# Model-Oriented Tracing Language:

## Producing Execution Traces from Tracepoints Injected into Code Generated from UML Models

**Hamoud Ibrahim H Aljamaan**

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the Doctorate in Philosophy degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

# Abstract

This thesis investigates the building of a textual tracing language that operates at the model level to allow trace specification of textually modeled UML constructs. Current tracing approaches focus on manually injecting tracepoints into targeted systems at the source code level. Such approaches are useful in code-centric development styles where the majority of the code is handwritten. However, in the case of Model Driven Development (MDD), where models are utilized to generate some or all of the code, current tracing technology results in low level trace specification and generation of execution traces that are not aware of or linked to the originating model-level constructs. Dynamic analysis hence becomes harder for a modeler adopting an MDD approach. This field, which we call model-oriented tracing, is currently immature, with little pre-existing research.

In this thesis, we present a textual model-level tracing language, implemented as part of Umple, that overcomes some of the limitations of existing tracing methods. The language facilitates model-level tracing, in a fashion very similar to code tracing. The language, which we call MOTL (Model-Oriented Tracing Language) allows tracing of UML associations, attributes and state machines. Constraints can be imposed to limit the scope of tracing.

As a result of this work, modelers will gain the ability to specify traces of UML constructs at the model level without the need to modify the generated code, and then generate execution traces when the generated system is run. The resulting trace links back to the model constructs. Modelers can choose from among several tracing technologies including basic file or console output, Java logging framework, Log4J and LTTng.

This thesis defines the language syntactically and semantically. Model-Driven Development (MDD) and Test-Driven Development (TDD) were followed to implement the language architecture to ensure high quality code generation. MOTL was used in the development in two of Umple subprojects. An empirical evaluation was conducted to evaluate the language's usability.

# Acknowledgment

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Timothy C. Lethbridge. Working under Tim supervision during my PhD journey, has been a rich and unforgettable learning experience to me. Tim dedication to his research and students is truly remarkable, and the knowledge I gained from interacting with him will impact my career as an academic researcher. Indeed, I was fortunate to work under the supervision of Tim.

Besides my supervisor, I would also like to thank my PhD committee members for their valuable feedback on this research.

I would like to thank members of the Complexity Reduction in Software Engineering (CRuiSE) research group. Special appreciation goes to my colleague and friend, Dr. Miguel A. Garzon, for all his critiques and feedback on my research. I would like to extend my gratitude to my close friend and mentor Dr. Fawaz Alsulaiman for his continuous encouragement while I was working on my research.

I thank my home university, King Fahd University of Petroleum and Minerals (KFUPM), for granting me a scholarship to pursue my PhD degree in Canada. I would like also to thank the University of Ottawa (uOttawa) for giving me the opportunity to pursue my degree, and providing me with all the support to carry out my research.

A thank you is not enough to express my appreciation to my wife Hadeel Alsolai. I am proud to share my life with you. We came to Canada as a newlywed couple, and now we have two angels in our family, my son Ibrahim and my daughter Jumaan.

Sincere admiration is for my father Dr. Ibrahim Aljamaan. I am proud to be your son. My mother was my first teacher, she taught me my first words and letters. She is my constant refuge as she always prays for me. I wish I made them happy and proud of what I have achieved so far in my life.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1. Introduction

In this thesis, we present a technology to allow tracing of software systems at the modeling level. This technology takes the form of a language we call Model-Oriented Tracing Language (MOTL) that can be used to inject tracepoints when generating code from models. MOTL allows modelers to trace software while thinking at a higher level of abstraction. MOTL generates code to allow interaction with a variety of low-level tracing technologies.

Software development has been continuously becoming more complex. Many researchers and software practitioners have strived to simplify the process by introducing abstract layers. In early days of the software industry, developers used assembly languages and machine code. As the need for greater productivity and quality became ever higher, programming languages were introduced as abstract layers over machine code. Nowadays, as software complexity continues to increase, researchers are directing their attention to Model Driven Development (MDD). Notable advantages are gained when software developers adopt the MDD approach to develop their systems, with models representing the main artifact in the development cycle [1–3]. One of main anticipated advantages is the ability to view targeted systems at a high level of abstraction as compared to programming languages.

The Unified Modeling Language (UML) [4] has become a standard for modeling object-oriented (OO) software systems. It has been widely used and adopted in both academia and industry to capture the structural (e.g. classes) and behavioral (e.g. state machine) aspects of software. Hence, UML is seen as a general purpose modeling language used to create, visualize and document software development artifacts.

Tracing is a well-known technique that has been used by software developers to understand the behaviour of systems in order to debug or monitor them. Data collected while tracing ranges from the output of simple programmed log commands to more sophisticated traces containing lower-level events triggered by tools that dynamically instrument user and kernel spaces. However, injecting tracepoints into a system is not a trivial task, and controlling tracing, to be both effective and efficient, is even more difficult. Historically, developers

have traced software using techniques such as adding simple print statements, breakpoints in a debugger, or more sophisticated tracing tools such as Dtrace [5, 6] or LTTng [7, 8] that allows tracing to be initiated either at compile time or run time.

In an MDD approach, where code is generated from models, these traditional tracing techniques tend to be limited to working with generated code. They therefore require the understanding of the generated code's structure, and require extra work to map changes and understanding to the original source. Besides, code that emits trace information needs replacing when the code is regenerated. Tracing thus occurs at a level of abstraction below the level at which the system is implemented. Furthermore, execution traces, resulting from traces injected manually in the source code generated from models, lack the links between model constructs and the execution trace elements.

In this thesis, we will solve the above problem by introducing a textual trace specification language that operates at the model level; Model-Oriented Tracing Language, hereafter called MOTL. MOTL describes tracing needs in terms of modeling constructs, generates code with injected tracepoints that will trace those modeling constructs, and injects model-construct information into traces. Traces can then be analyzed at the model level. We will expand on this in the coming sections.

## 1.1. Model-Oriented Tracing

The focus of existing tracing techniques is to perform tracing at the code level independently from the model. This involves tracing function or method calls, lines executed, variables being set, etc. There has been little research into the objective of this thesis: tracing at the model level. The concept of model-oriented tracing is the following: A developer modeling in UML (or some other modeling language), and generating much of his system directly from the model, should be able to indicate that he wants any of the model constructs to be traced. The developer should not be forced to inject tracing into generated code, which he or she may never otherwise look at, and which is subject to regeneration every time the system changes.

The focus in this thesis will be tracing of UML constructs. Suitable constructs to trace include:

- **Attributes**: Tracing attributes is conceptually similar to tracing variables, except that the level of abstraction is higher because the implementation of the attribute is deferred to the code generator, and the attribute may have automatically managed constraints, specialized initialization conditions, and triggers such that changes to the attribute cause system events.
- **Associations and association ends**: Associations can have many different implementations depending on multiplicity, reflexivity, direction of navigability and other factors. Tracing an association means tracing whenever an association link is added or deleted during run time.
- **State machines**: A state machine can have different elements: simple states, composite states, transitions, states with do activities, etc. Tracing can be performed at state entry and exit, as well when particular transitions occur or when named events occur. Since state machines can be nested at several levels of depth, tracing can be scoped to certain substates. Tracing of attributes and associations can be constrained to occur in specific states.

In addition to the above, it should still be possible to trace user-written (non-generated) methods and lines of code (in user-written methods) as has been traditionally possible. The key is that model-oriented tracing adds another level of abstraction to the elements that can be traced.

## 1.2. Problem Statement

The problem statement for this research is as follows:

*Developers frequently need to deploy tracing to debug programs, to test them in a white-box manner, to understand their internal behaviour and to detect anomalies such as hacker intrusion or performance degradation. Current technologies, however, only allow injecting tracepoints into functions (procedures or methods) and data items. The ability to systematically trace at the model level is generally missing, since there are no comprehensive tools available to meet this need).*

As a solution to this problem, we have designed and implemented a textual tracing modeling language to allow the abstract specification of tracing at the model level. We de-

cided to name the language Model-Oriented Tracing Language (MOTL). MOTL allows developers to specify tracing at the model level by tracing UML constructs (state machines, associations and attributes) and specifying how to inject tracepoints into generated code for these UML constructs. Therefore, MOTL adds a level of abstraction for tracing specification. MOTL is implemented as an extension of Umple [9–11], a textual language for representing UML models. We will describe Umple in more detail in Chapter 2.

## 1.3. Research Questions

The following are the research questions we are addressing throughout this thesis:

**RQ 1. What should the syntax and semantics of a model-oriented tracing language be?**

Tracing at the model level adds an abstract level over traditional tracing performed at the code level. Creating a textual modeling language to handle abstract trace specification requires the description of its syntax and semantics. In this research, we investigate how MOTL should handle abstract trace specification. We consider issues such as the following:

I. **Model entities**: the trace language syntax should be flexible enough and have the ability to allow modelers to specify tracing of any desired model entity. The question we tackle is how to provide flexibility, and how to specify tracing for a variety of types of entities. For the purpose of our research we will focus on attributes, state machines, and associations.

II. **Contents of trace output**: Tracing at the model level will inject tracepoints in the code generated from models, so the question is what kind of information should be collected when models are executed? In other words, what are the needed components that should be output when an instrumented system is executed.

III. **Tracing technology tools**: Output in the form of trace messages needs to be reported via some underlying tracing technology. The question we investigate is what different tracing technologies can be supported by MOTL, and how can they be integrated in such a manner that they can be easily selected.

IV. **Other capabilities**: What other capabilities ought to be available in the language.

**RQ 2. How can such a language best be implemented?**

We need to investigate the following language issues:

I. **Grammar**: What should be the formal description of the MOTL syntax?

II. **Metamodel**: What should be the internal metamodel of MOTL constructs?

III. **Code generation**: What should the generated code look like? In other words how should the compiler inject tracepoints into generated code?

**RQ 3. How to link trace specification at the model level with execution traces?**

We need to investigate:

I. **Tracepoint injection locations**: how and where tracepoints should be injected into code generated from model?

II. **Trace output**: What should the components of the trace output should be, and how can they be used to link execution traces to model constructs?

**RQ 4. How usable is the implemented language in practice?**

We need to investigate:

I. **Syntax learnability**: were new MOTL users able to grasp the language concepts and gain knowledge of its syntax in order to use it in actual practice?

II. **Likeability**: to what extents do users like the language?

## 1.4. Research Methodology

In this thesis, we followed the design science methodology [12] to address the research questions stated above. The major steps conducted in following the methodology were:

1. *Problem identification*: Perform manual tracepoint injection on code generated from Umple models. Use these manual injections to produce execution traces when the code is run. This results in trace specification at the code level, but loss of the

tracepoints whenever the code is regenerated from the model. Furthermore, execution traces at this stage are not linked to modeling constructs.

2. *Suggestion*: Trace specification at the model level can be accomplished by proposing a textual model-oriented tracing language (MOTL) that allows traces to be specified and maintained at the model level.
3. *Development*:
    a. Iteratively work on the syntax and semantics of a language for tracing of Umple modeling constructs (specifically attributes, state machines, and associations) using structured trace directives. Tracepoints injected in generated code initially support one primitive tracer (console).
    b. Explore ways to limit the scope of tracing in trace directives in the form of prefix and postfix subdirectives.
    c. After reaching an acceptable stable version of the language syntax and semantic, add support for other tracers technologies, and investigate how selection of tracer technology can be made at the model level.
4. *Evaluation and validation*:
    a. Conduct an empirical study with real participants to study and measure the language usability by asking them to perform tracing tasks at the model level using the language and receive feedback about their subjective satisfaction towards the language.
    b. Validate correctness of the language by designing a test suite for each MOTL architecture component using Multi-level Test Driven Development.
    c. Invite other researchers in the CRuiSE research group to use the language to perform tracing tasks in their research projects and report issues encountered or suggestions for further language enhancements.
5. *Conclusion*: Compare the work with similar approaches and develop ideas for future work.

## 1.5. Thesis Contributions

The key contributions of this thesis are summarized as follows:

- The core idea of the MOTL language; specifically allowing modelers to abstractly specify tracing of the following constructs at the model level:
    I. **Attributes**: Tracing of attributes is the simplest form of tracing at the model level. It is conceptually similar to tracing of variables at the source code level; however, have additional semantics such as being subject to constraints.
    II. **Association**s: Tracing of associations involves the tracing of all link modifications at run time, and can be specified at either association end. This type of model level tracing has not been explored nor introduced before.
    III. **State machines**: Tracing of state machines has been explored previously in other modeling tools, but not integrated with other tracing capabilities as we have done in this work. In MOTL, all state machine components can be traced and the depth of nested states tracing can be controlled.
- Flexibility of the language in terms of trace specification and limiting the scope of tracing at the model level:
    I. **Model constructs integration**: MOTL is the only language that allows the fully integration of trace specification among different modeling constructs. In other words, tracing of a modeling construct (e.g. state) can be linked to other modeling constructs (e.g. cardinality of an association link).
    II. **Constraints**: MOTL is the only language that allows modelers to constraint tracing at the model level with numerous constraints options.
- The specific syntax and detailed semantics of MOTL, including mechanisms for dealing with conditions, and the integration with modeling;

- The specific implementation of MOTL in the Umple technology. This is ready to use in real systems. MOTL has been used in practice by other researchers in the CRuiSE research team.
- An evaluation showing that MOTL is usable in the context of an empirical study.

## 1.6. Clarification of Terminology

The term *model-based tracing* as used in this thesis refers to the injection of tracepoints into code generated from models, resulting in the production of execution traces when systems are executed.

The terms *trace* and *tracing* as we use them in this thesis should not be confused with how the terms are used in certain other contexts. The following two contexts in particular use the terms in ways that could prove confusing unless the reader is aware of the important distinctions.

***Traceability*** is a very important software engineering concept and refers to making connections between different levels of abstraction (e.g. between requirements and design, or between elements of a model before and after a model transformation). A trace in this context is a connection, or the act of constructing such a connection. Examples of work related to traceability and models includes [13, 14]. Tools like Acceleo support tracing in this manner as they do model transformations [14–16]. However, there is slight similarity in this context with MOTL, as MOTL will inject model related information into tracepoints messages, which will be reported in the execution traces. Thus, creating a link between the traced modeling construct and the execution trace line (as will be shown later in this thesis).

This work is fundamentally different from ours, even though it uses related terms. In particular, traces in their context are static connections, whereas in our context they are dynamic records of execution.

The term ***model tracing*** also appears in the context of intelligent systems such as tutors and diagnosis systems. Here it means that a tool walks (i.e. traces) the model to help make decisions [17, 18].

Once again, that usage of the terms 'trace' is quite different from how we use it.

## 1.7. Publications

The following publications have been produced as a result of this research:

1. Hamoud Aljamaan, T. C. Lethbridge, M. Garzón, "MOTL: a Textual Language for Trace Specification of State Machines and Associations", In ACM proceedings of the 25th Conference of the Center for Advanced Studies on Collaborative Research (CASCON). IBM Corp., pp. 101-110, 2015. (Ref number: [19])
2. Hamoud Aljamaan, M. Garzón, T. C. Lethbridge, A. Forward, "UmpleRun: a Dynamic Analysis Tool for Textually Modeled State Machines using Umple", Workshop on Executable Modeling at the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), 2015. (Ref number: [20])
3. M. Garzón, Hamoud Aljamaan, and T. C. Lethbridge, "Umple : A Framework for Model Driven Development of Object- Oriented Systems," in 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 494-498, 2015. (Ref number: [9])
4. Hamoud Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying Trace Directives for UML Attributes and State Machines," in 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 79–86, 2014. (Ref number: [21])
5. Hamoud Aljamaan, T. C. Lethbridge, "Towards Tracing at the Model Level", in 19th Working Conference on Reverse Engineering (WCRE) – Doctoral symposium, IEEE, pp. 495-498, 2012. (Ref number: [22])

## 1.8. Outline

This thesis is organized as follows:

**Chapter 2**

This chapter serves as an introduction to Umple, a model-oriented programming language. Umple modeling constructs are explored, with the explanation of the language architecture. Development environments for Umple are shown.

**Chapter 3**

In this chapter, a technical overview of current source code tracing technologies is given with examples of their usage. These technologies can serve as potential MOTL supported tracer technologies that would be used by MOTL to inject tracepoints into generated code from models.

**Chapter 4**

MOTL core architecture is presented in this chapter. MOTL allows the textual trace specification of modeling constructs through the use of structured trace directives. MOTL trace directive architecture is explored. Syntax and semantics of tracing: attributes, state machines, and associations are defined. Tracing constraints in the form of prefix and postfix subdirectives are discussed.

**Chapter 5**

This chapter presents list of MOTL supported tracer technologies and explains how can they be selected textually at the model level through a structured tracer directive. MOTL tracer directive architecture is explored, with its defined syntax and semantics. Trace output components are outlined, and generation of execution traces are presented using two modeling examples.

**Chapter 6**

This chapter discusses the quality and validity aspects of MOTL. MOTL is an open source language developed in an agile manner using both Model-Driven Development and Multi-level Test-Driven Development to ensure correctness and quality of generated code. Usage of MOTL in two internal subprojects of Umple is examined.

**Chapter 7**

This chapter presents the empirical study conducted to examine MOTL usability. A group of participants were invited to learn the language and perform tracing tasks at the model level by writing trace directives. Subjective satisfaction of the participants was also gathered.

**Chapter 8**

Relevant work in the area of model tracing is explored and compared to our approach.

**Chapter 9**

Contributions are summarized and directions for future work are outlined.

# Chapter 2. Umple: a Model-Oriented Programming Language

Umple [9–11] is an open source modeling-oriented programming language for modeling UML constructs textually and graphically; it can be seen as both a modeling and a programming language. Umple allows modelers to perform model driven development of Object-Oriented systems by generating entire software systems of high quality code in a number of targeted programming languages.

Software developers can represent UML concepts such as classes, attributes, associations and state machines in Umple, and can embed ordinary methods in an Umple class. As a result, an Umple program looks like a standard program (e.g. in Java) with some extra features added. Umple has set a list of philosophies that direct its research vision:

- Umple sees modeling as programming and vice versa. UML models can be expressed textually. Thus, a modeler can see UML models both visually and textually, while a programmer can see UML coded abstractly using Umple code.
- Usage of Umple can start from an existing system and UML constructs can be added incrementally. Hence, Umple will parse programming languages code as part of Umple code.
- Umple uses base language code as an action language to handle algorithmic operations. This allows the generation of complete systems from an Umple model.
- Generated code should not be edited as in the process called "round tripping" [23], since any special-purpose code can be embedded in Umple as necessary. Therefore, Umple model should be the main and central artifact in the system development, where it is maintained as the system being modified.
- Software developers describing the system at a high level of abstraction will have less code to write and hence should have higher productivity.

Umple is being maintained and developed by the CRuiSE research group at the University of Ottawa. It is an open source mirrored live at various sites including GitHub [24].

## 2.1. Umple Constructs

In Umple, the concepts of 'model' and 'program' are unified [25]. A model, containing a collection of UML classes, associations, state machines and other constructs is written in Umple's textual form. With the addition of a main program and some other embedded methods, this becomes a complete system. Umple users have the capacity of specifying high level constructs, most of which are based on UML. The following sections explore in detail these concepts.

### 2.1.1 Classes

An Umple class is defined in a similar manner to other object-oriented languages. It starts with a 'class' keyword followed by class name, then class contents between curly brackets '{}'. The main elements of possible class content are:

- **Attributes**: These become variables with accessor methods and are discussed in the next section.
- **State machines**: These capture desired dynamic behavior and are briefly discussed in section 2.1.3.
- **Associations**: These describe relationships, between instances; they are discussed in section 2.1.4.
- ***Trace directives***: to specify tracing at the model level (the main focus of this thesis).
- **Methods**: These are parsed so they can be analyzed, but are output largely 'as is' in the generated code.
- **Comments**
- **Generalization directives**: The keyword 'isA' followed by the name of a class or interface indicates generalization.
- **Stereotypes**: Directives such as 'abstract', 'immutable' or 'singleton' direct code generation.
- **Extra code**: code in native programming language that is not recognized as one of the above, so is output in the generated code as is.

It should be noted that Umple interfaces are defined in a manner similar to classes, except that they can only contain methods with no body and comments. The keyword 'interface' is used instead of 'class'.

As an illustration, consider the Umple code in Listing 2.1:

**Listing 2.1: Simplified university student system**

Umple

```
1  class Student {
2     Integer id;
3     name;
4   }
5  class Graduate {
6     isA Student;
7     Thesis {
8       Ready {
9         submit -> Submitted ;
10        quit -> Withdrawn;
11      }
12      Submitted {
13        accept -> Graduated;
14        reject -> Ready;
15      }
16      Graduated {}
17      Withdrawn {}
18    }
19    * -- 1 ThesisSupervisor supervisor;
20  }
21 class Undergraduate {
22     isA Student;
23 }
24 class ThesisSupervisor {}
```

Umple code is designed to look similar to Java, and adds modeling abstractions to represent UML constructs textually. In this example, lines 6 and 22 are Umple statements that represent the generalization relation between two classes. Line 19 shows the association between some Graduates and a ThesisSupervisor. A graduate student must have a thesis supervisor and a supervisor can supervise many graduate students. Using UmpleOnline [26], we can render the above code as Yuml class diagram:

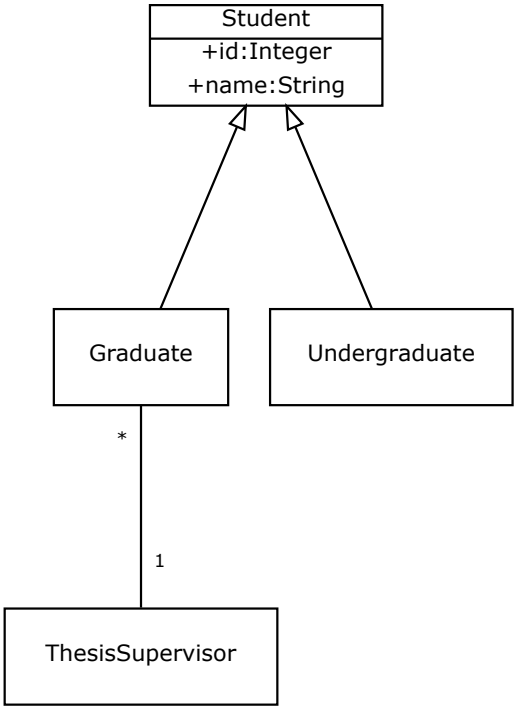


**Figure 2.1: University student system class diagram**

Lines 7 to 18 describe an aspect of the behavior of a Graduate instance as a state machine; the diagram of this generated by Umple is shown in Figure 2.2. When Graduate object is created, its initial state is Ready. When a thesis is ready to be submitted, the submit event will cause a transition to the Submitted state.

