Return true to indicate the event was processed.</td>
</tr>
</table>

Figure 31: Transition from an inner state

```
// Flattened state machines
enum StateMachine { A, B }
enum StateMachineB { Null, C }


// Construction
  public FromInnerState()
  {
    setStateMachineB(StateMachineB.Null);
    setStateMachine(StateMachine.A);
  }

// Event processing
public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachineB) {
    case C:
      setStateMachine(StateMachine.A);
      wasEventProcessed = true;
      break;
  }
}
```

Continued Figure 31: Transition from an inner state

The code generation for this case is similar to the previous case, which is an objective we strive to maintain in Umple; similar state machines should have similar code generation patterns.

The difference here is in the event processing method. In response to the event '*e'*, and if the state machine is in state *C*, we update the state machine state to *A*. This is also encapsulated in a single method call *setStateMachine( )*.

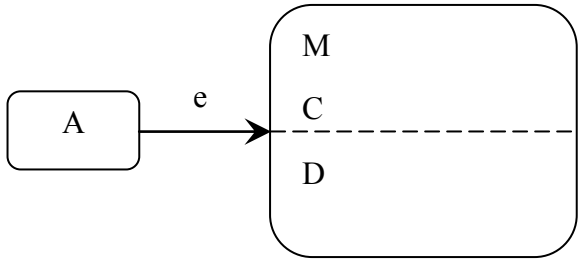
The coming cases entail regions and concurrency. In our implementation, we consider every region to be a full-fledged state machine; a region may have one or more state machine elements of any type, such as a start state, end states, ordinary states and transitions. This view of regions allows us to recursively define regions without having to define a new region element. This is similar to a nested state, where a state can itself contain a state (a substate).

5.2.3 Case 3: Transition to a concurrent state

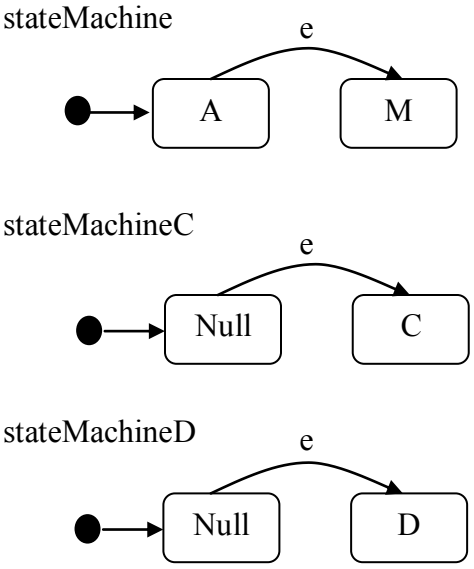
In this case (Figure 32), the state machine starts in *state A*. When the event '*e'* occurs, the transition from *state A* to the composite *state M* takes place. Instantaneously, the two regions C and D become active.

Umple creates internally three state machines; *StateMachine* that has two states, A and M; *StateMachineC* that has two states, null and C; and finally *StateMachineD* that has two states null and D.

Note that we use the dummy state null in a consistent manner. If a state machine is in state null, it means that the state machine is not active. In this case, if the state machine is in state A, then both regions C and D are set to *null*.




```
class ToConcurrentState {
  stateMachine {

    A {
      e -> M; }
    M {
      C {}
      ||
      D {}
} } }
```



1. Flatten by generating *stateMachine* and *StateMachineC* and *stateMachineD*.

2. Set *stateMachine* to A (the start state)

3. Set *stateMachineC* to Null.

4. Set *stateMachineD* to Null.

4. When event e occurs:

- Set *stateMachine* to M.
- Set *stateMachineC* to C
- Set *stateMachineD* to D.
- Return true to indicate the event was processed.

Figure 32: Transition to a concurrent state

```
// Flattened state machines
enum StateMachine { A, M }
enum StateMachineC { Null, C }
enum StateMachineD { Null, D }

// Construction
public ToConcurrentState() {
  setStateMachineC(StateMachineC.Null);
  setStateMachineD(StateMachineD.Null);
  setStateMachine(StateMachine.A);
}
// Event prcoessing
public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachine) {
    case A:
      setStateMachine(StateMachine.M);
      wasEventProcessed = true;
      break;
  }
  return wasEventProcessed;
}
```

Continued Figure 32: Transition to a concurrent state

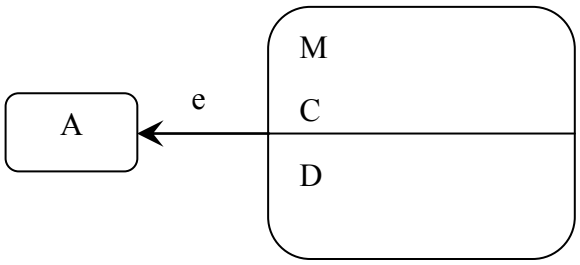
At construction, the state machine is set to state A. The two other state machines (*stateMachineC* and *stateMachineD*) are set to state null.

When the event '*e'* occurs, the state machine becomes in state M. The method *setStateMachine(stateMachine.M)* updates the states for the two regions C and D and calls entry and exit actions, if any.

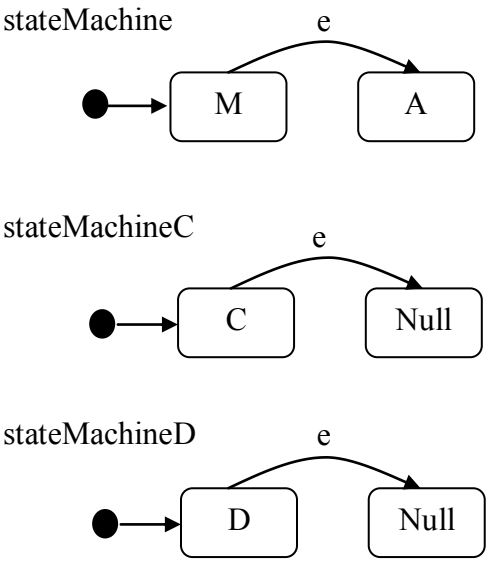
Notice the level of similarity between event processing methods in the previous cases, even though the transition is of a different nature. This similarity was achieved by means of hiding the transition details in a single method call.

5.2.4 Case 4: Transition from a concurrent state

This case demonstrates a scenario when a transition out of a composite state is taking place. In this example, the state machine starts in *state M*, which has two concurrent regions, *C* and *D*. The event '*e'* triggers a transition out of the composite state.




```
class FromConcurrentState {
  stateMachine {
    M {
      e -> A;
      C {}
      ||
      D {}
    }
    A {}
} }
```



1. Flatten by generating *stateMachine* and *StateMachineC* and *stateMachineD*.

2. Set *stateMachine* to M (the start state)

3. Set *stateMachineC* to C.

4. Set *stateMachineD* to D.

4. When event e occurs:

- Set *stateMachine* to A.
- Set *stateMachineC* to *Null*
- Set *stateMachineD* to *Null*.
- Return true to indicate the event was processed.

Figure 33: Transition from a concurrent state

```
// exiting a composite state

public boolean exitM() {
  boolean wasEventProcessed = false;
  switch (stateMachineC) {
    case C:
      setStateMachineC(StateMachineC.Null);
      wasEventProcessed = true;
      break;
  }

  switch (stateMachineD) {
    case D:
      setStateMachineD(StateMachineD.Null);
      wasEventProcessed = true;
      break;
  }
}
```

Continued Figure 33: Transition from a concurrent state

When exiting a simple state, a single switch statement suffices. In our case, a concurrent state with two regions requires two switch statements. The first switch statement checks if the region C is active, and if so, updates the state machine to null using the method *setStateMachineC,* which also handles any exit actions. The second switch statement performs the same steps for region D.

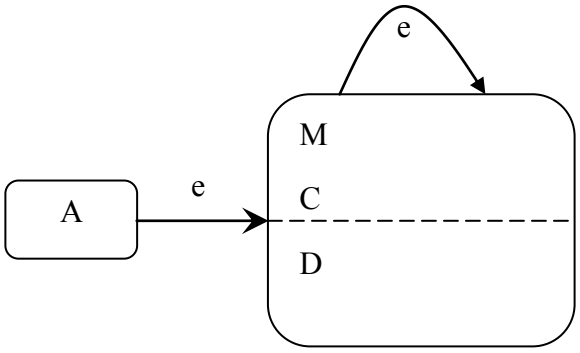
5.2.5 Case 5: Reflexive transition of a concurrent state

This case focuses on the implementation of a reflexive transition. A reflexive transition is just another transition whose source state and destination state are the same.

A reflexive transition of a composite state with two concurrent regions behaves as follows:

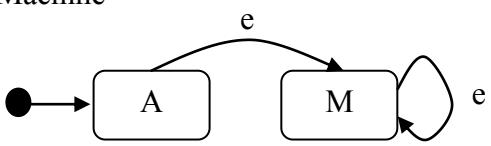
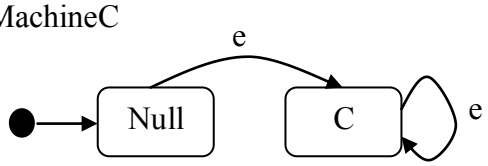
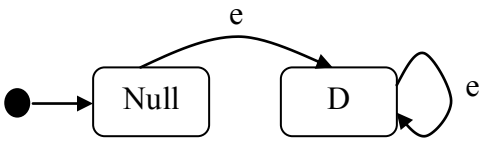
1. Call exit actions associated with any state being exited, including the composite state itself. Starting with the innermost state and working your way outward.
2. Exit all regions of the concurrent state;
3. Call transition actions, if any;
4. Re-enter the concurrent state;
5. Re-enter each concurrent region;
6. Call entry actions of any state being entered including the composite state itself.

According to the state machine semantics, exiting both regions takes place at *the same time*. However, if you are executing the state machine in a single threaded environment, one region will be exited before the other. Due to the sequential nature of the Umple textual notation, Umple determines that the region that is declared first will be exited first. To override such behavior, one can simply re-order the regions so that region D is declared before region C. The same applies for entering a concurrent region in step 5 above.




```
class Reflexive {
  stateMachine {
    A {
      e -> M;
    }

    M {
      e -> M;
      C {}
      ||
      D {}
} } }
```

Figure 34: Reflexive transition of a concurrent state

stateMachine  stateMachineC  stateMachineD  	1. Flatten by generating *stateMachine* and *StateMachineC* and *stateMachineD*. 2. Set *stateMachine* to A (the start state) 3. Set *stateMachineC* to Null. 4. Set *stateMachineD* to Null. 5. When event e occurs: • Set stateMachine to M, set stateMachineC to C, and set stateMachineD to D. 6. When event e occurs (triggering the reflexive transition): • Call *exitStateMachine*() method, which exits all regions of M and exits M itself. • Set *stateMachine* to M (re-entering composite state) • Set *stateMachineC* to *C* • Set *stateMachineD* to *D*. • Return true to indicate the event was processed.

Continued Figure 34: Reflexive transition of a concurrent

```
// Reflexive transition of a concurrent state

public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachine) {
    case A:
      setStateMachine(StateMachine.M);
      wasEventProcessed = true;
      break;

    case M:
      exitStateMachine();
      setStateMachine(StateMachine.M);
      wasEventProcessed = true;
      break;
  }
  return wasEventProcessed;
}
```

Continued Figure 34: Reflexive transition of a concurrent state

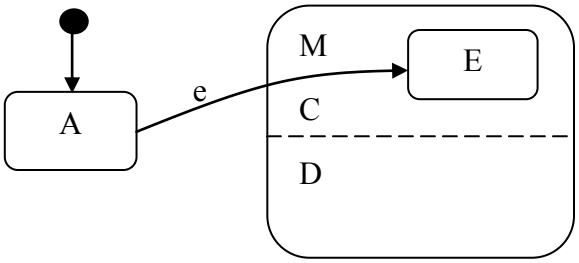
Note the switch statement in the generated code. The first case handles the behavior when the state machine is in state A. The second case handles the situation when the state machine is in state M. Our focus here is on the second case. The following takes place:

1. Calling the method *exitStateMachine( )* which encapsulates the logistics of exiting all regions.
2. Re-entering the state M by calling the method *setStateMachine(StateMachine.M( ))*
3. Updating the Boolean variable to indicate that the event was processed

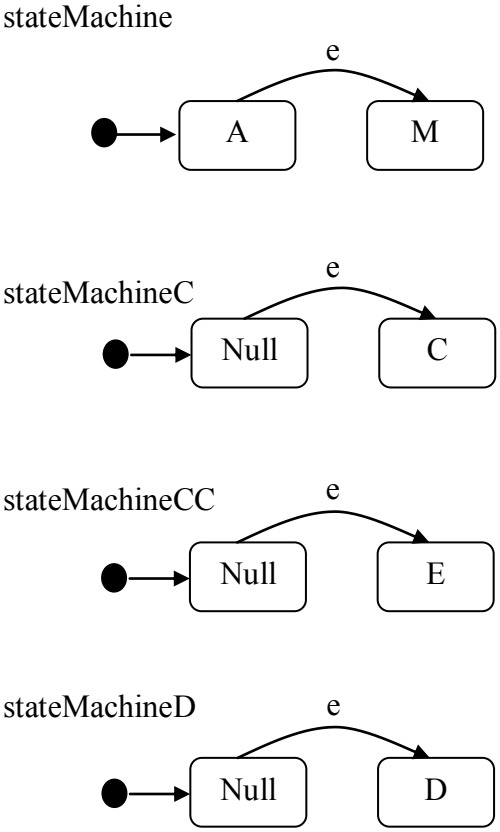
5.2.6 Case 6: Transition into an inner state in a concurrent region

This case explores a scenario when a transition to an inner state which lies inside a concurrent region. This case is special because even though the transition explicitly enters one region, the second region must also be activated.

In our example below, the state machine is initially in state A. When the event ‘e’ occurs, the state machine instantaneously enters the concurrent state M and also instantaneously enters state E. In that situation, the state machine is in state M, and in state E. Both regions C and D are active.




```
class ToConcurrentState {
  stateMachine {
    A {
      e -> E;
    }
    M {
      C {
        E { entry/{inside_E;} }
      }
      ||
      D {}
} } }
```



1. Flatten by generating *stateMachine* and *StateMachineC, stateMachineCC,* and *stateMachineD*.

2. Set *stateMachine* to A (the start state)

3. Set *stateMachineC* to Null.

4. setStateMachineCC to Null.

5. Set *stateMachineD* to Null.

5. When event e occurs:

- Set stateMachine to M.
- Set stateMachineC to C.
- Set stateMachineCC to E.
- Set stateMachineD to D.
- Return true to indicate the event was processed.

Figure 35: Transition to an inner state in a concurrent

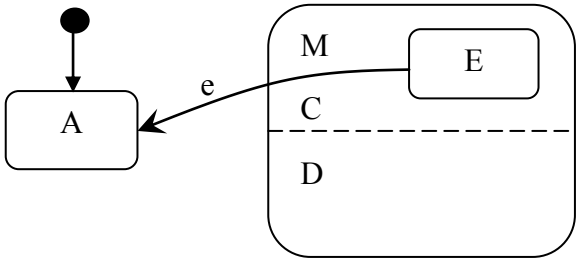
```
// Flattened state machines
enum StateMachine { A, M }
enum StateMachineC { Null, C }
enum StateMachineCC { Null, E }
enum StateMachineD { Null, D }

// Event processing
public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachine) {
    case A:
        setStateMachineCC(StateMachineCC.E);
        wasEventProcessed = true;
        break;
    }
  return wasEventProcessed;
}


private void setStateMachineCC(StateMachineCC aStateMachineCC) {
  stateMachineCC = aStateMachineCC;
  if (stateMachineC != StateMachineC.C && aStateMachineCC !=
  StateMachineCC.Null) { setStateMachineC(StateMachineC.C); }

  // entry action
  switch(stateMachineCC) {
    case E:
        inside_E;
        break;
    }
  }
}
```

Continued Figure 35: Transition to an inner state in a concurrent region

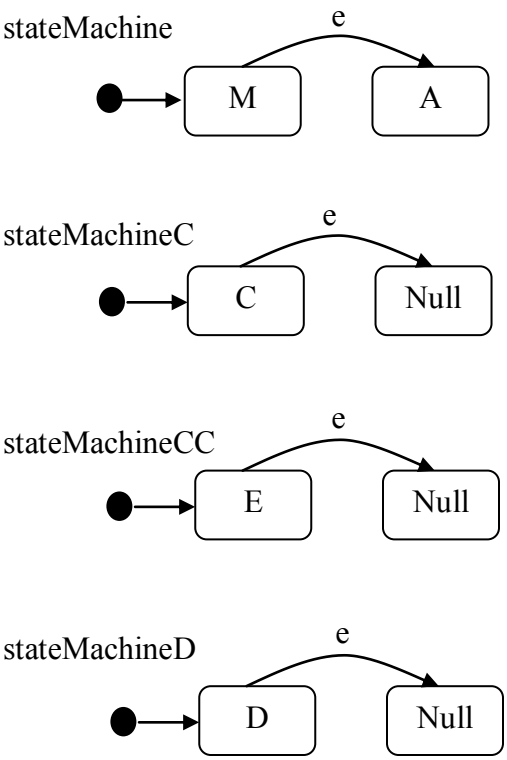
This case results in four internal state machines as shown in the generated code above. Notice how the event processing method is very similar to other cases. This is because the public method 'e' delegates to the method *setStateMachineCC* that calls the entry action and updates the state machine's states.

5.2.7 Case 7: Transition from an inner state of a concurrent region

This case is similar to the previous case. When the state machine is in the M state, and the event 'e' occurs, the transition from the inner state E, which lies in the concurrent region C, takes place.




```
class FromConcurrentState {
  stateMachine {

    M {
      C {
        E {
          e -> A;
        }
      }
      ||
      D {}
    }
    A {}
} }
```



1. Flatten by generating *stateMachine* and *StateMachineC, stateMachineCC,* and *stateMachineD*.

2. Set *stateMachine* to M (the start state)

3. Set *stateMachineC* to C.

4. setStateMachineCC to E.

5. Set *stateMachineD* to D.

5. When event e occurs:

- Call the method exitStateMachine() which handles exiting substates and calling actions.
- Set stateMachine to A.
- Return true to indicate the event was processed.

Figure 36: Transition from an inner state of a concurrent

```
// Event processing

public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachine) {
    case M:
      exitStateMachine();
      setStateMachine(StateMachine.A);
      wasEventProcessed = true;
      break;
    }
  return wasEventProcessed;
}
```

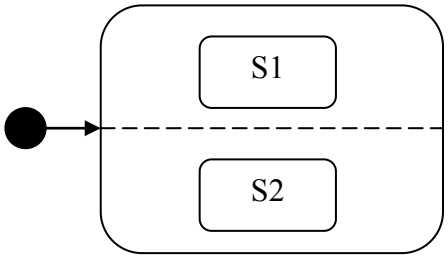
Continued Figure 36: Transition from an inner state of a concurrent region

5.2.8 Case 8: Concurrent state is the start state

This case shows a situation when the state machine start state is a concurrent state. This is a controversial model. We discussed this controversy in the previous chapter in the section "A higher level transition to composite states with regions without start state" on page 82. The execution semantics of such a model can be interpreted in one of three ways;

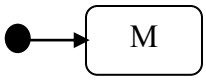
1. The model is invalid and Umple should throw a syntactic error.
2. The state machine becomes in state M, and enters the two concurrent regions, and enters states S1 and S2.
3. The state machine becomes in state M, but does not enter any of the states in the concurrent regions.

In Umple we adopt alternative 2, following the rule that when in a region, you must always be in a substate of that region.

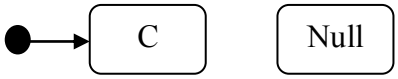



```
class ConcurrentState {
  stateMachine {

    M {
      C {
        S1 {}
      }
      ||
      D {
        S2 {}
} } } }
```

stateMachine



stateMachineC



stateMachineCC

stateMachineD

stateMachineDD

1. Flatten by generating *stateMachine* and *StateMachineC, stateMachineCC, stateMachineD*, and *stateMachineDD*.

2. Set *stateMachine* to M

3. Set *stateMachineC* to C.

4. setStateMachineCC to S1.

5. Set *stateMachineD* to D.

5. Set stateMachineDD to S2.

```
// Construction
    setStateMachineC(StateMachineC.Null);
    setStateMachineCC(StateMachineCC.Null);
    setStateMachineD(StateMachineD.Null);
    setStateMachineDD(StateMachineDD.Null);
    setStateMachine(StateMachine.M);
```

Figure 37: Concurrent state is the start state

As shown, the constructor initiates the state machine M and sets the start state for the two regions.

5.3 State transition method

As we have demonstrated in the previous code generation cases, there are many variations of state transitions. The following are the characteristics of such variations:

1. Is the source state a simple state or composite state?
2. If the source state is composite, is it nested or concurrent?
3. Are there any states being exited that have exit actions associated with them?
4. Does the transition have any transition action associated with it?
5. Is the destination state a simple state or a composite state?
6. Are there any entry actions associated with any state being entered?

The answers to the questions above demonstrate some of the complexity inherent in implementing transitions. Even though Umple's philosophy states that a software developer need not to look at or modify the generated code, we strived to make the generated code simple and easy to understand.

It turns out that simpler code generation is also easier to implement. If we are able to make event processing functions look similar, we will be able to use simpler code generation templates to implement them.

We were able to achieve this simplicity by abstracting common processing elements in any event processing method and encapsulating the details in other methods (typically private methods) that are called internally. It is worth mentioning that the abstraction process was achieved incrementally by means of trial and error. As we were adding additional features into the state machine, we hit roadblocks of highly complicated code generation templates. Rather than struggling with complicated code generation templates, we tried to take a few steps back, and reconsider the implementation of code generation. Encapsulation of details worked well in many situations. We will demonstrate by drilling down in the state transition function of two of the cases described before.

To demonstrate the complexity of implementing a transition, and how Umple handles this complexity, we will reuse two of the cases presented earlier in this chapter. For this analysis, we assume that all transitions have both a guard G and an action A associated with them. We also assume that every state has an entry and exit action.

5.3.1 *Entering a composite state*

This analysis is based on a modified state machine in case 3 above. The modified Umple model looks as follows

```
class ToConcurrentState {
  stateMachine {

    A {
      e [G] -> /{transition_action();} M; }
    M {
      entry/ {entering_M;}
      C { cState {entry/ {entering_C();} } }
      ||
      D {  dState {entry/ {entering_D();} } }
} } }
```

Listing 8: Entering a composite state

This model adds a guard, and two entry actions. The code that implements the transition from A to M is as follows:

Step1: Public function to handle the event processing

```
public boolean e() {
  boolean wasEventProcessed = false;
  switch (stateMachine){
    case A:
      if (G) {
        transition_action;
        setStateMachine(StateMachine.M);
        wasEventProcessed = true;
      }
      break;
  }
  return wasEventProcessed;
}
```

Listing 9: Step 1

The public method is named after the event name. In this case, the public method is named 'e'. This method returns a Boolean value to indicate whether the event has been processed or not. Checking for the guard takes place within this method (as highlighted above). The method also calls the transition action right after checking for the value of the guard. The method then delegates the rest of the transition execution to *setStateMachine(StateMachine.M)*.

Step 2: setStateMachine(StateMachine.M)

This is a generic method that is used to update the state of any state machine.

```
private void setStateMachine(StateMachine aStateMachine) {
  stateMachine = aStateMachine;
  // entry actions
  switch(stateMachine) {
    case M:
      entering_M;
      if (stateMachineC == StateMachineC.Null) {
        setStateMachineC(StateMachineC.C); }
      if (stateMachineD == StateMachineD.Null) {
        setStateMachineD(StateMachineD.D);
      }
      break;
   }
}
```

Listing 10: Step 2

This method will call any entry actions. In this case, *entering_M* is called.

We note here that the entry action is called prior to updating the state machine configurations (i.e prior to updating the state machine attributes). Therefore, if the entry action queries the state machine, inaccurate values will be returned.

Notice that initially, both regions' states are set to null (see Case 3: Transition to a concurrent state on page 99). This method checks if the region is in the null state, and if so, it will delegate to *setStateMachineC* and *setStateMachineD* respectively. For brevity, we only analyze *setStateMachineC*.

Step 3: setStateMachineC(StateMachineC.C)

```
private void setStateMachineC(StateMachineC aStateMachineC) {
  stateMachineC = aStateMachineC;
  if (stateMachine != StateMachine.M && aStateMachineC != StateMachineC.Null)
    { setStateMachine(StateMachine.M); }

  // entry actions

    switch(stateMachineC) {
      case C:
        if (stateMachineCC == StateMachineCC.Null) {
          setStateMachineCC(StateMachineCC.cState);
        }
        break;
    }
  }
}
```

Listing 11: Step 3

This method would call any entry actions. In this case, there are no entry actions associated with the *stateMachineC*. The method updates the state machine state to *cState* by means of delegation to *StateMachineCC.cState*.

Step 4: setStateMachineCC(StateMachineCC.cState)

```
private void setStateMachineCC(StateMachineCC aStateMachineCC) {
  // entry actions
  switch(stateMachineCC) {
    case cState:
      entering_C;
      break;
} } }
```

Listing 12: Step 4

This method finally calls the entry action for the *cState*.

5.3.2 Exiting a composite state

The steps for exiting a composite state machine are very similar to entering a composite state machine. Again, this similarity makes it easier to follow the generated code, and makes the code generation templates less complex. For brevity, we show the method for exiting the composite state M.

```
public boolean exitM() {
  boolean wasEventProcessed = false;
  switch (stateMachineC) {
    case C:
      exitStateMachineC();
      setStateMachineC(StateMachineC.Null);
      wasEventProcessed = true;
      break;
  }

  switch (stateMachineD)
  {
   ..
   ..
} }
```

Listing 13: Exiting the composite state

When exiting the composite state M, we also exit *stateMachineC* and *stateMachineD*. For brevity, we analyze the steps for exiting *stateMachineC*.

Again, we delegate to *exitStateMachineC* for the handling of exit actions, if any, and for updating the state machine state. Notice that when we exit the state machine, we set its state to null.

5.4 Code generation templates

Umple uses Java Emitter Templates (JET) technology to specify what the generated code should look like [48]. The JET templates are then compiled into Java code that generates the code in various languages, given an instance of the Umple Metamodel.

Each supported language in Umple has its own JET templates. For Java alone, there are 138 JET templates. The complete listing of Umple JET templates is part is available on the Umple Google Code project. The templates supporting Java is available at this location:

`http://code.google.com/p/umple/source/browse/#`*svn/trunk/UmpleToJava*

The following table summarizes key templates and briefly describes their function.

Table 10: Key code generation templates

	Template name (*.JET)	Function
1	*members_AllStateMachines*	Loops over all state machines and handles naming for state machine generated code.
2	*state_machine_Event*	Handles code for state machine events and events handling methods.
3	*state_machine_Event_StartStopTimer*	Outputs the method for starting and stopping timers for time-based events.
4	*state_machine_Events_All*	High level template that calls the state_machine_Event template.
5	*state_machine_IsFinal*	Handles code for final states.
7	*state_machine_SetSimple*	Handles the code for setting simple state machines.
8	*state_machine_Set_All*	High level template that loops over all state machines. For simple states the template calls state_machine_setSimple, and calls state_machine_Set.jet otherwise.
9	*state_machine_doActivity*	Handles code for do activities.
10	*state_machine_doActivityThread*	Handles the generated code for threading in Java.
11	*state_machine_doActivity_All*	A high level template for handling do activities.
12	*state_machine_timedEvent_All*	High level template for handling timed events.

5.5 Multiple state machines in the same class

An Umple class may contain an unbounded number of state machines. Those state machines may interact with each other in a number of ways. The following Umple model (Listing 14) illustrates two examples of such interactions.

In this example, the class Phone has three state machines; *ringerSound*, *screenLight* and *Vibration*. Initially, the ringer, the screen light and vibration are *Off*. When a call is received, the ringer sounds, the light turns on, and the vibration starts vibrating. The model abstracts some of the remaining common phone functionality.

```
class Phone {

  Integer t_ringer;
  Integer t_light;

  ringerSound {
    Off{
      callReceived -> On ;
    }

    On{
      silentButton -> Off ;
      pickUp -> / {setVibration(Vibration.Off);} Off ;
      rejectCall -> / {turnOffVibration();} Off ;
      after(t_ringer) -> Off ;
    }
  }

  screenLight {
    Off{
      callReceived -> On ;
    }

    On{
      callReceived -> / {resetTimer();} On ;
      after(t_light) -> Dimmed;
    }

    Dimmed{
      callReceived -> On ;
      after(t_light) -> Off;
    }
  }

  vibration {
    Off {
      callReceived -> On ;
      }

    On{
      turnOffVibration -> Off ;
} } }
```

Listing 14: Phone state machine

There is a difference between the semantics of multiple state machines in the same class, and concurrent regions in a composite state machine. In a concurrent state machine, the two regions are executing in parallel, while in a multiple state machine in the same class, the state machines are executing in sequence. The main benefit of supporting multiple state machines within the

same class is to allow every state machine to handle one aspect of the behaviour of the object. This approach helps in separation of concerns and can enhance the usability of mixins and state machine inheritance.

5.5.1 Single event causing multiple transitions

Within a single state machine, an event can at most cause a single transition. However, and because an Umple class may have more than one state machine, a single event may actually trigger a transition in more than one state machine.

In Listing 14, the *callReceived* event may cause a transition in the three state machines within the class *Phone*. Umple recognizes this special event, and groups all the behavior to implement the event handling into a single method (Listing 15).

```
public boolean callReceived() {
  boolean wasEventProcessed = false;
  switch (ringerSound) {
    case Off:
      ..
      break;
  }

  switch (screenLight) {
    case Off:
      ..
    case On:
      ..
    case Dimmed:
      ..
    }
  }

  switch (vibration){
    case Off:
      ..
    }
  return wasEventProcessed;
  }
}
```

Listing 15: Single event causing multiple transitions

5.5.2 Action in a state machine triggers an event of another state machine

An action within a state machine (entry, exit, or transition) can trigger an event that may cause a transition in another state machine. In the *ringerSound* state machine, when a call is rejected, a transition to Off is triggered. This transition calls an event of another state machine that results in another transition being triggered, a transition from *On* to *Off* in the *Vibration* state machine.

5.5.3 Action in a state machine updates the state of another state machine

An action within a state machine (entry, exit, or transition) can update the state of another state machine. Consider this transition in our example:

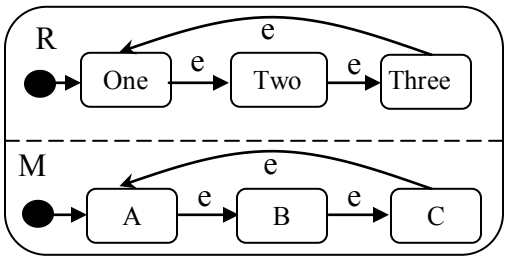
```
pickUp -> / {setVibration(Vibration.Off);} Off ;
```

When a call is picked up, a transition from *On* to *Off* takes place. The action on this transition updates the *Vibration* state machine to *Off*. This is commonly called a side effect of a transition; not a desirable feature of a state machine and developers must use it with care.

Notice that there is a difference between this action (*setVibration(Vibration.Off);)*) and (*turnoff Vibration();*). The first action updates the Vibration state machine without calling any entry, exit or transition actions within that state. However, the second action would result in execution of all involved actions in the transition. This feature enables the users to easily override a state machine behavior when needed.

5.6 Traditional flattening approach

The explosion phenomenon that occurs when flattening a composite state machine is explained here [49]. To briefly demonstrate this phenomenon, we present a modified example from Schaumont's book [50].




```
class TraditionalFlattening{

  status {
    R {
      One {
        e -> Two;
      }
      Two {
        e -> Three;
      }
      Three {
        e -> One;
      }
      ||
    M {
      A {
        e -> B;
      }
      B {
        e -> C;
      }
      C {
        e -> A;
      }
    }
  }
}
```

Figure 38: explosion phenomenon

This composite state machine can be in nine possible configurations (A and 1, A and 2, A and 3, B and 1, B and 2, B and 3, C and 1, C and 2, C and 3). Therefore, to flatten this state machine, the resulting simple state must have at least nine states (A1, A2, A3, B1, B2, B3, C1, C2, C3). If there was another region with another 3 states, the total number of flattened states jumps to 27 (3*3*3).

There are several research streams that are investigating the ability to generate code from state machines without the need for flattening the state machine to avoid un-scalable exponential growth in the generated code [51, 52] . However, these approaches typically ignore practical considerations for the generated code; as we explored in this chapter, one consideration for example is that similar state machine models should generate similar code.

In the case of 3*3 (Figure 38), using Umple results in eight states. Not a significant improvement over the standard flattening that results in nine states. But in the case of 3*3*3, the standard flattening results in 27 states, and Umple generates 12 states. Figure 39 summarizes the comparison for the number of generated states for Umple and the traditional flattening approach. The figure shows the number of generated states for the simple case of a state machine with 3

states, a concurrent state with 3*3 states, and up to a concurrent state machine with five concurrent regions with 3 states each.

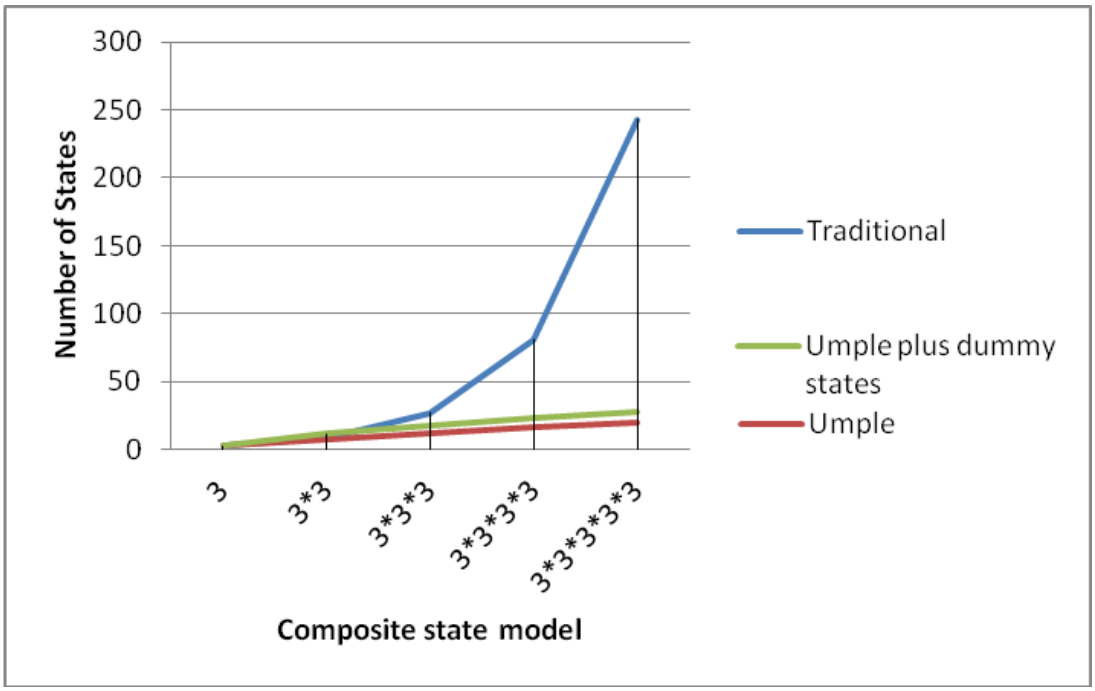


Figure 39: comparison of flattening approaches

As shown in the figure, the traditional flattening approach quickly outnumbers the number of states generated by Umple, even when the null dummy states are included.

5.7 Comparison of code generation approaches

In this section, we compare our CFCG code generation approach to that of a commercial tool (Rhapsody) and a research tool whose authors (Niaz et al) claim a novel approach of generating efficient and compact code for composite states.

Rhapsody implements state machines using the multiple-class pattern and creates objects that represents states upfront; i.e, as soon as the state machine becomes active. These objects stay in memory as long as the state machine is executing. Rhapsody uses a switch statement and a helper class to implement the state machine behavior. We discuss the pros and cons of multiple-class pattern in section " Multiple-class pattern" on page 35.

The research tool proposed by Niaz also uses multiple-class pattern where each state is implemented in a separate class. However, objects are not created upfront, rather, objects are created and deleted at run time. This makes the expected performance of this tool to be better than Rhapsody. Niaz's approach implements composite state machines by using object composition and delegation. In our comparison, we adopt a criteria similar to Niaz's [53] that

relies on the number of lines of code, number of bytes, and number of classes. For the base comparison, we consider the example in Figure 40.

In many cases, we were unable to compare our approach to other tools discussed in section "Code Generation from State Machines" on page 31 due to the fact that many of the available commercial and research tools do not support composite states in a way complete enough to allow this comparison. For example, Bridgepoint [28] does not allow substates or guards. Wasowski's approach[51] evaluates code generation for composite states with a focus on efficiency of the execution time of the generated code. Our focus is on the number of lines of the generated code.

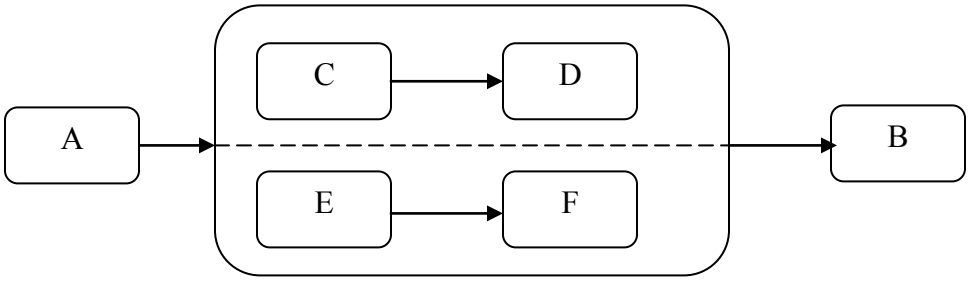


Figure 40: Composite state comparison example

The example is comprised of two simple states, and one state with two concurrent regions.

Table 11: code generation comparison

Generated code	Rhapsody	Niaz, I.A	Umple generated Code	Umple
Number of lines	675	250	125	8
Number of bytes	24,270	6,420	5,010	197
Number of classes	7	11	1	1

As shown, the number of lines of code is significantly lower in the case of Umple (reduction of about 50% as compared with Niaz`s approach). The number of bytes are less in the case of Umple (a reduction of about 22%).

5.7.1 Generated code growth analysis

The comparison in the previous section does not tell us how the generated code grows as the input model grows. We have conducted an estimate of the code generation by studying the generated code. We measured a factor of growth for every code section (a function of a code blocks) by analyzing how the code would grow when the number of states grows. For example,

an event is translated into a public event handler method (i.e one line of code). Two events are translated into two lines of code (a growth factor of 1).

This study results in a growth analysis summarized in Figure 41.

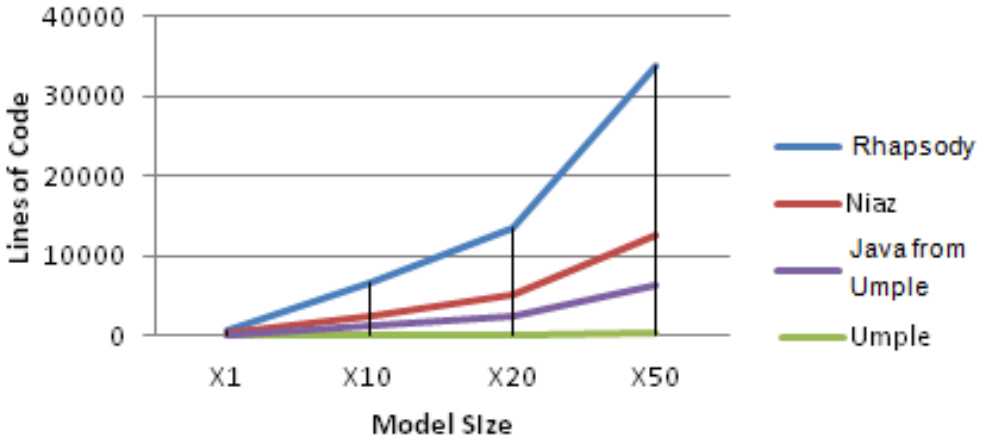


Figure 41: Factor of growth analysis

The study looks at models growth at a factor of 10, 20, and 50. The study compares Rhapsody, Niaz I.A, Umple generated Java, and Umple source models. This study implies that CFCG code generation approach results in significant reduction in code generation for larger models.

5.8 Summary

The majority of the modeling tools we surveyed did not handle code generation for composite state machines, maybe on the premise that any composite state machine model can be flattened into a simple state machine model. We quickly realized that we can further distinguish Umple by a careful analysis of all possible combinations of states and transitions. In the course of this analysis, we identified some undefined semantics in the UML specifications that we tried to handle in Umple.

We named our approach for code generation "Compress-Flatten Code Generation". This approach avoids explosions of composite state machines by internally creating dummy states and transitions. This chapter demonstrated this novel code generation approach by demonstrating a number of 'code generation cases'. Each case uncovers some aspects of the compress-flatten code generation technique.

We also presented, in great detail, how we implement the state transition method, and how Umple supports having an unbounded number of state machines in the same class. Finally, we compared Umple to two other modeling tools. This comparison indicates that Umple syntax is concise and tends to generate relatively fewer lines of code.

Chapter 6: A Grounded theory study of Umple

This chapter presents a grounded theory study to gather knowledge about perceptions of the usability of the Umple language. This is a long-running study that we conducted as the state machine capability in Umple was being developed. The objective is to gather user's feedback for existing aspects of Umple as they were being developed and use the findings to guide future development. Hence, a significant portion of the study addresses the work done prior to state machines being implemented in Umple, namely associations and attributes.

Grounded theory studies do not have hypotheses. Rather, the analysis of the data is expected to bring about theories about the domain being investigated. Our study does not have a hypothesis either. Our goal is that by studying Umple early adopters feedback, we can guide Umple's experimental development so that the resulting product can achieve the benefits claimed.

We start by first exploring the domain of using grounded theory (GT) studies in the area of software engineering. This survey helps us understand how GT have been used in the software engineering domain. We then present the grounded theory study of Umple users.

6.1 Survey of grounded theory in software engineering

Grounded theory (GT) is a systematic qualitative research methodology, originating in the social sciences, and emphasizes the generation of theory from qualitative data in the process of conducting research. Grounded theory, in its original form, was proposed by Glasser and Strauss in 1967 [54]. However, it was not until 1993 that we could find the first significant grounded theory work applied in software engineering [55]. Since that date, more researchers have adopted the process and the GT has been supported by promising results. There is a limited, but increasing, body of literature reporting the application of grounded theory in software engineering (SE) disciplines. Nevertheless, GT applications in SE are still very limited, mostly likely due to the complexities of conducting GT methodology in SE. The GT methodology, we argue, requires adaptation for successful employment in the SE world. The contribution of this chapter is to provide meta-codes that can be used to drive the initial coding phase of GT. We also provide an analysis of existing GT applications in software engineering and the characteristics of such applications as exhibited in the existing literature.

This section is organized as follows. We first present a brief history of grounded theory and its application in the software engineering arena. Then, we present the methodology we adopt to survey, categorize, and analyze GT coding. The subsequent three sections present a literature review and the meta-codes thematically organized by the application of grounded theory in agile development, distributed development, and requirements engineering. The remainder of this

section presents some GT characteristics that are specific to applications in software engineering and an overview of where GT has been successful and where challenges exist in the application of GT in software engineering.

6.1.1 Background and History

Grounded Theory is a systematic qualitative research methodology that emphasizes the generation of theory from data. Grounded theory operates almost in a reverse fashion to the traditional scientific method. Rather than proposing a hypothesis and gathering data to support it, data collection is pursued first, without any preconceptions. This ad-hoc characteristic is of great interest from our perspective because it allows us to study Umple user's perspective without having to have any hypothesis. The process continues by marking key points in the data with a series of 'codes', which are then grouped into similar concepts, or categories. These categories become the basis of a theory. The coding process is typically performed in two steps, initial then focused coding. The categorization process is normally referred to as *axial coding*.

Grounded theory emerged as a research methodology in the 1960s, during a time when sociological research practices were particularly reliant on quantitative methodologies. In 1967, Glaser and Strauss coined the term grounded theory in their book "The Discovery of Grounded Theory" [56]. The term refers to the idea of a theory that is generated by – or grounded in – an iterative process of analysis and sampling of qualitative data gathered from concrete settings, such as interviews, participant observation, and archival research.

The roots of this methodology can be traced back to the work of Wilhelm Dilthey who argued against the pursuit of causal explanations at the expense of establishing understanding. Grounded theory methodology can also be traced back to the symbolic interactionist perspective of Herbert Blumer [57]. The term "symbolic interaction" refers to the peculiar and distinctive character of interaction as it takes place between human beings. The peculiarity consists in the fact that human beings interpret or "define" each other's actions instead of merely reacting to each other's actions.

Since GTs' inception in the social sciences, grounded theory has become increasingly popular in information systems as a research methodology. This is evident by the growing literature on the methodology and its applications. The first publication we were able to identify as an application of grounded theory in the area of software engineering was the work by Calloway and Ariav [58] and Torasker [59] in 1991. In these publications, the researchers described how they adopted grounded theory in understanding how managerial users evaluate their decision support systems.

The first international journal publication of a grounded theory application in software engineering is that of Orlikowski in 1993 [55]. In this work, the researcher presents findings of a study into the adoption of CASE tools. The researcher justified the use of grounded theory as a

research methodology on the basis that it provided "a focus on contextual and processual elements as well as the action of key players associated with organizational change elements that are often omitted in IS -Information Systems- studies" [55].

More recently, Baskerville and Pries-Heje [60] employed grounded theory combined with action research to enhance the rigor and traceability in the theory-development part of their work. Action research is a reflective process of progressive problem solving led by individuals working with professionals to improve the way they address issues and solve problems. Other work has employed grounded theory to initiate more focused data collection activities [61].

Grounded theory applications have extended to other arenas within software engineering. While the literature is limited, the most prominent discipline of grounded theory work is in software development methodologies, as evident in the quantity of published work in this discipline. Out of the 60 research papers identified as applications of grounded theory in software engineering, 25 addressed software development methodology. Other sub-disciplines with significant bodies of GT research include requirements engineering and distributed software development practices.

We believe that GT is a research methodology particularly useful for software engineering research for reasons that include:

- Software development is a human intensive process; software is used by humans with complex interaction and usage patterns, where quantitative evidence is nonexistent or difficult to formulate,
- GT provides an effective approach for qualitative validation.

The low and slow adoption of GT methodology in SE is due to a number of factors. GT originated in the social sciences, and since its adoption in SE, there is little guidance on how to employ the methodology, in addition, it is not clear what characteristics of the GT needs adaptation to better fit the nature of SE research. Some researchers in the software engineering field are not familiar with the GT methodology, and can frequently be skeptical of its effectiveness. In addition, as our survey highlights, the number of researchers that have reported using GT is small which contributes to barriers of more GT adoption.

6.1.2 Discussion of Sources

Surveying the application of grounded theory in software engineering turned out to be more challenging than anticipated. Grounded theory work is published in a large variety of journals and conference proceedings. A significant portion of grounded theory research can be located in journals dealing with empirical studies. Nevertheless, a growing number of grounded theory projects deal with development processes, requirements engineering, tooling, and development practices. Such work is typically published in journals not related to empirical studies. What

follows is a review of the methodology used to identify candidate GT sources to ensure that we covered the full gamete of papers on the subject.