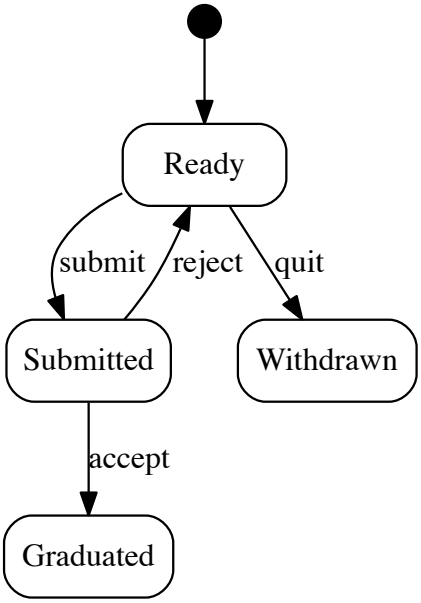


**Figure 2.2: State Machine generated from Listing 2.1**

### 2.1.2 Attributes

An Umple attribute is a simple property of a class. They are abstract entities that hold data and can be constrained in various ways. Declaration of an Umple attribute, in its simplest form, starts with the data type followed by the attribute name. Umple supports a set of data types that covers modeling needs as shown below. In cases where there is no data type specification, string is the default type as in line 9 of Listing 2.2.

**Listing 2.2: Umple attributes types**

Umple

```
1  class UmpleAttributes {
2    Integer number;
3    Double number2;
4    Float number3;
5    Boolean flag;
6    String str;
7    Time t;
8    Date e;
9    name;
10 }
```

Certain Umple stereotypes and other notations allow modelers to impose special characteristics upon model attributes:

**Lazy attributes**: Lazy attributes will not be passed as a constructor argument at the time of their creation. Objects including string are initialized with null, numbers with zeros, and Booleans to false.

```
lazy String str;
```

**Immutable attributes**: These are settable once at construction time and can not be changed afterwards. An immutable attribute's value is passed to constructor as an argument for its setting. Keyword 'immutable' is used to declare immutable attributes.

```
immutable String greeting = "Welcome";
```

The combination of lazy and immutable characteristics will form an immutable attribute whose value is not set at the time of its construction, but has to be assigned in a set method with a condition that it is settable only once.

```
lazy immutable String greeting;
```

**Derived attributes**: when an attribute value can be calculated from other attributes, then it is a derived attribute. Umple supports declaration of derived attributes by placing computation upon which the derived attribute is based between '{}'.

```
Double salary = {getTotalWorkingHours() * getHourRate()};
```

**Unique attributes**: An attribute value is unique if its value is not repeated in newly created objects. Preceding the data type with the 'unique' keyword will declare the uniqueness of its value. Situations requiring this might include an attribute that holds an Internet IP address, or a bank account number. Objects instantiated from a class declared to have a unique attribute will be constrained to have distinct values for the given attribute.

```
unique String ipAddress;
```

Another variant of this constraint is the automatic uniqueness, where for every object created, a new integer value is assigned incrementally. The attribute's value can not be changed; in other words, there is no set method generated.

```
autounique counter;
```

**Defaulted attributes**: attributes can have defaulted values indicating that they will be initialized with the default value, and can be reset to it when desired. Umple's generated API will have, in addition to the set/get methods, a reset method to reset the attribute value to its default value and a special get method to query the default value for an attribute.

```
defaulted String str = "DefaultedString";
```

**Multivalued attributes**: Attributes normally are used to hold a single value in a model, however, they can also manage multiple values of the same data type, using the square bracket syntax '[]' following the data type declaration. For instance, a phone line attribute can have multiple different phone lines assigned to it.

```
Integer[] phoneLine;
```

**Constant attributes**: constant attributes can be declared at the model level to imply unchangeable values.

```
const Integer MAX = 100;
```

Table 2.1 summarizes the API generated from Umple attributes.

**Table 2.1: Umple attributes API generation**

| | Constructor argument | set | get | reset | add | remove |
|---|---|---|---|---|---|---|
| *Lazy* | - | x | x | - | - | - |
| *Unique* | x | x | x | - | - | - |
| *Autounique* | x | - | x | - | - | - |
| *Defaulted* | x | x | x | x | - | - |
| *Derived* | - | - | x | - | - | - |
| *Multivalued* | - | - | x | - | x | x |
| *Immutable* | x | - | x | - | - | - |
| *lazy immutable* | - | x | x | - | - | - |
| *Constant* | - | - | - | - | - | - |

### 2.1.3 State Machines

In this section, we will overview Umple support for state machine textual definition. Umple provides a complete textual specification of state machines at the model level. Umple users have the ability to specify nested or concurrent states, transitions with guards, entry or exit actions, and do activities.

State machines are declared in similar manner to Umple attributes, and a state machine can be viewed as an attribute with a restrained number of states, along with a special mechanism for changing those states. Umple states can be classified simple composite.

For illustration purpose, we consider a garage door example as a motivating example to explain the usage of different Umple state machines features, as shown in Listing 2.3. The GarageDoor class has a state machine named 'status' to describe the behavior of a normal automatic garage door. The state machine consists of two states: Open and Closed. To further define the behavior of state machines we will add more features such as transitions, events and actions.

**Listing 2.3: Garage Door class with a state machine – initial version without transitions**

Umple

```
1 class GarageDoor {
2    status {
3      Open {}
4      Closed {}
5   }
6 }
```

State transitions occur when an Umple event is triggered to transit from one state to another. Umple's generated event-handling code calls exit, transition and entry actions if they exist, while setting the state machine to the new state. In Umple state machines, the first state defined (in the textual order in which the state machine is processed) is always the initial state (i.e. start state) of the state machine. Any state that does not have a transition out of it is an end state (i.e. final state).

In Listing 2.4, the garage door example is further refined with transitions. A transition now is possible from state Open to state Closed by the triggering of event 'button'. In addition, as the state machine is currently defined state Open is the initial state and state Closed is the only final state.

**Listing 2.4: GarageDoor with transitions**

Umple

```
1 class GarageDoor {
2    status {
3      Open {
4        button -> Closed;
5      }
6      Closed {}
7   }
8 }
```

Transitions can be constrained by the addition of guards, which are arbitrary Boolean expressions. If a guard is evaluated to true, then the transition will be completed, otherwise, the event will be ignored. Guards can be a Boolean variable, expression, or a method, and they are added before or after the transition name between square brackets '[]'. Listing 2.5 shows the addition of guarded transitions to both Open and Closed states. As a result, transition between the two states will not occur through the triggering of event button unless Boolean method pathClear() is evaluated to true. Method pathClear() is a custom method written in native language code to check whether the path for the garage door is clear or there is an obstacle preventing the door from closing or opening.

**Listing 2.5: GarageDoor with guarded transitions**

Umple

```
1  class GarageDoor {
2     status {
3       Open {
4         button [pathClear()] -> Closed;
5       }
6       Closed {
7         button [pathClear()] -> Open;
8       }
9    }
10 }
```

Umple state machine actions can be transition actions, entry actions, or exit actions. Transition actions are actions associated with the occurrence of the transition. Entry actions are actions executed upon the entry of a state, while exit actions are executed when a state is exited. Actions are methods written by modelers in native programming language, with the intention of providing modelers with the ability to reuse the same action in multiple states as a transition, entry, or exit actions. Listing 2.6 shows state Open with both an entry and an exit action. Upon transiting to state Open, method turnOnLights() is executed, while method turnOffLights() is executed when transiting out of state Open.

**Listing 2.6: Garage Door with entry/exit actions**

Umple

```
1  class GarageDoor {
2     status {
3       Open {
4         entry / { turnOnLights(); }
5         button [pathClear()] -> Closed;
6         exit / { turnOffLights(); }
7       }
8       Closed {
9         button [pathClear()] -> Open;
10      }
11   }
12 }
```

Unlike actions that are viewed as static points executed instantaneously upon being triggered, do activities are long-term activities that needs to remain running while we are inside a state. Umple supports the declaration of do activates inside a state, requiring the creation of a thread to handle the do activity while making the state available to respond to events. A state do activity thread is terminated once we leave (i.e. exit) from that state. Listing 2.7 shows state Closed with a do activity.

**Listing 2.7: Garage Door with a do activity**

Umple

```
1  class GarageDoor {
2     status {
3       Open {
4         entry / { turnOnLights(); }
5         button [pathClear()] -> Closed;
6         exit / { turnOffLights(); }
7       }
8       Closed {
9         button [pathClear()] -> Open;
10        do { doContinuouslyWhileInside(); }
11      }
12    }
13 }
```

Umple composite states can be either a simple nested states or concurrent states. Umple allows the flexibility of defining nested states of unlimited nesting depth and states with unbounded numbers of concurrent regions.

The best way to illustrate concurrent states is to show an example. We present a simplified traffic light system textually modeled using Umple in Listing 2.8; the visual rendering of this is in Figure 2.3. The traffic light state machine consists of two simple states Ready and Malfunction and a state TrafficLight with two concurrent states PedestrianSignal and CarSignal; each concurrent state has its own nested states. The state machine's initial state is Ready, and event initiateOperation will trigger a transition from Ready to TrafficLight state. Upon transiting to composite TrafficLight state, we enter both concurrent states (i.e. state machine is in state PedestrianSignal and state CarSignal). Both concurrent states contain substates with auto transitions to transit among them. However, any triggering of event detectMalfunction will result in a transition out of both composite states to Malfunction state.

**Listing 2.8: Traffic light system Umple code**

Umple

```
1  class TrafficLightSystem {
2    status {
3      Ready {
4        initiateOperation -> TrafficLight;
5      }
6      TrafficLight {
7        PedestrianSignal {
8          Walk {
9            after(50) -> StopFlash;
10         }
11         StopFlash {
12           after(10) -> Stop;
13         }
14         Stop {
15           after(60) -> Walk;
16         }
17         detectMalfunction -> Malfunction;
18       }
19       ||
20       CarSignal {
21         Red {
22           after(60) -> Green;
23         }
24         Green {
25           after(50) -> Yellow;
26         }
27         Yellow {
28           after(10) -> Red;
29         }
30         detectMalfunction -> Malfunction;
31       }
32     }
33     Malfunction {
34       repair -> Ready;
35     }
36   }
37 }
```

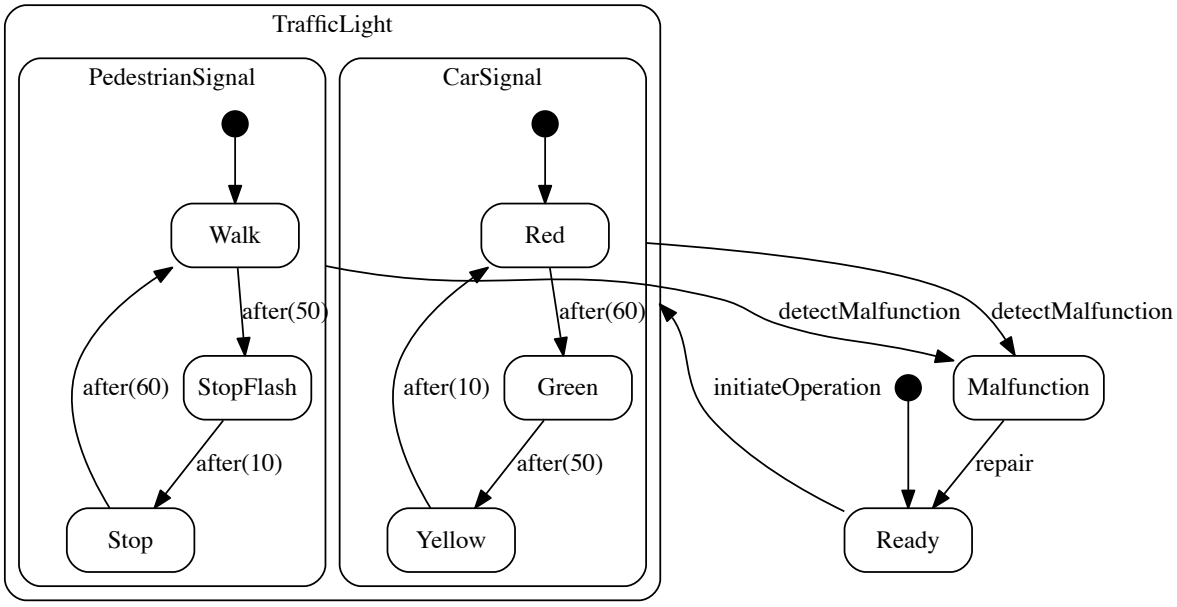


**Figure 2.3: Traffic light system state machine generated from Listing 2.8**

Table 2.2 provides a summary of Umple state machine notation explained to this point.

**Table 2.2: Summary of Umple state machine notation**

| Notation | Description |
|---|---|
| -> | Indicates a transition |
| entry / | Actions associated with state entry |
| exit / | Actions associated with state exit |
| […] | Transition Boolean guard |
| do | Indicates the declaration of a do activity inside a state |
| \|\| | Implies concurrent state |

### 2.1.4 Associations

UML associations define the relationship between classes in a class diagram. They specify the sets of links among objects that can exist at run time. Umple supports the definition of associations with two ends (i.e. binary associations) that can be written inside a class as an inline association declaration. In Listing 2.9, there is a bidirectional association between a Mentor and a Student, where a mentor can be assigned to many students, but a student can only be assigned to a single mentor. By declaring the inline association inside Mentor class, we say that Mentor class is the containing class for the association (i.e. relationship).

**Listing 2.9: Inline association declaration**

Umple

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

Alternatively, the previous association between Mentor and Student classes can be declared outside the scope of classes as a separate first class entity in a model as presented in Listing 2.10.

**Listing 2.10: Association declaration in a separate class**

Umple

```
1 class Mentor {}
2 class Student {}
3
4 association {
5   1 Mentor -- * Student;
6 }
```

Umple textual notation allows modelers to declare association navigability, plus explicitly impose association multiplicity constraints at both ends of an association. Umple associations can be unidirectional or bidirectional. A unidirectional association, expressed by symbol '->', means the association is navigable in one direction since only one association end is aware of the relationship between the two classes. However, in the case of bidirectional associations, both association ends are aware of the relationship (i.e. association links) between them. Symbol '--' is used to specify bidirectional associations.

Association multiplicity describes the number of association links that are permitted at an association end; Umple follows UML conventions for these. Multiplicity constraints made by modelers must be carefully managed and constraints are enforced in code generated by Umple. Different variations of multiplicity are presented in Table 2.3. In most cases, a multiplicity may have a lower bound and an upper bound.

**Table 2.3: Association multiplicity bounds**

| Multiplicity | Description |
|---|---|
| *1* | Mandatory One (lower and upper bound both1) |
| *0..1* | Optional One (lower-bound 0; upper bound 1) |
| *n* | Exactly n (lower and upper bound both n), where n is an integer |
| *0..n* | Up to n (special case of 0..1) |
| *n..m* | Between n and m, where both are integers |
| *n..** | N or more (in practice n is most commonly 1) |
| * | Many (lower bound zero; unbounded upper limit; can also be written 0..*) |

An association role name explains how each association end participates in the relationship. For example, in Listing 2.11, role name 'supervisor' adds more clarification to the nature of the relationship between a Student class and a Mentor class, meaning, that each Student instance he will have one and only one Mentor assigned to him *as a supervisor*. Usage of role names is important in case of reflexive associations. Where role names are omitted, the class name is implicitly used as the role name where required. Umple will generate an error message in circumstances where there are multiple associations between the same classes and no role name is provided to distinguish them.

**Listing 2.11:Umple association role name**

Umple

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

Umple's generated API methods for managing associations have been tested thoroughly to respect imposed constraints on multiplicity and manage referential integrity. Multiplicity constraints determine the particular set of generated API methods as outlined in Table 2.4. The last three rows are supportive methods that will facilitate querying the required lower bound or upper bound in an association end.

**Table 2.4: Association multiplicity constraint mapping to Umple generated methods API**

| | *Multiplicity* | | | | | | |
|---|---|---|---|---|---|---|---|
| ***API Methods*** | ***1*** | ***0..1*** | ***n*** | ***0..n*** | ***n..m*** | ***n..**** | ***** |
| *set* | x | x | x | x | x | x | - |
| *get* | x | x | x | x | x | x | - |
| *add* | - | - | - | x | x | x | x |
| *addAt* | - | - | - | x | x | x | x |
| *addOrMoveAt* | - | - | - | x | x | x | x |
| *remove* | - | - | - | x | x | x | x |
| *delete* | x | x | x | x | x | x | x |
| *index* | - | - | x | x | x | x | x |
| *requiredNumber* | - | - | x | x | x | x | - |
| *minimumNumber* | - | - | x | x | x | x | x |
| *maximumNumber* | - | - | x | x | x | - | - |

### 2.1.5 Other Modeling Features Relevant to this Thesis

In addition to the above modeling constructs, Umple supports other features worth mentioning as follows:

**Patterns**: Umple supports the singleton and immutable patterns. In Listing 2.12, only one object will be allowed to be instantiated from class Manager. The 'immutable' keyword in class Employee will enforce all its content to be settable only once in the constructor.

**Listing 2.12: Umple singleton and immutable patterns**

Umple

```
1  class Manager {
2    singleton;
3  }
4  class Employee {
5    immutable;
6    String name;
7    Date dob;
8  }
```

**Code injections**: Umple users have the capability to insert code that can be run before (i.e. precondition) or after (i.e. postcondition) Umple-defined actions on attributes, associations and the components of state machines. Listing 2.13 shows a code injection before and after the setting of attribute name. Before code will act as a precondition before setting the attribute name and the after code will print a confirmation message after setting attribute name.

**Listing 2.13: Umple before and after code injections**

Umple

```
1  class Student {
2    String name;
3    before setName {
4      if (aName != null && aName.length() > 20)
5          { return false; }
6    }
7    after setName {
8      System.out.println("Name was set successfully.");
9    }
10 }
```

## 2.2. Umple's Architecture

Umple's main intent is to be used to perform forward model engineering of software systems, with the Umple source being the master artifact in the development process. The Umple compiler has a layered and pipelined architecture as shown in Figure 2.4. The forward engineering components of Umple includes: a parser, an analyzer as well as several code-generators and model-to-model transformation engines. These are described below:

1. **Parser:** The parser receives an input model, written in the Umple language, tokenizes it and passes it to the next component in the pipeline.
2. **Analyzer:** This component processes the tokens previously obtained and converts them into an internal representation consistent with Umple's metamodel.
3. **Code Generator(s):** The internal representation is then translated into other artifacts; either additional models like Papyrus XMI, Ecore, Yuml; various diagrams, or source code such as Java, C++, PHP, Ruby or SQL. The compiler generates various types of methods including mutators (to control changes to a variable) and accessors (to return the value of a variable) from the various Umple features. Sophisticated code for managing state machines, tracing, generation templates, patterns, aspects and concurrency can also be generated from models.

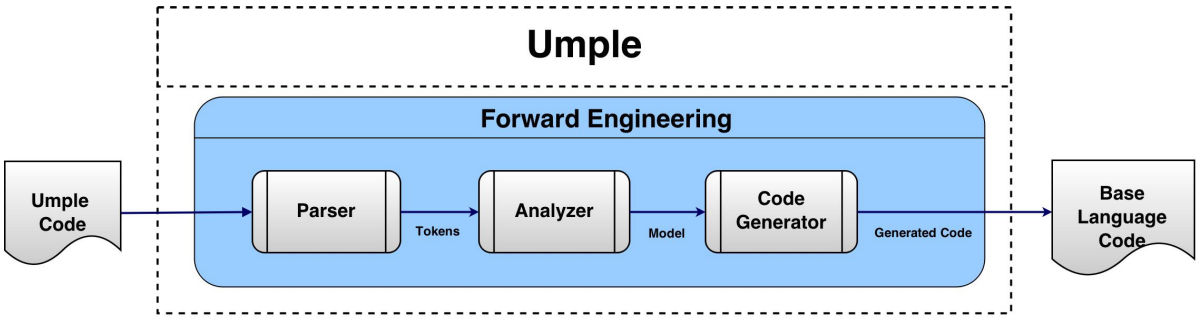


**Figure 2.4: Umple Architecture** [9]

### 2.2.1 Grammar

Grammar notations have been used to specify the syntactic structure of many languages and rules set forms the core component of any grammar. Extended Backus-Naur Form (EBNF) is the most famous and widely used notation for programming languages syntax specification. Umple use its own grammar notation to describe its language syntax for textual representation of UML models.

Umple grammar notation is similar to EBNF notation, however, with differences. Components of the Umple grammar are below:

- **Rules:** grammar rules consists of two parts: (1) rule name followed by ':' symbol, (2) rule definition, which can be either a non-terminal or terminal definitions.
- **Terminal:** terminals can be either (1) keywords (2) symbols (3) a non-whitespace sequence of characters surrounded by '[]' to name identifiers, that are delimited by the next space, newline, or terminal symbol.
- **Non-terminal:** non-terminals are surrounded by double square brackets '[[]]' and they are references to subrules.

Table 2.5 contains reference to Umple grammar notation with symbols controlling how the grammar is parsed.

**Table 2.5: Key elements of Umple grammar notation**

| Notation | Usage |
|---|---|
| […] | Terminal definition |
| [[…]] | Reference to another rule |
| \| | Alternation (OR) |
| ? | Optional |
| * | Zero or more |
| + | At least one |

The Umple grammar [27] is divided into several files based on separate concerns for readability and extensibility purposes. As an example, there is a separate grammar file that contains the specification of state machines in Umple. The Umple parser parses the Umple grammar files and builds its parser; the resulting parser is then used to parse Umple code and create an abstract syntax tree.

In most cases, grammar rule names are included as labels on tokens in the abstract syntax tree, however, there are certain rules that act as placeholders created for the grammar to be more clear and readable. Such rules do not generate their own tokens in the AST. Such rules are marked by placing a minus '-' sign after their name in the grammar.

As an illustration, we consider creating a simple rule for variable declaration (not a real part of Umple grammar) using the Umple grammar notation as shown below. There is one rule named 'Declaration' that has two terminals 'type' and 'name' followed by a ';' symbol.

| R1 | `Declaration- : [type] [name] ;` |
|---|---|

As shown, the terminal [name] will match an unrestricted non-whitespace sequence of characters that might include special symbols (e.g. underscore) as identifiers. However, the terminal [name] could instead be directed to match alphanumeric identifiers only by adding '~' symbol before its name, thus: [~name]. In addition, there are other symbols that help to parse special cases. Symbol '*' preceding the terminal name will match every character until newline is reached. This is used to match inline comments. Another symbol '**' preceding the terminal name will match every character, including newlines, until the next character sequence appearing in the grammar rule is matched. This is beneficial to deal with multiline comments, and in Umple it is used it to match native programming code (e.g. algorithmic methods).