We located more than 60 published papers that explicitly reported the use of grounded theory in the analysis of their data in an area related to software engineering. While the determination of the use of grounded theory as a research methodology was relatively clear, the scope that defines what software engineering is, is more challenging. Hence, we found a thematic presentation was most appropriate. The surveyed resources are organized under three main themes; agile development, distributed development, and requirements engineering. These three disciplines contain a major portion of the grounded theory work within software engineering.

Some grounded theory approaches recommend starting with high level codes to drive theory building [62]. This is particularly challenging due to the small amount of literature available on the application of GT in software engineering. In order to help SE researchers, we collected all codes and categories that were reported in each GT application theme. We then analyzed those codes in a GT approach to create what we call *meta-codes*, or codes of codes. We first collected all codes and sub codes from the grounded theory papers in each theme separately. Those codes were then analyzed, rearranged, and merged to create a final shallow hierarchy of meta-codes. Each meta-code is associated with tags that summarize a larger number of codes and sub codes as exhibited in the literature within a specific theme. It is our conjecture that the meta-codes can be of value to future applications of GT in the software engineering themes presented in this chapter; they can function as high level codes that drive theory building in these areas.

6.1.3 Grounded Theory in Agile Development Methodologies

We were able to identify 32 published papers that applied grounded theory to study software development methodologies. Of these, nine reported studying agile methodologies.

Agile software development refers to a group of methodologies that share and promote principles such as development with short iterations, teamwork, collaboration, and process adaptability throughout the life-cycle of the project [63]. The roots of agile development can be traced back to 1974 when an adaptive software development process was introduced by Edmonds [64]. However, the definition of modern agile development processes evolved in the 1990s. For example, eXtreme Programming was formally introduced in 1996 [65].

Out of all surveyed papers, nine reported research into agile methodologies using grounded theory. This number reflects the fact that agile development processes are a relatively new and evolving concept. In addition, applications of grounded theory work in software development methodology in general are limited [61]. The earliest work that reported a grounded theory methodology in an agile development process setting is that of Kähkönen and Abrahamsson [66].

Some of the most prominent work is that of Coleman et al. [65, 67, 68], who report on how software process and software process improvement (SPI) is applied in the practice of software development. Their study focused on a number of indigenous Irish software companies at various stages of development. In the first phase of the study, they performed four interviews in three different companies; each interview contained 53 questions. In the second phase, they investigated 11 more companies, performing interviews of about an hour each. They initially performed focused and axial coding, which resulted in three themes and 17 core categories. The theory they present represents a form of 'experience' road map illustrating some of the potential pitfalls a software product company could face and how others have avoided or resolved them. Their findings also included supporting evidence and justifications regarding the low level of adoption of CMM/CMMI and ISO 9000 by Irish software companies. They cited cost of implementation and maintenance, the added burden on the development efforts, and increased documentation and bureaucracy as the main factors behind the low adoption of the SPI initiatives. For example, they report that smaller companies believed SPI would negatively impact their creativity and flexibility.

Another example of use of the grounded theory approach in an agile environment involved exploring the socio-psychological characteristics of agile teams and to learn about the type of experiences acquired in such software development teams [69, 70]. The findings contribute a better understanding of the link between agile practices and positive team outcomes such as motivation and cohesion.

Meta-codes for Agile development methodologies

We collected codes and sub codes from the 9 studies that adopted GT to investigate agile development methodology. We constructed the meta-codes by analyzing 50 codes, and 206 sub codes. Meta-codes and tags are summarized in Table 12.

Table 12: Meta codes for agile development methodologies

No.	Agile development Meta-codes	Tags / Description
1	Characteristics/Practices of agile development	communications, processes, negotiations, skills, team, commitment, management, implementation, knowledge sharing trust, software builds, team rooms, workspaces, meetings.
2	Challenges of agile development	Requirements, communications, people oriented process, formality, team cohesion
3	Company characteristics	Domain, number of projects, market sector.
4	Project Characteristics	Duration, complexity, development sites, customer locations, team size.
6	Lessons	Tools, expertise, culture, trust, training, commitment, resource management.

Table 12 presents a summary of the meta-codes we constructed in the agile methodology theme. Each meta-code represents a large number of codes and sub-codes, samples of which are presented in Table 12. Here we provide a description for each of the meta-codes.

Characteristics/Practices of agile development. This meta-code is used to group codes and sub codes that refer to a characteristic specific to an agile software development project. This includes the nature of communication within teams, knowledge sharing, and the characteristics of trust within a development team, management, and the client. It also includes team rooms, and the nature of the workspaces and meetings.

Challenges of agile development. This code groups challenges in agile development related to requirement gathering activities, requirement stability, nature and frequency of changes in requirements, communications, about the people-oriented rather than process-oriented control, lack of formality, and lack of team cohesion.

Company characteristics. Company-related codes were reported in two studies. This meta-code groups tags related to the company domain, the number of agile projects in execution and in total, as well as the targeted market sector.

Project characteristics. This meta-code groups all codes related to the agile project characteristics. This includes duration of the projects on average and individually, complexity of the project as perceived, and objectively, the number of development sites and development team size.

Lessons. This meta-code collects all lessons learned that are related to agile development. Lessons learned were related to the tools being utilized, the importance of expertise within the team, the culture role in the success of projects, and the role of trust. In addition, it includes the importance of formal training, and the commitment of every team member to the success of the agile activities, and the importance of proactive resource management.

6.1.4 Grounded Theory and Geographically Distributed Development (GDD)

Out of our surveyed literature, we identified seven studies on Geographically Distributed Development (GDD) using GT. GDD, also known as Distributed Software Development (DSD), has grown to be a common practice in today's industry [71]. Despite the limited number of publications, GDD seems to be a fertile discipline for grounded theory application for the following reasons:

- GDD has grown, and is still growing, exponentially in the last decade [72].
- GDD brings about additional complexity to any development process.
- There is a wealth of data sources that can be analyzed using grounded theory analysis. For example, communications in GDD are typically written communications (Email, chat sessions) that can be easily recorded over an extended period of time with little effort and little disruptions to existing business activities. Such data are typically absent in normal settings, or require significant effort to facilitate data collection.

There are situations when a surveyed GT work addressed both GDD and agile methodology at the same time, as we show in this section. In such situations, we actually classified the paper under both themes, including their codes and sub-codes in the analysis and construction of meta-codes in both themes.

GDD becomes extensively complex and challenging when an agile method is adopted [73]. Agile processes depend heavily on information, short informal meetings, and face-to-face communications. Ramesh [72] has reported a grounded theory approach that analyses data from three different organizations, attempting to answer the question whether distributed software development can be agile. Ramesh has identified a number of challenges specific to distributed agile development processes, nevertheless, he concluded that distributed and agile can be combined.

Layman [71] pursued a different approach. Layman studied a successful distributed agile development project in the U.S and Czech Republic in an attempt to uncover the characteristics of these successful projects. They collected the data from archives of emails, as well as semi-structured interviews. Quantitative data (number of source file lines for example) was supplementary to their qualitative data. Their work's main contribution is the recommendation of four success factors for a distributed XP methodology; the facilitation of communication by the management, short asynchronous communication loops, identifiable customer authority to resolve requirement related issues, and a high process visibility.

It is typical for grounded theory research activities to take place in real life situations, by interviewing or collecting data from real projects. However, one study [74] reported grounded theory methodology using student subjects comprising 21 virtual teams collaborating in the completion of a given task. In this study, the researcher aimed at uncovering how distributed projects are managed and executed. The study concludes with characteristics of managing a distributed project, as well as proposing a model for distributed project management. A similar work [75] also utilized students in a study of distributed development using student participants. The study relied on the analysis of electronic communications collected during the performance of a distributed development task by the students.

Managing requirements in a distributed development setting presents unique challenges. Requirements engineering is a communication-intensive and dynamic task. When stakeholders are geographically distributed, requirement engineering tasks become even more complex. Damian and Zowghi [76] present their field study work that investigates requirements engineering challenges introduced by stakeholders' geographical distribution in a multi-site organization. Their goal is to examine requirements engineering practice in global software development and formulate recommendations for improvements. In the next section, we discuss grounded theory-based requirements engineering research in non-distributed projects.

Meta-codes for geographically distributed development

Out of the seven identified GT studies on GDD, we analyzed the codes extracted from six studies. One study did not provide adequate reporting on their codes and subcodes. We collected 31 codes, and 95 sub codes resulting in eleven meta-codes presented below in Table 13.

Table 13: Meta-codes for geographically distributed development

No.	GDD Meta-codes	Tags / Description
1	Communication	communication patterns (generating ideas, confirmation, consensus, conflict, humor, attitude), positive and negative,
2	Coordination	Time zone (delay in responses) collaboration, Involvement
3	Adaptation	social, work, technological, conflict resolution, lateral thinking
4	Company background	company size, maturity levels, existing development approaches, company's culture.
5	Stakeholders	project under study's stakeholders related information, years of experience, etc..
6	Collaboration technologies	simple emails, advanced collaboration technologies
7	Requirements challenges due to distance	inadequate communication, knowledge management, cultural diversity, time difference
8	Requirements activities	elicitation, prioritization, negotiation, validation, examining current system, managing uncertainty specification
9	Involvement of users	achieving appropriate participation of system users and field personnel,
10	Trust	checking project status, concern about a member doing his task, trust built progressively,
11	Delay	Sources and nature of delay, perceived causes, delay mitigation actions

Table 13 presents a summary of the meta-codes we constructed in the geographically distributed development theme. Each meta-code represents a large number of codes and sub-codes, samples of which are presented in the table above. Here we provide an analysis and description for each of the meta-codes.

GDD projects are after all software development projects, so it was expected to see a number of codes that can be found in a typical software engineering project. Communications in a GDD project plays a more prominent role, and it was found in almost every set of codes analyzed. Coordination and adaptation meta-codes are closely associated with the GDD nature of the project. That code represented codes related to time zone issues, collaboration, level of involvement, and social and cultural issues. All these aspects are related to the geographical

nature of the project. Collaboration technologies, requirement challenges due to distance, involvement of users, delay and trust are meta-codes that were found specific to GDD projects.

6.1.5 Grounded Theory and Requirement Engineering

Requirements engineering is particularly attractive for a grounded theory methodology for a number of reasons. Applications and systems are growing increasingly more complex, and involve ever-increasing numbers of users and stakeholders. Grounded theory can help discover patterns from a stakeholders' perspective of the system under development that may increase our knowledge of the users' needs, and how those stakeholders may perceive aspects related to the new system, like organization impact of the new system, and changes in business tasks and activities.

Requirements management tools now incorporate discussions, communications, and issues related to requirements. This large amount of data can serve as the basis for extensive grounded theory work. Data collection techniques that are typically applicable in social sciences and psychology, from which grounded theory has emerged, are not always as applicable in software engineering. Such requirements management tools provide unbiased data that is otherwise hard, or sometimes impossible, to collect without some level of disturbance of existing business activities.

A prominent work in applying grounded theory to the requirements engineering discipline is the work of Damian and Zowghli [76], where they report on the investigation of requirements engineering challenges introduced by stakeholder's geographical distribution in a multi-site organization. In addition to conducting semi-structured interviews, they also analyzed existing documents, and observed requirement meetings. In this work, and due to the geographically distributed organization, stakeholders heavily relied on Emails and automated requirements engineering tools that provided recorded, as well as detailed, history of discussions and communications.

Qureshi and Liu [74] applied grounded theory methodology on a case study of distributed software project management activities. Their study spanned all project phases and used observations and transcripts of electronic communications as their data sources.

A study of the Hewlett-Packard requirements engineering process considered two projects [77]. The first project was small, agile, and characterized by quick releases, while the second project was large, complex, and outsourced some of the development. Hewlett-Packard has a large and varying collection of requirements engineering processes. The selection of such processes is influenced by business drivers and constraints, as well as characteristics of the project itself. Alan Padula [77] has reported on how Hewlett-Packard selects the requirement engineering process based on project attributes. Requirements engineering is inherently dynamic due to the nature of continuous change put forward by the various stakeholders. It is argued that

information system contexts are soft and ambiguous, and are therefore mainly characterized by qualitative data. Such characteristics make grounded theory a suitable methodology for research in the requirements engineering discipline. Galal and Paul [78] presented an analytical technique, based on grounded theory, for developing qualitative scenarios against which statements of requirements can be evaluated [78].

Meta-codes for requirements engineering

For requirements engineering, we collected 26 codes and 54 sub-codes for analysis that resulted in eleven meta-codes. We summarize meta-codes in Table 14.

Table 14: Meta-codes for requirements engineering

No.	Requirement Engineering Meta-codes	Tags / Description
1	Challenges	Distance, communications, knowledge management, customer culture, awareness of processes for RE,
2	Elicitation	These meta-codes refer to the standard RE activities [79].
3	Prioritization	
4	Negotiation	
5	Validation	
6	Specification	
7	Examining current system	Existing system attributes.
8	Business objectives	Business objectives of the current software project to which RE is being performed.
9	Primary business attributes	Nature of users, expected project contribution to business goals, etc..
10	Primary project attributes	Type of requirements, RE process, complexity of requirements, ..
11	RE process attributes	Iterative development, development team and business analysis team, ..

Our meta-codes include five of the standard requirement engineering activities. It is possible that the referenced studies has used the standard requirement engineering activities as code seeds to initiate their coding process.

6.1.6 Other Applications of Grounded Theory

Grounded theory has been applied to a number of other subjects within software engineering that do not fall under our three main themes. Grounded theory has been employed to investigate tool and technology adoption [80], the impact of background knowledge of the performance of software developers [81], the motivation of open source software developers [82], questions developers ask during software maintenance tasks [83], knowledge repositories in software companies [84], barriers to adoption of software reuse [85], cognitive patterns used when

explaining or understanding software [86], and new product development management issues and decision-making approaches of development managers [87].

We opted not to include the analysis of codes and sub-codes of this type of GT application for a number of reasons. First, we could not find sufficient literature of the application of GT to create meaningful new themes. And due to the lack of papers, construction of meta-codes using our approach will inevitably result in biased meta-codes that reflect more the surveyed studies, rather than the emergence of a pattern observed from a broader coding or sub-coding processes.

6.1.7 Opportunities and Challenges of GD Application in Software Engineering

Our analysis of the surveyed papers, as well as our experience in applying GT in software engineering, highlights a number of opportunities unique to the software engineering field that makes GT an even more promising research methodology. With opportunities, come challenges that SE researchers should be aware of while preparing for their research. The analysis we present in this section is extracted from the surveyed literature, and does not reflect our own experience with GT.

Opportunities:

- The lack of integrated theories in the literature related to a number of areas in software engineering practices suggest the use of an inductive approach that allows theories to emerge based on pragmatic accounts of professionals themselves. For example, the role of communication and trust in distributed development is not formalized in a theory. However, a number of studies reported in this chapter have addressed the role of communication and trust in distributed development as reflected by the experience of professionals in GDD projects.

- Grounded theory has well-established guidelines for conducting inductive, theory-generating research.

- Software development is a human-intensive activity and development processes are characterized by heavy reliance on human compliance, emphasizing the human aspects of software engineering. Grounded theory is renowned for its application to the analysis of human behavior.

- Grounded theory is a burgeoning methodology in the information systems arena, and has been an established and credible methodology in sociological and health disciplines.

- Grounded theory (for the novice researcher or the experienced researchers new to interpretive studies) provides a useful template and as such serves as a comfort factor in the stressful and uncertain nature of conducting qualitative research [88].

- Software engineering relies significantly on software tools for managing artifacts, as well as documentation and communications. These tools make available recorded communications, potentially over an extended period of time, that become valuable assets for grounded theory analysis. In comparison to typical social settings, such information is either non-existent, or significantly harder to collect.

Challenges:

- Data collection within an organization for research purposes is typically challenging. There are a number of business priorities that take precedence over participation in research activities. Ethics committee and managerial approval is required prior to performing such research. The requirement for management approval raises the question of whether the data sampling is actually unbiased and is an honest representation of the organization or activities under study or not.

- The use of semi-structured interviews, centers data collection on users' opinions. This can lead to an over-emphasis on the participants' perception of what is taking place, which could be at odds with reality. Despite the occasions when there is no supporting evidence, the researcher is obliged to accept what the respondents say during the interviews [89]. However, in certain situations such as decision-making processes, managers base their decisions on their own perceptions, and therefore it is the perception that matters [68]. In addition, semi-structured interviews need not be the only data collection activities: As discussed earlier, there are a number of papers reporting the utilization of electronic communications, documentations, and archives as data sources.

6.1.8 Adaptation of Grounded Theory

Because grounded theory as a methodology has emerged from the social sciences, one could justifiably adapt the methodology when adopting it in software engineering research. The existence of some variations of grounded theory in social sciences is reported in the literature [90]. In addition, rigid application of grounded theory has been critically questioned [76]. We have noted three major characteristics specific to grounded theory work when applied to software engineering. Those characteristics are related to a) literature review prior to the study, b) selection of participants, and c) data sources. In this section, we briefly highlight those characteristics.

A prominent characteristic of grounded theory is captured by the advice offered by Glaser [54, 91]: "There is a need not to review any of the literature in the substantive area under study. This dictum is brought about by the desire not to contaminate...it is vital to be reading and studying from the outset of the research, but in unrelated fields".

Contrary to this advice, a number of grounded theory researchers have explicitly advocated the benefits of literature and background knowledge of the researcher prior to conducting data

gathering activities [61, 88]. It is reported that prior knowledge helps in guiding research and the use of seed categories (such as our meta-codes) helps inform analysis. This deviance from the original methodology is justifiable as some background knowledge is needed to help the researcher in the process of interviewing and data collection. The researchers' personal constructs and skills help structure data, and it is the researcher's hermeneutic perspective that maintains the interpretive style rather than the grounded theory method [88].

Selection of participants was particularly challenging for a number of reported research activities in the software engineering arena. While there is normally a criterion for the selection of subjects (based on their role in the study case for example), subject selection was largely affected by management and the participants' availability [89]. In such situations, management could deliberately select participants that would present a favorable picture. Informing management of the objective and purpose of the research, and guaranteeing an adequate level of privacy and confidentiality of the data can help mitigate such risks.

Data sources in grounded theory work seem to be overwhelmingly reliant on semi-structured interviews. However, in a number of studies, particularly those addressing distributed development, researchers made significant use of documented email communications and chat session histories. Such data sources are typically nonexistent in normal social sciences settings. Researchers also made use of existing manuals and archives.

6.1.9 Analysis of meta-codes

We present the following remarks about our meta-codes presented in Table 12, Table 13, and Table 14:

1. Communications and trust tend to be central to all GT applications in the three themes under study.

All themes included communications and trust at the first or second level. This may reflect the importance of communication and trust in software development activities. It may also indicate that the existing GT studies focused on studying communications and trust. This may be due to the nature of how GT is developed from interviews, meetings, etc. Such data sources may inevitably reflect communications and trust aspects of software development projects.

2. Meta-codes derive their significance from the specific GT application.

In the process of developing our meta-codes, we were careful to only select codes from studies that sufficiently addressed its corresponding theme using the GT approach. However, each study had its own unique settings, procedures, and study objective and findings. For example, Padula [77] was studying the requirements engineering process with Hewlett-Packard, and focused on the study of two particular HP products, while other studies had a different focus. In our analysis, we were careful to select codes and sub codes that were, as much as possible, not

related to a specific product or study. At the same time, we also want our meta-codes to represent adequate coverage of the existing literature.

3. Overlapping of themes and codes.

The three themes presented in this paper are not mutually exclusive. For example, the study [76] addressed GT application for requirements engineering in a GDD environment. This paper was justifiably classified under the two themes GDD and RE. During the process of code analysis, additional care was required to properly select codes that addressed aspects of the project that corresponded to the theme under which the meta-codes were listed. For example, in the study by Damian, D [76] codes that related to GDD was listed under GDD theme, and similarly for RE codes.

6.2 Grounded Theory study of Umple

Umple is being used by a variety of people, including students, in an exploratory manner. The aim of this study is to gather information about Umple as it stands, and also to gain experience in studying Umple that will be applied as Umple is improved and maintained.

The research activities are comprised of a questionnaire and an interview of participants who have experience using Umple. The questionnaire provides data for simple statistical analysis, and the interviews are analyzed using a grounded theory approach.

6.2.1 Purpose

We claim that using Umple will enhance the quality and reduce the effort required to develop software systems and enhance learning about UML and object-oriented programming. While we maintain neutrality in the process of conducting this research activity, we aim at verifying the claim that Umple enhances developer's productivity. The study will also help the researchers understand how users implement systems using Umple, possible usage patterns, and discover new features useful for Umple users. The results of this research activity will help guide ongoing and future Umple development activities.

We obtained ethical approval from the University of Ottawa's Health Sciences and Science Review Ethics Board number H12-08-04 approved on February, 18th, 2009.

6.2.2 Objective

The main objective of the study is to determine what is needed to enhance Umple with adjustments or features directly related to the needs of end users. Other objectives are to enhance modeling practices, increase consistency among models and code, and provide a tool

aimed at educating students with the value of UML models. We have iteratively used findings from this study to help guide Umple development activities.

6.2.3 Methodology

Our methodology is to conduct a questionnaire and an interview of users who have previous experience with Umple. The interview results were analyzed following a grounded theory approach. The analysis aims at identifying implementation patterns and helping in the process of assessing the modeling and coding consistency of the implemented solutions.

6.2.4 Participants

The community of users of Umple consists of people who have downloaded it for use in developing software, or have used it live online at UmpleOnline. Since Umple is developed at the University of Ottawa, early adopters include graduate and undergraduate students. The process of encouraging adoption of a new programming technology includes putting it on the web, and publishing papers about it.

We were able to recruit seven participants. These participants have been graduate and undergraduate students who have used Umple in their university course work. The participants experience with Umple includes creating class diagrams to model a small class domain problem, modeling behavior using state machine concepts, generate code, and analyze the generated code.

6.2.5 Participants' tasks

Participants are asked to fill a short questionnaire and are interviewed for about 30 minutes each. The questionnaire and interview questions are related to their experience using the Umple language, major challenges, and code generation experiences using Umple. The questionnaire data is analyzed quantitatively, while the interview is analyzed in a grounded theory approach.

6.2.6 Questionnaire

The questionnaire is estimated to take 10 minutes to answer. The questionnaire is anonymous and the questions focus on the participants' Umple user experience, their perception of the syntax, the generated code, and Umple's role in past, present, or future software development activities. The questionnaire questions are presented in

Table 16: Questionnaire responses summary on page 142.

6.2.7 Interview

The objective of the interview is to better understand user's experience using Umple as a modeling, programming and code generation tool. The interview will also clarify decisions the participants made during their work with Umple.

The interview is semi-structured and is comprised of the following open ended questions.

Table 15: Interview questions

1. What were the main things you liked about Umple?
2. What were the main difficulties you experienced using Umple?
3. For the system you used Umple with, did you start by drawing a diagram on paper, by using RSM [Rational Software Modeler], using some other tool or by writing Umple code?
4. An objective of Umple is that it becomes a full programming language; you shouldn't need to look at the generated code. Nevertheless what do you think of the generated Java code?
5. Did you have to edit the generated code from Umple? What were the problems you had to fix, if any?
6. How much code did you have to write to complete the system implementation? For which components of your system?
7. For future work, how would you go about modeling and implementing the system? How would your approach change, if at all, if you are solving a different problem?
8. What new features would be useful in Umple? Do you think embedding state diagrams would be useful? How about design patterns?
9. Assuming Umple was turned into a production quality compiler with good error messages, do you think you would use Umple in your future development projects? Why or why not?

The interview questions were made purposely generic so that they apply to Umple class and state machine models. The exception being question number eight that explicitly referred to state

machines and patterns as being features planned in the future. This question therefore was modified when state machines were developed.