The following examples show syntactically valid statements that satisfy the previous rule (please note that the second statement is valid in terms of syntax, declaring an attribute 'name' of type 'return'):

```
int id;
return name;
```

To further refine the previous rule, we decomposed the 'Declaration' rule into two sub rules 'LHS' and 'RHS'. In 'LHS' sub rule, we constrained the terminal 'type' to be a constant by strictly accepting either provided strings (i.e. int, float or double). RHS is an optional sub rule as indicated by '?'. It starts with an equality symbol followed by terminal 'value'. Parentheses were used to group the rule elements together so they can be flagged as optional using the question mark.

```
R1 | Declaration- : [[LHS]] [[RHS]] ;
R2 | LHS : [=type: int | float | double ] [name]
R3 | RHS : ( = [value] )?
```

Examples of valid syntax statements:

```
int number;
double number = 10.4;
```

Based on the grammar, Umple parser will tokenize the previous statements as shown below. The 'Declaration' rule name was omitted from the tokenization string, because it has a minus sign after its name.

```
[LHS][type:int][name:number]
[LHS][type:double][name:number][RHS][value:10.4]
```

### 2.2.2 Metamodel

Umple's metamodel is used to define the different modeling language constructs and code elements. The metamodel classes (metaclasses) represent the various language constructs and features. Changes to Umple's metamodel occur when a new language feature is being added or an existing feature is refined/refactored. The Umple metamodel is written and modeled using Umple itself. For instance, the Umple state machine metamodel was written in Umple, in file StateMachine.ump containing the modeling concepts and StateMachine_Code.ump covering algorithmic operations that can be applied to state machines during model analysis and code generation. Figure 2.5 shows the Umple state machine metamodel. The full Umple metamodel can be viewed at [28].

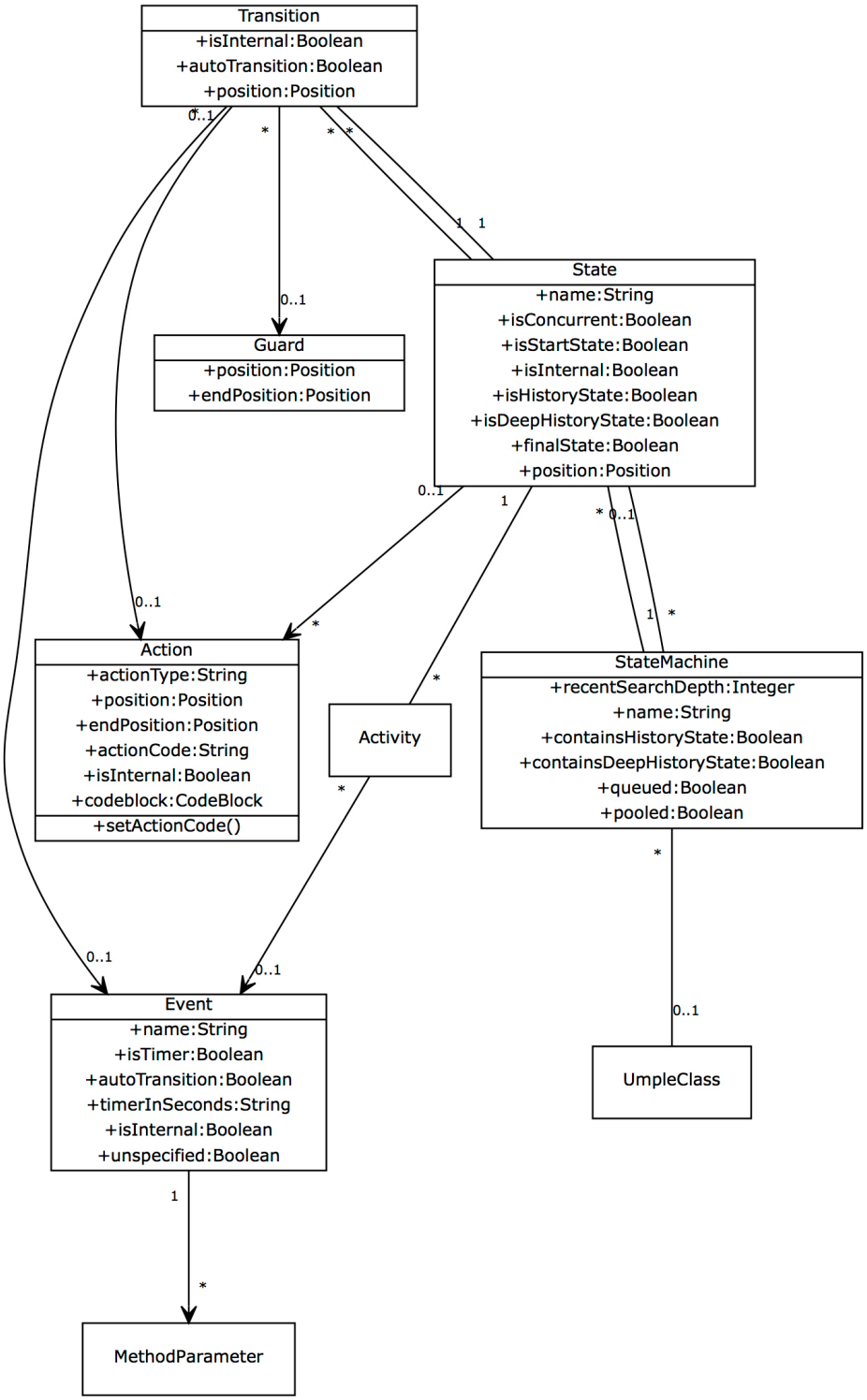


**Figure 2.5: Umple state machine metamodel**

### 2.2.3 Multi-level Test Driven Development

Test Driven Development (TDD) [29, 30] is a software development style where test cases drive the development of the code; in other words, developers create tests at the beginning before the actual implementation. Therefore, written code must pass all tests. In software systems with a layered architecture, multi-level test driven development is achieved by applying the TDD principles to every independent component of the system. As a result, each layer will have its own test suite that must pass whenever a new feature is introduced to the system.

Umple embraces multi-level test-driven development in its layered architecture. Each component is tested independently to ensure that the input is processed correctly and the output produced is valid. The testing process, as illustrated in Figure 2.6, consists of different test suites:

- **Tokenization tests:** ensures that Umple code is correctly tokenized and a correct abstract syntax tree is generated.
- **Metamodel tests:** ensures that valid metamodel instances are produced, where the abstract syntax tree is processed into an internal representation consistent with the Umple metamodel.
- **Syntax tests:** ensures that the metamodel instance is correctly transformed into other artifacts (e.g. Java source code).
- **Semantic tests:** Generated code is executed to ensure that the generated system behaves as desired.

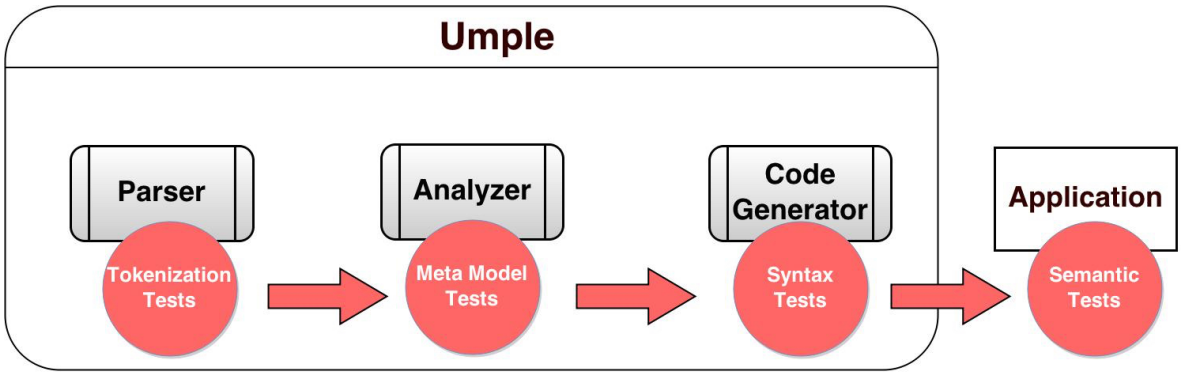


**Figure 2.6: Umple Multi-level Test Driven Development Infrastructure**

Currently, the core test suite in Umple is comprised of over 5000 tests that span its multiple architecture layers. The execution results of the test suite can be accessed online at [31].

There are reported benefits of adopting test driven development style when developing software systems [32, 33]. In Umple, we have experienced noticeable rewards applying TDD in each component of our architecture:

I. **Validation**: Multi-level test driven development encourages the emphasis on quality and correctness by creating tests at each level of Umple architecture.

II. **Bugs should not be hidden**: Reporting of bugs is seen as unimplemented set of tests that needs to be added to our test suite. Fixing bugs eventually leads to better quality and better test coverage.

III. **Refactoring**: Having test suites at each level in Umple's architecture raises the confidence in performing refactoring tasks of existing language features without introducing future hidden bugs nor breaking any currently implemented features.

## 2.3. Umple Development Environments

Developing in Umple can be approached in a bottom up or top down manner. In the bottom-up approach, a modeler can start with a legacy system and add Umple modeling constructs by reverse engineering. A tool has been developed to incrementally reverse engineer an existing system into Umple in a process called Umplification [34]. In the top-down approach, a developer can start modeling UML constructs in Umple and add action code (in Java, C++ etc.) to handle algorithmic operations.

Umple development comes in three flavours: a command line stand-alone jar, an Eclipse plugin, and a Web-based application called UmpleOnline. Details on those environments are provided in the next sections.

### 2.3.1 Eclipse Plugin

Eclipse is a widely used integrated development environment (IDE) that supports the development of a variety of languages by providing useful development tools such as: syntax highlighting, error detection, code completion, etc. Having an Umple Eclipse editor is without a doubt a necessity for developers accommodated to the Eclipse environment. We have

developed an Umple editor as an Eclipse plugin. The Umple Eclipse jar should be placed in eclipse plugin folder.

Umple's Eclipse plugin provide eclipse developers with Umple syntax highlighting and tree hierarchy view of the Umple code as shown in Figure 2.7. Umple files are recognized by the (.ump) as an extension. There are two Umple icons, which indicate the proper installation of the plugin, the first icon compiles the Umple code, while the second compiles and then run's the model when a main program is associated with the Umple model.

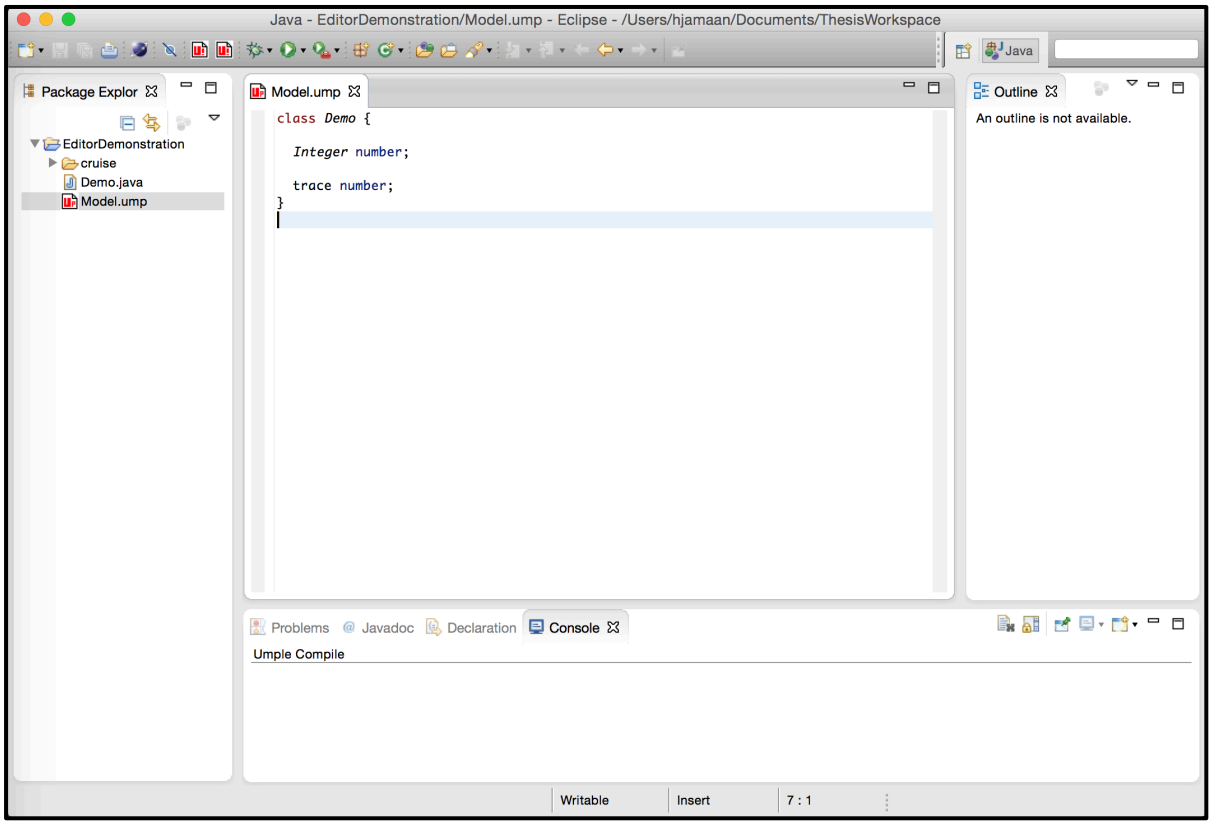


**Figure 2.7: Umple Eclipse Editor**

### 2.3.2 Command Line

Umple code can be compiled from the command line using Umple's standalone jar file as shown below, where Umple files are passed as <arguments> to the Umple compiler and there are a list of options that can direct the output of the compiled Umple code (e.g. –g to specify the output targeted programming language).

```
> java -jar umple.jar <options> <arguments>
```

This approach is intended for developers who wish not to constraint their development to a specific environment.

### 2.3.3 UmpleOnline

UmpleOnline [26] is an interactive online development environment that allows anyone to instantly experiment with Umple on the web and create small Umple models or programs on the fly. It does not require any installation, thus making it a simple and easy tool for demonstration and teaching purposes by university professors [35, 36]. Although it's simple, it is also a powerful tool with a rich set of examples of modeled systems that can be explored, and for which source code can be generated in any supported programming language (C++, Java, PhP etc.).

UmpleOnline has two panels as shown in Figure 2.8: the left panel is used to write Umple code and right panel to draw UML diagrams. Syntax highlighting and error detection is provided for Umple code writing. When users write Umple code in the left panel, the corresponding UML diagrams are shown in the right panel. Alternatively, users can draw UML diagrams and Umple code will be generated from these diagrams. In addition, UmpleOnline users may chose a wide variety of object oriented programming languages (e.g. Java/Ruby/PhP) as a targeted generated code.

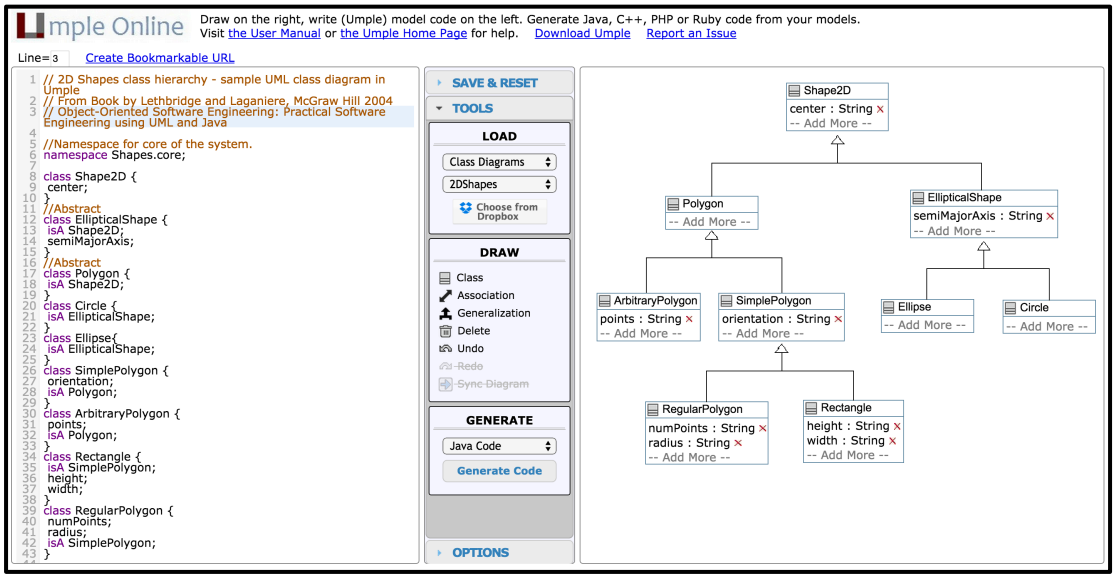


**Figure 2.8: UmpleOnline** [26]

## 2.4. Summary

In this chapter, we presented the Umple model-oriented programming language and explored how it can used to model different modeling constructs. Umple supports several attribute types with the ability to impose special stereotypes on them at the model level. Bi- and Uni-directional associations can be modeled textually with multiplicity constraints. In addition, Umple supports an enriched textual syntax to model different state machine components.

Umple's layered architecture was overviewed by presenting the Umple grammar and metamodel. Multi-level test driven development was adopted as a process for language development. At the end of the chapter, we outlined the Umple development environments.

# Chapter 3. Background: Tracing Technologies

Since the beginning of the software industry, code instrumentation has been used by developers. It has allowed them to detect bugs and reason about program behavior by generating informative messages as the program executes. Code instrumentation involves adding new code lines to existing software or modifying existing lines.

Instrumentation was discussed as early as 1974 [37], when a tool named SMT was introduced that allowed users to instrument their programs and control data collection. Several approaches were created for code instrumentation, where a software system under inspection can use one or a combination of approaches to satisfy the instrumentation objectives [38, 39]:

I. **Source code tracing**: This approach collects low-level information during program execution in the form of trace messages. Tracing is widely used by developers to trace execution flow by collecting information that may include: method entry, method exit, setting of variables, and thrown exceptions. A high volume of output is expected when collecting these low level details resulting in size concerns.

II. **Logging**: Logging is similar to tracing, however the key difference is the type of information collected during program execution. System administrators use logging to collect high level information related to program execution like the start of a program or failure to launch it. Logging should be kept switched on, in contrast to tracing that should be switched off when not needed to reduce the negative impact on performance.

III. **Profiling**: Profilers are designed primarily for performance analysis. A profiler will monitor parts of the program during execution and gather statistics for optimization purposes. They often intermittently record the status of the program to gather statistics. An example is using a profiler with a Java programs to detect memory leaks and CPU performance issues [40, 41]. Studies have been conducted in the academia to propose new profiling techniques and

compare existing profilers, as in [42], where authors performed an accuracy evaluation among a number of Java profilers in detecting hot methods (i.e. methods that consume the most execution time).

Code instrumentation can be applied directly to the source code or to the byte code. Insertion of code instrumentation lines is done in the form of a ***tracepoint***, and it can be done statically by inserting static tracepoints into source code before the execution, or dynamically while the program is being executed. In case of static tracepoints, when one is added to source code, it becomes part of program execution. During the execution of the instrumented program, if the tracepoint is reached, it will be invoked and data such as the identity of the traced entity item, the time of tracepoint invocation, and any other information specified will be gathered.

Injected static tracepoints should not intervene or modify the dynamic behavior of instrumented programs. There are two issues that need to be handled well when inserting static tracepoints into source code:

i. *Quantity*: Number of tracepoints should be sufficient to achieve the instrumentation objectives, but not so many as to unduly increase the size of the code or result in excessive execution time.
ii. *Location*: Determining whether tracepoints locations should be inserted randomly or systematically following the program structure.

To overcome any confusion in terminology that might arise from the distinction between the functionality of 'trace' and 'logging', we would like to state that we will use these terminologies interchangeably based on the context of the technology being presented, and we define both 'tracing' and 'logging' as 'deployment of a set of tracepoint injections that produces messages once invoked with the purpose of understanding the executed program behavior'.

In this chapter, we will give an overview of the technical background for tracing technologies that perform logging and tracing tasks. We start with Java primitive tracing methods, progress to well-established Java logging frameworks, and end with the exploration of more sophisticated and advanced tracing tools.

The MOTL language that we will discuss later in this thesis is designed to be able to generate output using most of the technologies to be discussed.

## 3.1. Print Statements

Tracing using print statements has been used since the beginning of software due to its simplicity, and is the most primitive approach. Developers in most programming languages use print statements to perform quick debugging tasks by sending informal trace messages to standard output (console) and standard error streams. It is recommended that trace messages should be forwarded to the error stream rather than the console so as to separate error output from actual program output. Integrated development environments, such as Eclipse, show messages printed in the error stream in a different color (e.g. red) than output to the console. Another important reason to direct the trace messages to the error stream is that developers have the option to direct the error stream to a file while leaving console output unchanged.

Listing 3.1 shows an example of a traced Java method. There are two tracepoints located at the method entry and exit in line 2 and 8 respectively, and they pass an informal trace message to the error stream. This example presents a common practice done by Java developers to assert whether the method was entered and then exited probably when called.

**Listing 3.1: Java method tracing**

Java

```
1 public void JavaMethod() {
2   System.err.println("JavaMethod entry");
3   // Method body begins
4   // ........
5   System.out.println("Printing stuff related to the method");
6   // ........
7   // Method body ends
8   System.err.println("JavaMethod exit");
9 }
```

The simplicity and ease of use of print statements comes with a cost of performance overhead and cluttering the program source code with print statements. Add to that the cost of manually inserting/modifying/deleting print statements lines [43]. Consequently, many logging frameworks have been introduced and implemented in the software industry to facilitate advanced program tracing. We will focus on the ones for Java.

## 3.2. Logging Frameworks

Java logging frameworks provide advanced architecture for developers to have more control over their tracing objectives. The best-known logging frameworks for Java are Java logging API [44] and Apache log4j [45]. These API have many features, but there are two key features that distinguish logging frameworks from simple print statements [46]:

1. **Output destination**: logging frameworks provide developers with various output destinations to direct logging messages such as: network socket, database, email, etc.
2. **Logging levels**: this functionality allows developers to log their messages to different predefined logging levels based on severity. Thus, severe error messages can be treated one way, while less important informative messages can be treated another way, or even ignored.

In this section, we will shed light on the Java native logging API and log4j frameworks.

### 3.2.1 Java Native Logging API

Java provides its own built-in logging API implemented in the java.util.logging package [44]. This package forms a framework that contains all classes and interfaces needed to facilitate logging of Java programs as shown in Figure 3.1. To use the Java logging API, first we define a logger object from 'Logger' class, and then use it to record log messages at different log levels based on message severity. There are seven logging levels labeled: severe (highest level), warning, info, config, fine, finer, finest (lowest level). Moreover, there are two special levels: level (off) that can be used to turn off the logging and level (all) to turn logging on for all levels. Logging calls are made to the Logger object, which creates a LogRecord object and passes it to the handler.

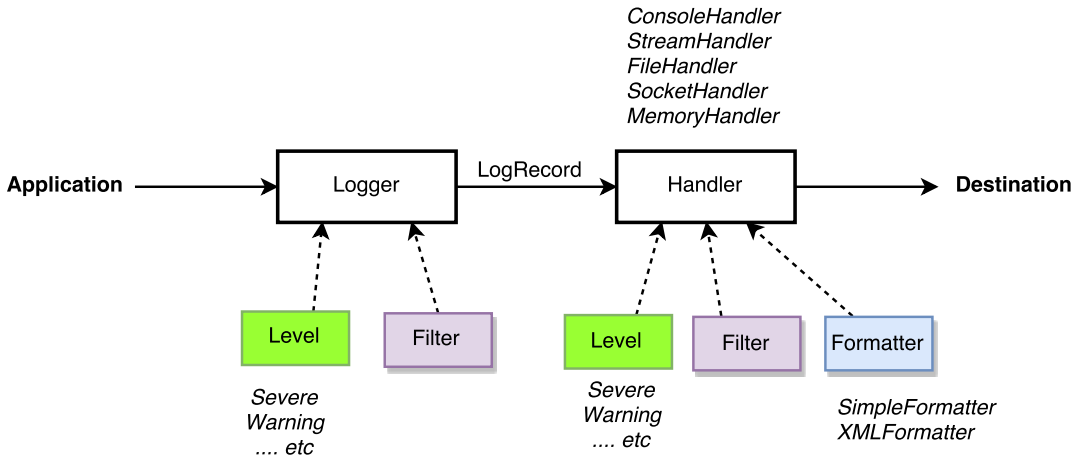


**Figure 3.1: Java logging API (Util package)**

There are five built-in handlers in the Java logging API as shown in Table 3.1. A handler is a component of the Java logging API framework that is responsible for writing log messages to a target destination (e.g. console or file), and logging levels can be assigned to one or more handlers. e.g. all messages at log level 'severe' should be directed to file.

The formatter component is used optionally to format the message before sending it to the destination. There are two built-in formatters in the Java logging API: SimpleFormatter to format the message into a human readable format, and XMLFormatter to produce the messages in XML structure. The Java logging API is extensible, allowing developers to add more user-defined handlers and formatters.

**Table 3.1: Java Logging API Handlers**

| Handler | Description |
|---|---|
| Console | Default handler. Writes all log messages to "System.err.". |
| File | Writes all log messages to a file. |
| Stream | Writes all log messages to an OutputStream. |
| Socket | Writes all log messages to a network stream connection. |
| Memory | Writes all log messages into a memory buffer. |

Listing 3.2 provides an illustration of the usage of Java logging API, as it would appear in plain Java code. Please note that to simplify the explanation we removed various details of the code like declaring a file object. In this example, we will try to open a file that does not exist, resulting in the throw of an exception. The first line shows the importing of

the Java logging package. Line 4 contains the definition of a logger object that will be used to pass logging messages to the default console handler. For the method that opens the file, we will inject two tracepoints, however, each one is for a unique log level. In line 9, we will log an informal 'unimportant' message to indicate that we are entering this method. The logger object is passing the message with 'info' as the logging level. However, in line 15, the logging object passes the thrown exception to the 'severe' level.

**Listing 3.2: Logging using Java API**

Java

```
1  import java.util.logging.*;
2
3  public class JavaLogEx {
4    public static final Logger logger =
5                Logger.getLogger(JavaLogEx.class.getName());
6
7    public void openFile()
8    {
9      logger.info("Entered openFile method");
10
11     try {
12     reader = new BufferedReader(new FileReader(file));
13     String text = null;
14     } catch (FileNotFoundException e) {
15       logger.severe(e.toString());
16     }
17   }
18 }
19
```

A sample run of the above Java program with a non-existing file to open, will result in the following output. By default, both log levels 'info' and 'severe' will forward logged messages to the console, but this can be changed. The output will report the time and date of the logged message; the class and method where the log was triggered, the level of the logged message, and the logged message contents. Output contents can be changed and controlled in the configuration.

```
Jul 10, 2015 12:15:27 PM JavaLogEx openFile
INFO: Entered openFile method
Jul 10, 2015 12:15:27 PM JavaLogEx openFile
SEVERE: java.io.FileNotFoundException: file.txt (No such file or direc-
tory)
```

### 3.2.2 Apache Log4j

Apache log4j (version 2), commonly named log4j2, [45] is a prominent open source Java logging framework. It is similar to Java's built-in logging mechanism, but with more advanced logging options. Log4j2's full architecture is shown in Figure 3.2. Details can be accessed online in log4j2 project website [47]. We will give an overview of part of the architecture that is most important to developers using the framework.

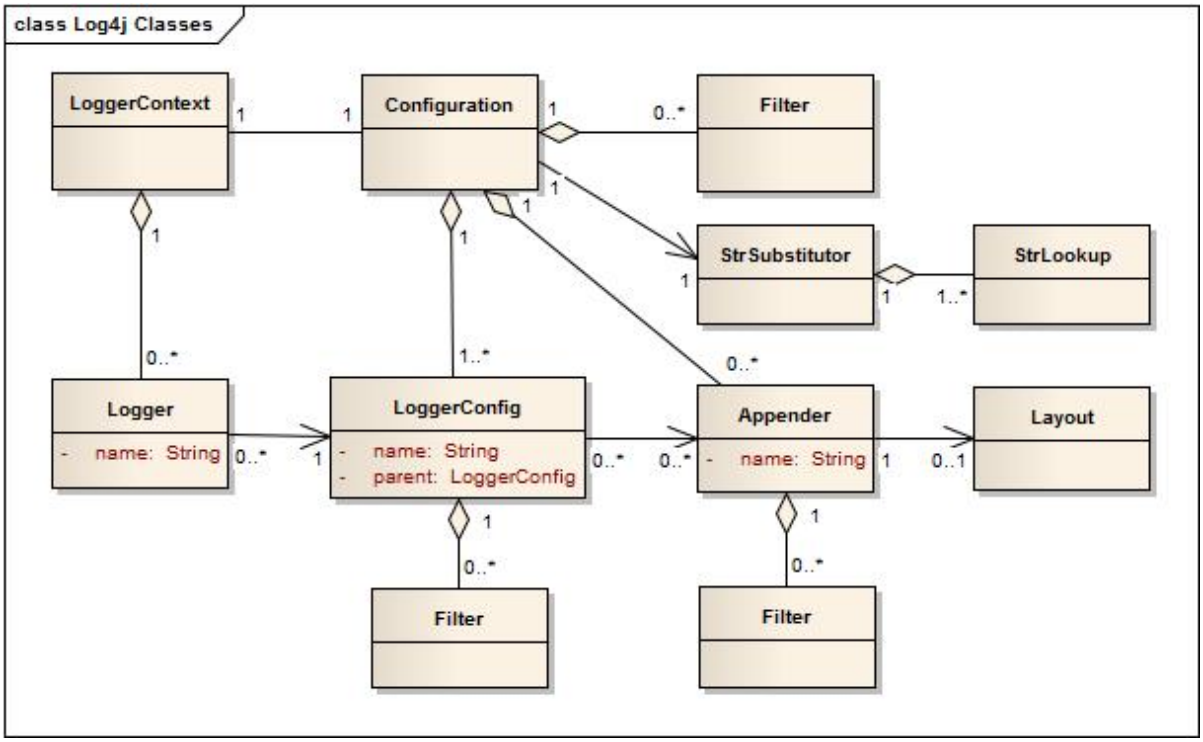


**Figure 3.2: log4j2 architecture** [47]

The Log4j2 component that sends log messages to the destination is the Appender class. Numerous appenders are provided by log4j2 with various configuration options, making it much richer than the Java logging API. A list of important appenders is in Table 3.2.

**Table 3.2: log4j2 appenders**

| Appender | Description |
|---|---|
| ConsoleAppender | Basic appender. Sends all log messages to the console (System.err). |
| FileAppender | Writes all log messages to a file. |
| RollingFileAppender | Similar to FileAppender, however, a file size is set, and once the used file to record logging reaches the maximum size, it will open a new file and move the old one as a backup. |
| SMTPAppender | Sends log messages to an email |
| JDBCAppender | Writes all log messages to a database using Oracle JDBC. |
| SocketAppender | Sends log messages to a network socket. |

Log4j2 implements six logging levels ranked in decreasing order based on severity of logged messages: fatal (highest level), error, warn, info, debug, and trace (lowest level). Logged messages are directed to appenders to handle the actual output. Formatting of sent logged messages is accomplished through the Layout component. Log4j2 has many layout options that produce log messages in JSON, HTML, and XML formats.

Configuration of the logging mechanism (e.g. appenders assigned to different log levels) is done in a separate XML configuration file, and developers can create their own customized configuration file based on their logging objectives. A default configuration file will be used by log4j2 in case a customized file was not found in program classpath. A key feature in log4j2 is that reconfiguration is possible during program execution, meaning logging can be stopped while the program is running.

As in the Java logging API, log4j2 can be extended by allowing developers to create their own appenders, layouts, etc. Listing 3.3 gives the same example used in the previous subsection, but we used log4j logging code instead. The first two lines imports the necessary log4j2 classes located in log4j2 jars. Line 5 creates the logger object that will be used to pass log messages. Line 15 will log the thrown exception at level 'fatal'.

**Listing 3.3: Logging using log4j2**

Java

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j. Logger;
3
4  public class JavaLogEx {
5    public static final Logger logger =
6                LogManager.getLogger (JavaLogEx.class);
7
8    public void openFile()
9    {
10
11     try {
12     reader = new BufferedReader(new FileReader(file));
13     String text = null;
14     } catch (FileNotFoundException e) {
15       logger.fatal(e.toString());
16     }
17   }
18 }
```

Executing the above Java program with a non-existing file to open will generate the following output. The default output (i.e. no customized XML configuration) will include system time, thread outputting the log message, level of the logged message, class where the logging is triggered and contents of logged message.

```
01:22:42.178 [main] FATAL JavaLogEx - java.io.FileNotFoundException:
file.txt (No such file or directory)
```

## 3.3. Linux Trace Toolkit Next Generation (LTTng)

Linux Trace Toolkit Next Generation (LTTng) [7, 8, 48] is an open-source tracing framework targeting Linux systems, allowing kernel and userspace tracing. Currently, LTTng is gaining momentum in the software industry since it is prepackaged with different Linux distributions (e.g. Ubuntu). It has been an active project since 2005 and many upgrades and features have been added since that time. Many major software companies have adopted LTTng for various debugging and performance analysis purposes such as IBM to debug issues related to distributed file systems, and Siemens to monitor and debug performance of their systems [8, 49].

Figure 3.3 shows the current architecture for a recent version of LTTng. It supports kernel and user space tracing. In kernel tracing mode, traces are designed to debug kernel performance-related problems such as memory management, scheduling. Likewise user space tracing aims at providing information related to user space activity.

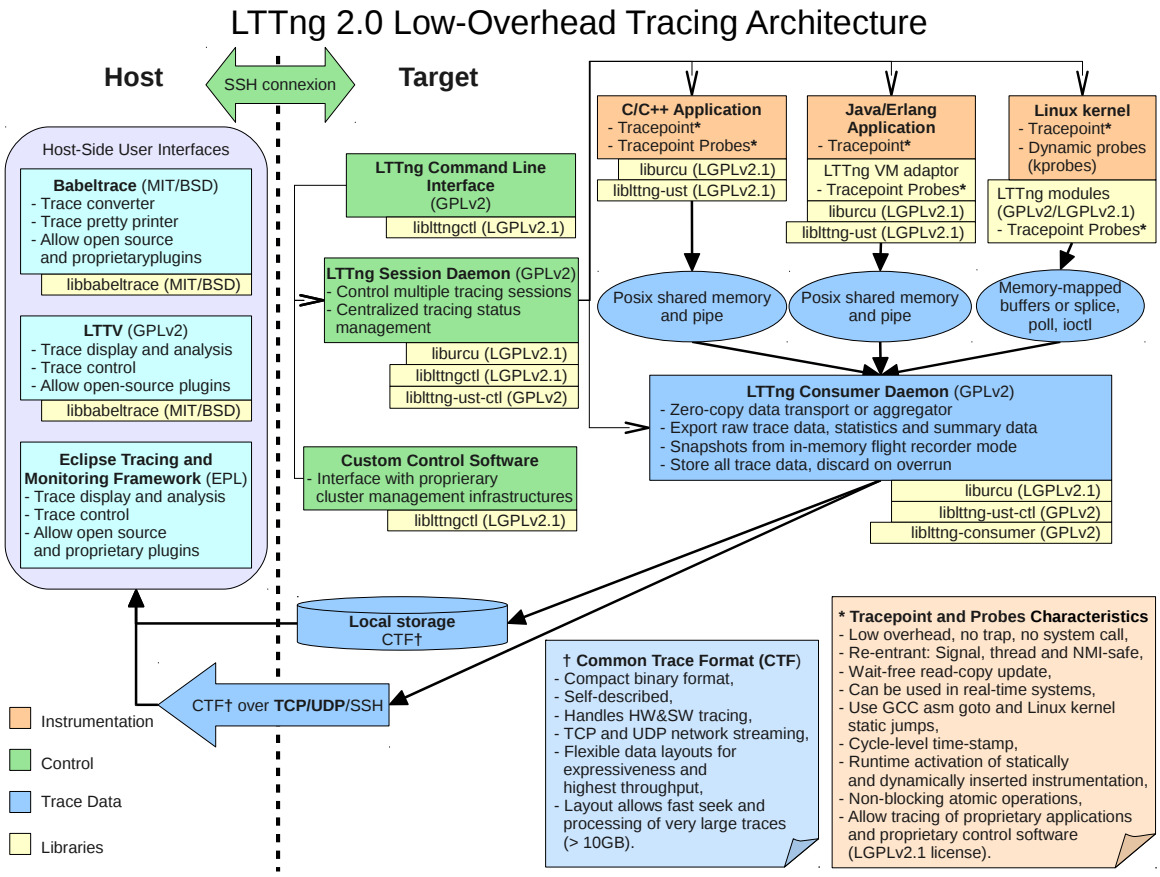


**Figure 3.3: LTTng 2 architecture** [48]

To start kernel tracing, commands need to be executed to start, control, and stop the kernel tracer. Once started, it will capture all events triggered in the kernel space; the scope of events captured can be controlled. In user space tracing, static tracepoints can be injected into programs with arguments to specify details of the trace.

Resulting LTTng traces are produced in the Common Trace Format (CTF) to allow them to be application-, architecture-, and programming-language agnostic. Therefore, LTTng traces can be used to debug and analyze complex multicore systems [50]. However, traces in CTF are not human-readable, therefore users need to use BabelTrace [51] or Trace Compass [52] to covert them into human readable form and visualize them to make analysis possible. In addition to producing tracers in CTF format at the local host, LTTng can send traces over a network.

As the name indicates, LTTng is used to trace Linux C/C++ program execution in both kernel and user spaces. However, tracing of Java programs is possible through the

LTTng-UST agent that is responsible for communications with an LTTng session daemon. This agent acts as a logging back-end for Java programs traced using either Java native logging or Apache log4j. The LTTng-UST Java agent must be initialized before any logging commences in the instrumented Java program. The agent adds its handler to the root logger, so that all loggers may generate LTTng events.

Listing 3.4 is an example of how to use LTTng-UST Java agent to trace Java programs. In line 2, LTTngAgent is imported, and line 11 initializes the LTTng agent before any call to injected logging as occurs in line 16.

**Listing 3.4: Tracing using LTTng**

Java

```
1  import java.util.logging.Logger;
2  import org.lttng.ust.agent.LTTngAgent;
3
4  public class LttngEx
5  {
6     public static final Logger logger =
7               Logger.getLogger(LttngEx.class.getName());
8
9     public LttngEx()
10    {
11      LTTngAgent lttngAgent = LTTngAgent.getLTTngAgent();
12    }
13
14    public boolean JavaMethod(int aId)
15    {
16      logger.info("Trace Message");
17    }
18 }
```

## 3.4. Dtrace

Dtrace [5, 6] is a comprehensive dynamic tracing toolkit developed by Sun Microsystems and integrated into the Solaris operating system, Mac OSX, and FreeBSD. It enables developers to write trace scripts using the D programming language to debug and trace program execution at the user and kernel levels. Dtrace is designed to be integrated with the underlying operating system and support the tracing of different computer resources such as CPU scheduling, input/output activities, etc.

The Dtrace toolkit uses probes to handle events, and associates them with appropriate actions. Once a targeted event triggers, probes handles this event and gathers deep event information. A Dtrace script may have one or more probe clauses that accomplish a specific debugging objective. The following code shows a simple Dtrace script with one probe clause that reports the time on every system call entry. Overall, Dtrace syntax is complicated and difficult. Thus, Dtrace toolkit provided for its users a list of written tracing scripts that can be used for system debugging and monitoring.

**Listing 3.5: Dtrace script**

Dtrace