6.3 Results and Analysis:

Questionnaire data are analyzed quantitatively. Interviews are transcribed and analyzed using a grounded theory approach. During the analysis activities, we removed all data that may reveal participants identities, and ignored all quotes that may compromise the identity of the participants.

6.3.1 Questionnaire results

Table 16 presents the results of the participants' responses to the questionnaire. Note that some participants did not provide answers to all questions on the questionnaire. This happened when participants did not have experience with Umple feature that relates to the question. Or, in other cases, the participant opted to skip the question.

Table 16: Questionnaire responses summary

Q	Question description	SA =5	A =4	N =3	D =2	SD =1	weighted Average
1	Umple syntax for attributes and associations is easy to learn?	2	4	0	0	0	4.3
2	Umple syntax covers all my programming needs, except user interface and database access?	0	3	0	3	0	3.0
3	Programming in Umple allows me to be more productive than programming in Java?	0	3	3	0	0	3.5
4	I find it easier to edit Umple code than to edit Java code?	0	0	3	2	0	2.6
5	Java code generated from Umple is easy to understand?	0	4	0	0	0	4.0
6	I am satisfied with how Umple implements associations in the generated Java code?	3	3	1	0	0	4.3
7	I am satisfied with how Umple implements attributes in the generated Java code?	3	3	0	0	0	4.5
8	Syntax errors in Umple are easy to identify?	0	2	0	0	0	4.0
9	Error messages that appear when I compile Umple are very useful?	0	0	0	3	0	2.0
10	It will not be necessary to look at the generated Java code to use Umple, just like you do not need to look at object code or bytecode?	0	3	2	2	0	3.1
11	I usually had to edit the generated Java code?	0	2	0	2	0	3.0
12	It was easy to reuse the generated Java code?	0	3	1	0	0	3.8
13	It was easy to reuse the Umple code?	0	4	0	0	0	4.0
14	Umple has the potential to be a major advance in programming?	0	3	2	0	0	3.6
15	Umple code is easier to understand than the equivalent Java code?	2	4	1	0	0	5.8
16	Umple has lots of bugs?	0	0	4	0	0	3.0
17	The code generated from Umple is bug-free?	0	3	2	0	0	3.6
18	I will recommend Umple to my team members or colleagues?	0	3	3	0	0	3.5
19	I will use Umple in my future development projects?	0	3	1	1	0	3.4
20	Umple can help students to understand how to create proper models?	0	3	1	0	0	3.8
21	Umple can help make object-oriented programming easier?	0	2	2	0	0	3.5

Question number 15 “Umple code is easier to understand than the equivalent Java code?” has yielded highest weighted average on our likert scale. This result could be because either Umple code is in fact easy to read and understand, or that the equivalent Java code is relatively complex and harder to understand. By investigating other questions, we note that question number 6 “Java code generated from Umple is easy to understand?” has high score in the weighted average scale. This means that participants find the generated Java code is relatively easy to understand. We can, therefore, with more confidence conclude that Umple code is in fact easy to understand.

This aspect of comprehensibility of the Umple and the equivalent Java code inspired us to conduct a formal experimentation that we present in “Chapter 7: Experimentation”. Briefly, the experiment results imply that Umple is indeed more comprehensible than the equivalent Java code.

The least weighted average question is question number 9 “Error messages that appear when I compile Umple are very useful?” This was expected because Umple, being a research platform under constant development, has few error messages that are not yet very meaningful; in particular the Umple compiler usually only refers to where the error has occurred, and provides little to no support on how to fix the error. Participants have also reported error messages that were unjustified and hard to interpret.

We also interpret this low average for this question to mean that Umple users did consider Umple to be a programming language, and therefore had expectations of the editor similar to a typical IDE for other high level programming languages. Users familiar with, for example, the Eclipse IDE expect the Java editor to provide features such as auto-complete, syntax highlighting, syntax errors reports in near real-time fashion, and others. This result motivated us to implement two components in Umple; first the sophisticated editor, second, syntax and semantic error messages. Umple editor is presented in the section “Umple textual editor and automated update site” on page 69. The error messages are in early stage of development and is documented as part of the Umple open source project [7].

6.3.2 Interview qualitative analysis

We conducted interviews with seven participants who had used Umple for about four hours each. The recorded interviews made up about 145 minutes of recorded audio, and their transcripts are about 27 pages of text (~ 10,800 words). The coding process was performed by the same researcher who performed the transcription of the interviews.

6.3.3 Coding process

We used a word processor to conduct our coding process. The resulting codes are presented in Figure 42.

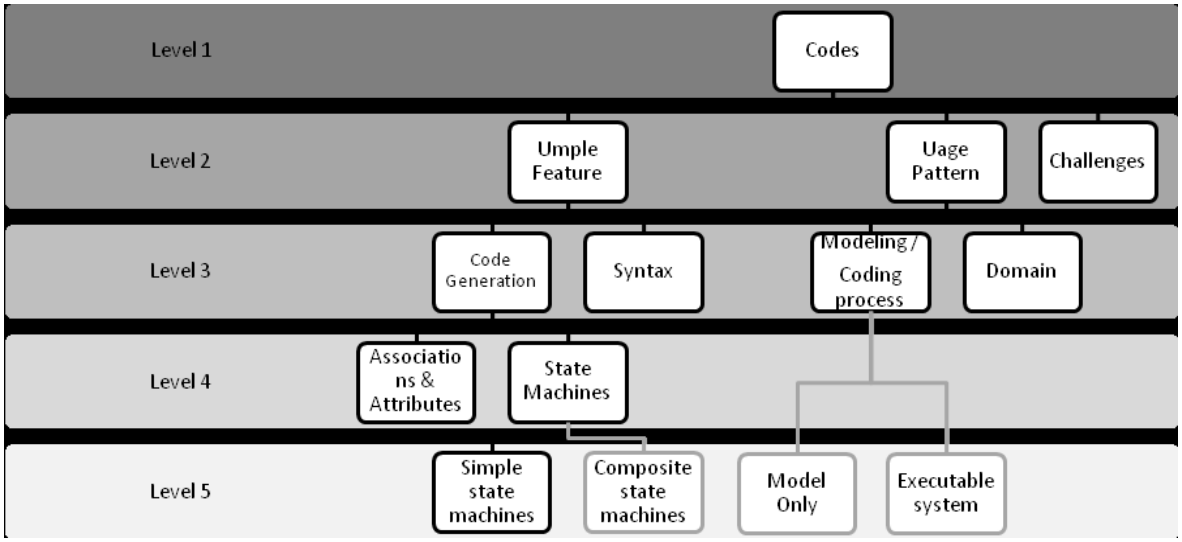


Figure 42: Codes

The coding process was performed iteratively with the interviewing process. In addition, as new major features were added to Umple, answers to the generic interview questions addressed more recent features of Umple. This means that codes presented in Figure 42 were not equally embedded in the data. For example, Code generation for associations and attributes were more embedded in the data than the codes on state machines. This is because state machines in Umple were developed at a later stage than associations and attributes. Moreover, the development of state machines in Umple were influenced by user's feedback on associations and attributes. For example, we noticed that some provided answers that imply that they did not analyze the generated code carefully, even when the tasks they were performing required closer analysis. As an example of this, a user reporting that the code generated for associations are sometime unnecessarily replicated in the generated code. We therefore realized that we need to enhance the use of inline comments in the generated code so that when and if a user investigates the generated code, he shall be able to locate and comprehend the piece of code he is trying to investigate.

6.3.4 Codes summary

We now present a short description of each code along with some representative quotations from the interviews, and discuss some of the codes we anticipate to emerge had this study included users whose Umple experience was more extensive.

Umple feature

This is a top-level code that groups participants' feedback relating to an Umple feature.

Usage pattern

This is a top-level code that groups participants' feedback relating to the way Umple was used and the environment in which it was used.

Challenges

This is a top-level code that we opted not to break down into sub-codes. This is because we routinely referred to text under this code when justifying or weighing on a design or an implementation decision. Samples of this code are:

- Associations being generated twice in the Java code (which was misconception of some Umple users)
- Web based application being slow.
- Umple reported an error message for no apparent reason.
- Error messages unclear.
- The generated code is too long.
- Relative simple classes result in large Java code.
- Attributes always had a setter and getter.
- Umple documentation can be improved
- Misleading/meaningless error messages.
- The generated code, once edited, cannot be refactored into Umple.
- Limitations of what you can do with only Umple, i.e., building interfaces.
- Unexpected or unexplained behavior.

Note that many of these issues have been dealt with as Umple was developed.

Code Generation

The Code Generation code was used to highlight all transcript text that related to the Umple code generation process, and the generated code itself. Examples are:

- Generated setters and getters for attributes;
- Handling of Associations and implementation approach for associations;
- Assessment of the generated code.
- The need, or lack thereof, to edit the generated code;

Syntax

The Syntax code refers to Umple textual syntax, and was used to highlight all portions of the transcripts that reflected participants experience with creating the textual model. This covered aspects that related to attributes, associations, state machines, impression of the participant on the Java versus Umple code, effectiveness of writing code in Umple, etc.

Modeling / Coding process

The modeling-code process code covers aspects of the user experience that is related to the software development process they adopt, or anticipate to adopt, using Umple technology. For example, some users reported that they modeled on paper and then wrote the model in the Umple language, others reported on their experiences trying to debug Umple as compared to debugging Java. Some users reported that they used the visual editor to draft the model, and then fine tune specifics textually.

Domain

This referred to the characteristics of the environment or framework under which Umple is used. For example, the characteristics of the problems tackled by the participant using Umple. This sub-code also covers the characteristics of the environment that can influence Umple's effectiveness. For example, some participant's argued for and against using Umple in place of their current IDE, others argued that it is good for larger systems, but not as effective for smaller ones, others had the opposite argument. Further, some user's debugged the generated code and then implemented the fixes in the Umple because they had tool support for Java debugging and none for Umple. This code also covered user's experiences in using Umple in real development projects.

Associations & Attributes

All code generation issues related to associations and attributes are under this sub-code.

State machines

All code generation issues related to state machines.

Simple state machines

All code generation experiences of simple state machines are under this sub-code.

Composite state machines

We did not conduct interviews with participants that used Umple's composite state machine features. This is why this code is grayed out in Figure 42.

Model only

Some participant's had used Umple to model only, and not to build a full executable system.

Executable system

This code covers user's who used Umple to build an executable system.

6.4 Findings

We have classified the codes under three main categories; *Umple Features*, *Challenges*, and *Usage Patterns*. Umple features are further broken down onto *Code Generation* and *Syntax*. Similarly, *Usage Pattern* is broken down onto *Coding/Modeling Pattern* and *Domain*.

The "*Code generation*" code groups users experience with the generated Java/PhP code of Umple. We noted that some participants have, in one way or another, analyzed or edited the generated executable Umple code.

The "Syntax" code is based on user's remarks on the Umple syntax. Most of the text under this code implies that Umple users are satisfied with Umple syntax.

(1) *"it would be much easier for me, you know, to [write in Umple] compared to Java."*

(2) *" I like its basic syntax .. .. having to do your associations in one class or the other, that was straight forward .. that makes associations easy.."*

(3) *" You do not have to write setters or getters, these are automatically generated for you"*

(4) *"I will use Umple to generate code for my state machines online. I do not have to install RSA [Rational Software Architect]if I need to quickly implement a simple state machine "*

(5) *"It is easy to implement state machine this way."*

There were no negative remarks in all of our transcripts about Umple syntax. We would expect the Questionnaire to reflect a particularly positive user's experience with Umple syntax. However, question 1 resulted in weighted values of 4.3, out of a maximum of 5. The results of this question did not particularly reflect the anticipated positive user experience with Umple syntax.

Analyzing the syntax sub-code, we could not understand why some participants did not find Umple to satisfy all their programming needs (see question 2 in the questionnaire results in

Table 16). This may mean that users are satisfied with Umple syntax, even though it may not cover all their programming needs, or that the syntax is comparable to users' functionality expectation, or that participants accept less functionality in return of simplicity.

The "Challenges" code is used for Umple experiences that were particularly challenging, or negative. This code can be further broken down into sub-codes that reflect the source of the challenge. The source of the problem can be attributed to a number of root causes. Some of those root causes are related to the Umple compiler's sophistication, error messages, interface functionality, code-assist-related features, and code generation. Many of challenges have since been fixed or are on our research agenda. The root causes of some issues were most likely a result of the participants' unfamiliarity with the Umple specifications or modeling concepts, as in quote number 7 below.

(6) *"The generated code in our assignment was too long"*

(7) *[automatic generation of setters and getters] "That was useful, and also that was annoying in some cases, because I am not sure whether you are able to define whether that was an accessor or not, it would automatically create setters and getters for you."*

(8) *"if you declare the association in certain way, it would end up generating it twice in the generated code"*

"Modeling/Coding pattern". This code is concerned with the way participants have reported their use of Umple as a modeling and coding tool. Our research's ultimate objective is to enhance modeling by the introduction of textual modeling approaches that fit traditional programmers who are more used to efficiently write code in text.

(9) *"you can look at the code, and it is almost visual"*

(10) *[Being presented by the visual and the textual] "I think that was the best thing, because you kind of get the best of the two worlds."*

(11) *"like the traditional approach of writing the code, and that is nice and all, but when you get a huge system, that needs to be modeled.. "*

Participants have frequently linked the usability of Umple approach to the domain and/or the application characteristics.

(12) *"Umple is good for simple applications"*

(13) *".. but if you are going to make something visual that has GUI components, .. and classes that adds listeners and stuff like that, that sort of thing, I do not know, I have not done that with Umple"*

We can tentatively conclude that participants find Umple suitable for small and medium size applications that do not require GUI components or special programming aspects, like connecting to a server or listing on a channel. This users' perception is, probably, a result of their focusing on using the modeling aspects of Umple. Umple is built to be an extension to existing high level programming languages, and therefore, should be suitable for implementing all types of systems. The findings here bring about the following questions:

Q.1. Is the perceived limitation in Umple due to the prototype nature of Umple as it is today? Or is the limitation due to the basic Umple approach?

Q.2. Typically, larger applications are comprised of smaller size sub-components. If Umple is suitable for smaller applications; one would assume that it is also suitable for larger applications.

Participants

Our study participants have so far been university students at the undergraduate and graduate levels who have had limited experience with Umple. This limited experience with Umple has to some extent hindered our ability to extract more extensive usage information. For example, a number of participants used only the Umple online tool [3] that has limited capabilities compared to the Umple Eclipse plug-in. We expect to conduct more studies with professional industry participants when such users are available.

6.5 Challenges

As we have discussed in this chapter, GT studies emerged from the social sciences where the pool of potential participants is large. In this study, a major challenge, which was identified early on, is the size of the pool of the potential participants. Another challenge is the level of Umple experience the participants have with Umple.

Open sourcing Umple technology, and making it available to the public has had the effect of increasing the pool of Umple users and contributors. However, by the time Umple open source project was set up and the number of new visitors started to increase (Figure 43), the grounded theory study was being finalized.

Figure 43: Number of unique visitors to the Umple Google Code site from March 1st to December 1st, 2011 (this does not include UmpleOnline).

Using Umple as an educational tool in classrooms has served well in increasing the pool of Umple users [92]. But these users have typically used Umple in a classroom environment where the modeling tasks they performed are limited.

6.6 Summary

We presented a survey of the use of grounded theory in the area of software engineering. Particularly, in agile development methodologies, geographically distributed development, and requirement engineering. We presented an overview of the challenges and opportunities of such a research methodology in the software engineering domain.

We then presented a new approach to apply grounded theory methodology which has the objective of supporting and guiding Umple experimental development and using participants' feedback as an input when making design decisions. The study participants were undergraduate and graduate students that have used Umple within a university environment. We also presented our quantitative analysis of survey data and qualitative analysis of interview data using a grounded theory approach.

Chapter 7: Experimentation

We conducted a controlled experiment aiming at understanding the effectiveness of using Umple in performing simple software development tasks. This chapter reports on this experiment that takes a human comprehension perspective on such tasks. Three different notations were investigated: UML, Java, and Umple. Our experiment asked participants to answer questions that reflect their level of comprehension. The results reveal that for simple comprehension tasks, a pure visual notation and a model-oriented (textual) language are comparable. Java's comprehension levels were lowest of all three notations. Our results align with the intuition that raising the abstraction levels of common object-oriented programming languages enhances comprehensibility.

7.1 Experiment definition, context, and steps

The goal of this experiment is to evaluate the Umple textual modeling notation in comparison with UML and Java. One of the objectives behind the development of Umple was that it should gain the advantages of both visual modeling (UML) and textual programming (i.e. Java). This experiment, therefore, seeks to validate the hypothesis that Umple has retained the advantages of UML with respect to comprehension. We leave it as separate research to assess whether Umple has also retained any advantages of textual programming.

A sub-goal of this experiment is to evaluate the effect on experimental results of using software artifacts named with identifiers derived from the domain versus abstract names.

Step 1
Participants are asked pre-experiment profiling questions. Duration: 2 minutes.

Step 2
Participants watch a short video on class and state machine modeling using Umple and UML [93, 94]. Participants are expected to be familiar with Java and require no Java training. Duration: 6 minutes.

Step 3
Participants are provided with three instances of the experiment models, one at a time. Participants are given 1 minute to review the model, and they are not allowed to ask any questions. After each presentation, the participants are asked to answer a number of questions. Duration: 25 minutes.

7.2 Experiment Metrics

The experiment employed the following metrics:

1. Number of questions answered correctly and incorrectly
2. For each question:
    2.1 Time elapsed for the first answer
    2.2 Time elapsed for a correct answer
    2.3 Number of incorrect answers (i.e number of trials)
3. Pre-experiment profiling questions

7.3 Null Hypotheses (H0)

The experiment employs three hypotheses:.

H1: A system written in Umple is more comprehensible than an equivalent Java implementation of the system.

In other words, participants take on average less time to respond to questions when presented with an Umple version of a system as opposed to a Java version.

The corresponding null hypothesis is:

H1o: Umple and Java do not differ in comprehensibility.

The next hypothesis is similar, comparing Umple and UML diagrams:

H2: A system written in Umple is more comprehensible than an equivalent UML diagram of the system.

H2o: Umple and UML diagrams do not differ in comprehensibility.

The third hypothesis was of secondary interest:

H3: Whether names derived from the domain or abstract names has an effect on experiment results regarding comprehensibility.

H3o: The use of abstract or domain-derived names makes no difference to comprehensibility.

Using systems with domain names or abstract names may or may not have an effect on response time and number of inaccurate responses. The purpose of testing H3 is to determine whether, in future experiments, we need to care about controlling for this factor.

7.4 Experiment Planning

We recruited 9 participants (none of whom had participated in the grounded theory study presented in the previous chapter). Each participant was presented with three different models (one, two, and three in Table 17: System example instances distribution) in three different notations (Umple, UML, and Java).

Table 17: System example instances distribution

	Umple	**UML**	**Java**
Participant 1	One	Three	Two
Participant 2	Two	One	Three
Participant 3	Three	Two	One
Participant 4	One	Three	Two
Participant 5	Two	One	Three
Participant 6	Three	Two	One
Participant 7	One	Three	Two
Participant 8	Two	One	Three
Participant 9	Three	Two	One

Three of the system examples used names derived from the domain (student-supervisor domain), while the remaining six models used abstract names (i.e. a, b, c) (see Table 18). Abstract names were used since we wanted to test the 'pure' comprehensibility of the notations, and wanted to avoid the threat to validity that people might understand the system simply because they understand the underlying domain. On the other hand, we also used names derived from the domain to reduce the opposite threat to validity, which is that systems with abstract names are less realistic.

Prior to use, in the experiment, the example systems and the renderings of the systems in each notation were reviewed by three independent researchers to help maintain consistent complexity levels across the modeling examples.

Table 18: Domain and abstract naming distribution

	Umple	UML	Java
Participant 1	domain	abstract	abstract
Participant 2	abstract	domain	abstract
Participant 3	abstract	abstract	domain
Participant 4	domain	abstract	abstract
Participant 5	abstract	domain	abstract
Participant 6	abstract	abstract	domain
Participant 7	domain	abstract	abstract
Participant 8	abstract	domain	abstract
Participant 9	abstract	abstract	domain

7.5 Experiment objects

There are three system examples, each are presented in three notations; UML, Umple, and Java. In total, there are nine variations of system examples. The system examples are very simple with simple associations and state machine transitions. The objective of this variation is to eliminate any learning a participant may accumulate from one experiment round to the other. The first system examples are illustrated in Figure 44, Figure 45, and Listing 16: Example one Umple code. The rest of the experiment system examples are listed in the appendix.

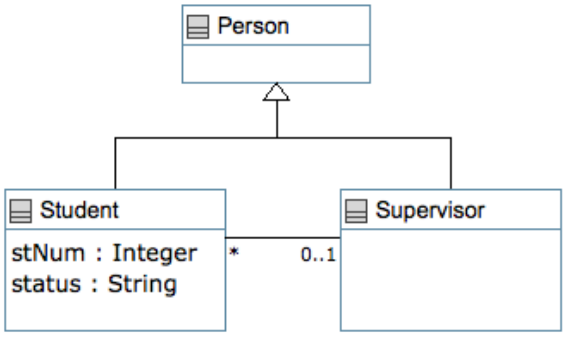


Figure 44: Example one class diagram

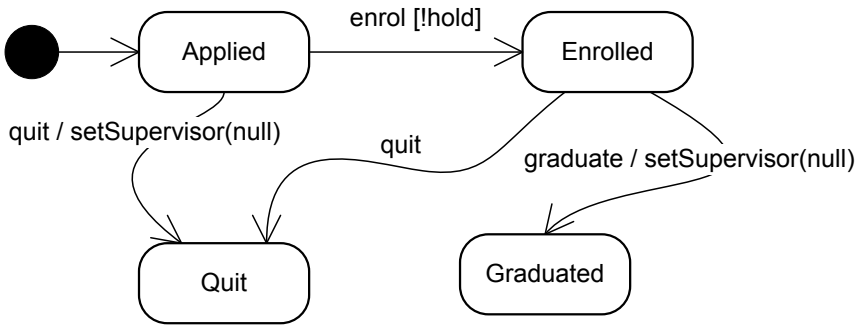


Figure 45: Example One state machine diagram

```
class Person {
  name;
}

class Student {
  isA Person;

  Integer stNum;
  status {
    Applied {
      quit -> Quit;
      enroll [hold] -> Enrolled;
    }
    Enrolled {
     quit -> /{setSupervisor(null)}Quit;
      graduate->/{setSupervisor(null)}Graduated;
    }
    Graduated {}
    Quit {}
  }

  * -- 0..1 Supervisor;
}

class Supervisor {
  isA Person;
}
```

Listing 16: Example one Umple code

The following two tables (Table 19 and Table 20) show system examples properties. The objective is to have three system examples with similar complexity levels.

The complete list of system examples are published in a technical report [95].

Table 19: Example model properties

			Number of							
	Domain Names	**Notation**	**Classes**	**Assoc.**	**Attr.**	**States**	**Trans**	**Guards**	**Unique events**	**Unique Actions**
1	NO	UML	3	3	3	4	4	1	3	1
2	NO	UMPLE								
3	NO	JAVA								
4	YES	UML	3	3	3	4	4	1	3	1
5	YES	UMPLE								
6	YES	JAVA								
7	NO	UML	3	2	1	3	5	1	5	5
8	NO	UMPLE								
9	NO	JAVA								

Table 20: Line and character numbers for Java and Umple examples

Example	**Number of lines**	**Number of characters**[11]	**Domain names?**	**Notation**
1	30	254	Yes	Umple
1	42	570	Yes	Java
2	29	154	No	Umple
2	55	469	No	Java
3	28	227	No	Umple
3	49	576	No	Java
Average Java Lines			146	
Average Umple Lines			87	

7.6 Question List

There are 12 core questions. The questions wording and correct answers have 9 variations. This is because a question posed on a Java implementation must use different wording than a question posed on a UML model. The questions for UML and Umple are almost identical. Presented below are questions for UML and Umple for the first example instance.

[11] Spaces are not counted.

Table 21: Question list for version E1 (UML and Umple)

	Questions for Version E1	
	Umple and UML questions	**Correct Answer**
Q1	Let's assume the state machine is in the Applied state and hold is false. Also assume the following events occurred in sequence, enrol, quit, enrol. What is the resulting state?	Quit.
Q2	Assume the student has one supervisor. Can you add another supervisor to the same student?	No.
Q3	Assume a supervisor has 6 students. Can we add another student to this supervisor?	Yes.
Q4	Assume the state machine is in the Applied state, and the value of hold is true. What happens when the event enrol occurs?	Nothing. No transition occurs.
Q5	How many students can a supervisor have?	Many. Unlimited number.
Q6	What are the possible states the state machine status can have?	Applied, Enrolled, Graduated, and Quit. (in any order)
Q7	What actions are called when the following transition occurs : From Applied to Enrolled	Nothing. No actions are called.
Q8	Can the state machine go directly from Quit to Enrolled?	No.
Q9	Can the state machine go from Graduated to Applied?	No.
Q10	Assume we are in the Applied state, what happens when the event graduate occurs?	Nothing.
Q11	Can you create a Person Object?	No.
Q12	Assume the state machine is in the Applied state. Also assume the following events occur in sequence: graduate, quit, quit, enrol. What is the resulting state?	Quit.

The other question lists, along with all experimental objects, can be found at [95].

7.7 Profiling information

The following profiling information is collected prior to the experiment.

Table 22: Information collected prior to the experiment

Education	Q1	Are you a bachelor's student? Masters? PhD?
	Q2	What year?
	Q3	Which university?
	Q4	How many Software Engineering courses have you successfully completed?
Background		This scale is used for the following three questions: ? (Never heard of it, know about it but not used it, used it a little, Have used it regularly, expert)
	Q5	How familiar are you with Java?
	Q6	How familiar are you with UML?
	Q7	How familiar are you with Umple?
Experience	Q8	How many years/months of working experience in the software development industry?

7.8 Selection of Participants

The 9 participants are selected randomly from the pool of university students who have completed at least two courses in OO-programming.

Participation in the experiment is anonymous and voluntary. Participants are not compensated for their participation.

7.9 Variables in the Study

This section presents the extraneous, independent, and dependent variables in the study.

7.9.1 Extraneous Variables

These variables may have an effect on the dependent variables. The experiment design attempts to eliminate or minimize the effect of extraneous variables.

Table 23: Extraneous variables

Independent variable	Description
Domain knowledge	Participants knowledge of the domain should have no impact on their responses. The system examples are very simple and domain knowledge should not have an impact on answers.
System examples complexity levels	All system examples instances used in the experiment have complexity levels identical, as possible. This is insured by using the similar number of model properties across system example instances (i.e number of classes, associations, attributes, states, transitions, actions, guards, and events)
Java, Umple, and UML background and experience	Participants are expected to have good knowledge of Java. That is because the pool of potential participants are university students who have completed at least two courses in OO programming. Knowledge of UML and Umple are variable. The following is performed in order to isolate the impact such knowledge and background on results: 1. Training examples are presented to the participants prior to commencing the experiment. 2. The system examples used are very simple, requiring only very basic knowledge of the different notations (Java, Umple, UML).
Learning during the experiment	Participants will inevitably learn aspects about the system examples during the experiment. The following is performed in order to isolate the impact of learning on results: 1. Example instances are slightly different. Participants cannot answer questions about instance 2 based on knowledge of instance 1. 2. Every system example instance is presented in a different notation.
Environment factors	Every effort is taken to eliminate or minimize external environmental factors, such as noise and interruptions.

7.9.2 Independent Variables

These are the variables that are manipulated during the experiment.

Table 24: Independent variables

Independent variable	Description
Notation	The notation used to present a system example. This variable is manipulated (UML, Umple, Java).
Domain Names and Abstract Names	System example element naming are manipulated (Domain names vs abstract names).

The example system could be considered as a third independent variable, but this is introduced only to minimize the learning effect, therefore, we do not consider it to be an independent variable.

7.9.3 Dependent Variables

These are the responses observed during the experiment. The independent variables manipulations should ideally be solely responsible for the variations in the dependent variables .

Table 25: dependent variables

Dependent Variable	Description
Time to respond to questions	The time the participant takes to find out the answer to the question is dependent on the notation used.
Number of correct responses	The number of correct responses on the first trial is dependent on the notation used.

7.10 Threats of Validity

Table 26 summarizes the threats of validity for this experiment, along with mitigation strategy for each threat.

Table 26: Threats of validity

Threat	Description (Thread and Mitigation)
Expertise and background of participants may affect how fast they respond to questions	Participants may have varying background and experience with modeling using UML and Umple. They may also have a varying background on object orientation and state machine modeling. Their background may affect how fast they are able to respond to questions.
	1. We collect demographic information on the users and make sure that distribution of background and experience is balanced. 2. At the beginning of the experiment, we present a short tutorial that covers knowledge required for answering the experiment questions. 3. Experiment examples are very simple and require minimal expertise. This has the effect of shifting the focus on the notation, rather than the subject's technical expertise 4. The experiment is balanced, participants with high levels of expertise will most likely answer better and quicker answers in all three instances of the model.
Number of examples and participants may not be representative of the overall population	This is an external validity threat. Can we generalize the results in this study to the general population? This threat is strengthened by use of student participants. This validity threat exists in many controlled experiment reported in the literature.
	1. Using three examples, rather than one. 2. Randomly recruiting nine participants. 3. Making experiment design, procedure, and models public to invite replication. 4. Use of non-parametric analysis techniques to minimize assumption of data set distribution.
Question and system examples variations have an impact on response time	Each question has 6 variations; the question essence is the same, but the wording is different. There is a threat that these variations affect the response time of participants.
	1. The questions have been reviewed by at least 3 experts independently to make sure that the variations have no, or little, impact on response time. 2. Conducted at least one pilot study for each question list to make sure the questions are unambiguously understood by participants.

7.11 Results

A total of nine participants provided answers to thirty six questions; 12 questions per example system for three instances. In total, there are 324 response times recorded.

Seven participants reported to have a PhD or enrolled in a PhD program. Two participants reported being a Master's student. The majority came from University of Ottawa (six). There was one student from Carleton and one from Cornell University.

Participants had an average of four software engineering courses. They had an average Java familiarity score of 3.3, average UML familiarity of 2.7, and an average Umple familiarity of 1.67. Participants had an average of 9.9 months of professional software development experience.

Table 27 summarizes the average response time for the 9 participants. For example, participant number 1 responded to Umple questions in 5.5 seconds on average, and 4.7 to UML questions on average, and 9.3 seconds to Java questions on average.

The table also shows the average per notation. For example, Umple's overall response time is 3.57, compared with 3.61 for UML, and 6.87 for Java.

Also in the table is the average per participant. For example, participant number 1's response time was 6.5 on average across the three different notation types.

Table 27: Average results

Participant	Umple	UML	Java	Overall average
1	5.5	4.7	9.3	**6.5**
2	5.0	4.2	9.0	**6.1**
3	3.7	3.6	6.2	**4.5**
4	2.3	3.1	6.2	**3.9**
5	2.1	3.0	6.3	**3.8**
6	2.2	2.4	3.4	**2.7**
7	4.3	2.4	6.0	**4.2**
8	3.0	4.0	6.3	**4.4**
9	4.1	5.1	9.0	**6.1**
Average	**3.57**	**3.61**	**6.87**	

It is our intention that the questions be straightforward and participants should be able to provide the correct answer at the first attempt. However, it was not always the case; there were a total of 37 incorrect responses out of the 324 questions posed. Incorrect responses were distributed among all three notations as follows: Umple with 8, UML 12, and Java 17.

The average response time per system example is summarized in Table 28.

Table 28: Average response time per example

Average Per Example	
E1	4.67
E2	4.74
E3	4.67

Questions posed for system examples that use domain names had an average response time of 4.67 seconds. On the other hand, models that used abstract names had an average response time of 4.79 seconds.

7.12 Results Analysis

This section presents our analysis and interpretation of the experiment data and is organized as follows. We first re-examine the experiment threats of validity and make an assessment of the identified threats. We then present the results of the parametric and non-parametric statistical analysis and results.

7.12.1 Assessment of threats of validity

Profiling information indicate that participants have more Java background than UML, and much less background on Umple. Therefore, if background were to have an impact, it will not be in favor of Umple.

The results of the average response time per example (Table 28) reveals that system model used does not seem to have had an impact on the response time of the participants.

7.12.2 Examining Data for Umple and Java

Figure 46 illustrates the average response time for Umple and Java.

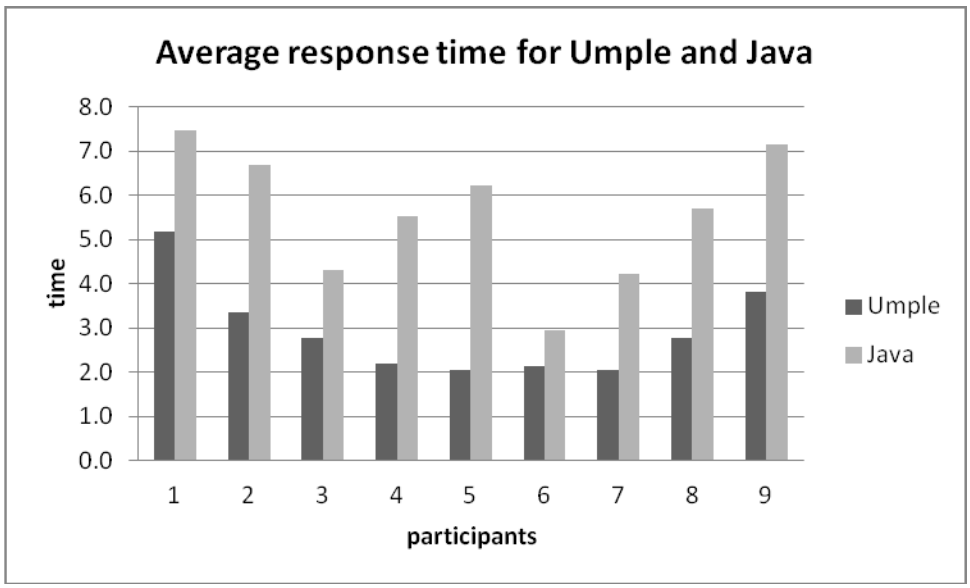


Figure 46: Average response time for Umple and Java

Using a two-tailed t-test to measure the statistical significance, Umple is better than Java ($p=1.5x10^{-8}$).

Using Mann-Whitney test (U-test) Umple is better than Java ($p = 8.9x10^{-9}$) and a W value of 2722.5.

Using the sign-test, Umple was better than Java in 83 occurrences, while Java was better than Umple in 13 occurrences. The sign test results indicate Umple is better than Java ($p=6.0x10^{-14}$).

7.12.3 Examining data for Umple and UML

Figure 47 illustrates the average response time for Umple and UML.

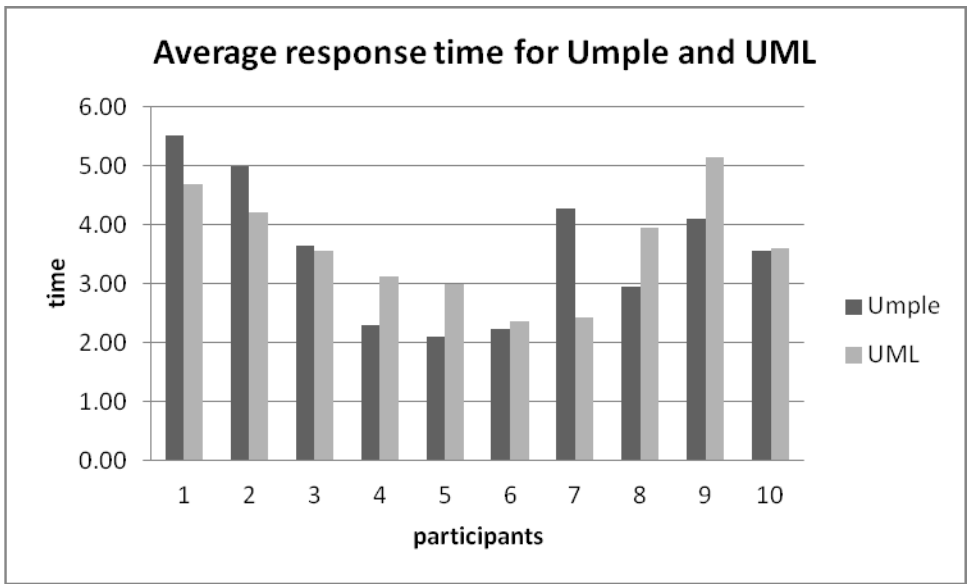


Figure 47: Average response time for Umple and UML

Using a two-tailed t-test to measure the statistical significance, Umple is not significantly better than UML (P=0.9).

Using Mann-Whitney test (U-test) Umple is not significantly better than UML (P = 0.2) and a W value of 4477.5.

Using the sign-test, Umple was better than UML in 53 occurrences, while UML was better than Umple in 30 occurrences. The sign test results indicate Umple is not significantly better than UML (P=0.864).

We also conducted mean and standard deviation analysis. For each participant's results, we test to see if the mean of Umple lies in the range of the mean of UML minus one standard deviation and the mean of Java plus one standard deviation. The answer was positive in all nine participants' results. This technique is used to show that two data sets come from different populations [96]. Here, we use it to show that the two data sets (Umple and UML) are not significantly different to conclude that they come from different populations. Elsewhere in the literature, this technique is also used to identify outliers [97].

Regarding the third hypothesis (H3), using the Mann-Whitney test (U-test), we cannot conclude that using domain or abstract names can have any significant impact of the average response time (W= 3, and P =0.6). We come to the same conclusion by using the standard deviation analysis described earlier.

7.13 Discussion

There is evidence that Umple performed significantly better than Java. But such evidence is lacking in the case of UML. Our interpretation of the results is as follows.

The tasks involved in this experiment focused on simple model comprehension and tracing questions. These tasks resemble realistic software engineering tasks [98]. These tasks, however, do not cover the wide spectrum of tasks performed by software engineers. In particular, the tasks do not address model creation, tuning, implementation, and maintenance tasks. Therefore, interpretation of the results must take into consideration the scope on which conclusion can be drawn.

We can therefore infer that Umple is better than Java in understanding a system. We can also infer that Umple is not significantly better than UML in this regard. We can with significant confidence claim that Umple is not worse than the visual UML models. After all, Umple is not meant to replace UML, but to complement it. Indeed, the UmpleOnline tool [3] allows both to be used interchangeably.

A core lesson from this experiment is that people whose program development approach is primarily textual, for any reason, should with confidence consider Umple as a viable textual technology. It retains the advantages of text, while being easier to understand than Java, and being just as comprehensible as UML diagrams when it comes to UML concepts such as state machines and associations.

7.14 Related Work

One of the challenges with the evaluation of textual and visual modeling is the wide variety of textual and visual modeling approaches available. The work of Hendrix [99] adopts a similar approach towards measuring comprehensibility levels. In his work, Hendrix evaluated textual code and control structure diagrams by measuring the time subjects took to respond to questions. We, on the other hand, evaluated UML, Java, and Umple. Our work is the first that provides empirical evaluation of the Umple modeling approach.

Briand el al. [100] evaluated two types of object-oriented documents. Similar to our experiments, it is an evaluation of two different ways of presenting equivalent information. Briand el al concluded that “Good object-oriented design is easier to understand than good structured design”. He also found no evidence that “good structured design is easier to understand than bad structured design”.

7.15 Future work

This experiment cannot be a final word on model notation effectiveness, and it is not intended to be so. Future work to replicate this experiment can be of great value in two ways. First, by increasing the number of participants; second, by recruiting more professional software engineers and making conclusions on this group of subjects; and third, by using a variety of more complex systems.

It is yet to be seen in future studies how Umple, UML, and Java compare in the performance of other, possibly more elaborate, software engineering tasks. One variant of this experiment can ask participants to spot flaws or defects in model elements, or match pieces of Umple models and Java artifacts to UML models. Such tasks can shed more light on the nature of comprehension of textual modeling.

We also noted that during the experiment, participants have used the visual notation to trace the execution of events in the state machine. The experiment did not focus on tracing tasks and we cannot draw any conclusion about the effectiveness of the visual notation as compared to the textual notation in relation to tracing tasks. A tracing language is currently under development in Umple. Experiments focusing on tracing may therefore reveal insights on this issue.

7.16 Summary

This chapter presented our evaluation of the Umple modeling approach, with a focus on state machines. We have conducted a controlled experiment where participants were presented with an example system and answered some questions. The systems were presented using three notations, UML, Umple, and Java. The experiment took a human perspective on the comprehensibility of models.

The results indicate that both Umple and UML outperformed Java. There was no statistically significant difference between Umple and UML. These results aligned with our vision; Umple is not meant to replace the visual UML models, but to complement them. This experiment did not explore whether text-oriented individuals, like programmers, find it easier to comprehend and edit models textually, while other individuals, like modelers, may find it easier to deal with visual models. This is left for future experimentation.

Chapter 8: Related work

In this chapter, we position our research with respect to the existing related work. We classify related work under two classes;

1. Work related to textual modeling.

In this section, we provide an overview of the growing trend of textual modeling in the software engineering domain. We also explore in greater detail three instances of related work; state machine development in the Ruby technology, State Machine Compiler (SMC) [101], and Specification and Description Language (SDL) [14].

2. Work related to standardization of execution semantics of UML.

The latest UML action language emerging standards are presented in this section. We demonstrate the need for action languages and briefly introduce Alf, The Action Language for Foundational UML [102].

8.1 Textual modeling

There exist a number of textual UML approaches that target textual representation for UML models. Some of these approaches are motivated by the claim that traditional mouse-centric and drag-and-drop techniques to create models require a considerable amount of clicking and dragging to fine tune the model. Therefore, textual model creation can be relatively effective. For example, MetaUML [103] has the main objective of creating UML diagrams to be readily usable in LaTeX documents. Other approaches facilitate the online creation of models, for example, yUML [104] allows users to create UML models online by including the textual description as part of the URL. Similarly, modsl [105] enables the textual creation of UML diagrams by providing an online service that translates the textual representation, being part of a wiki for example, into a visual model. WebSequenceDiagrams [106] allows users to create the textual model and see the visual model on the same web page. Human-Usable Textual Notation [107] was created to conform to human usability criteria, and was later sponsored by OMG. The notation supported class diagrams only, and was aimed at replacing XMI; it does not support the embedding of native code.

Textual representation of UML models is less evident in existing commercial tools, other than the XMI format which is not meant for human consumption. Telelogic Tau and VP Suite 4.1 SP1 are two commercial tools that have some level of support for a textual notation. Diagrammr [108] attempts to use a natural-language-like syntax to create diagrams that do not follow

specific modeling notation. SinelaboreRT [109] generates efficient code targeted towards embedded applications, and relies on XMI for integration with third party tools.

Other textual modeling tools have taken a programming-like approach. For example, UML Graph [110] uses Java syntax complemented by javadoc tags to create UML diagrams, and uses Light UML [111] to integrate with Eclipse.

TextUML [112] is an Eclipse-based tool that supports textual UML class diagrams, with good syntax highlighting and debugging features. TextUML claims creating models just like you write code, but they do not support any code generation, and instead rely on integration with a number of modeling tools for code generation, if code generation is desired. Such approaches are characterized by their objective of manipulating visual models in a way that is easy and creates neatly aligned models.

Other research directions have the objective of creating an executable subset of UML textually. Such approaches are more challenging because the objective is not to only create a UML model, but to also to define execution semantics and generate executable artifacts. For example, Ragel [113] presents a usable syntax to specify finite state machines and generates code for a number of high level languages (C, C++, Objective-C, D, Java and Ruby). Ragel supports finite state machines targeted towards parsing text and creating lexical analyzers. Ruby on Rails has recently added a built-in support for state machines.

In the next two sections, we present in more depth the analysis of State Machine Compiler and Ruby on Rails implementation of state machines. These two tools were chosen because they support two important aspects available in Umple; support of an executable subset of the UML standard, and the use of a textual notation for state machine diagrams.

To demonstrate the basic syntax for these two tools, we will use a simple state machine model (Figure 1).

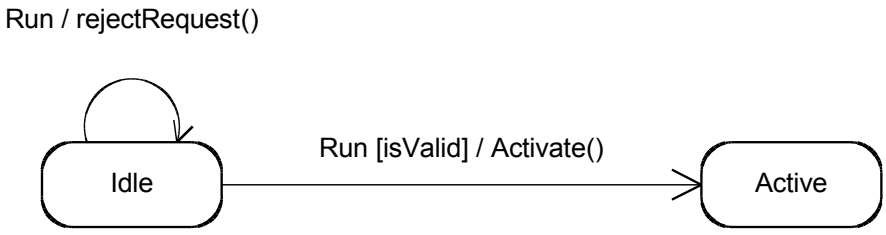


Figure 48: Simple state machine

The state machine has two states (Idle and Active), and two transitions. One transition is guarded, and the other is reflexive.

Umple state machine

```
class Engine {
  Motor {
    Idle {
      run [isValid] -> /{Activate();} Active;
      run ->  /{rejectRequest()} Idle;
    }
  }
}
```

8.1.1 State machines in Ruby

Similar to Umple, Ruby's state machine syntax can be described as declarative. States and transitions are declared similar to declaring variables in any high level programming language.

Figure 49 demonstrates our example state machine implemented in Ruby.

Figure 49: State machines in Ruby

Ruby state machine

```
class Engine < ActiveRecord::Base
include ActiveRecord::StateMachine
state_machine do

  state :Idle
  state :Active
  event :Run do
    transitions
      :to => :Idle, :from =>[:Idle],
      :on_transition =>:rejectRequest()

    transitions
      :to => [isValid] :Active, :from => [:Idle],
      :on_transition => : Activate()
  end
end
```

The declarative nature of implementing state machine syntax is evident in all state machine elements. States, events, and transitions are declared independently. This declarative nature extends to declaring other state machine elements, such as entry and exit actions (Listing 17).

Listing 17: Entry and exit actions in Ruby

Ruby on Rails

`state :Idle,`
`:enter => [:startTimer(), :checkQueue()],`
`:exit => :stopTimer()`

This declares an entry and exit actions in the Idle state.

8.1.2 State Machine Compiler

The State Machine Compiler (SMC) [101] is a Java based tool that enables developers to create event-driven applications by providing state machine specifications. The state machine specifications are provided textually. SMC uses multiple-class pattern to generate the executable code to implement the state machine.

Listing 18: State Machine Compiler

State machine Compiler

`// State`
`Idle {`
`  // Trans`
`  Run`
 	
`  // Guard condition`
`  [isValid]`
 	
`  // Next State`
`  Active`
`  // Actions`
`  { Activate(); }`
`  Run  nil {rejectRequest();}`
`}`

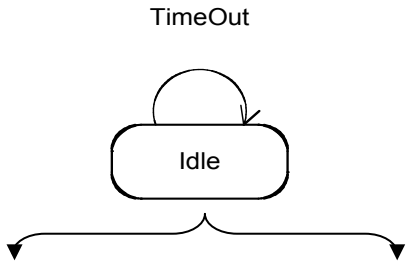
Notice that the comment like lines are compiler directives needed to tell the compiler what elements are being parsed.

State machine compiler

```
// State
  Idle
    Entry {
      StartTimer( );
      CheckQueue( );}

    Exit {
      StopTimer( );}

{
    // Transitions
}
```

SMC's Transition to 'nil' state

SMC implements a dummy nil state to which any transition can transit to. Transiting to a 'nil' state means the next state is the same as the starting state. This is useful in availing the ability to choose whether to execute the entry and exit actions of a state, or not. Listing 19 illustrates two types of reflexive transition.