```
1 #!/usr/sbin/dtrace -s
2 syscall:::entry
3 {
4   trace(timestamp);
5 }
```

## 3.5. Tracing in Modeling Tools

In this section, we briefly introduce the tracing support in existing modeling tools. Tracing of state machines has been available in FXU tracer [53] and StateForge [54]. Both modeling tools allows the tracing of state machines and capturing all the behavior captured the traced state machine. No mechanisms were provided to limit the scope of state machine tracing or to trace other modeling constructs (e.g. association link cardinality).

In another modeling tool named Papyrus [55], visionary perspectives have been mentioned as planned tool enhancements to support tracing at the model level.

Existing tools, however, leave the following list of gaps with regards to model level trace specification:

- The ability to trace different modeling constructs in a modeling tool is generally missing, and lacks the flexibility to connect the tracing of a model construct to another modeling construct.
- No mechanisms have been proposed to limit the scope of tracing at the model level in terms of trace conditions, depth, etc.

- The necessary outputs of model-level tracing tools have been systematically examined, and there is no clear mapping between the trace specification and the resulting injected tracepoints into code generated from models. There has also been no attempt to supported multiple tracer technologies to output the trace messages.

Detailed discussion of these modeling tools will be given in Chapter 8 and they will be compared to current work presented in this thesis.

## 3.6. Conclusion

In this chapter, we discussed general principles of tracing technologies, and described the most popular. There are gaps in existing modeling tools that limit modelers' ability to systematically specify tracing at the model level. The MOTL language discussed starting in the next chapter aim at filling these gaps. MOTL is capable of generating tracepoints for primitive tracing, Java logging and LTTng as discussed in this chapter, and is flexible enough such that it could be extended fairly easily to handle Dtrace, but that is future work.

# Chapter 4. MOTL Trace Directive: Syntax and Semantics

Tracing at the model level adds an abstract level over traditional tracing performed at the code level. Creating a textual modeling language to handle abstract trace specification requires the description of its syntax and semantics. In this chapter, we investigate how MOTL handles abstract trace specification taking into consideration a number of requirements.

The requirements we have identified include the following:

I. **All model constructs are traceable**: Trace language syntax will have the ability to allow modelers to specify tracing of any desired model construct. For the purpose of our research we will focus on the tracing of attributes, state machines, and associations.

II. **Trace specification is flexible**: Modelers should have as many degrees of freedom as possible in constructing useful trace directives. In particular, they should be able to specify tracing of an association in the context of either associated class; they should also be able to trace one model construct with reference to other model constructs. For instance, tracing of a state entry can be linked to a monitoring of an attribute value; i.e. when an attribute changes, the current state is output as part of the generated trace output, or when the state changes, the current value of the attribute could be output.

III. **Transparently integrated language**: MOTL is a textual language; hence trace directives need to be written in a format that is harmonious with the textual representation of the model.

IV. **Constraints**: Modelers should have the ability to have control over the tracing scope of different model constructs, both temporally (starting and ending at certain times according to a variety of constraints) and at different levels for nested constructs such as state machines.

V. **Halt of tracing**: A mechanism should be provided for modelers to allow them to halt tracing when desired.

Tracing is specified using structured trace directives that can be placed anywhere inside an Umple class, other than method bodies, which are not considered in this thesis. The generalized syntax of a trace directive is as follows; it start with a keyword 'trace' and ends with a semicolon ';'. After the trace keyword, an Umple construct to be traced is specified (attribute, association, state, etc.). The scope of tracing can be limited using appropriate optional constraints and conditions that can switch tracing on or off in certain situations. An additional 'record' clause can be used to specify data that will be output.

**trace** <Prefix_Subdirectives> < Umple_constructs > <Postfix_Subdirectives> **;**

Trace directives specified at the model level inject appropriate tracepoints into source code generated from Umple constructs. The language is designed such that tracepoints should not be edited nor modified in the generated code. Rather, any required modification should be made at the model level. In this chapter, we introduce our trace directives to specify tracing of Umple constructs. First, we explain MOTL architecture by discussing its grammar and metamodel. Then, we explore trace directive syntax and semantics for modeling constructs: attributes, state machines, and associations. After that, we show how constraints can be imposed upon trace directives to control the scope of tracing.

## 4.1. Architecture

Umple was chosen as a base language to implement MOTL for a number of reasons. First, Umple has an extensive compiler infrastructure that is easy to extend with new features. Second, Umple has full implementation of associations, generating correct code for all combinations of multiplicity and navigability, as well as for state machines with unlimited nesting and concurrency [56, 57]. Lastly, Umple has an extensive testing framework allowing careful control of the quality of what we create [58].

In a forward model driven engineering approach, trace directives blended with Umple code will result in model-generated code instrumented with tracepoints. MOTL was designed to extend Umple architecture components as follows:

- **Parser**: The MOTL grammar was created to define the language features, and it was integrated into Umple as one of Umple's grammar files.
- **Analyzer**: The MOTL metamodel was written textually as part of the Umple metamodel.
- **Code generator**: Umple uses JET templates to generate Java systems. MOTL extended the JET templates to support tracepoint injections.

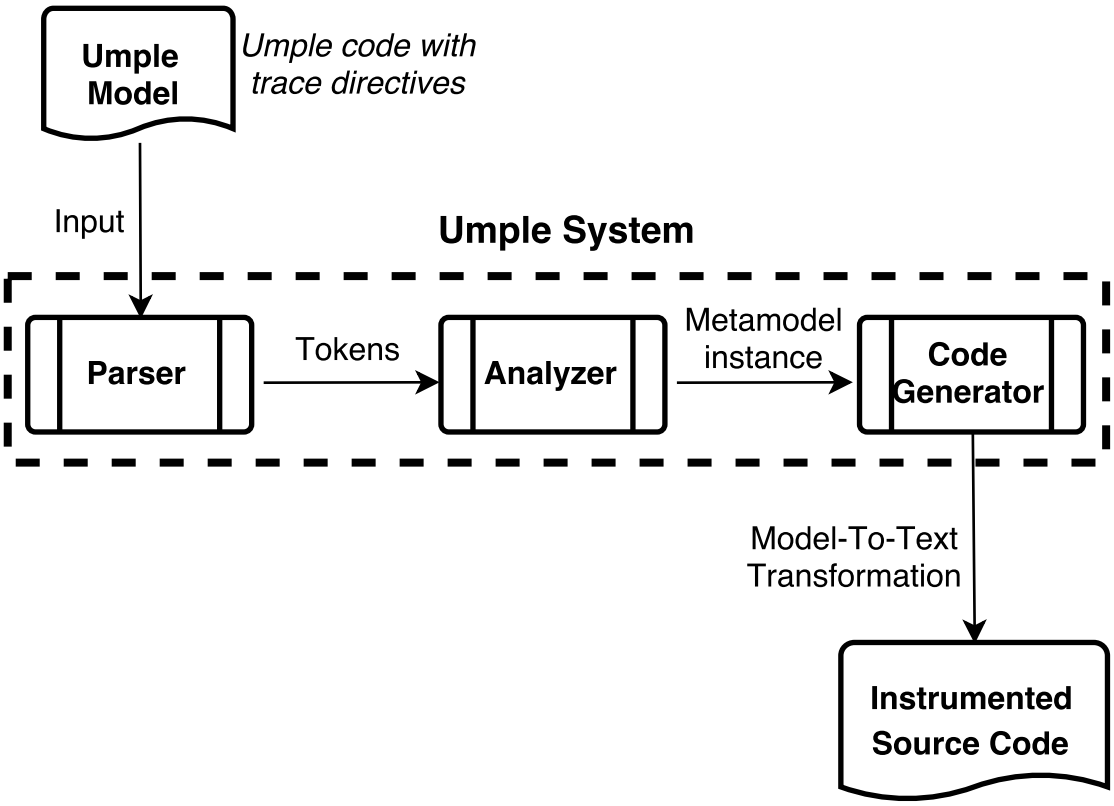


**Figure 4.1: MOTL architecture**

### 4.1.1 Grammar

We have developed a set of grammar rules to satisfy MOTL requirements and implemented them as part of Umple. There are a total of 15 grammar rules for the MOTL trace directive as listed in Table 4.1. Keywords and symbols are highlighted in red, terminals in green, and non-terminals in blue surrounded by double square brackets.

The first rule describes the general structure of a MOTL trace directive consisting of three sub rules. A trace entity can be preceded by an optional prefix and/or followed by many postfix sub directives.

R2 is a subrule that shows a trace entity being parsed as a terminal and additional trace entities can also be parsed. Declaration of different entities to be traced should be separated by a comma ‘,’.

R3 and R4 handle a prefix in a trace directive. Terminal ‘option’ will accept any of the provided strings in the listing to be parsed as a value.

R5 to R15 define postfix subdirectives in a trace directive. R5 is a subrule of the main trace directive rule (i.e. R1), where a postfix can be any of stated non-terminals. R6 defines the structure of a postfix condition with terminal ‘conditionType’ storing the type of condition parsed and details of the condition itself is captured in rule (constraintToken). Rule constraintToken is defined in the main Umple grammar, and MOTL is using this rule to delegate the handling of condition details to that rule. R7 defines how to handle occurrences postfixes. R8 and R9 define timing constraints. R10 is a rule designated to handle the depth of tracing in the case of nested state tracing.

R11 and R12 handle the record clause in a postfix subdirective. It allows the developer to record a string or multiple model constructs (e.g. attribute / state). These rules satisfy previously stated requirement 2.

R13 and R14 are special rules that can assign a trace directive to a logging level. All logging levels listed as an option in R14 are levels predefined in Java logging API and log4j.

The final rule (R15) is a rule that handles the execute clause in a trace directive, with terminal ‘traceExecute’ accepting any native code with two curly brackets ‘{}’.

**Table 4.1: Trace Directive Grammar Rules**

| Rule | Umple Grammar |
|---|---|
| *R1* | **traceDirective** : trace [[**Prefix**]]? [[**traceEntity**]] [[**Postfix**]]* ; |
| *R2* | **traceEntity**- : [traceEntity] ( , [traceEntity])* |
| *R3* | **Prefix** : [[**PrefixOption**]] ( , [[**PrefixOption**]] )* |
| *R4* | **PrefixOption**- : [=option:set \| get \| entry \| exit \| add \| remove] |
| *R5* | **Postfix**- : ( [[**traceCondition**]]<br>\|[[**traceFor**]]<br>\|[[**tracePeriod**]] |

| | |
|---|---|
| | \|[[**traceDuring**]]<br>\|[[**traceLevel**]]<br>\|[[**traceRecord**]]<br>\|[[**executeClause**]]<br>\|[[**logLevel**]] ) |
| *R6* | **traceCondition** :<br>[=conditionType:where\|until\|after\|giving]? [ [[**constraintToken**]] ] |
| *R7* | **traceFor-** : for [traceFor] |
| *R8* | **tracePeriod-** : period [tracePeriod] |
| *R9* | **traceDuring-** : during [traceDuration] |
| *R10* | **traceLevel-** : level [traceLevel] |
| *R11* | **traceRecord-** : record [[**recordEntity**]] |
| *R12* | **recordEntity-** : ( only )? ( " [**recordString] " \| [traceRecord] ) ( , [traceRecord] )* |
| *R13* | **logLevel-** : logLevel [[**logLevelOption**]] ( , [[**logLevelOption**]] )* |
| *R14* | **logLevelOption-** : [=logLevel:trace<br>\| debug<br>\| info<br>\| warn<br>\| error<br>\| fatal<br>\| all<br>\| finest<br>\| finer<br>\| fine<br>\| config<br>\| warning<br>\| severe ] |
| *R15* | **executeClause-** : execute { [**traceExecute] } |

Visualization of the MOTL trace directive grammar is provided as a syntax diagram in Figure 4.2, where dark blue rounded rectangles are keywords and terminal symbols and light blue rectangles are terminals. Syntax diagram was generated based on the grammar using Railroad Diagram Generator [59].

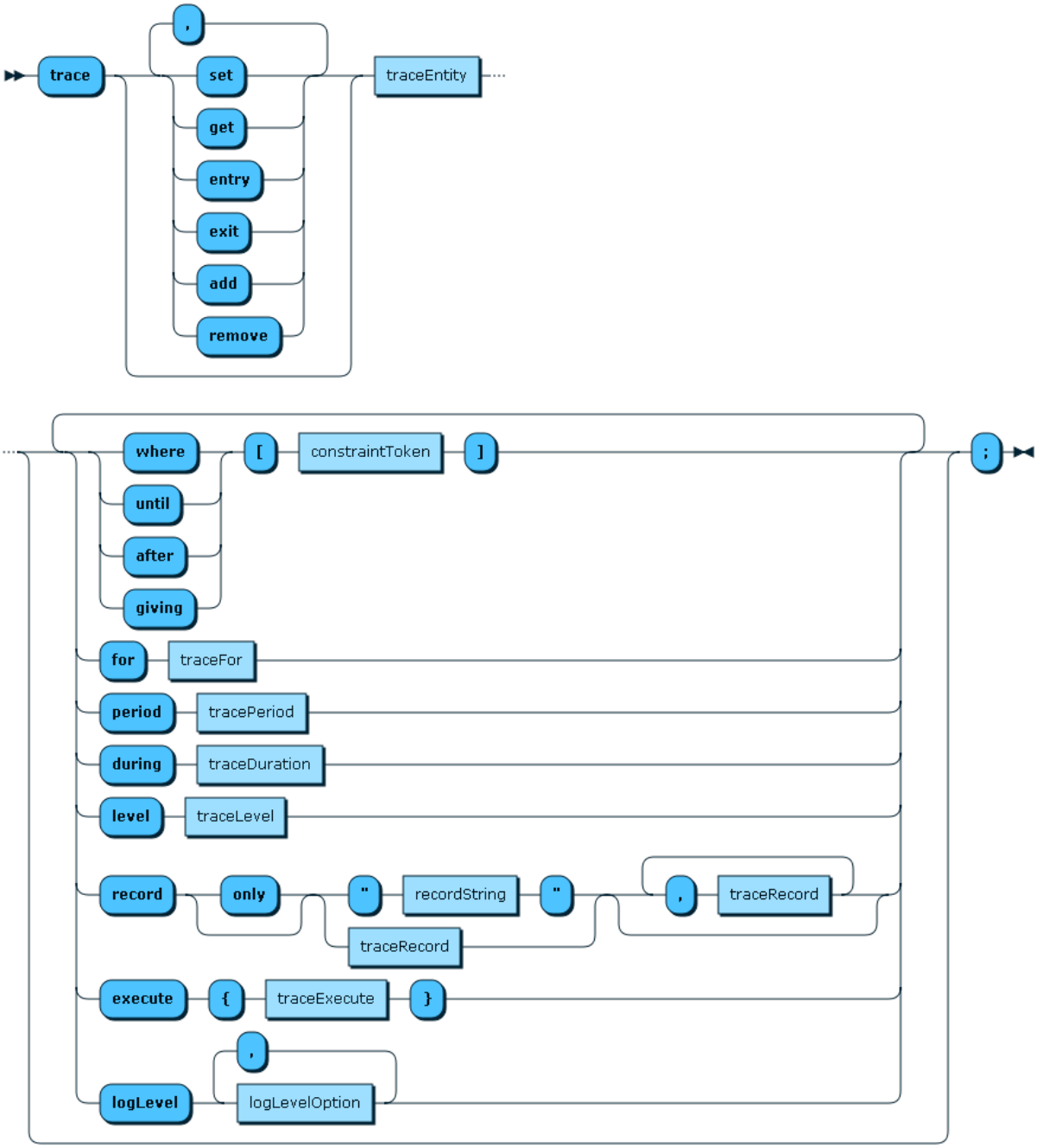


**Figure 4.2: Trace directive syntax diagram**

### 4.1.2 Metamodel

The trace directive metamodel is explored in Figure 4.3 and Figure 4.4:

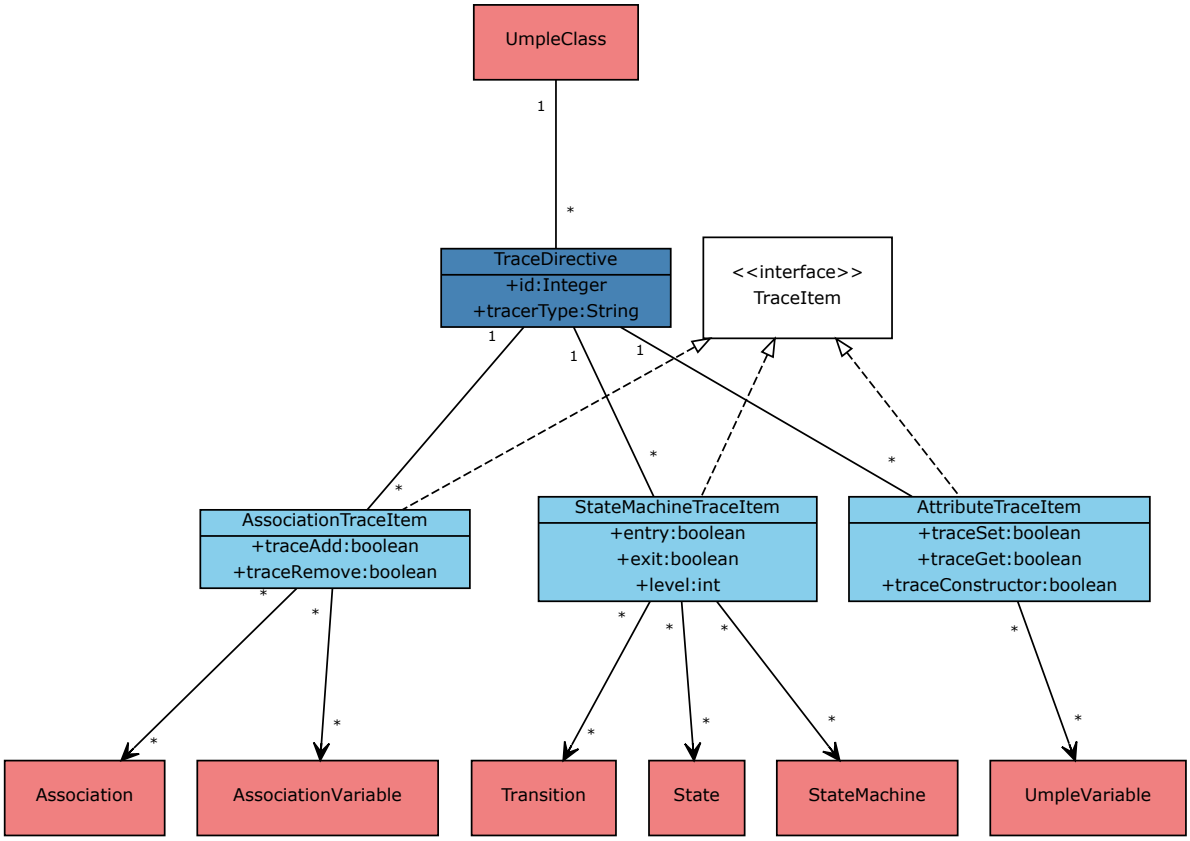


**Figure 4.3: Trace directive metamodel – part I**

- *TraceDirective* is a class at the center of the trace directive metamodel. An Umple class can have multiple trace directives but a trace directive belongs to only one Umple class. Each trace directive is uniquely identified by an integer id, and can be associated with multiple trace items of different Umple constructs. In cases where a trace directive has a postfix constraint, handling of the constraints is delegated to the postfix class.
- *TraceItem* is an interface that needs to be implemented whenever we create a trace item for a specific modeling construct. Any specialized trace item class that is targeting the tracing of an Umple construct needs to extend this class.
- *AttributeTraceItem* is a specialized class that manages the tracing of an attribute. This class can be associated with any Umple attribute with the objective of tracing that attribute. There are three Boolean attributes controlling attribute tracing mode: when value is modified, when the value is accessed, and when the attribute is initialized at the time of construction.

- *StateMachineTraceItem* handles what needs to be traced in a state machine. We can trace a specific event or multiple events. In addition, a state machine can be traced as a whole. Limiting tracing to state entry or exit can be accomplished through entry and exit Boolean attributes. Controlling the tracing depth of a nested state is indicated using the level integer attribute.
- *AssociationTraceItem* is another specialized trace item similar to StateMachine-TraceItem that handles what needs to be traced in an association. There are two Boolean attributes to indicate whether the tracing should be limited to association link addition or link removal.

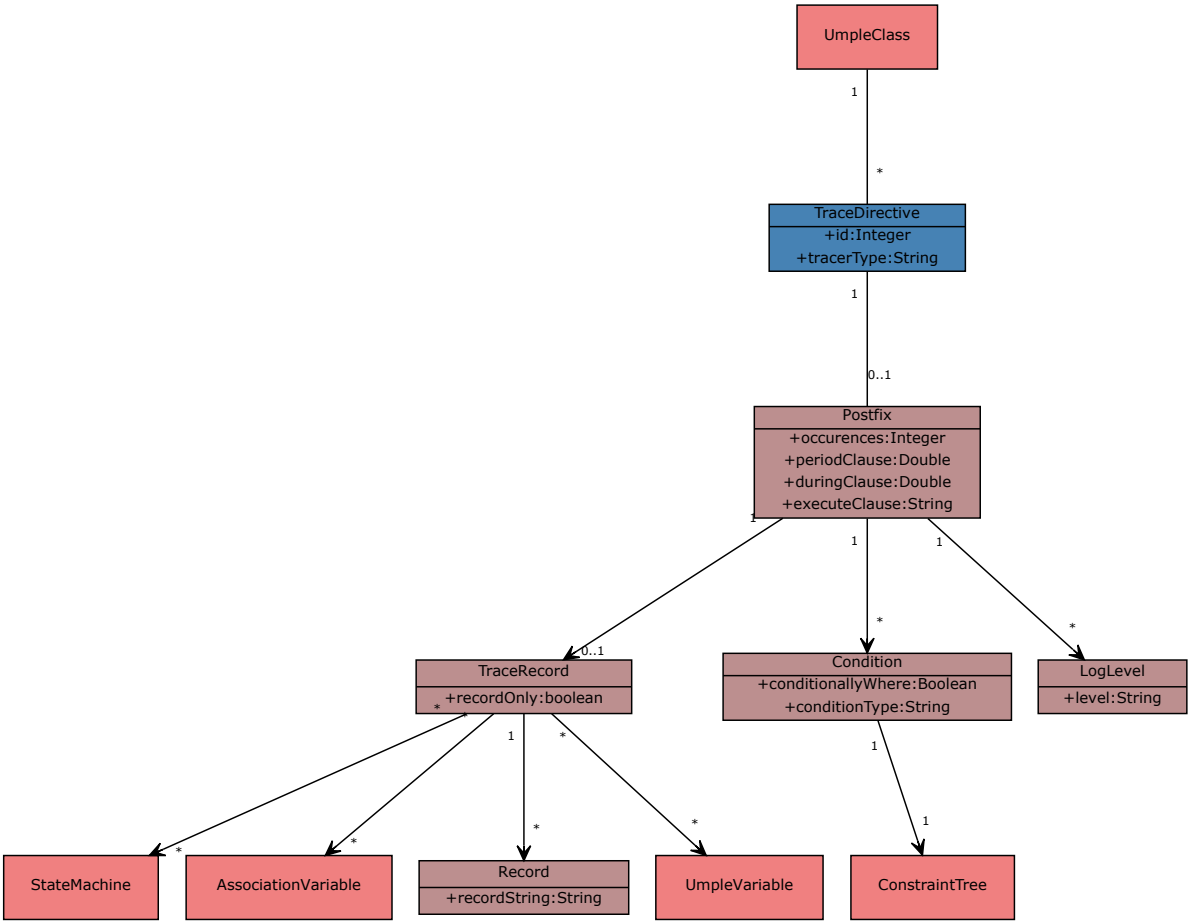


**Figure 4.4: Trace directive metamodel – part II**

- *Postfix* is a class that holds the control and specification of any postfix subdirective declared in a trace directive. It has three numeric attributes that store time values for time constraint postfixes, as well as another attribute to store native code passed by the execute clause, which will be explained in section 4.6.

- *TraceRecord* is a class that manages any 'record' statement in a postfix, where it can record: multiple strings, value of any Umple attribute, number of association link at run time, or current state in state machine.
- *Condition* is a class that manages conditionally constrained postfixes in form of (where/giving/after/until) conditions by creating an instance of the ConstraintTree class to contain the condition specification. ConstraintTree is part of the Umple metamodel core.
- *UmpleVariable*, *StateMachine*, *State*, *Transition*, *AssociationVariable* and *Association* classes are Umple classes that define the behavior of Umple constructs in the compiler, and pre-exist the work of this thesis. For readability of the trace directive metamodel, contents of these classes and their interaction with the other Umple metamodel classes have been hidden. Full details of Umple metamodel can be accessed online at [28].

Trace directive metamodel was textually modeled using Umple as shown in Listing 4.1.

**Listing 4.1: Trace directive metamodel Umple code**

Umple

```
1  class UmpleClass {
2    1 -- * TraceDirective;
3  }
4  class TraceDirective {
5    autounique id;
6    String tracerType = {getTracer().getName()}
7    1 -- * AttributeTraceItem;
8    1 -- * StateMachineTraceItem;
9    1 -- * AssociationTraceItem;
10   1 -- 0..1 Postfix;
11 }
12 class Postfix {
13   Integer occurences = 0;
14   Double periodClause = 0;
15   Double duringClause = 0;
16   executeClause = null;
17   1 -> * Condition;
18   1 -> 0..1 TraceRecord;
19   1 -> * LogLevel;
20 }
21 class Condition {
22   1 -> 1 ConstraintTree constraint;
```

```
23      Boolean conditionallyWhere = true;
24      conditionType = "where";
25  }
26  class TraceRecord {
27    Boolean recordOnly = false;
28    * -> * UmpleVariable;
29    * -> * StateMachine;
30    * -> * AssociationVariable;
31    1 -> * Record;
32  }
33  class Record {
34      String recordString;
35  }
36  class LogLevel {
37      String level;
38  }
39  interface TraceItem {
40    depend java.util.*;
41    public String getTracerType();
42    public Boolean getIsPre();
43    public Boolean getIsPost();
44  public String getExtremities(CodeTranslator
45  gen, String name);
46    Position position = null;
47  }
48  class AttributeTraceItem {
49    isA TraceItem;
50    Boolean traceSet = false;
51    Boolean traceGet = false;
52    Boolean traceConstructor = false;
53    * -> * UmpleVariable;
54  }
55  class StateMachineTraceItem {
56    isA TraceItem;
57    Boolean entry = false;
58    Boolean exit = false;
59    Integer level = -1;
60    * -> * StateMachine;
61    * -> * State;
62    * -> * Transition;
63  }
64  class AssociationTraceItem {
65    isA TraceItem;
66    * -> * AssociationVariable;
67    * -> * Association;
68    Boolean traceAdd = false;
69    Boolean traceRemove = false;
70  }
```

### 4.1.3 Separation of Concerns

Modelers can use MOTL in two modes: either directly annotating model elements, or writing a tracing script that is independent of the model. When used with Umple, such separate tracing scripts would be merged into an Umple model using Umple's mixin capability [10]. The last mode (i.e. independent trace script) allows tracing to be separated from the model code, and not pollute it with scattered tracing directives. However, selecting either mode is left for the modeler's preference.

Figure 4.5 shows a system modeled in three files with trace directives to trace modeling constructs in each file. As a separation of concern, all trace directives are grouped into a single trace file (trace.ump) independent from the model.

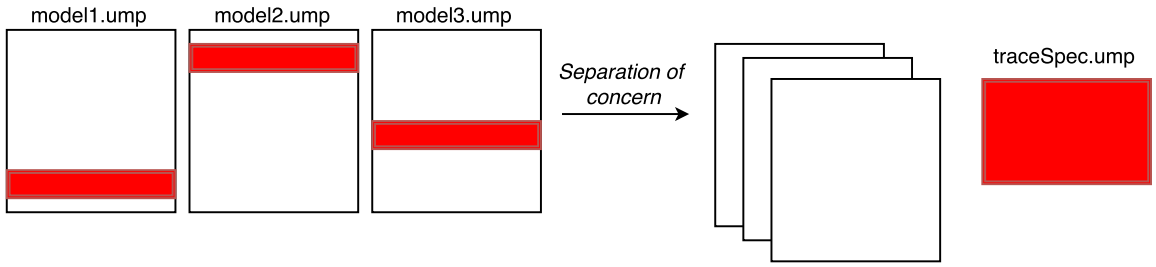


**Figure 4.5: Separation of concerns**

## 4.2. Syntax and Semantics of Attributes Tracing

Umple attributes defined at the model level can have special characteristics imposed by Umple stereotypes and other notations. Semantics of Umple attribute tracing is coupled with those stereotypes resulting in distinctive tracing handling for each different situation. In MOTL, Umple attribute tracepoint injection can occur in three modes: when the attribute value is accessed, modified, or both:

i. **Attribute value modified:** Umple attribute value manipulation can happen at the time of its construction or whenever its value has been changed.
ii. **Attribute value accessed:** Umple attribute value can be read through the get method.
iii. **Attribute value modified or accessed:** Tracing is desired at every location that modifies or accesses the value of traced attribute.

### 4.2.1 Tracepoints Injections for Attributes

In this section, we explore the possible tracepoint injection locations in the Umple attribute generated API with respect to different Umple attribute stereotypes. First, we start with the simplest form of Umple attribute declaration that does not have any additional characteristics as shown in the following listing.