Listing 19: Nil states in SMC

External loopback transition	Internal loopback transition

`// State`
`Idle`
`  {`
`    // Trans Next State`
`    Actions`
`    Timeout  Idle  {}`
`}`
	`// State`
	`Idle`
	`{`
	`    // Trans Next State`
	`    Actions`
	`    Timeout    nil {}`
	`}`

Listing 19 illustrates the usage of *nil* state. In SMC, there are two ways to implement this reflexive transition; *external loopback transition* and *internal loopback transition*. In external loopback transitions, the exit and entry actions of the Idle state are executed, as well as transition actions, if any exists. On the other hand, in the case of the internal loopback transitions, only the transition action is executed. This feature in SMC is not directly supported Umple. However, this behavior can be simulated using composite states in Umple by having a reflexive transition

in an inner state. In Umple, we adopt UML semantics for the execution of entry and exit actions.

8.1.3 Comparison with Umple approach

There are core differences between Umple, Ruby and SMC. We summarize these differences as follows:

1. Language independence

Umple is language independent. Umple currently supports code generation for Java, C++, PhP and can easily be extended to support other languages. Neither Ruby nor SMC are language independent.

2. Readability

Umple's syntax is very similar to high level programming languages. We claim that Umple is more comprehensible than both Ruby and SMC syntaxes. Verification of this claim is left as a future work. However, one factor supporting our claim is that SMC's uses compiler directives to identify and parse state machine elements. The state machine becomes immediately less readable by humans. This is evident in the increased number of lines and reserved words.

3. Embedding of native code

Umple supports embedding of native code to implement various state machine elements. We actually like to say that Umple's syntax, for those modeling elements, happened to be identical to that of Java or PhP.

4. Support for associations and attributes

Umple state machine elements are well integrated with other UML modeling elements in Umple, such as associations and attributes.

5. Support for multiple state machines within the same class.

Umple supports unbounded number of state machines within the same class. See "Multiple state machines in the same class" on page 117.

6. Support for composite state machines

Support for composite state machine was discussed in the previous chapters. Ruby and SMC only support basic state machines.

7. The need for round tripping

One of Umple's objectives of supporting native code is to eliminate the need to edit the generated code. From Umple's perspective, the generated code is just like the byte code generated by other compilers. The need to edit the generated code is eliminated in Ruby, but not in SMC.

8.1.4 Specification and Description Language (SDL)

SDL is an object-oriented formal language defined by the International Telecommunications Union-Telecommunications Standardization Sector (ITU-T). The objective of the language is to model event-driven real-time systems. Particularly systems that involve extensive signals and communicating components.

SDL is designed to describe both the structure, data, and behaviour of real-time systems. The language eliminates ambiguities by having precise interpretation for each symbol. There are compilers that can generate high level programming language code for SDL. SDL has two semantically equivalent notations, a graphical representation (SDL-GR) and a phrase representation (SDR-PR). It is reported that the two equivalent notations enhance on clarity and ease of use.

In this section, we give an overview of the main features of SDL, and make a comparison with Umple's approach.

Parallelism

Parallelism is a key concept in SDL. Each process has its own memory and processing space. In other words, each process is an independent thread. A number of processes can be grouped to form blocks. A sub-system is a number of logically grouped blocks. This is SDL's approach to representing a system hierarchy.

Process instances can be created and terminated at run time. Each process instance has a unique process identifier (PId) allowing users of SDL to send signals and parameters to specific instances of a process.

Communication

Signals are the means of asynchronous communications between blocks. Signals can have optional parameters transmitted along with the signals. SDL also supports synchronous communications by means of remote procedure calls. For real time systems, time is a critical concept. SDL supports sophisticated timers and can measure and control response time of other processes.

8.1.5 Comparing Umple and SDL

SDL has emerged from the communications and signals domains. Umple, on the other hand, is a general purpose model oriented programming language. SDL provides better asynchronous support and better handling of parallelism in general. Let's consider the comparison at two levels, syntax and semantics.

Syntax

In this comparison, we consider the language coverage of modeling abstractions, terminology used in the notation and comprehension of the notation. Figure 50 is a simple example presented in SDL graphical notation and textual notation.

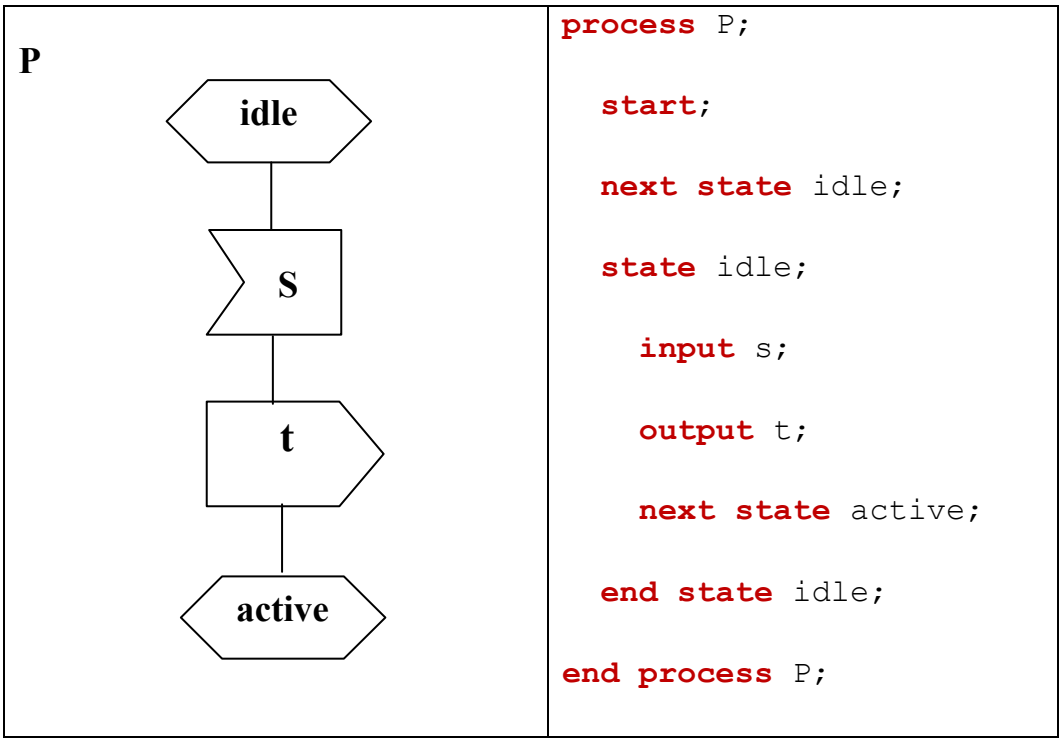
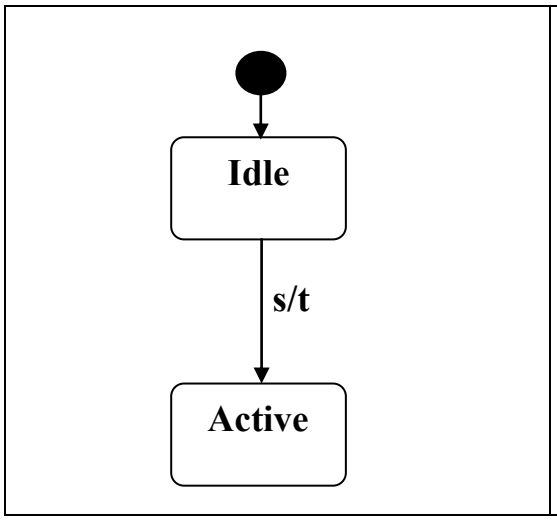


Figure 50: SDL graphical and textual notation

The same model is represented in UML and Umple as follows.




```
Status {
  Idle {
    s -> / {t} Active;
  }
  Active {
  }
}
```

Figure 51: UML and Umple notations

Coverage

Umple provides a textual notation for UML constructs that define state machine, attributes and associations. The textual notation is not intended to replace the UML visual language, rather, it is supposed to complement it. The development of SDL is similar with respect to this aspect of Umple. Both languages address the value added of the textual notation. In both cases, the textual notation complements the visual representation.

Many of the modeling notations in SDL are also available in Umple. One can argue that any model written in SDL can also be modelled in Umple. However, visa versa is not true. Umple provides better support for data types (attributes) and also for associations, which are not available in SDL.

Terminology

Because SDL has emerged from the communications domain, some of the terminology used is unique to SDL. For example, input and output in SDL are similar to UML's event and action. However, with the UML profiling becoming part of the UML standard, using the real-time profiles can address this issue. Umple, however, does not yet support profiles, but Umple user's can chose to name their modeling elements as they wish.

Comprehension

We can claim that Umple's notation is significantly better suited for human consumption and comprehension. As shown in the SDL and Umple textual notation examples, Umple uses significantly fewer key words, and a model can be represented in Umple using fewer lines and less overall text. In addition, Umple's notation is similar to modern high level programming languages, making it more familiar to more users.

Semantics

Here we compare the two languages based on platform independence, fitness for general-purpose programming, support for parallelism, support for timer-based computations and support for asynchronous computations.

Platform Independence

Both Umple and SDL claim to be a portable notation. Using OMG MDD concepts, both can be used as Platform Independent Models or PIMs. Umple, when purely modeling elements are included, is completely platform independent. Umple supports the embedding of target language code, a feature not supported in SDL. When such code elements are in the model, Umple becomes Platform Dependent.

General purpose programming and modeling

Umple is designed to be a general-purpose modeling and programming language. Umple syntax and semantics is most suited for general-purpose system development. SDL is most suitable for communications and real-time systems, and would probably not be a good choice for a web-based information system, for example.

Support for concurrency

Natively, the modeling notation in Umple supports concurrency using the do activity construct in state machines. SDL's native support for concurrent execution is superior. SDL's processes run in separate memories and can execute simultaneously on the same machine or in a distributed environments. While many communications systems can be implemented in Umple, the language itself does not provide much support 'out of the box' as compared with SDL. However, Umple users can rely on the target language for the creation and management of independent running threads.

Support for timer-based computations

Umple provides two types of timer that are also available in SDL. Timers in Umple are discussed in section "State machine timers" on page 67. SDL provides superior capabilities in supporting timers. In particular, it provides a mechanism to measure the time a remote running process has taken to process or respond to a specific signal. Such sophisticated time management in distributed systems has not been within the scope of the Umple language.

Asynchronous computations

A synchronous communications can be implemented in Umple using the do activity construct, or by using embedded target-language code, if such communications is supported in the target language. The rest of Umple's semantics for events and actions are synchronous. SDL supports synchronous communications natively by means of remote procedure calls. SDL is superior in its

support for asynchronous signals. A process in SDL can fire off a signal and continue execution as normal without caring whether such a signal was ever processed or received.

8.2 Standardization of execution semantics of UML

In this section, we introduce and discuss the latest UML action language standard. We also demonstrate the need for such languages, and show why existing high level programming languages cannot be used as an action language. Alf, The Action Language for Foundational UML [102], is the latest action language standard sponsored by OMG. We briefly introduce the language and compare it to Umple.

8.2.1 Background and Introduction

A UML action language (UAL) is geared towards describing elements of a system, such as actions, algorithms, and navigation paths, which are not readily described by typical UML diagramming notation. Snippets of languages like C++ and Java can be used as a UAL, but such languages are unaware of UML abstractions, resulting in mixed levels of abstractions and 'boilerplate' code.

OMG made available an RFP for a concrete syntax for a UML Action Language (UAL) [114]. Responses had to define a textual language for representing the UML subset defined in the Foundation Subset for executable UML Models (fUML) [115].

The OMG proposal required that the UAL be suitable for use in executable UML models. A proposed UAL had to meet a number of objectives including:

1. It must be computationally complete, meaning it must include standard arithmetic and logical capabilities supported natively or by the use of libraries.

2. It must allow the invocation of user-specified external code such as legacy code.

3. It must allow embedding of native code. For example, if the target platform is Java, the UAL should allow the embedding of java statements and constructs.

In November 2009, OMG published two proposals, one from IBM and one from Mentor Graphics. In mid-February 2010, the two proposals were consolidated into one to be called Action Language for Foundational UML (Alf). Alf version 1.0 – Beta-1 is the latest Alf standard published to date. In the rest of this section, and without loss of generality, we routinely refer to Alf as a representative UML Action Language.

8.2.2 Emergence of Action Languages

Action languages emerged to fill in the gap between the highly abstract (and visual) model notations to manage structure and relationships, with the more algorithmic manipulation of the model's structure (i.e. programming language-like-statements). This gap, commonly referred to as 'execution semantics', has not yet been completely formalized. Action languages can help both modelers and coders to achieve the following goals:

Define the execution semantics of models

Models, by definition, are an abstraction of a system, where some details are purposely left out. But to execute the model, missing details need to be defined using a Turing-complete language, so that all needed computations can be performed and so that executing the same model results in the same behavior. This is much like how different compilers for the same language should result in a system with the same behavior (although perhaps different machine language implementation and different characteristics like performance).

Express actions that natively interact with UML constructs.

UML introduces concepts that are more abstract than what is normally found in programming languages. This includes associations, state machines, preconditions, etc. A UML Action Language should define constructs that interact with, and fill in missing details of, such modeling constructs. For example, an Action Language should define statements to add or remove objects in an association, execute state machine actions, and define executable checks for pre- and post-conditions where appropriate.

Express algorithmic details in the most usable and maintainable way.

To support an executable modeling environment, the need to unambiguously define algorithmic computations is imperative. A UML Action Language should enable the modeler to define such algorithmic computations at a level of abstraction that is as high as possible and which builds on and complements modeling elements in a simple and elegant way.

Avoiding, or delaying, commitment to an execution platform.

The Action Language should allow modelers, and developers, to produce an executable system and, at the same time, to delay commitment to an execution platform. For example, a modeler should be able to define state machine actions in the UAL, and later in the development life cycle, a developer can choose to generate or embed Java code (or both), after committing to a Java execution platform. This is desirable in a model driven environments, where the same model may be eventually implemented on more than one execution platform.

Early verification and enhancement of reuse [116]

Because a UAL would be defined over an executable subset of UML, it must be possible to execute the UML models, along with the associated Action Language, early in the modeling activities. Modelers can then see an executable prototype of their system, and refine their modeling accordingly .

8.2.3 Why not use an existing programming or constraint language?

Reasons for not using an existing programming language can be summarized in the following three points. These mirror the points expressed by Mellor et al [116]:

Programming languages provide much more than what an Action Language needs

The java console I/O statements, and the variety of UI frameworks for Java are examples where the programming language is too powerful for what is needed from an action language. A Programming Language (such as Java) provides a large number of statements and libraries to accomplish the same or similar effect, which is to display output. Similarly, Java Programming Languages provides considerable freedom regarding how instance variables and methods can be used to represent and manipulate properties and relationships. The abstract UML concepts of attributes and associations therefore have many concrete mappings; when presented with implementation code, the software developer has a hard time seeing the abstractions. A UML action language can hence abstract the most commonly used concepts and make the algorithmic elements in models easier to understand.

Commitment to implementation

When programming an abstraction such as an association in a language like Java, one is forced to choose the low-level details, such as the names of methods and the algorithms. It is hard to change these later. As another example, when implementing a state machine one may choose to use a string attribute, but one may later on decide to change to an `enum` and hence have to change the code considerably. On the other hand, if using a UAL, this decision would be made by the compiler or code generator, and could be changed simply by changing the some configuration option.

Programming languages do not support directly UML concepts such as association or states

As mentioned, a language like Java does not have constructs for the representation of associations or state machines, and consequently does not promote abstract thinking on the part of programmers.

Declarative constraint languages, such as OCL, lack the support for algorithmic level specification. OCL-like languages do a good job in navigating associations and defining pre and post conditions, but generally, do not support effective implementation of algorithms.

8.2.4 Umple as an Action Language

The Umple approach to implementing a UML action language is distinct from the official OMG approach in three aspects. First, Umple makes a textual representation available for UML modeling elements and integrates the textual action language with the textual diagram representation. This is, as we have demonstrated, done without loss of the visual representation of UML models that can still be used along with the textual representation. Modelers can create and edit models diagrammatically or textually, and can embed the action language textually. This allows the modelers and the developers to reason uniformly about models and action language statements. Second, Umple's attempts to raise the abstraction level of the widely adopted programming languages to include modeling abstractions and action semantics, effectively overcomes limitations associated with programming languages used as action languages in UML models.

Looking at Umple as an action language, Umple raises the abstraction level of programming languages by availing the following language refinements (LRs):

L.R-1. Make available additional, and more abstract, language constructs

L.R-2. Restrict and modify statements so they become language independent

Thirdly, Umple provides native support for class and state machine abstractions. Current action language approaches address class diagram abstractions only. An action language statement should interact with abstractions from both class and state machine abstractions. For example, action language statements that define the life cycle of an object may be more effectively represented by a state machine diagram.

8.2.5 Overcoming limitations with existing programming languages

Umple addresses the limitations in programming languages for use as an action language as follows:

Programming languages provide much more than what an Action Language needs.

This limitation can be easily overcome in Umple by limiting the scope of the programming language into the subset required in the action language. Currently, Umple supports all statements within the supported languages. For example, a state machine action or a guard condition can be any valid statement or a function call.

Commitment to implementation.

Umple does not require the programmer to implement many abstract concepts; as in ordinary compilers, the many implementation decisions are left to the compiler designers. The compiler will select a suitable implementation based on the target environment.

Take for example a for loop in a typical high-level language compiler. The for loop is implemented in a machine language in a number of different ways, all are deemed acceptable as long as the semantics of the for loop is maintained. Taking the same concept to the modeling abstraction, consider a state machine. There are a variety of approaches to the implementation of state machine behavior (discussed in Chapter 2: Background); from an action language perspective, all are acceptable as long as the semantics of the state machine is maintained.

Programming languages do not support directly UML concepts such as association or states.

This is one core aspect of Umple. Umple makes available those UML constructs in the language itself. This becomes evident when we present the language syntax.

Declarative and constraint languages, such as OCL, lack the support for algorithmic level specification.

Because Umple is based on object-oriented programming languages, this limitation is not applicable to Umple. In addition, Umple supports aspects of the OCL, an example being pre and post conditions. Discussion of pre and post conditions are not within scope of this thesis. The reader can refer to the Umple open source project for additional information [7].

8.2.6 Comparison between Umple and UAL

Table 29 presents a summary of our interpretation of OMG's stated objective of UAL. The table also illustrates how Umple addresses these objectives. The listed objectives are only a brief summary of the stated UAL objectives. The reader is referred to [114] for information on OMG RFP.

Table 29: Objective of UAL standard

	The new standard objectives	Umple Position
1	Define a concrete textual syntax so that modelers can use text in executable UML models.	Umple is a concrete syntax since we are able to unambiguously generate executable artifacts.
2	Define a computationally complete language.	By allowing for native code, Umple is computationally complete.
3	Reuse existing OMG language specifications where possible.	Umple is based semantically on UML 2. We are able to generate UML diagrams from Umple.
4	Provide a mapping from statements in the concrete syntax to the foundational subset of actions in the Executable UML Foundation.	Umple achieves objectives 4, 5, 6,7, and 8 by delegating to high-level language. Umple does not make any assumptions, or dependencies, on the target executable language. Developers can use capabilities provided by the target language, or the platform, to accomplish input/output operations, interfacing with other packages, and use the target language arithmetic support. With regard to objective 4, Umple does not generally, with minor exceptions, provide mapping since Umple does not support a concrete syntax separate from the base programming language. We are investigating adding such a concrete syntax in state machine actions code.
5	Provide mechanisms for input/output that map to the corresponding input/output constructs specified in the Executable UML Foundation.	
6	Provide mechanisms for interfacing to user-specified input/output packages.	
7	Include standard arithmetic and logical capabilities	
8	Include mechanisms to invoke user-specified operations.	
9	Include a notation for comments.	Umple uses the widely adopted comments notation; // for line comments and /* */ for multi line comments.
10	Have a mechanism for transferring control to non-UML “programs”.	This is supported since arbitrary methods can be embedded in Umple, and these can call any code.
11	Allow inline embedding of target programming code.	The IBM and *MentorGraphics* proposal does not address native code embedding. This is one powerful aspect of Umple that enables the users to accomplish a significant portion of the UAL objectives using native code embedding.

Continued Table 29: Objective of UAL standard

12	Define a label so that modelers not fully committed to an action language may embed action language fragments in Opaque Expressions.	Umple does not support labeling. Native code is not labeled. We may provide labels in the case we support a concrete syntax, in addition to native code. This is still under investigation.
13	Define extension points	Umple supports extension points and libraries as much as the native code, or platform, supports it.
14	Support for libraries	

8.3 Summary

This chapter discussed related work; namely, work related to textual modeling and work related to code generation from state machine models. We have chosen to discuss state machines in Ruby and State Machine Compiler in greater depth. These two approaches bear some similarities to the state machine in Umple, but there are a few core differences. In particular, state machines in Umple are language-independent. We also claim that Umple is designed with readability and comprehensibility in mind. The state machines in SMC for example use compiler directives that have a significant negative impact on readability. Another core difference is the tight integration in Umple between state machine elements and associations and attributes. For example, a state machine action can add a member to one side of an association. Umple supports multiple state machines in the same class, and supports the reusing of actions within the same state machine, and across multiple state machines. Umple has an extensive support for composite state machines. At the time of writing, SMC had no support for composite state machines, and Ruby supported only the concept of a parent state, or a super state.

Object Management Group is sponsoring an emerging standard to standardize the execution semantics of a subset of UML. This chapter presented Alf, the latest OMG standard that defines an executable subset of UML. We discussed the need for Action Languages, and explained why the existing high level programming languages cannot be used in lieu of an action language. We also explained how Umple overcomes the existing limitations of action languages. Finally, we compared Umple to the objectives of action languages as stated in OMG RFP.

Chapter 9: Summary and conclusion

The evolution of human language [117] exhibits an increasing level of abstraction; new words emerge to facilitate the effective representation of more complex situations and concepts. More recently, the Oxford dictionary has accepted new words such as Google. Google means "*search for information about (someone or something) on the Internet, typically using the search engine Google"*. There is no evidence yet to indicate that this trend will come to an end.

The financial sector has witness a similar trend of ever increasing levels of complexity with new concepts emerging along with new terminology to represent such complex financial instruments. To name one, a *derivative*, defined to be "*A security whose price is dependent upon or derived from one or more underlying assets"*. Derivatives enabled some financial institutions to hide the real risks associated with some financial assets.

Biological evolution exhibits a similar trend. More complex organisms emerge that are more capable of manipulating the environment or defending themselves.

Across all human activates, it seems that more complexity keeps on emerging as does the means to deal with such complexity. The story of this ever-increasing complexity is well described in Wright's book "Non zero. The logic of human destiny." [118].

The same argument can be applied to programming languages. At the beginning, it was all zeros and ones. Assembly language was introduced to facilitate the reading and comprehension of instructions. Constructs such as loops can be represented in an assembly language using go-to statements, but these are hard to read and maintain. Sophisticated compilers emerged that take procedural programming languages with abstract constructs as input and automatically generate the assembly code. Object orientation abstracts entities and encapsulates them in classes with procedures and attributes.

We like to think of Umple as the inevitable next level of abstraction in programming languages after object orientation. We therefore use the term '*Model-Oriented Programming Language'* to describe Umple. Our view is that many UML modeling elements, which are typically more abstract than what is available in today's high level programming languages, can be effectively represented textually. In fact, our view is that both model and code are just two manifestations of the same underlying concepts. Therefore, a system can be equally specified textually or diagrammatically. In this thesis, we have sufficiently proved that this is the case for most of the UML state machine concepts.

If reality does support such a vision, we would expect that programming languages to continue to incorporate ever more abstract concepts. We can speculate that next generation programming

languages to fully support many of the existing UML abstractions, and future languages to incorporate system abstractions, maybe to be drawn from SysML [119]. Programmers may then one day program a system using a super-high programming language, and then apply domain specifications, and be able to quickly test and refine the system.

In this thesis, we have presented our research work that brought state machine abstractions to Umple. We have demonstrated how some of the visual abstractions of state machine modeling can be effectively represented textually using a programming-like syntax.

We outlined the hypothesis and the research direction in "Chapter 1: Introduction". We presented background research and investigation of existing state machine technologies in "Chapter 2: Background".

The core of our experimental development is presented in chapters three, four, and five. These three chapters together present the implementation of state machines in the Umple technology.

The evaluation of our work is the topic of chapters six and seven. Chapter six presented a grounded theory study of Umple users; chapter seven presented a controlled experiment. We then presented the related work in Chapter 8.

The main contributions this thesis makes can be summarized in the following points:

- Incorporating a textual representation of most of the state machine abstractions into a high-level-programming-like language where software developers (programmers and modelers) can utilize to build complete systems.

- Introducing an interpretation and an implementation of the latest UML state machine specifications. Exposing some of the ambiguities in the latest specifications and providing an alternative for dealing with such ambiguities.

- Presenting an implementation for composite state machine code generation that avoids explosion of the generated code.

- Providing a Model Driven Development environment where model and code are united; where the need to modify the generated code is minimized or eliminated. An environment where the generated artifacts are similar to those that are written by hand.

- Presenting an empirical evaluation for Umple that takes a human perspective on the comprehensibility of such a technology.

Future work with Umple can be summarized as follows:

- Work to integrate model-level tracing. A textual modeling language like Umple can be extended to include tracing at the modeling level. This direction is being investigated by a PhD student.

- Support for an event queue. Currently, the state machine implementation in Umple consumes events as they become available. This can cause problems when two events occur and the processing of the second event starts before the processing of the first event is completed (e.g. because action code triggered by the first event directly or indirectly triggers the second event, or because of code executing in concurrent threads). The solution for this can be achieved by the implementation of an event queue from which events ready for processing can be processed in FIFO sequence.

- Support for firing of events when a do activity completes. The current implementation in Umple is such that nothing happens when a do activity execution is completed. One useful approach would be to take a certain transition when this occurs.

- Work to enhance the Umple language with additional modeling abstractions possible from other UML modeling notations, such as sequence diagrams.

- Enhancements to address communications protocols and thread safety. Umple's semantics are mainly synchronous. Additional future work would involve providing better handling of signals and asynchronous communication semantics.

- Enhancements to support state machine inheritance, as for example found in tools such as IBM RSA-RT. Umple's ability to add or delete elements of state machines using mixins would also be used in this context.

- Empirical evaluation of larger systems built in Umple. The evaluation can involve real life systems and professional software engineers. This work requires some enhancement to the Umple platform to make it suitable for an industrial level system production.

Glossary

Action Semantics: The specification of operations and their interactions in a modeling language. Action Semantics was added to the UML specification in 2001.

BNF Grammar (Backus–Naur Form): A notation used to describe the syntax of languages used in computing, such as computer programming languages.

Church's Thesis: (also known as the Church–Turing conjecture, Church's thesis, Church's conjecture, and Turing's thesis) is a combined hypothesis about the nature of functions whose values are effectively calculable; in more modern terms, algorithmically computable. In simple terms, it states that "everything algorithmically computable is computable by a Turing machine.

Composite state machine: States that contain other states are called composite. In UML and Umple, composite state machines are either concurrent or nested.

Compress-Flatten Code Generation (CFCG): An approach to flatten composite state machines that avoids the explosion of the generated code.

Do Activity: In UML state diagrams, a do activity represents a long running computation; i.e a computation that is expected to take a long period of time. If present, it is initiated in a separate thread upon entry into a state.

Extended-multiple-class pattern: A state machine code generation pattern where more state machine elements are implemented in dedicated classes than in the multiple-class pattern. See also Multiple-class pattern and In-class pattern.

Event: A state machine event is any occurrence that the state machine responds to if it is in a state with an outgoing transition labeled with that event. Events trigger state machine transitions and may have actions associated with them.

Explicit transition: A transition that is explicitly defined in the state machine model. See also implicit transition.

Final state: A special kind of state signifying that the enclosing region is completed. In Umple, any transition to a Final state causes the object to be deleted.

Grounded Theory (GT): A systematic qualitative research methodology, originating in the social sciences, that emphasizes the generation of theory from qualitative data in the process of conducting research. Grounded theory, in its original form, was proposed by Glasser and Strauss in 1967.

Implicit transition: A transition that is not explicitly defined in the state machine model. In Umple, a transition to the start state is always implicit. See also explicit transition.

In-class state pattern: A code generation pattern for state machines where all the state machine elements are implemented in a single class. See also Multiple-class pattern and Extended-multiple-class pattern. Umple uses this pattern.

Mealy and Moore state machines: Mealy and Moore state machines are the basic models of state machines. A state machine whose only actions are entry actions is called a Moore machine. A state machine that instead relies on transition actions is called a Mealy machine.

Meta-Model: A model of models; a modeling methodology whereby a model is used to specify what qualifies to be a valid model conforming to a specific meta-model.

Model-Code Duality: A notion or a view that code is model and model is code. Both are a representation of an underlying system.

Model-Driven Development (MDD): Also known as Model Driven Software Development (MDSD), Model Driven Engineering (MDE), and Model Driven Architecture (MDA). It is a software development methodology that puts models as the main development artifact, and not just as documentation.

Multiple-class pattern: A code generation pattern for state machine where some of the state machine elements are implemented in a dedicated class. See also In-class pattern and Extended-multiple-class pattern.

Simple state machine: A state machine where states do not contain other states. See also Composite state machine.

Start state: In a state machine model, the first state that is active upon the creation of the object (or upon entry into a sub-state in the case of a composite state machine) is the start state. Umple by default specifies that the state which definition comes first is the start state.

State machine: A model defining the behavior of an entity based on a finite number of states, transitions between those states based on events, and the actions or activities that occur in the system as the entity changes states (e.g. entry and exit actions, or activities while in a state).

State Pattern: The state pattern is a design pattern that allows an object to change its behavior depending upon its current internal state. The state pattern is useful when creating object-oriented state machines, where the functionality of an object changes fundamentally according to its state.

Test Driven Development (TDD): A software development process that relies on iterative and short development cycles and where tests are written first to specify what must be developed. The developer writes a failing automated test case that defines a desired improvement or new

function, then produces code to pass that test. Test cases can also be derived or generated from the user requirements or use cases.

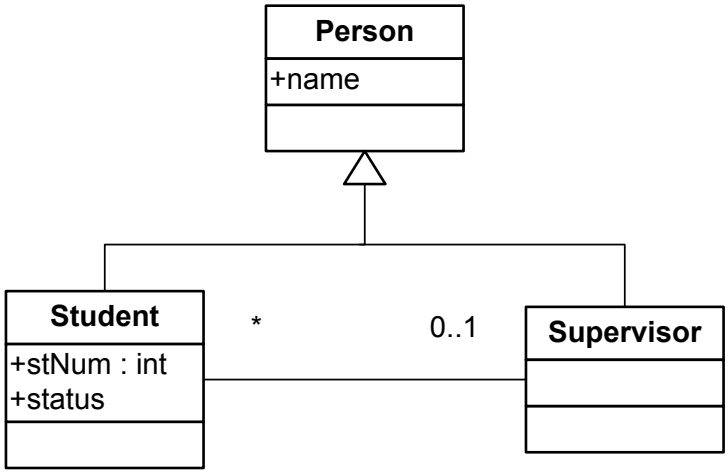
Unified Modeling Language (UML): A standardized general-purpose modeling language that is managed by the Object Management Group.

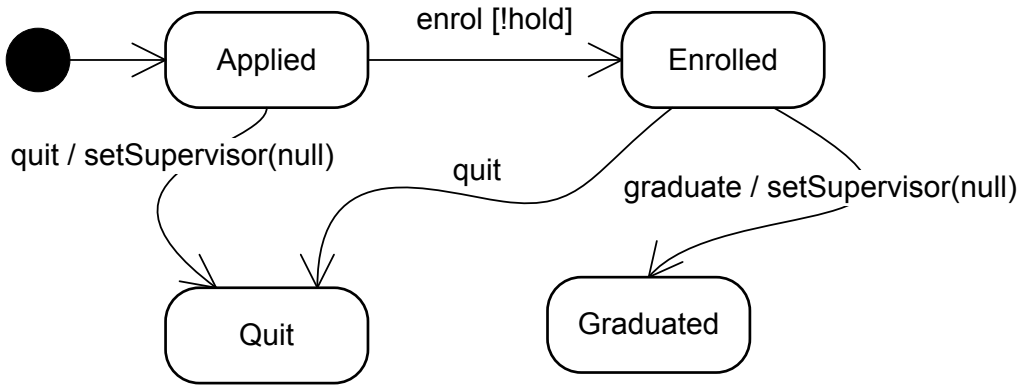
Umple: A model-oriented programming technology that adds additional modeling elements as first-class entities on top of existing object-oriented languages like Java, PHP and Ruby. Umple supports class diagram entities like associations, attributes and multiplicity, it supports state machine entities like states, events, transitions as well as software patterns like singleton, equality, software mix-ins and aspect-oriented code weaving.

Xtext: A framework for building programming languages and domain specific language (DSL) editors. Xtext is meant to be the EMF for IDEs.

Appendix

A.1 Example System One (UML)





A.2 Example System One (Umple)

Umple

```
class Person {
  name;
}

class Student {
  isA Person;

  Integer stNum;

  status {
    Applied {
      quit -> Quit;
      enrol [hold] -> Enrolled;
    }
    Enrolled {
      quit -> /{setSupervisor(null);}
        Quit;
      graduate -> /{setSupervisor(null);} Graduated;
    }
    Graduated {}
    Quit {}
  }

  * -- 0..1 Supervisor;
}

class Supervisor {
  isA Person;
}
```

A.3 Example System One (JAVA)

Student

```
class Student extends Person {
   private int stNum;

   boolean hold;
   private int status;
   private Supervisor mySupervisor;

   public Student(int stNum) {
     this.stNum= stNum;
     status=0;
    }

   public int stNum() {return stNum;}

   public void enrol() {
     if (!hold){
       if(status ==0) status=1;}
  }
   public void graduate() {
      if(status==1) {
         removeSupervisor();
         status=2;
       }
    }

   public void quit() {removeSupervisor(); status=3;}
   public boolean setSupervisor(Supervisor newSupervisor) { }
   public boolean removeSupervisor() { }
 }
```

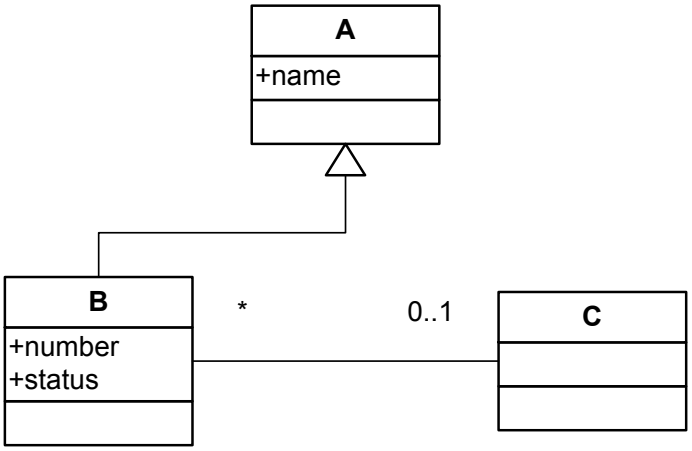
Person

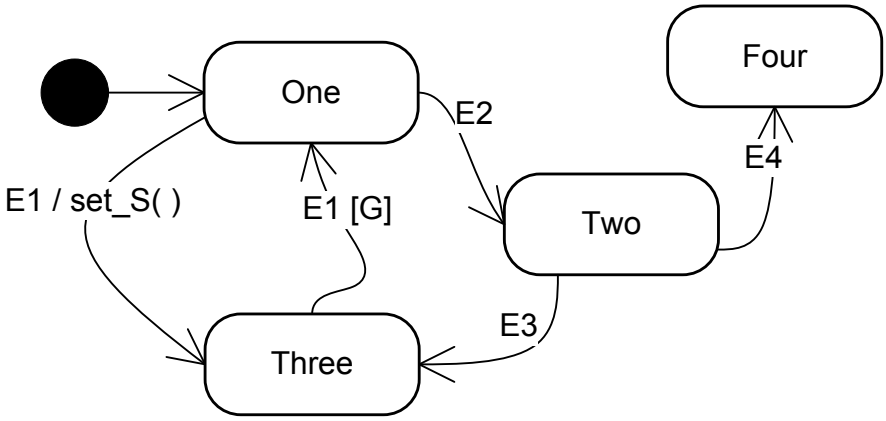
```
class Person {
  public String name;
}
```

Supervisor

```
class Supervisor extends Person {
   List<Student> mentees = new
ArrayList<Student>();

   Supervisor() {}
}
```

A.4 Example System Two (UML)





A.5 Example System Two (Umple)

Umple

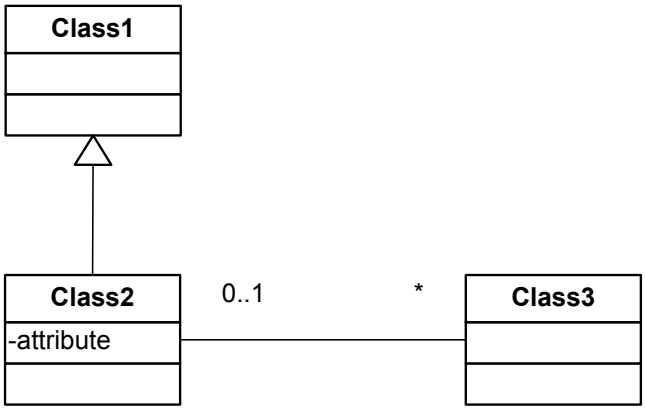
```
class A {
  name;
}

class B {
  isA A;
  * -- 0..1 C;

  Integer number;

  status {
    One {
      E1 -> /{set_S();} Three;
      E2 -> Two;
    }
    Two {
      E4 -> Four;
    }
    Three {
      E1 [G] -> One;
    }
    Four { }
  }
}

class C { }
```

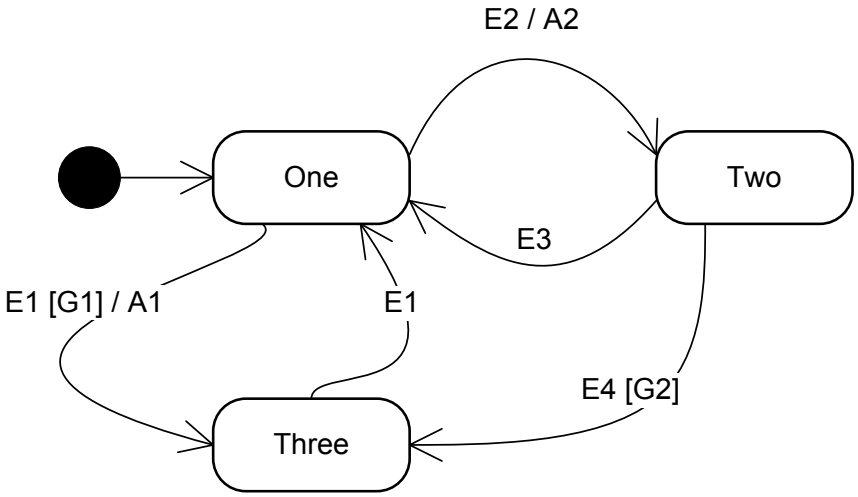
A.6 Example System Two (JAVA)

Class B

```
class B extends A {
   public int number;

   private int status;
   private C myC;

   public B(int number) {
     this.number = number;
     status = 1;
    }
   public int number() {return number;}

   public void E1() {
     if(status == 1) {
       set_S();
       status = 3;
     }
     if(status == 3 {
       if (G)  {privateInt_2 = 3;}
     }
  }
   public void E2() {
     if(status == 1) {status = 2;}
    }

   public void E3() {
     if (status == 2) status = 3;
    }

   public void E4() {
     if (status == 2) status = 4;
    }

   public boolean setC(C newC) { }
   public boolean removeC() { } }
```

A.7 Example System Three (UML)





A.8 Example System Three (Umple)

Umple

```
class class1 {
}

class class2 {
  isA Person;

  attribute;

  stateMachine {
    StateOne {
      E1 -> / {A1} StateThree;
      E2 -> / {A2} StateTwo;
    }
    StateTwo {
      E3 -> / {A3} StateOne;
      E4 -> / {A4} StateThree;
    }
    StateThree {
      E5 -> / {A5} StateOne;
    }
  }

  * -- 0..1 Supervisor;
}

class class 3 {
  isA class1;
}
```

A.9 Example System Three (JAVA)

Class2

```
class Class2 extends Class1 {

   private int attribute;
      List<Class3> role = new ArrayList<class3>();

   public Class2(int attribute) {
     this.attribute = 0;
   }

   public void E1() {
     A1();
     if(attribute ==1) attribute =3; }

   public void E2() {
     A2();
     if(attribute ==1) attribute =2; }

   public void E3() {
     A1();
     if(attribute ==2) attribute =1; }

   public void E4() {
     A1();
     if(attribute ==2) attribute =3; }

   public void E5() {
     A1();
     if(attribute ==3) attribute =1; }
     }
   }

   public void A1(){  }
   public void A2(){  }
   public void A3(){  }
   public void A4(){  }
   public void A5(){  }
}
```

Class1

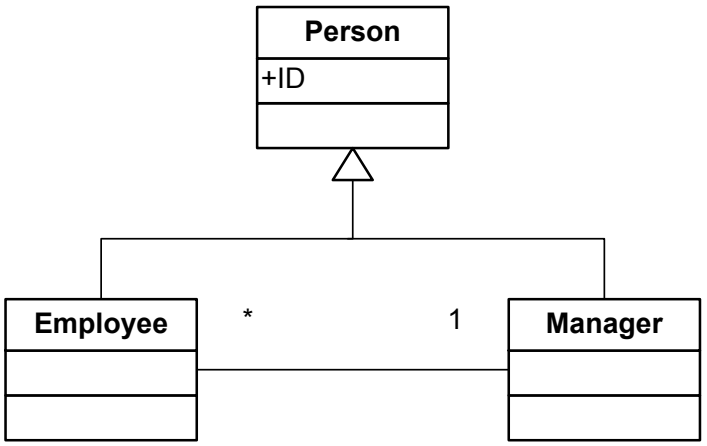
```
class Class1 { }
```

Class3

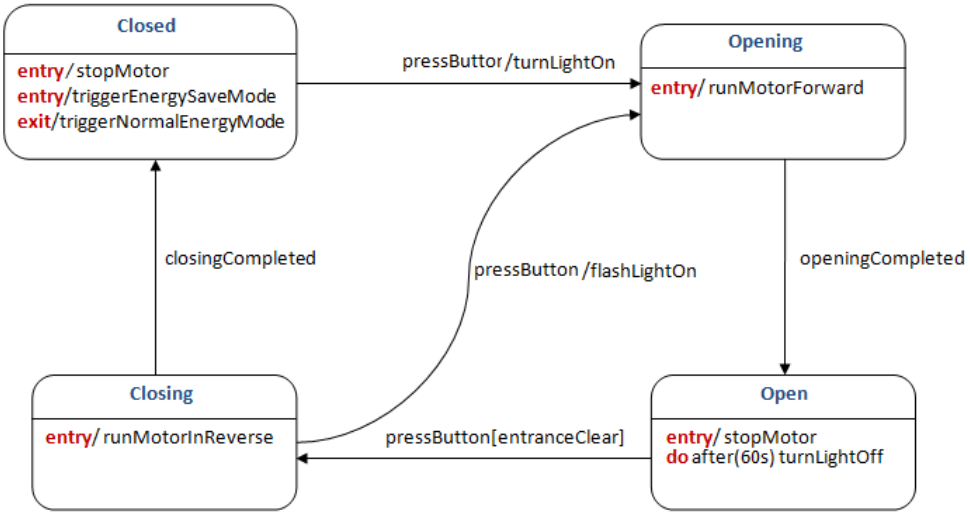
```
import java.util.*;

class Class3 extends Class1 {
  private Class2 myClass2;
  Class3() {}
}
```

A.10 Training Example One (Classes, attributes, Associations)

UML	Umple

	`class Person {`
	`  ID;`
	`}`
	
	`class Manager {`
	`  isA Person;`
	`  1 -- * Employee;`
	`}`
	
	`class Employee {`
	`  isA Person;`
	`}`

A.11 Training Example 2 (State Machines)



A.12 Question list for example system one

Table 30: Question list for version E1 (UML and Umple)

Questions for Version E1		
Umple and UML questions		**Correct Answer**
Q1	Let's assume the state machine is in the Applied state and hold is false. Also assume the following events occurred in sequence, enrol, quit, enrol. What is the resulting state?	Quit.
Q2	Assume the student has one supervisor. Can you add another supervisor to the same student?	No.
Q3	Assume a supervisor has 6 students. Can we add another student to this supervisor?	Yes.
Q4	Assume the state machine is in the Applied state, and the value of hold is true. What happens when the event enrol occurs?	Nothing. Not transition occurs.
Q5	How many students can a supervisor have?	Many. Unlimited number.
Q6	What are the possible states the state machine status can have?	Applied, Enrolled, Graduated, and Quit. (in any order)
Q7	What actions are called when the following transition occurs : From Applied to Enrolled	Nothing. No actions are called.
Q8	Can the state machine go directly from Quit to Enrolled?	No.
Q9	Can the state machine go from Graduated to Applied?	No.
Q10	Assume we are in the Applied state, what happens when the event graduate occurs?	Nothing.
Q11	Can you create a Person Object?	No.
Q12	Assume the state machine is in the Applied state and that hold is false. Also assume the following events occur in sequence: graduate, quit, quit, enrol. What is the resulting state?	Quit.

Table 31: Question list for version E1 (Java)

	Questions for Version E1	
	Java questions	**Correct Answer**
Q1	Let's assume the variable status equals zero, and the following functions are called in sequence, enrol(), quit(), enrol(). What is the value of the variable status?	Two.
Q2	Assume the student has one supervisor. Can you add another supervisor to the same student?	No.
Q3	Assume a supervisor has 6 students. Can we add another student to this supervisor?	Yes.
Q4	Assume the value of status equals zero, and the value of hold is true. What happens when the function enrol() is called?	Nothing.
Q5	How many students can a supervisor have?	Many. Unlimited number.
Q6	What are the possible values the variable status can have?	Zero, One, Two, Three, and Four. (in any order)
Q7	What functions are called when the value of the variable status goes from zero to one?	Nothing.
Q8	Can the value of the variable status go directly from two to one?	No.
Q9	Can the value of the variable status go from three to zero?	No.
Q10	Assume the value of status equals zero, what happens when the function graduate() is called?	Nothing.
Q11	Can you create a Person Object?	No.
Q12	Assume the value of status equals to zero. Also assume the following functions are called in sequence: graduate(), quit(), quit(), enrol(). What is the resulting value of the variable status?	Two.

A.13 Question list for example system two

Table 32: Question list for version E2 (UML and Umple)

	Questions for Version E1	
	Umple and UML questions	**Correct Answer**
Q1	Let's assume the state machine is "One" state and G is false. Also assume the following events occurred in sequence, E1, E1, E2. What is the resulting state?	Three.
Q2	Assume B has one instance of C. Can you add another instance of C to B?	No.
Q3	Assume C has 6 instances of B. Can we add another instance of B to C?	Yes.
Q4	Assume the state machine is in the "One" state, and the value of G is true. What happens when E3 occurs?	Nothing. Not transition occurs.
Q5	How many instances of B can be associated with C?	Many. Unlimited number.
Q6	What are the possible states the state machine can have?	One, Two, Three, Four. (in any order)
Q7	What actions are called when the following transition occurs : From One to Three.	Set_S()
Q8	Can the state machine go directly from One to Four?	No.
Q9	Can the state machine go from Two to One?	No.
Q10	Assume we are in the One state, what happens when the event E3 occurs?	Nothing.
Q11	Can you create an A Object?	No.
Q12	Assume the state machine is in the One state. Also assume the following events occur in sequence: E2, E3, E3, E1. What is the resulting state?	One.