```
Integer id;
```

When tracing basic attributes with no stereotypes or special conditions, there are three locations for tracepoint injection as shown in below listing:

**Tracepoint 1)** Constructor – attribute value is initialized

**Tracepoint 2)** Setter – attribute value is modified

**Tracepoint 3)** Getter – attribute value is accessed

**Listing 4.2: Generated code for simple Umple attribute with tracepoints**

Java

```
1  public class Student
2  {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6
7     //Student Attributes
8     private int id;
9
10    //------------------------
11    // CONSTRUCTOR
12    //------------------------
13
14    public Student(int aId)
15    {
16      id = aId;
17      // Tracepoint [1] – Attribute initialization
18    }
19
20    //------------------------
21    // INTERFACE
22    //------------------------
23
24    public boolean setId(int aId)
25    {
26      boolean wasSet = false;
27      // Tracepoint [2] – Attribute setter
```

```
28        id = aId;
29        wasSet = true;
30        return wasSet;
31      }
32
33      public int getId()
34      {
35        // Tracepoint [3] – Attribute getter
36        return id;
37      }
38  }
```

```
immutable String name;
```

Immutable attributes are settable once at the time of construction, resulting in tracepoints injected at:

**Tracepoint 1)** Constructor – immutable attribute value is initialized indefinitely

**Tracepoint 2)** Getter – immutable attribute value is accessed

**Listing 4.3: Generated code for immutable Umple attribute with tracepoints**

Java

```
1   public class Student
2   {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6
7     //Student Attributes
8     private String name;
9
10    //------------------------
11    // CONSTRUCTOR
12    //------------------------
13
14    public Student(String aName)
15    {
16      name = aName;
17      // Tracepoint [1] – Immutable attribute initialization
18    }
19
20    //------------------------
21    // INTERFACE
22    //------------------------
23
24    public String getName()
25    {
26      // Tracepoint [2] – Immutable attribute getter
```

```
    return name;
  }
}
```

```
lazy immutable name;
```

Umple immutable attributes can constrained further to be lazy requiring the generation of a set method that allows the attribute setting once. Tracepoints injection in case of immutable attributes:

**Tracepoint 1)** Setter – lazy immutable attribute value is settable once

**Tracepoint 2)** Getter – lazy immutable attribute value is accessed

**Listing 4.4: Generated code for lazy immutable Umple attribute with tracepoints**

Java

```
public class Student
{
  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //Student Attributes
  private String name;

  //Helper Variables
  private boolean canSetName;

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

  public Student()
  {
    canSetName = true;
  }

  //------------------------
  // INTERFACE
  //------------------------

  public boolean setName(String aName)
  {
    boolean wasSet = false;
    if (!canSetName) { return false; }
    canSetName = false;
    // Tracepoint [1] – lazy immutable attribute setter
    name = aName;
```

```
33      wasSet = true;
34      return wasSet;
35    }
36
37    public String getName()
38    {
39      // Tracepoint [2] – lazy immutable attribute getter
40      return name;
41    }
42  }
```

```
Double height;
Double width;
Double area = {height*width}
```

A single tracepoint is possible when tracing derived attributes due to the fact that their value is calculated based on the value of other attributes. As shown in the following code listing, the area attribute is calculated based on attributes height and width. Tracing of the derived attribute area will inject a single tracepoint

**Listing 4.5: Generated code for derived Umple attribute with tracepoints**

Java

```
1  public class Rectangle
2  {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6
7     //Rectangle Attributes
8     private double height;
9     private double width;
10
11    //------------------------
12    // CONSTRUCTOR
13    //------------------------
14
15    public Rectangle(double aHeight, double aWidth)
16    {
17      height = aHeight;
18      width = aWidth;
19    }
20
21    //------------------------
22    // INTERFACE
23    //------------------------
24
25    public boolean setHeight(double aHeight)
```

```
26      {
27        boolean wasSet = false;
28        height = aHeight;
29        wasSet = true;
30        return wasSet;
31      }
32
33      public boolean setWidth(double aWidth)
34      {
35        boolean wasSet = false;
36        width = aWidth;
37        wasSet = true;
38        return wasSet;
39      }
40
41      public double getHeight()
42      {
43        return height;
44      }
45
46      public double getWidth()
47      {
48        return width;
49      }
50
51      public double getArea()
52      {
53        // Single Tracepoint – Derived Attribute
54        return height*width;
55      }
56  }
```

```
unique Integer id;
```

Unique Umple attributes ensure a given attribute value is not repeated among the class instances. Manipulation of such attributes requires more generated API methods. Tracing unique Umple attributes will inject tracepoints in its setter and all three generated getter methods.

**Listing 4.6: Generated code for unique Umple attribute with tracepoints**

Java

```
1  public class Student
2  {
3     //------------------------
4     // STATIC VARIABLES
5     //------------------------
6
7     private static Map<Integer, Student> studentsById
8                     = new HashMap<Integer, Student>();
9
10    //------------------------
11    // MEMBER VARIABLES
12    //------------------------
13
14    //Student Attributes
15    private int id;
16
17    //------------------------
18    // CONSTRUCTOR
19    //------------------------
20
21    public Student(int aId)
22    {
23      if (!setId(aId))
24      {
25        throw new RuntimeException("duplicate id");
26      }
27    }
28
29    //------------------------
30    // INTERFACE
31    //------------------------
32
33    public boolean setId(int aId)
34    {
35      boolean wasSet = false;
36      Integer anOldId = getId();
37      if (hasWithId(aId)) {
38        return wasSet;
39      }
40      // Tracepoint [1] – unique attribute setter
41      id = aId;
42      wasSet = true;
43      if (anOldId != null) {
44        studentsById.remove(anOldId);
45      }
46      studentsById.put(aId, this);
47      return wasSet;
48    }
49
50    public int getId()
```

```
51      {
52        // Tracepoint [2] – unique attribute getter
53        return id;
54      }
55
56      public static Student getWithId(int aId)
57      {
58        // Tracepoint [3] – unique attribute getter
59        return studentsById.get(aId);
60      }
61
62      public static boolean hasWithId(int aId)
63      {
64        // Tracepoint [4] – unique attribute getter
65        return getWithId(aId) != null;
66      }
67
68      public void delete()
69      {
70        studentsById.remove(getId());
71      }
72  }
```

```
autounique id;
```

In case of attribute autouniqueness, Umple's generated API can handle attribute uniqueness automatically by declaring a static incremental integer attribute uniquely assigned to each new class object. Tracing autounique Umple attributes will inject tracepoints in the following:

**Tracepoint 1)** Constructor – attribute value is initialized to unique value

**Tracepoint 2)** Getter – accessing attribute unique value

**Listing 4.7: Generated code for autounique Umple attribute with tracepoints**

Java

```
1  public class Student
2   {
3     //------------------------
4     // STATIC VARIABLES
5     //------------------------
6
7     private static int nextId = 1;
8
9     //------------------------
10    // MEMBER VARIABLES
11    //------------------------
12
```

```
13      //Autounique Attributes
14      private int id;
15
16      //------------------------
17      // CONSTRUCTOR
18      //------------------------
19
20      public Student()
21      {
22        id = nextId++;
23        // Tracepoint [1] – autounique attribute initialization
24      }
25
26      //------------------------
27      // INTERFACE
28      //------------------------
29
30      public int getId()
31      {
32        // Tracepoint [2] – autounique attribute getter
33        return id;
34      }
35  }
```

```
defaulted String faculty = "fgps";
```

Umple defaulted attributes generate API containing methods that allows the retrieval of the default value and the reset of current value to the default in addition to setter/getter methods. Tracing defaulted Umple attributes will inject tracepoints in the following:

**Tracepoint 1)** Setter – defaulted attribute value is modified

**Tracepoint 2)** Reset – attribute value is reset to default

**Tracepoint 3)** Getter – accessing attribute value

**Tracepoint 4)** Default getter - requesting attribute default value

**Listing 4.8: Generated code for defaulted Umple attribute with tracepoints**

Java

```
1  public class GraduateStudent
2   {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6
7     private String faculty;
8
9     //------------------------
10    // CONSTRUCTOR
11    //------------------------
12    public GraduateStudent()
13    {
14      resetFaculty();
15    }
16
17    //------------------------
18    // INTERFACE
19    //------------------------
20
21    public boolean setFaculty(String aFaculty)
22    {
23      boolean wasSet = false;
24      // Tracepoint [1] – defaulted attribute setter
25      faculty = aFaculty;
26      wasSet = true;
27      return wasSet;
28    }
29
30    public boolean resetFaculty()
31    {
32      boolean wasReset = false;
33      // Tracepoint [2] – reset attribute to defaulted
34      faculty = getDefaultFaculty();
35      wasReset = true;
36      return wasReset;
37    }
38
39    public String getFaculty()
40    {
41      // Tracepoint [3] – defaulted attribute getter
42      return faculty;
43    }
44
45    public String getDefaultFaculty()
46    {
47      // Tracepoint [4] – defaulted value getter
48      return "fgps";
49    }
50  }
```

```
Integer[] phoneLine;
```

Umple multivalued attributes generate a wide range of API methods for value manipulation. Tracing multivalued Umple attributes will inject tracepoints in the following:

**Tracepoint 1)** Addition – a value is added to attribute

**Tracepoint 2)** Removal – a value is removed from attribute

**Tracepoint 3)** Getter – accessing attribute value

**Tracepoint 4)** Second getter – requesting all attribute values

**Tracepoint 5)** Count – calculating number of attribute values

**Tracepoint 6)** Has value – inquiring if there is at least one value

**Tracepoint 7)** Index – requesting index of a specific attribute

**Listing 4.9: Generated code for multivalued Umple attribute with tracepoints**

Java

```
1  public class Student
2   {
3     //------------------------
4     // MEMBER VARIABLES
5     //------------------------
6
7     //Student Attributes
8     private List<Integer> phoneLine;
9
10    //------------------------
11    // CONSTRUCTOR
12    //------------------------
13
14    public Student()
15    {
16      phoneLine = new ArrayList<Integer>();
17    }
18
19    //------------------------
20    // INTERFACE
21    //------------------------
22
23    public boolean addPhoneLine(Integer aPhoneLine)
24    {
25      boolean wasAdded = false;
26      // Tracepoint [1] – attribute value addition
27      wasAdded = phoneLine.add(aPhoneLine);
28      return wasAdded;
29    }
```

```
30
31      public boolean removePhoneLine(Integer aPhoneLine)
32      {
33        boolean wasRemoved = false;
34        // Tracepoint [2] – attribute value removal
35        wasRemoved = phoneLine.remove(aPhoneLine);
36        return wasRemoved;
37      }
38
39      public Integer getPhoneLine(int index)
40      {
41        Integer aPhoneLine = phoneLine.get(index);
42        // Tracepoint [3] – attribute value getter
43        return aPhoneLine;
44      }
45
46      public Integer[] getPhoneLine()
47      {
48        Integer[] newPhoneLine = phoneLine.toArray(new Inte-
49  ger[phoneLine.size()]);
50        // Tracepoint [4] – attribute value getter
51        return newPhoneLine;
52      }
53
54      public int numberOfPhoneLine()
55      {
56        int number = phoneLine.size();
57        // Tracepoint [5] – number of attribute values
58        return number;
59      }
60
61      public boolean hasPhoneLine()
62      {
63        boolean has = phoneLine.size() > 0;
64        // Tracepoint [6] – multivalued attribute inquiry
65        return has;
66      }
67
68      public int indexOfPhoneLine(Integer aPhoneLine)
69      {
70        int index = phoneLine.indexOf(aPhoneLine);
71        // Tracepoint [7] – multivalued attribute index
72        return index;
73      }
74  }
```

Finally, tracing of constant attributes is meaningless due to their unchangeable values and non-generation of API methods.

### 4.2.2 Prefix Subdirectives

Prefix subdirectives aim to control the injection of tracepoints for traced Umple attributes. The following are keywords can be used as prefixes for Umple attributes:

i. *Attribute value modified***:** Using keyword 'set' as a prefix indicates that tracing is desired in when attribute is initialized and modified. This is also the default case when no prefix is specified before the traced attribute in a trace directive.

**trace set** <Umple_Attribute>;

ii. *Attribute value accessed***:** Using the 'get' keyword as a prefix indicates that tracing is desired only whenever an attribute value is accessed by a call to its get method.

**trace get** <Umple_Attribute>;

iii. *Attribute value modified or accessed***:** Full tracing of an Umple attribute can be accomplished when both keywords 'set' and 'get' separated by a comma ',' is used as a prefix.

**trace set,get** <Umple_Attribute>;

Table 4.2 summarizes the mapping between prefix subdirectives and tracepoints injection locations in Umple generated API for Umple various supported attribute stereotypes. As an example, let us consider the case of defaulted attribute tracing. Tracing of such attributes with a 'set' prefix results in the injection of tracepoints when attribute value is initialized in the constructor, whenever the value is modified in the set method, and whenever the attribute value is reset using the reset method. As stated, handling of a trace directive with a prefix 'set' keyword is the same as the handling of a trace directive with no prefix, meaning that having a trace directive with only 'set' as prefix is dispensable and can be omitted.

**Table 4.2: Attributes prefix subdirectives mapping to tracepoint injection locations**

| | | API generated from Umple Attributes where Tracepoints are injected | | | | | |
|---|---|---|---|---|---|---|---|
| **Attribute** | **Prefix** | **Constructor argument** | **set** | **get** | **reset** | **add** | **remove** |
| *Lazy* | set | - | x | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | - | x | x | - | - | - |
| *Unique* | set | x | x | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | x | x | x | - | - | - |
| *Autounique* | set | x | - | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | x | - | x | - | - | - |
| *Defaulted* | set | x | x | - | x | - | - |
| | get | - | - | x | - | - | - |
| | both | x | x | x | x | - | - |
| *Derived* | set | - | - | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | - | - | x | - | - | - |
| *Multivalued* | set | - | - | - | - | x | x |
| | get | - | - | x | - | - | - |
| | both | - | - | x | - | x | x |
| *Immutable* | set | x | - | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | x | - | x | - | - | - |
| *lazy immutable* | set | - | x | - | - | - | - |
| | get | - | - | x | - | - | - |
| | both | - | x | x | - | - | - |
| *Constant* | none | - | - | - | - | - | - |

## 4.3. Syntax and Semantics of State Machine Tracing

As discussed in Chapter 2, state machines are representations of system behavior. States can have entry, transition and exit actions as well as do activities; they can also be composite (i.e. nested) or concurrent. Trace directives provide modelers with the capability of specifying tracing of any state machine element:

- **Simple State:** Tracing a simple state will trace whenever we transit into this traced state by reporting the event that caused this transition and any entry actions executed. In addition, tracing will report any transition out from a state by reporting the event that caused this transition and any exit actions executed.
- **Nested state:** Tracing of a nested state will trace all of its substates. In other words, all substates will be traced individually as simple states to the full depth of nesting.
- **Concurrent state:** Tracing of a state with concurrent states will trace all of its concurrent states in a same manner as tracing an individual trace.
- **State with a Do-Activity:** Tracing a state with a do activity, will trace whenever a do-activity starts (i.e. thread created). Upon exiting that state the thread is interrupted, and a tracepoint is injected to say it is interrupted (i.e. thread was interrupted).
- **Events:** Triggering of an event will result in a transition from an original state to another destination state. Tracing of an event will trace any occurrences of the traced event reporting the name of original state, the destination state and the triggering event. A given event can trigger multiple transitions; each will be traced separately.
- **Parameterized event:** tracing a parameterized event is similar to tracing of an event. However, event parameter is traced whenever the event is triggered.

In addition to the capability of tracing different state machine elements, tracing a state machine as a whole is possible, causing the tracing of all its elements.

### 4.3.1 Tracepoint Injection for State Machines

Tracing specification of different state machine elements injects tracepoints in state machine generated code where appropriate. To further explain our state machine tracing semantics and where tracepoints will be injected based on traced state machine element, we consider tracing of different state machine elements in the symbolic state machine drawn in Figure 4.6.

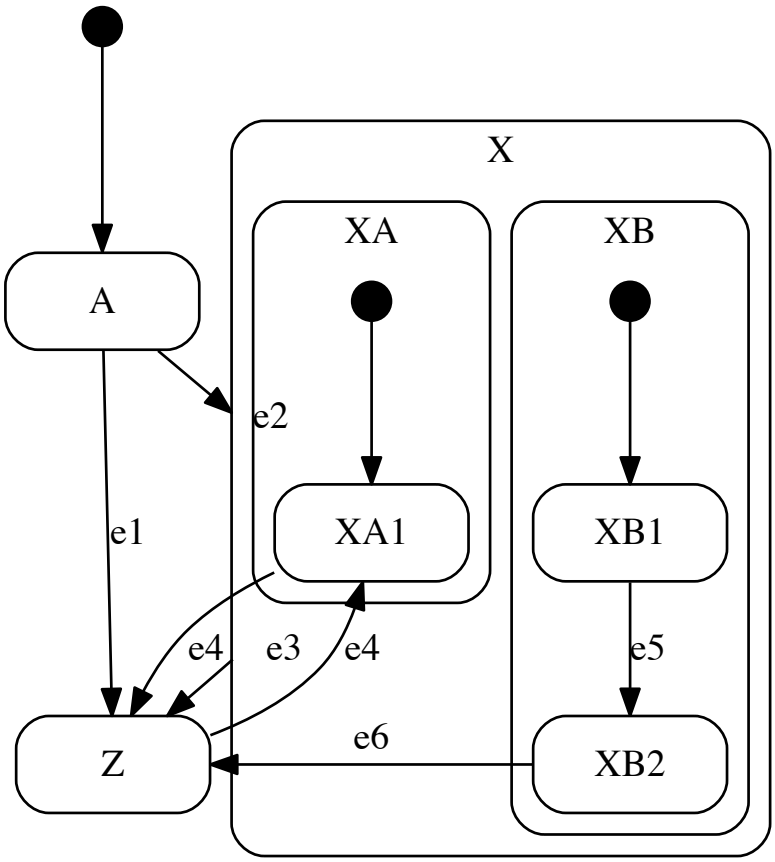


**Figure 4.6: Exploring state machine tracing semantics**

```
trace A;
```

In Figure 4.7, tracing simple state A will result in tracing of:

- Entry to state A as the initial state when state machine instance is created.
- Event e1, which will trigger an exit transition from simple state A to simple state Z.
- Event e2, which will trigger an exit transition from simple state A to concurrent state X.
- Reporting of any exit actions associated with the transitions taken by either e1 or e2.

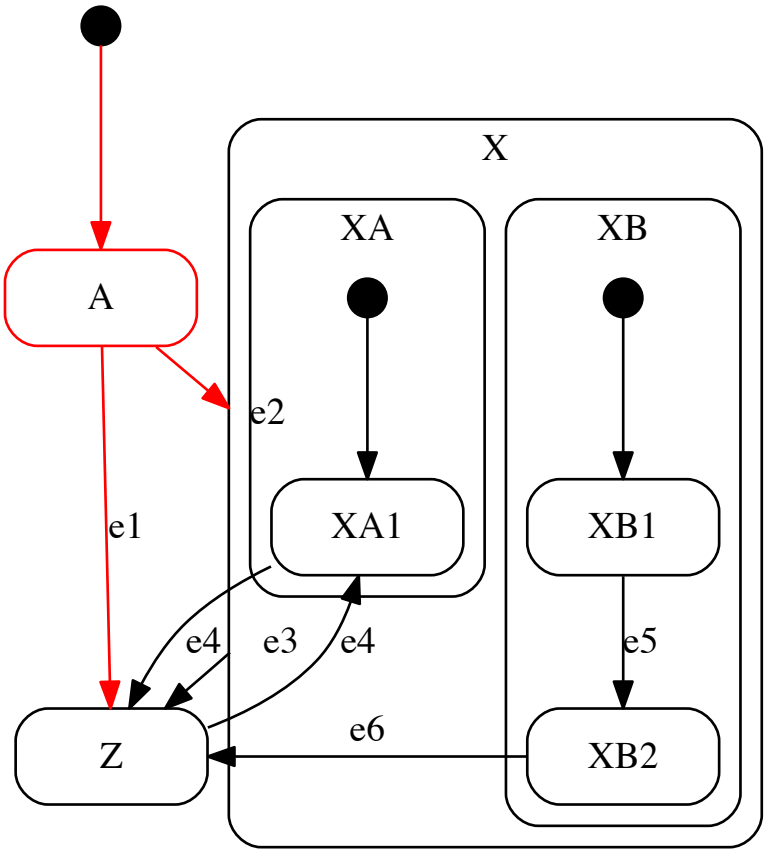


**Figure 4.7: Tracing simple state A in symbolic state machine**

```
trace Z;
```

In Figure 4.8, tracing simple state Z will result in tracing of:

- Event e1, which will trigger an entry transition to simple state Z from simple state A.
- Event e4, which will trigger an entry transition to simple state Z from nested state XA1.
- Event e3, which will trigger an entry transition to simple state Z from composite state X.
- Event e6, which will trigger an entry transition to simple state Z from nested state XB2.
- Event e4, which will trigger an exit transition from simple state Z to nested state XA1.
- Any entry actions associated with e1, e4, e3, or e6.
- Any exit actions associated with e4.

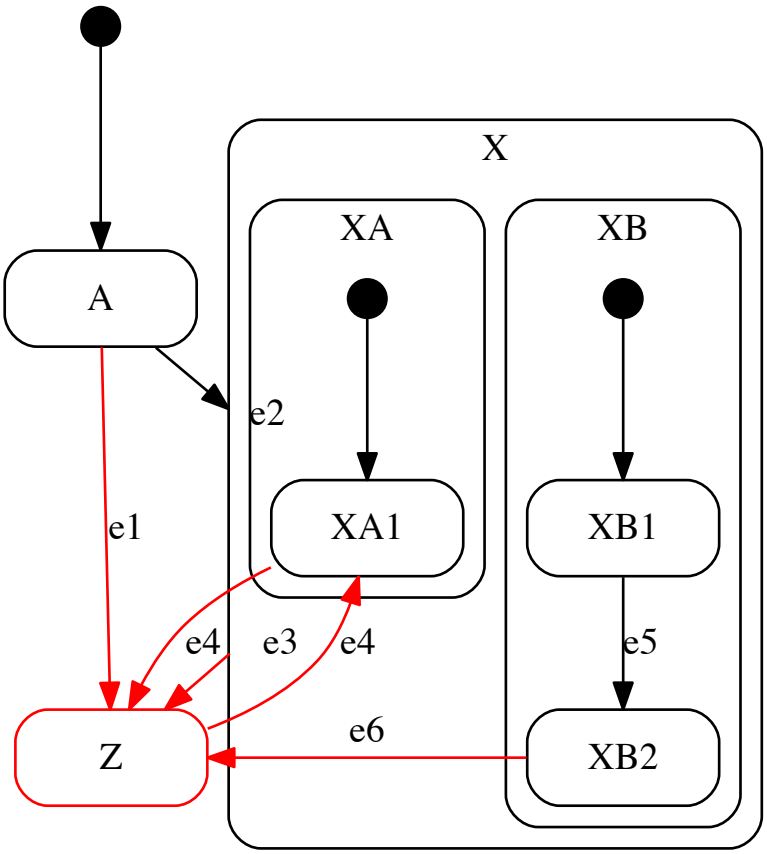


**Figure 4.8: Tracing simple state Z in symbolic state machine**

```
trace XB;
```

In Figure 4.9, tracing nested state XB will trace all its substates until the nesting depth is reached:

- Tracing entry of initial state XB1 when transiting to nested state XB.
- Tracing of event e5 that will trigger a transition from substate XB1 to substate XB2.
- Tracing of event e6 that will trigger an exit transition from nested state XB2, then forcing an exit from composite state X to simple state Z.
- Reporting of any entry/exit actions associated with e5.
- Reporting of any exit actions associated with e6.

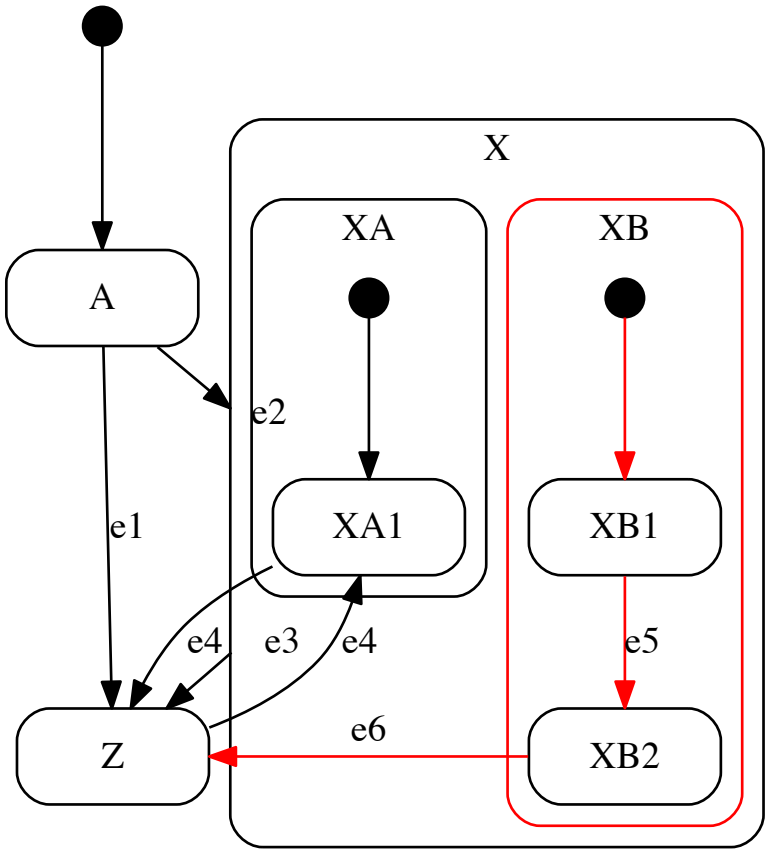


**Figure 4.9: Tracing nested state XB in symbolic state machine**

```
trace X;
```

In Figure 4.10, tracing composite state X will trace all its concurrent states XA and XB:

- Tracing entry of initial state XA1 when transiting to nested state XA.
- Tracing of event e2 that triggers an entry to state X from simple state A.
- Tracing of event e4 that will trigger an exit transition from nested state A, then forcing an exit from composite state X to simple state Z.
- Tracing of event e3 that will force an exit from state X (i.e. exit from all its concurrent states) to simple state Z.
- Tracing entry of initial state XB1 when transiting to state XB.
- Tracing of event e5 that will trigger a transition from substate XB1 to substate XB2.
- Tracing of event e6 that will trigger an exit transition from state XB2, then forcing an exit from composite state X to simple state Z.
- Reporting of any entry/exit actions associated with e5.
- Reporting of any exit actions associated with e4, e3, or e6.
- Reporting of any entry actions associated with e2 or e4.

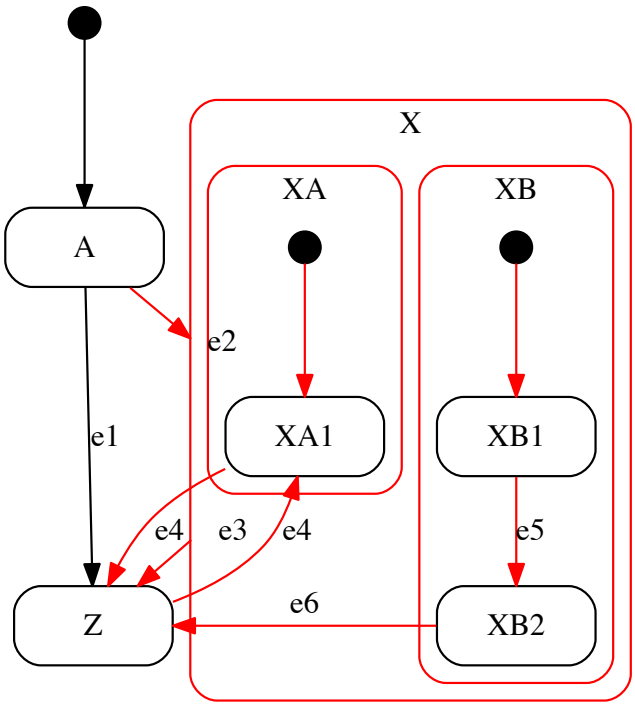


**Figure 4.10: Tracing composite state X in symbolic state machine**

```
trace e4;
```

Tracing of an event will trace any transitions caused by the triggering of this event:

- Tracing of event e4 that triggers a transition from nested state XA1 to simple state Z forcing an exit from composite state X.
- Tracing of event e4 that triggers a transition from simple state Z to nested state XA1.

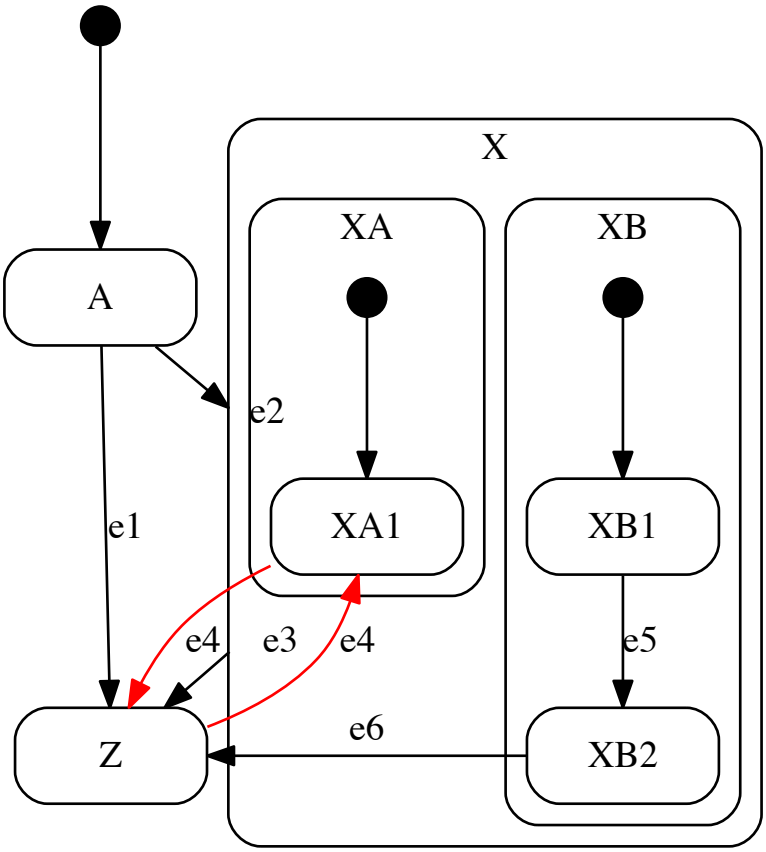


**Figure 4.11: Tracing event 4 in symbolic state machine**

### 4.3.2 Prefix Subdirectives

Tracing a state normally involves the tracing of its entries and exits, but tracing can be limited to either case. In a trace directive, we can use the 'entry' keyword as a prefix before a traced state to indicate tracing of its entries only, or the 'exit' keyword as a prefix to indicate tracing of its exits:

i. *State entry*: Monitors any incoming transition to the traced state. Records the previous state with the event that was triggered to transit to the traced state. Entry actions will be recorded if there are any.

**trace entry** <Umple_State> ;

ii. *State exit*: Monitors outgoing transitions from the traced state. Records the traced state with the event that triggered the transition. Any exit actions are recorded.

**trace exit** <Umple_State> ;

## 4.4. Syntax and Semantics of Association Tracing

Associations describe the relationships between class instances; role names are used to clarify the relation at both ends of an association. Tracing associations as the linked objects vary over time is an interesting capability introduced by MOTL. The tracing of associations in MOTL is similar to the tracing of attributes, but with some key differences:

- The name of the association to be traced is either the class at the 'other' end of the as-association or the role name (if this exists). If there are multiple associations to other classes, then a role name is required. A role name is also required for reflexive associations.
- The actual trigger for adding or removing a link can be from either end of a bi-directional association. The resulting trace would be the same.

Trace directives trace associations by referring to the role names:

**trace** <Umple_RoleName> **;**

In Umple if a role name is not explicitly included, then the name of the class is used as the role name implicitly. Umple raises an error message if this is not possible due to name clashes.

Tracing can occur whenever an association link is added or deleted at run time; possible uses of this are to understand how the cardinality of an association varies over time, or to detect unusual association links. Semantically, in Umple, a trace directive tracing an association end through the role name should be written in the containing class of the inline association declaration as shown in Listing 4.10. In this example, the trace directive will trace any addition or deletion of a link of the association between Student and Mentor (referred to as the supervisor).

**Listing 4.10: Trace directive of an association role name**

Umple

```
1  class Student {
2    * -- 1 Mentor supervisor;
3    trace supervisor;
4  }
5  class Mentor {}
```

### 4.4.1 Tracepoint Injections for Associations

Tracepoints are injected into the generated methods that are part of the Umple association API. The code generated will vary depending on details of the association's specification. We will consider the following inline association, where class Student is linked to class Mentor:

```
* -- 0..1 Mentor supervisor;
```

Tracing of an association end with an optional-one multiplicity, such as above, will generate the following tracepoints:

**Tracepoint 1)** Setter – association link is being set

**Tracepoint 2)** Delete – Optional link is deleted

Listing 4.11 shows the generated code with the tracepoint highlighted.

**Listing 4.11: Generated code for traced association end of optional one multiplicity**

Java

```
1  public class Student
2  {
3    //------------------------
4    // MEMBER VARIABLES
5    //------------------------
6
7    //Student Associations
8    private Mentor supervisor;
9
10   //------------------------
11   // CONSTRUCTOR
12   //------------------------
13
14   public Student()
15   {}
16
17   //------------------------
18   // INTERFACE
19   //------------------------
20
21   public Mentor getSupervisor()
22   {
23     return supervisor;
24   }
25
26   public boolean hasSupervisor()
27   {
28     boolean has = supervisor != null;
29     return has;
30   }
31
32   public boolean setSupervisor(Mentor aSupervisor)
33   {
34     boolean wasSet = false;
35     Mentor existingSupervisor = supervisor;
36     supervisor = aSupervisor;
37     // Tracepoint [1] – association link setter
38     if (existingSupervisor != null &&
39        !existingSupervisor.equals(aSupervisor))
40     {
41       existingSupervisor.removeStudent(this);
42     }
43     if (aSupervisor != null)
44     {
45       aSupervisor.addStudent(this);
46     }
47     wasSet = true;
48     return wasSet;
49   }
50
```

```
51    public void delete()
52    {
53      if (supervisor != null)
54      {
55        Mentor placeholderSupervisor = supervisor;
56        this.supervisor = null;
57        placeholderSupervisor.removeStudent(this);
58        // Tracepoint [2] – association link deletion
59      }
60    }
61  }
```

Now, let us consider a 'mandatory' association end.

```
* -- 1 Mentor supervisor;
```

Similarly to the tracing of an optional-one association end, tracing of an association end with a mandatory-one multiplicity will generate tracepoints in similar locations, with an additional tracepoint in the set method although it is being called in the constructor to set the association link. Thus, tracepoints will be injected in:

**Tracepoint 1)** Constructor – when object is created, the mandatory association link must be created, otherwise an exception will be generated. A tracepoint is injected in the constructor to trace whether the link has been set properly.

**Tracepoint 2)** Setter – an association link is being set

**Tracepoint 3)** Delete – link is deleted due to object deletion

Listing 4.12 shows the generated code with the tracepoint highlighted.

**Listing 4.12: Generated code for traced association end of mandatory one multiplicity**

Java

```
1  public class Student
2  {
3    //------------------------
4    // MEMBER VARIABLES
5    //------------------------
6
7    //Student Associations
8    private Mentor supervisor;
9
10   //------------------------
11   // CONSTRUCTOR
12   //------------------------
13
```

```
14    public Student(Mentor aSupervisor)
15    {
16      boolean didAddSupervisor = setSupervisor(aSupervisor);
17      // Tracepoint [1] – association link construction
18      if (!didAddSupervisor)
19      {
20        throw new RuntimeException("Unable to create student");
21      }
22    }
23
24    //------------------------
25    // INTERFACE
26    //------------------------
27
28    public Mentor getSupervisor()
29    {
30      return supervisor;
31    }
32
33    public boolean setSupervisor(Mentor aSupervisor)
34    {
35      boolean wasSet = false;
36      if (aSupervisor == null)
37      {
38        return wasSet;
39      }
40
41      Mentor existingSupervisor = supervisor;
42      supervisor = aSupervisor;
43      // Tracepoint [2] – association link setter
44      if (existingSupervisor != null &&
45          !existingSupervisor.equals(aSupervisor))
46      {
47        existingSupervisor.removeStudent(this);
48      }
49      supervisor.addStudent(this);
50      wasSet = true;
51      return wasSet;
52    }
53
54    public void delete()
55    {
56      Mentor placeholderSupervisor = supervisor;
57      this.supervisor = null;
58      placeholderSupervisor.removeStudent(this);
59      // Tracepoint [3] – association link deletion
60    }
61  }
```

Next, we will consider a mandatory association where the fixed number is greater than one.

```
* -- 7 Mentor supervisor;
```

Tracing of an association end with a mandatory (n) multiplicity will generate the following tracepoints:

**Tracepoint 1)** Constructor – Upon object creation, the mandatory association links must be created at the time of object construction.

**Tracepoint 2)** Add – The Umple API was designed so that the setter method will call the add methods (n) times to set the (n) mandatory links. Tracing should be done whenever a link has been created. Thus, we inject a tracepoint in the add method to trace every newly created link between the relationship ends.

**Tracepoint 3)** Remove – a link is removed

**Tracepoint 4)** Delete – links are cancelled due to object deletion

Listing 4.13 shows the generated code with the tracepoint highlighted.

**Listing 4.13: Generated code for traced association end of mandatory (n) multiplicity**

Java

```
1  public class Student
2  {
3    //------------------------
4    // MEMBER VARIABLES
5    //------------------------
6
7    //Student Associations
8    private List<Mentor> supervisor;
9
10   //------------------------
11   // CONSTRUCTOR
12   //------------------------
13
14   public Student(Mentor... allSupervisor)
15   {
16     supervisor = new ArrayList<Mentor>();
17     boolean didAddSupervisor = setSupervisor(allSupervisor);
18     // Tracepoint [1] – association link construction
19     if (!didAddSupervisor)
20     {
21       throw new RuntimeException("Unable to create Student");
22     }
23   }
24
25   //------------------------
```

```
26    // INTERFACE
27    //------------------------
28
29    public boolean addSupervisor(Mentor aSupervisor)
30    {
31      boolean wasAdded = false;
32      if (supervisor.contains(aSupervisor)) { return false; }
33      if (numberOfSupervisor() >= maximumNumberOfSupervisor())
34      {
35        return wasAdded;
36      }
37
38      supervisor.add(aSupervisor);
39      // Tracepoint [2] — association link addition
40      if (aSupervisor.indexOfStudent(this) != -1)
41      {
42        wasAdded = true;
43      }
44      else
45      {
46        wasAdded = aSupervisor.addStudent(this);
47        if (!wasAdded)
48        {
49          supervisor.remove(aSupervisor);
50        }
51      }
52      return wasAdded;
53    }
54
55    public boolean removeSupervisor(Mentor aSupervisor)
56    {
57      boolean wasRemoved = false;
58      if (!supervisor.contains(aSupervisor))
59        return wasRemoved;
60
61      if (numberOfSupervisor() <= minimumNumberOfSupervisor())
62        return wasRemoved;
63
64      int oldIndex = supervisor.indexOf(aSupervisor);
65      supervisor.remove(oldIndex);
66      // Tracepoint [3] — association link removal
67      if (aSupervisor.indexOfStudent(this) == -1)
68      {
69        wasRemoved = true;
70      }
71      else
72      {
73        wasRemoved = aSupervisor.removeStudent(this);
74        if (!wasRemoved)
75        {
76          supervisor.add(oldIndex,aSupervisor);
77        }
78      }
```

```
79      return wasRemoved;
80    }
81
82    public boolean setSupervisor(Mentor... newSupervisor)
83    { /* omitted for simplicity */ }
84
85    public void delete()
86    {
87      ArrayList<Mentor> copyOfSupervisor =
88                      new ArrayList<Mentor>(supervisor);
89      supervisor.clear();
90      for(Mentor aSupervisor : copyOfSupervisor)
91      {
92        aSupervisor.removeStudent(this);
93      }
94     // Tracepoint [4] – association links deletion
95     // (due to Object deletion)
96    }
97  }
```

The next case we will consider is the common 'many' association, where any number of links may be present.

```
* -- * Mentor supervisor;
```

Tracing of an association end with zero-to-many (*) multiplicity will generate the following tracepoints:

**Tracepoint 1)** Add – link added

**Tracepoint 2)** Remove – a link is removed

**Tracepoint 3)** AddAt – link added at specific index

**Tracepoint 4)** addOrMoveAt - link added at specific index

**Tracepoint 5)** Delete – links are removed

Listing 4.14 shows the generated code with the tracepoint highlighted.

**Listing 4.14: Generated code for traced association end of zero-to-many (*) multiplicity**

Java

```
1  public class Student
2  {
3    //------------------------
4    // MEMBER VARIABLES
5    //------------------------
6
7    //Student Associations
```

```
 8   private List<Mentor> supervisor;
 9
10   //------------------------
11   // CONSTRUCTOR
12   //------------------------
13
14   public Student()
15   {
16     supervisor = new ArrayList<Mentor>();
17   }
18
19   //------------------------
20   // INTERFACE
21   //------------------------
22
23   public boolean addSupervisor(Mentor aSupervisor)
24   {
25     boolean wasAdded = false;
26     if (supervisor.contains(aSupervisor)) { return false; }
27     supervisor.add(aSupervisor);
28     // Tracepoint [1] — association link addition
29     if (aSupervisor.indexOfStudent(this) != -1)
30     {
31       wasAdded = true;
32     }
33     else
34     {
35       wasAdded = aSupervisor.addStudent(this);
36       if (!wasAdded)
37       {
38         supervisor.remove(aSupervisor);
39       }
40     }
41     return wasAdded;
42   }
43
44   public boolean removeSupervisor(Mentor aSupervisor)
45   {
46     boolean wasRemoved = false;
47     if (!supervisor.contains(aSupervisor))
48     {
49       return wasRemoved;
50     }
51
52     int oldIndex = supervisor.indexOf(aSupervisor);
53     supervisor.remove(oldIndex);
54     if (aSupervisor.indexOfStudent(this) == -1)
55     {
56       wasRemoved = true;
57     }
58     else
59     {
60       wasRemoved = aSupervisor.removeStudent(this);
```

```
61        if (!wasRemoved)
62        {
63          supervisor.add(oldIndex,aSupervisor);
64        }
65      }
66      // Tracepoint [2] – association link removal
67      return wasRemoved;
68    }
69
70    public boolean addSupervisorAt(Mentor aSupervisor,
71                                   int index)
72    {
73      boolean wasAdded = false;
74      if(addSupervisor(aSupervisor))
75      {
76        if(index < 0 ) { index = 0; }
77        if(index > numberOfSupervisor())
78        { index = numberOfSupervisor() - 1; }
79        supervisor.remove(aSupervisor);
80        supervisor.add(index, aSupervisor);
81        // Tracepoint [3] – association link addition
82        wasAdded = true;
83      }
84      return wasAdded;
85    }
86
87    public boolean addOrMoveSupervisorAt(Mentor aSupervisor,
88                                         int index)
89    {
90      boolean wasAdded = false;
91      if(supervisor.contains(aSupervisor))
92      {
93        if(index < 0 ) { index = 0; }
94        if(index > numberOfSupervisor())
95        { index = numberOfSupervisor() - 1; }
96        supervisor.remove(aSupervisor);
97        supervisor.add(index, aSupervisor);
98        // Tracepoint [4] – association link addition
99        wasAdded = true;
100     }
101     else
102     {
103       wasAdded = addSupervisorAt(aSupervisor, index);
104     }
105     return wasAdded;
106   }
107
108   public void delete()
109   {
110     ArrayList<Mentor> copyOfSupervisor = new Ar-
111 rayList<Mentor>(supervisor);
112     supervisor.clear();
113     for(Mentor aSupervisor : copyOfSupervisor)
```