Table 33: Question list for version E2 (Java)

	Questions for Version E1	
	Umple and UML questions	**Correct Answer**
Q1	Let's assume the variable status equals to 1, and G is zero. Also assume the following functions are called in sequence, E1(), E1(), E2(). What is the value of the variable status?	Three.
Q2	Assume B has one instance of C. Can you add another instance of C to B?	No.
Q3	Assume C has 6 instances of B. Can we add another instance of B to C?	Yes.
Q4	Assume the status variable equals 1, and the value of G is 1. What happens when the function E3() is called?	Nothing. Not transition occurs.
Q5	How many instances of B can be associated with C?	Many. Unlimited number.
Q6	What are the possible values the variable status can have?	1, 2, 3, 4. (in any order)
Q7	When the variable status's value changes from 1 to 3, what function gets called?	Set_S()
Q8	Can the variable status value change directly from 1 to 4?	No.
Q9	Can the variable status's value change from 2 to 1?	No.
Q10	Assume the value of status is 1, what happens when the function E3() is called?	Nothing.
Q11	Can you create an A Object?	No.
Q12	Assume the value of the variable status is 1, and G is false. Also assume the following functions are called in sequence: E2(), E3(), E3(), E1(). What is the resulting value of the variable status?	1.

A.14 Question list for example system three

Table 34: Question list for version E3 (UML and Umple)

	Questions for Version E1	
	Umple and UML questions	**Correct Answer**
Q1	Let's assume the state machine is in the One state and G1 is false. Also assume the following events occurred in sequence, E1, E1, E1. What is the resulting state?	One.
Q2	Assume the Class3 has one Class2. Can you add another Class2 to the same Class3?	No.
Q3	Assume a Class2 has 6 instances of Class3. Can we add another Class3 to this Class2?	Yes.
Q4	Assume the state machine is in the One state, and the value of G2 is true. What happens when the event E4 occurs?	Nothing. No transition occurs.
Q5	How many Class3 can a Class2 have?	Many. Unlimited number.
Q6	What are the possible states the state machine can have?	One, Two, and Three. (in any order)
Q7	What actions are called when the following transition occurs : From Two to One	Nothing. No actions are called.
Q8	Can the state machine go directly from Three to Two?	No.
Q9	Can the state machine go from Two to Three?	Yes.
Q10	Assume we are in the One state, what happens when the event E3 occurs?	Nothing.
Q11	Can you create a Class1 Object?	No.
Q12	Assume the state machine is in the One state. Also assume the following events occur in sequence: E2, E3, E3, E2. What is the resulting state?	Two.

Table 35: Question list for version E3 (Java)

	Questions for Version E1	
	Umple and UML questions	**Correct Answer**
Q1	Let's assume the attribute status equals to 1. Also assume the following functions are called in sequence, E1(), E1(), E1(). What is the resulting value of status?	Three.
Q2	Assume the Class3 has one Class2. Can you add another Class2 to the same Class3?	No.
Q3	Assume a Class2 has 6 instances of Class3. Can we add another Class3 to this Class2?	Yes.
Q4	Assume the value of the variable status equals to 1. What happens when the function E4() is called?	Nothing. No transition occurs.
Q5	How many Class3 can a Class2 have?	Many. Unlimited number.
Q6	What are the possible values the status variable can have?	One, Two, and Three. (in any order) (also possible zero since the code initialize to zero.)
Q7	What methods are called when the value of the variable status changes from 2 to 1?	Nothing. No functions are called.
Q8	Can the value of te attribute status changes from Three to Two?	No.
Q9	Can the value of the variable status change from 3 to 1?	Yes.
Q10	Assume variable status is equal to 1, what happens when the function E3() is called?	A1.
Q11	Can you create a Class1 Object? (be flexible here)	No.
Q12	Assume the value of the attribute status equals to 1. Also assume the following methods are called in sequence: E2(), E3(), E3(), E2(). What is the resulting value of the variable status?	Two.

References

[1] Dyson, P. and Anderson, B. "State Patterns". 1998. *Pattern languages of program design,* vol 3, Addison-Wesley Longman Inc.

[2] Forward, A. " Computer Science PhD Thesis, Appendices, and Supplementary Material", accessed 2011, http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/.

[3] Forward, A. " Umple Language Online.", accessed 2012, http://try.umple.org.

[4] Henry Babbage. *Babbage's Calculating Engines.* Massachusets, U.S.A: Tomash, mitpress.mit.edu, 1982.

[5] Forward, A., Badreddin, O. and Lethbridge, T. C. "Perceptions of Software Modeling: A Survey of Software Practitioners," in *5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M:EEMDD),* 2010. Available: http://www.esi.es/modelplex/c2m/papers.php

[6] Skorkin, A. " Why Developers Never use State Machines", accessed 2011, http://www.skorks.com/2011/09/why-developers-never-use-state-machines/.

[7] Lethbridge, T. C., Forward, A. and Badreddin, O. "Umple Google Code Project". 2012. Available: code.umple.org

[8] Hodges, A. *Alan Turing: The Enigma.* ACM: Walker & Company; First Edition, 2000.

[9] Cohen, D. I. A. *Introduction to Computer Theory.* New York: Prentice-Hall, Second Edition, 1997.

[10] Wagner, F., Schmuki, R., Wagner, T. and Wolstenholme, P. *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach Publications, 2006.

[11] Börger, E. and Stärk, R. F. *Abstract State Machines: A Method for High-Level System Design and Analysis.* New York: Springer Verlag, 2003.

[12] Börger, E. "The Origins and the Development of the ASM Method for High Level System Design and Analysis". 2002. *Journal of Universal Computer Science,* vol 8, pp. 2-74.

[13] G. v. Bochmann, G. Gerber and J.M. Scrre. "Abstract State Machine Semantics of SDL". 1989. *IEEE Transactions Software Engineering.,* pp. 989-1000.

[14] SDL Forum Society. " Towards SDL-2010", accessed 2010, http://www.sdl-forum.org/ftp/pub/SDL-2010/index.htm.

[15] Piefel, M. and Scheidgen, M. "Modelling SDL, Modelling Languages," in *Cybernetics and Information Technologies, Systems and Applications (CITSA),* 2006. pp. 298.

[16] NTNU. " SDL-2000", accessed 2010, http://www.item.ntnu.no/fag/ttm4115/sdl-2000.htm.

[17] Harel, D. "Statecharts: A Visual Formalism for Complex Systems". 1987. *Science of computer programming,* vol 8, Department of Applied Mathematics, Weizmann Institute of Science. pp. 231-274.

[18] Douglass, B. P. "UML Statecharts". 1999. *Embedded systems programming,* vol 12, pp. 22-42.

[19] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M. and Sherman, R. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," in *Proceedings of the 10th International Conference on Software Engineering,* 1988. pp. 396-406.

[20] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[21] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1991.

[22] Booch, G., Rumbaugh, J. and Jacobson, I. "The Unified Modeling Language". 1996. *Unix Review,* vol 14, pp. 5.

[23] OMG. " UML Specifications", accessed 2011, http://www.omg.org/spec/UML/.

[24] Adamczyk, P. "Selected Patterns for Implementing Finite State Machines," in *The 11th Conf. on Pattern Languages of Programs,* 2004.

[25] Michael J. Blechar, "Magic Quadrant for OOA&D Tools, 2H06 to 1H07". Gartner Inc., Tech. Rep. G00140111, 30 May 2006,

[26] Norton, D., "Open-Source Modeling Tools Maturing, but Need Time to Reach Full Potential". Gartner, Inc., Tech. Rep. G00146580, 20 April 2007, 2007.

[27] Telelogic. " Rhapsody", accessed 2012, http://modeling.telelogic.com/.

[28] Mentor Graphics Corporation. " Mentor Graphics BridgePoint", accessed 2012, http://www.mentor.com/products/sm/model_development/bridgepoint/.

[29] Anonymous " Borland Together for Eclipse", accessed 2012, http://www.borland.com/us/products/together/index.html.

[30] Anonymous " SmartState", accessed 2012, http://www.smartstatestudio.com/.

[31] Pavel Bekkerman. "FSMGenerator". [OpenSource]. 2002-2003. Available: http://fsmgenerator.sourceforge.net/

[32] Croll, P., Duval, P. Y., Jones, R., Kolos, S., Sari, R. and Wheeler, S. "Use of Statecharts in the Modelling of Dynamic Behaviour in theATLAS DAQ Prototype-1". 1998. *IEEE Trans.Nucl.Sci.,* vol 45, pp. 1983-1988.

[33] Knapp, A. and Merz, S. "Model Checking and Code Generation for UML State Machines and Collaborations," in *Proceedings of 5th Workshop on Tools for System Design and Verification, Technical Report,* 2002. pp. 59–64.

[34] Knapp, A., Merz, S. and Rauh, C. "Model Checking-Timed UML State Machines and Collaborations". 2002. *Lecture Notes in Computer Science,* Springer. pp. 395-416.

[35] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* New Jersey: Addison-Wesley Reading, MA, 1995.

[36] Van Gurp, J. and Bosch, J. "On the Implementation of Finite State Machines," in *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications,* 1999. pp. 172-178.

[37] Yacoub, S. M. and Ammar, H. H. "A Pattern Language of Statecharts," in *Proc. Fifth Annual Conf. on the Pattern Languages of Program (PLoP'98),* 1998. pp. 98-29.

[38] Tomura, T., Kanai, S., Uchiro, K. and Yamamoto, S. "Object-Oriented Design Pattern Approach for Modeling and Simulating Open Distributed Control System," in *IEEE International Conference on Robotics and Automation,* 2001. pp. 211-216.

[39] Mohan, M. A. "Spatial Complexity Metrics: An Investigation of Utility". 2005. *IEEE Trans.Softw.Eng. vol 31, pp. 203-212,*

[40] Gold, N., Mohan, A. and Layzell, P. "Spatial Complexity Metrics: An Investigation of Utility". 2005. *IEEE Trans.Software Eng.,* vol 31, pp. 203-212.

[41] Stevens, W., Myers, G. and Constantine, L. *Structured Design.* Yourdon Press Upper Saddle River, NJ, USA, 1979.

[42] Terence, P. " ANTLR Parser Generator", accessed 2012, http://www.antlr.org/.

[43] IBM. "Telelogic, TAU SDL Suite". [http://www-01.ibm.com/software/awdtools/tau/]. vol. 4.3, 2012.

[44] The Eclipse Foundation. " Xtext - a Programming Language Framework", accessed 2012, http://www.eclipse.org/Xtext/.

[45] Samek, M. *Practical UML Statecharts in C/C : Event-Driven Programming for Embedded Systems.* Newnes, 2008.

[46] G Sunyé, D Pollet, Y Le Traon. "Refactoring UML Models". 2001. *Springer,* Springer. pp. 134-148.

[47] Gupta, A. and Jalote, P. "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development". 2007. *Empirical Software Engineering and Measurement, 2007.ESEM 2007.First International Symposium on,* pp. 285-294.

[48] The Eclipse Foundation. " Eclipse Modeling - M2T - Home (Jet Project)", accessed 2009, http://www.eclipse.org/modeling/m2t/?project=jet.

[49] Wagner, F. and Wolstenholme. "Misunderstandings about State Machines". 2004. *Computing and Control Eng,* vol 15, pp. 40-45.

[50] Schaumont, P. R. *A Practical Introduction to hardware/software Codesign.* Springer Verlag, 2010.

[51] A Wasowski. "Flattening State Machines without Explosions". 2004. *ACM Sigplan Notices,* vol 39, ACM. pp. 257-266.

[52] Lano, K. and Clark, D. "Direct Semantics of Extended State Machines". 2007. *TOOLS'07,* pp. 35-51.

[53] Niaz, I. A. and Tanaka, J. "Code Generation from UML Statecharts," in *Proc. 7Th IASTED International Conf. on Software Engineering and Application (SEA 2003),* 2003. pp. 315-321.

[54] Glaser, B. G. and Strauss, A. L. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* New York: Aldine de Gruyter, 1977.

[55] Orlikowski, W. J. "CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development". 1993. *MIS quarterly,* The Society for Information Management and The Management Information Systems Research Center of the University of Minnesota. pp. 309-340.

[56] John T. E. Richardson. *Handbook of Qualitative Research Methods for Psychology and the Social Sciences.* Massachusetts, U.S.A: Wiley, 1996.

[57] Blumer, H. *Symbolic Interactionism: Perspective and Method.* California: Univ of California Press, 1986.

[58] Calloway, L. and Ariav, G. "Developing and using a Qualitative Methodology to Study Relationships among Designers and Tools". 1991. *Information Systems Research: Contemporary Approaches and Emergent Traditions,* pp. 175-193.

[59] Toraskar, K. "How Managerial Users Evaluate their Decision Support: A Grounded Theory Approach," in *Proceedings of the IFIP WG 8.2 Working Conference,* 1991. pp. 195-225.

[60] Baskerville, R. and Pries-Heje, J. "Grounded Action Research: A Method for Understanding IT in Practice". 1999. *Accounting, Management and Information Technologies,* vol 9, Elsevier. pp. 1-23.

[61] Fitzgerald, B. "The use of Systems Development Methodologies in Practice: A Field Study". 1997. *Information Systems Journal,* vol 7, Blackwell Science Ltd. pp. 201-212.

[62] Bowen, G. A. "Grounded Theory and Sensitizing Concepts". 2006. *International Journal of Qualitative Methods,* vol 5, pp. 1-9.

[63] Markku, O. and Seija, K. S. "Product Focused Software Process Improvement 4th International Conference, Profes 2002 Rovaniemi, Finland, December 9-11, 2002".

[64] Edmonds, E. "A Process for the Development of Software for Non-Technical Users as an Adaptive System". 1974. *General Systems,* vol 19, Society for General Systems Research. pp. 215-217.

[65] Coleman, G. "EXtreme Programming (XP) as a'Minimum'Software Process: A Grounded Theory," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International,* 2004. pp. 30-31.

[66] Kahkonen, T., Abrahamsson, P., Center, N. R. and Espoo, F. "Digging into the Fundamentals of Extreme Programming Building the Theoretical Base for Agile Methods," in *Euromicro Conference, 2003. Proceedings. 29th,* 2003. pp. 273-280.

[67] Coleman, G. and O Connor, R. "Software Process in Practice: A Grounded Theory of the Irish Software Industry". 2006. *Lecture Notes in Computer Science,* vol 4257, Springer. pp. 28-39.

[68] Coleman, G. and O'Connor, R. "Investigating Software Process in Practice: A Grounded Theory Perspective". 2008. *The Journal of Systems & Software,* vol 81, Elsevier. pp. 772-784.

[69] Whitworth, E. and Biddle, R. "Motivation and Cohesion in Agile Teams". 2007. *Lecture Notes in Computer Science,* vol 4536, Springer. pp. 62-69.

[70] Whitworth, E. and Biddle, R. "The Social Nature of Agile Teams". 2007. *AGILE 2007,* vol 3, pp. 26-36.

[71] Layman, L., Williams, L., Damian, D. and Bures, H. "Essential Communication Practices for Extreme Programming in a Global Software Development Team". 2006. *Information and software technology,* vol 48, Elsevier. pp. 781-794.

[72] Ramesh, B., Cao, L., Mohan, K. and Xu, P. "Can Distributed Software Development be Agile?". 2006. *SPECIAL ISSUE: Flexible and distributed software processes: old petunias in new bowls?,* vol 49, ACM New York, NY, USA. pp. 41-46.

[73] Martin, A., Biddle, R. and Noble, J. "When XP Met Outsourcing". 2004. *Lecture notes in computer science,* vol 3092, Springer. pp. 51-59.

[74] Qureshi, S., Liu, M. and Vogel, D. "A Grounded Theory Analysis of e-Collaboration Effects for Distributed Project Management," in *Proceedings of 38th Annual Hawaiian International Conference on Systems Sciences, Big Island, HI,* 2005. pp. 59-87.

[75] Last, M. "Understanding the Group Development Process in Global Software Teams". 2003. *Frontiers in Education, 2003.FIE 2003.33rd Annual,* vol 3, pp. 20-25.

[76] Damian, D. and Zowghi, D. "Requirements Engineering Challenges in Multi-Site Software Development Organizations". 2003. *Requirements engineering,* vol 8, pp. 149-160.

[77] Padula, A. "Requirements Engineering Process Selection at Hewlett-Packard," in *12th IEEE International Requirements Engineering Conference, 2004. Proceedings,* 2004. pp. 296-300.

[78] Galal, G. and Paul, R. "A Qualitative Scenario Approach to Managing Evolving Requirements". 1999. *Requirements Engineering,* vol 4, Springer. pp. 92-102.

[79] Tat, E. H. and Kuan, M. H. C. "Requirements Engineering Processes". 1998. *Requirements Engineering,* vol 4, pp. 2.

[80] Oliver, D., Whymark, G. and Romm, C. "Researching ERP Adoption: An Internet-Based Grounded Theory Approach". 2005. *Online Information Review,* vol 29, Emerald Group Publishing Limited. pp. 585-603.

[81] Carver, J. "The Impact of Background and Experience on Software Inspections". 2004. *Empirical Software Engineering,* vol 9, Springer. pp. 259-262.

[82] Ye, Y. and Kishida, K. "Toward an Understanding of the Motivation of Open Source Software Developers," in *Software Engineering, 2003. Proceedings. 25th International Conference on,* 2003. pp. 419-429.

[83] Sillito, J., Murphy, G. C. and De Volder, K. "Questions Programmers Ask during Software Evolution Tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* 2006. pp. 23-34.

[84] Dingsoyr, T. and Royrvik, E. "An Empirical Study of an Informal Knowledge Repository in a Medium-Sized Software Consulting Company," in *Software Engineering, 2003. Proceedings. 25th International Conference on,* 2003. pp. 84-92.

[85] Sherif, K. and Vinze, A. "Barriers to Adoption of Software Reuse a Qualitative Study". 2003. *Information & Management,* vol 41, Elsevier. pp. 159-175.

[86] Murray, A. R. "Discourse Structure of Software Explanation: Snapshot Theory, Cognitive Patterns and Grounded Theory Methods". *Doctoral Thesis, University of Ottawa,* 2006.

[87] Yahaya, S. Y. and Abu-Bakar, N. "New Product Development Management Issues and Decision-Making Approaches". 2007. *Management Decision,* vol 45, Emerald, 60/62 Toller Lane, Bradford, West Yorkshire, BD 8 9 BY, UK,. pp. 1123-1142.

[88] Hughes, J. and Jones, S. "Reflections on the use of Grounded Theory in Interpretive Information Systems Research". 2004. *Electronic Journal of Business Research Methods,* vol 7, pp. 1-10.

[89] Hansen, B. H. and Kautz, K. "Grounded Theory Applied-Studying Information Systems Development Methodologies in Practice," in *Proceedings of 38th Annual Hawaiian International Conference on Systems Sciences,* 2005. pp. 264-264.

[90] Goulding, C. "Grounded Theory: The Missing Methodology on the Interpretivist Agenda". 1998. *An International Journal,* vol 1, Emerald, 60/62 Toller Lane, Bradford, West Yorkshire, BD 8 9 BY, UK,. pp. 50-57.

[91] Glaser, B. G. *Basics of Grounded Theory Analysis: Emergence Vs Forcing*. California: Sociology Press Mill Valley, 1992.

[92] Timothy C. Lethbridge, Gunter Mussbacher, Andrew Forward and Omar Badreddin. "Teaching UML using Umple: Applying Model-Oriented Programming in the Classroom". 2011. *CSEE&T,* pp. 421-428.

[93] Timothy C. Lethbridge, Omar Badreddin. "Umple - Associations and Generalizations". vol. youtube video, 2011.

[94] Timothy C. Lethbridge, Omar Badreddin. "Umple - State Machines Details". *Http://www. Youtube. com/watch?v=mFczzVkTZ9g;,* vol. youtube video, 2011.

[95] O. Badreddin. (2012) "An Empirical Experiment of Comprehension on Textual and Visual Modeling Approaches". University of Ottawa, Available: http://www.site.uottawa.ca/~tcl/gradtheses/obadreldin/

[96] Mohammad, S. "From Once upon a Time to Happily Ever After: Tracking Emotions in Novels and Fairy Tales". 2011. *ACL HLT 2011,* pp. 105-114.

[97] Mohammad, S. M. and Turney, P. D. "Crowd-Sourcing a Word--Emotion Association Lexicon". 2011. *Computational Intelligence,* Wiley Blackwell Publishing Ltd.

[98] Sjoberg, D. I. K., Anda, B., Arisholm, E., Dyba, T., Jorgensen, M., Karahasanovic, A., Koren, E. F. and Vokác, M. "Conducting Realistic Experiments in Software Engineering," in *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n,* 2002. pp. 17-26.

[99] Hendrix, D., Cross II, J. H. and Maghsoodloo, S. "The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities". 2002. *IEEE Trans.Software Eng.,* Published by the IEEE Computer Society. pp. 463-477.

[100] Briand, L. C., Bunse, C., Daly, J. W. and Differding, C. "An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents". 1997. *Empirical Software Engineering,* vol 2, Springer. pp. 291-312.

[101] Charles W. Rapp. "The State Machine Compiler". vol. 6.0.1, December, 2009.

[102] OMG, "Action Language for Foundational UML - ALF". OMG, Tech. Rep. 1.0 - Beta 1, 2010.

[103] MetaUML, U. *Diagrams for La T E X, Internet,*

[104] [104] Harris, T. " YUML", accessed 2012, http://yuml.me/.

[105] [105] avishn. " ModSL - Text-to-Diagram UML Sketching Tool", accessed 2012, http://code.google.com/p/modsl/.

[106] Hanov, S. " Web Sequence Diagram", accessed 2012, http://www.websequencediagrams.com/.

[107] Steel, J. and Raymond, K. "Generating Human-Usable Textual Notations for Information Models," in *Proceedings of the Fifth International Conference on Enterprise Distributed Object Computing (EDOC 2001), Seattle, Washington, USA,* 2001. pp. 250-250.

[108] Kushal. "Diagrammr". *Last Accessed January 2010.,* 2012. Available: http://www.diagrammr.com/

[109] Mueller, P. "SinelaboreRT: Generate Efficient Source Code from UML State Diagrams". vol. MD5, 2008-2009.

[110] Spinellis, D. "On the Declarative Specification of Models". 2003. *IEEE Software,* vol 20, IEEE Computer Society. pp. 95-96.

[111] Hakala, A. " LightUML", accessed 2012, http://lightuml.sourceforge.net/.

[112] Chaves, R. " TextUML", accessed 2012, http://abstratt.com/.

[113] Thurston, A. D. "Parsing Computer Languages with an Automaton Complied from a Single Regular Expression". 2006. *Lecture Notes in Computer Science,* vol 4094, Springer. pp. 285-286.

[114] Object Management Group (OMG). " Concrete Syntax for a UML Action Language RFP", accessed 2012, http://www.omg.org/cgi-bin/doc?ad/2008-9-9.

[115] Object Management Group (OMG). " Semantics of a Foundation Subset for Executable UML Models", accessed 2012, http://www.omg.org/spec/FUML/.

[116] Mellor, S. J., Tockey, S. R., Arthaud, R. and Leblanc, P. "An Action Language for UML: Proposal for a Precise Execution Semantics". 1999. *Lecture notes in computer science,* Springer. pp. 307-318.

[117] Merritt Ruhlen. "The Origin of Language: Tracing the Evolution of the Mother Tongue". 1994. *Wiley,*

[118] Robert Wright. "Nonzero: The Logic of Human Destiny". 2001. *Vintage,*

[119] Weilkiens, T. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* 2006.