```
114       {
115         aSupervisor.removeStudent(this);
116       }
117       // Tracepoint [5] – association link deletion
118     }
119   }
```

We have shown the key locations of tracepoint injection in the Umple generated API for associations. Table 4.3 summarizes the mapping between tracepoints injection locations and Umple generated API methods. For example, tracing an association end with multiplicity (0..n) will inject tracepoints in all shown API methods.

**Table 4.3: Mapping between tracepoints injection and Umple association API**

| | **API generated from Umple associations** | | | | | |
|---|---|---|---|---|---|---|
| *Multiplicity* | **set** | **add** | **addAt** | **addOrMoveAt** | **remove** | **delete** |
| ***1*** | x | - | - | - | - | x |
| ***0..1*** | x | - | - | - | - | x |
| ***n*** | x | - | - | - | - | x |
| ***0..n*** | x | x | x | x | x | x |
| ***n..m*** | x | x | x | x | x | x |
| ***n..**** | x | x | x | x | x | x |
| ***** | - | x | x | x | x | x |

### 4.4.2 Prefix Subdirectives

Tracing prefixes for associations tracing aims at limiting the scope of tracing to whether an association link was added or deleted from the relationship. There are three modes that defines the prefix's purpose:

i. *Link added*: using the keyword 'add' as a prefix indicates that tracing is desired when an association link is added (i.e. traces when two instances of the associated classes are connected). Consequently, tracepoints are injected only in API methods that deal with link addition/setting.

**trace add** <Umple_RoleName> ;

ii. *Link removed*: using the keyword 'remove' as a prefix results in reducing the scope of association tracing to whenever an association link is deleted. Thus, only API methods that deal with link removal/deletion will be injected with tracepoints.

**trace remove** <Umple_RoleName> ;

iii. *Link added and removed*: using both 'add' and 'remove' keyword together separated with a comma ',' will indicate the tracing is desired in both cases. However, this is the same as the default case when no prefix is specified before the traced association. Omitting out this will result in exact same trace handling.

**trace add,remove** <Umple_RoleName>;

Table 4.4 summarizes the mapping between prefix subdirectives and tracepoints injection locations in Umple generated API for various multiplicity constraints. For example, using prefix 'add' when tracing an association end with (*) multiplicity constraint will inject tracepoints in the three API methods that deals with link additions.

**Table 4.4: Association prefix subdirectives mapping to tracepoints injection locations**

| | | **API generated from Umple associations where Tracepoints are injected** | | | | | |
|---|---|---|---|---|---|---|---|
| *Multiplicity* | *Prefix* | **set** | **add** | **addAt** | **addOrMoveAt** | **remove** | **delete** |
| ***1*** | both | x | - | - | - | - | x |
| | add | x | - | - | - | - | - |
| | remove | - | - | - | - | - | x |
| ***0..1*** | both | x | - | - | - | - | x |
| | add | x | - | - | - | - | - |
| | remove | - | - | - | - | - | x |
| ***n*** | both | x | - | - | - | - | x |
| | add | x | - | - | - | - | - |
| | remove | - | - | - | - | x | x |
| ***0..n*** | both | x | x | x | x | x | x |
| | add | x | x | x | x | - | - |
| | remove | - | - | - | - | x | x |
| ***n..m*** | both | x | x | x | x | x | x |
| | add | x | x | x | x | - | - |
| | remove | - | - | - | - | x | x |
| ***n..**** | both | x | x | x | x | x | x |
| | add | x | x | x | x | - | - |
| | remove | - | - | - | - | x | x |
| ***** | both | - | x | x | x | x | x |
| | add | - | x | x | x | - | - |
| | remove | - | - | - | - | x | x |

## 4.5. Tracing of Non Modeling Constructs

In addition to trace specification for modeling constructs, modelers have the ability to trace non-API methods using MOTL in a trace directive. Such methods can be written by programmers in Umple to handle algorithmic operations, while Umple code defines the modeling constructs. As in traditional tracing, tracing of user-written methods that are embedded in Umple will inject tracepoints at method entry or/and exit.

The syntax for the MOTL method trace directive is:

**trace** <NonAPI_Method> **;**

Consistent with the previous trace specification design, MOTL provides prefixes for non-API methods tracing based on three modes:

i. *Method entry*: using the keyword 'entry', tracepoint is injected at traced method entry.

**trace entry** < NonAPI_Method >;

ii. *Method exit*: using the keyword 'exit', tracepoint is injected at every location inside the method body that enforces an exit from the method (i.e. before return statements).

**trace exit** < NonAPI_Method >;

iii. *Method entry/exit*: combining the two previous keywords as a prefix will inject tracepoints at method entry and exit(s).

**trace entry,exit** < NonAPI_Method >;

To illustrate method tracing, we consider Listing 4.15, where an entry action is executed upon the entry to state On. Entry action is a call to user-defined method named computationMethod. The trace directive in line 9 indicates that tracing is desired when the method is called and when method terminates.

**Listing 4.15:Trace directive with non-API method**

Umple

```
1  class TraceMethod
2  {
3    status {
4      On {
5        entry / { computationMethod(100) };
6      }
7      off {}
8    }
9    trace entry,exit computationMethod;
10   // user defined method
11   int computationMethod( int x ) {
12     x += 5;
13     return x;
14   }
15 }
```

The previous Umple code with trace directives will inject following tracepoints in non-API method as shown in Listing 4.16:

**Tracepoint 1)** Entry – method entry tracepoint

**Tracepoint 2)** Exit – Any possible method exit will be injected with a tracepoint, as in the case of methods with multiple returns. In other words, MOTL will parse the method body for any possible return statements.

**Listing 4.16: Generated code for traced non-API method**

Java

```
1  public int computationMethod(int x){
2    // Tracepoint [1] – method entry
3    x += 5;
4    // Tracepoint [2] – method exit
5    return x;
6  }
```

## 4.6. Postfix Subdirectives

Constraints can be imposed upon the traced Umple entity to limit the scope of tracing in a form of prefix or postfix subdirectives. Until this point, unique prefix subdirectives have been explored for each supported Umple construct in MOTL. In this section, postfix subdirectives are outlined with examples. There are various situations where such postfix subdirectives are needed. For instance, a bug might exist in the model that relates to the behavior of a state machine, and that occurs only in a certain set of circumstances. Tracing can be controlled and limited to one or more substates in order to further isolate the bug from the rest of the state machine behavior. Another situation might require switching the tracing on when certain model-level conditions becomes true, such as detection of a possible intrusion. The next subsections elaborate on the postfix subdirectives we have designed.

### 4.6.1 Conditions

Basic conditions are used to control tracing by injecting code that outputs trace message only upon condition satisfaction. Conditions can be either pre-conditions or post-conditions. Keyword 'where' is used to inject tracing code as a pre-condition, while the 'giving' keyword is used to inject tracing code as a post-condition. Note that these condition expressions work the same regardless of the entity being traced – i.e. they apply to state machines and methods. The 'giving' keyword is so-named because the tracing occurs when the operation 'gives' a certain result.

Trace only when the condition is true prior to the operation being traced:

**trace** < Umple_construct> **where** <Condition> ;

Trace only when a traced operation results in the Umple construct being changed to match the condition:

**trace** < Umple_construct> **giving** <Condition>;

Conditions consist of a left hand side, Boolean operator and a right hand side. Either side can refer to an Umple attribute, role name, state, state value or constant. In addition, conditions can be simple or compound conditions.

### 4.6.2 Occurrences

In addition to basic conditions, the functionality of tracing for a certain number of occurrences (i.e. appearances) of trace output is implemented. This is achieved by using the 'for' keyword followed by an integer to specify the number of trace output occurrences desired. The trace will be recorded for a specific number of appearances, after which tracing will cease.

**trace** <Umple_construct> **for** <Numeric>;

### 4.6.3 Life Timeline

Tracing can be limited for a period bracketed by a condition. Two keywords were designated for this purpose: The 'until' keyword triggers tracing to start and continues tracing until a given condition is satisfied, after which tracing stops permanently.

**trace** < Umple_construct> **until** <Condition> ;

Figure 4.12 shows that with until postfix subdirective, any setting of 'attr' value will be traced from its creation until the condition has been satisfied and then tracing will be halted.

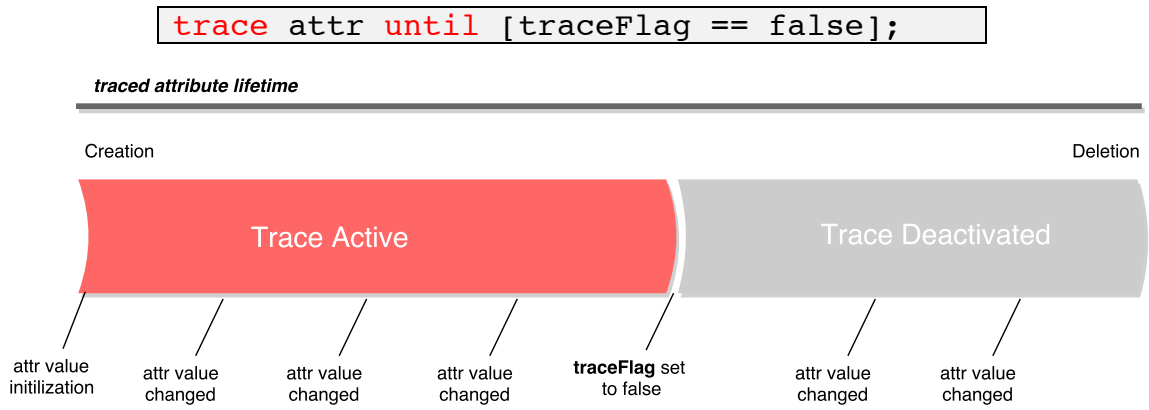


**Figure 4.12: Tracing until condition is satisfied**

The 'after' keyword provides the opposite behaviour; tracing will start once a given condition is satisfied and then continues indefinitely without any interruption.

**trace** < Umple_construct> **after** <Condition> ;

Figure 4.13 shows the effect of after postfix on attribute tracing. Tracing of attribute setting will start after 'traceFlag' equals true, and will continue the tracing onwards.

```
trace attr after [traceFlag == true];
```

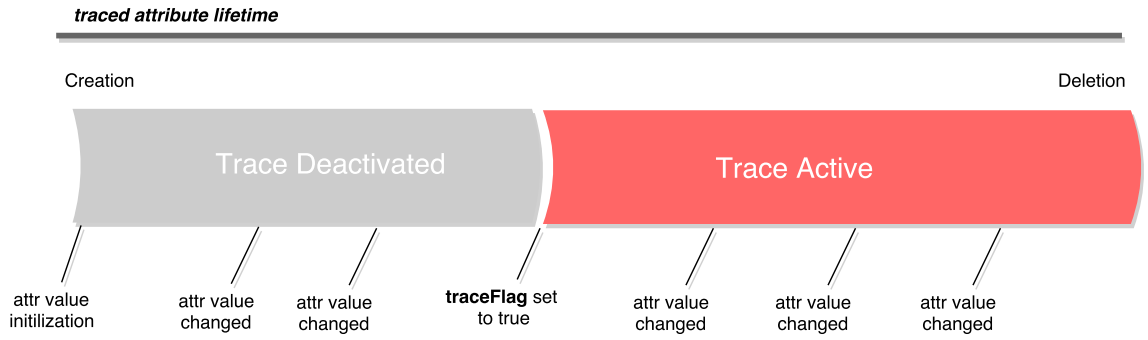


**Figure 4.13: Tracing indefinitely after condition is satisfied**

### 4.6.4 Periodical Time

Tracing will be turned on periodically at intervals determined by given time expressed in seconds. After a trace is output, no attempt to output another trace will occur until the period of time has once again passed.

**trace** <Umple_construct> **period** <Numeric> ;

### 4.6.5 Nested States Depth Level

Umple supports nested state of unlimited depth. Tracing of nested states means trace output will be recorded for each incoming and outgoing transition, in addition all substates will be traced, recursively. But, if tracing of substates is not desired and tracing scope should be limited to a certain level, then the 'level' keyword can be specified followed by an integer to represent the tracing level (i.e. recursion depth); level 0 would mean trace the current level and no substates.

**trace** < Umple_state> **level** <Numeric> ;

Listing 4.17 provides an example of nested state machine written in Umple, with the state machine visualization shown in Figure 4.14.

**Listing 4.17: Nested state example**

Umple

```
1  class Example {
2    status {
3      nestedState {
4        level1a {
5          e4 -> level1b;
6          e5 -> level1c;
7          level2a {
8            e1 -> level2b;
9          }
10         level2b {
11           e2 -> level2c;
12         }
13         level2c {
14           e3 -> outsideNested;
15         }
16       }
17       level1b {}
18       level1c {}
19     }
20     outsideNested {}
21   }
22 }
```

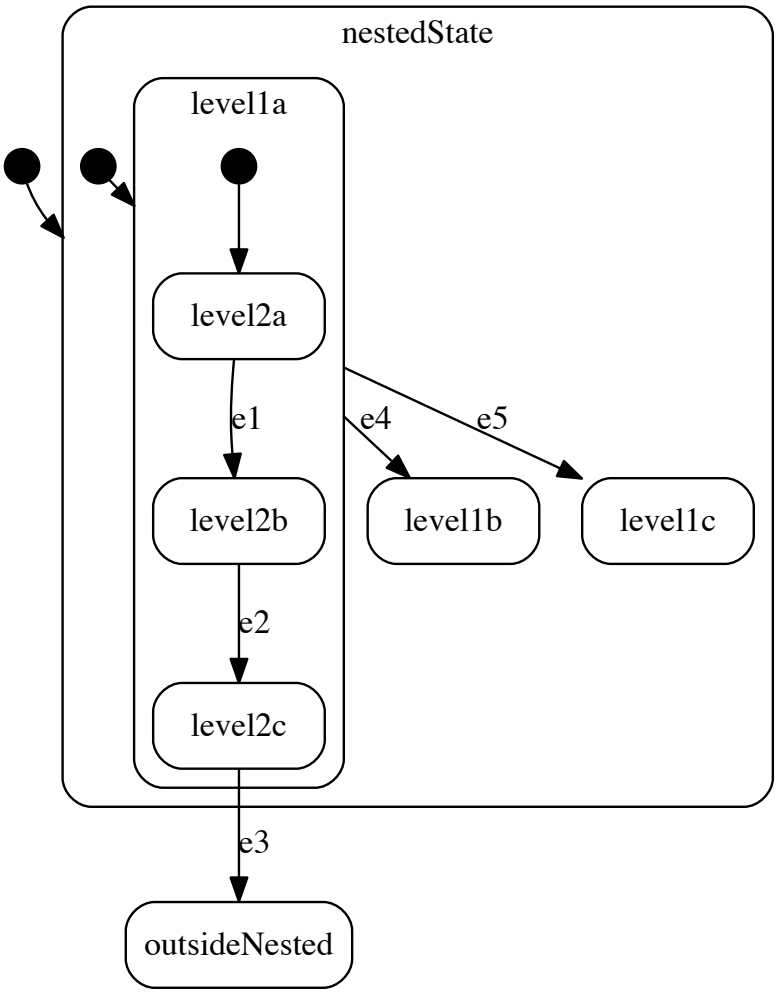


**Figure 4.14: Nested state with multi-level of depth**

The trace directive below indicates that tracing of nested state is desired with tracing depth level set to 1.

```
trace nestedState level 1;
```

Figure 4.15 visualizes the traced state machine elements with respect to this constrained trace directive. First level of depth tracing will trace all substates at that level, and they are ‘level1a’, ‘level1b’, and ‘level1c’. However, substates of ‘level1a’ will not be traced. Events that are within the traced states, or exit or enter them will be traced also, such as the event exiting the nested state ‘e3’.

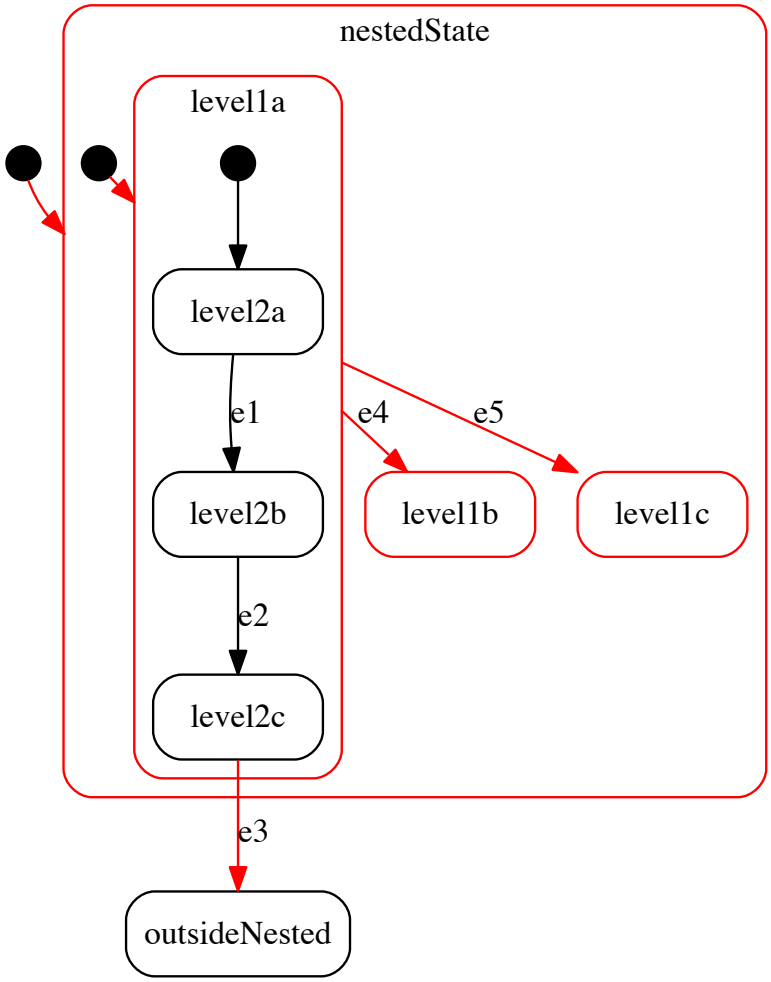


**Figure 4.15: Limiting the tracing depth of traced nested state**

### 4.6.6 Record Clause

There are situations where tracing may necessitate the monitoring of other Umple model constructs for debugging and analysis purposes such as the tracing of additional attributes and/or state machines. The record clause postfix subdirective allows the output in a trace to include other modeling constructs or arbitrary strings to improve the usefulness of the trace. Using the 'record' keyword, the postfix is specified as follows:

**trace** < Umple _construct> **record** <Record_Clause> ;

Listing 4.18 contains a light bulb state machine with two states ‘On’ and ‘Off’. Upon the entry of each state there is an entry action manipulating the Boolean attribute ‘lightOn’ value. The record postfix subdirective can be used to report Boolean attribute, when we trace states, e.g. trace directive located in line 12 trace state ‘Off’ and reports the value of Boolean attribute ‘lightOn’, which allows us to monitor the value of the attribute and make sure that entry action was executed probably.

**Listing 4.18: Trace directives with record clauses**

Umple

```
1  class LightBulb {
2    Boolean lightOn = false;
3    status {
4      On {
5        entry / { setLightOn(true); }
6        flip -> Off;
7      }
8      Off {
9        entry / { setLightOn(false); }
10       flip -> On;
11     }
12     trace Off record lightOn;
13     trace On record “Recording a message”;
14   }
15 }
```

### 4.6.7 Execute Clause

The execute clause postfix subdirective allows modelers to execute programming language native code when a tracepoint is triggered. This code will be executed after the trace output is generated. An example of use of this feature might be to count the entry or exit actions associated with a state. There is no restriction on the native code specifiable in the execution clause, since it will not be parsed and will be output into the generated system as is.

The trace directive below indicates that upon entry to a state called ‘state’, the system should execute the code that increments the attribute ‘traceCounter’. First the tracepoint is triggered that generates trace output indicating the state was entered and then the native code is executed.

```
trace entry state execute {++traceCounter;};
```

### 4.6.8 Constraint Combination

Combining conditions is possible and different scenarios might occur. If there are multiple trace directives tracing the same entity, each with its own condition, then they are treated independently. However, if multiple constraints are specified in the same trace directive for the same traced entity, precedence rules are applied to handle multiple constraints:

- The 'after' and 'until' keywords have precedence over 'where' and 'giving'.
- Ideally 'after' and 'until' conditions can be simple or compound conditions. In cases where there are multiple 'after's in a single trace directive, their condition will be ored ("| |") together. While if there are multiple 'until's in a single trace directive, their condition will be anded ("&&") together.
- If both 'after' and 'until' are specified, tracing does not start until the 'after' condition is satisfied, and then stops permanently when the 'until' condition is satisfied. The 'until' condition is ignored until the 'after' condition has become satisfied.
- If the 'for' count has not yet been reached, then tracing is controlled by the 'after', 'until', 'where' and 'giving' conditions. However, after the count has been reached, then the 'for' clause takes priority, and output ceases.
- Tracing works on a per-object basis: Essentially, there are private attributes added to each object for each traced attribute, association, etc. to help control the intervals during which the 'before', 'after' and 'for' clauses have effect.
- If there are multiple trace directives tracing the same entity, then they are treated independently, resulting in the potential for duplicate output; this is normally not desirable. However, it can be useful in cases where completely different and non-overlapping periods of tracing are to be specified.

## 4.7. Adding Tracing for All Constructs of a Given Category

So far we have discussed the tracing of individual modeling constructs and constraining them with postfix subdirectives. In this last section, we introduce the support for tracing of all modeling constructs of a certain category in a single trace directive using a reserved keyword for each modeling construct. For instance, in the case of tracing attributes, if all attributes of

an Umple class need to be traced, then keyword (AllAttributes) is used as a traced entity resulting in the tracing of all attributes where that trace directives is encapsulated.

In Listing 4.19, a trace script is shown including traces for two classes. In the first class, tracing of all attributes and state machines defined in that class is anticipated. Likewise, in the second class, entries of all non-API methods will be traced. Postfix subdirectives can be added to these trace directives, too.

**Listing 4.19: Wildcard notation in trace directives**

Umple

```
1 class GarageDoor {
2   trace AllAttributes;
3   trace AllStateMachines;
4 }
5 class GarageDoorAction {
6   trace AllNonApiMethods;
7 }
```

## 4.8. Summary

At the beginning of this chapter, we stated a number of requirements for trace specification at the model level; these drove our language design. In MOTL, trace specification was accomplished through textually trace directives that are part of the Umple language. The MOTL architecture was described by presenting its grammar and metamodel. Trace directives can be written directly in the model's textual specification, or can be written independently in a trace script.

Trace directives allow tracing of attributes, state machines, and associations. Syntax and semantics of attribute tracing using trace directives was outlined and we showed how semantics of attribute tracing differs in the presence of several stereotypes. After that, we explored the syntax and semantics of state machine tracing and presented the tracing specification of state machine components. The final modeling construct that can be traced is associations. Handling of association tracing varies based on the multiplicity at each end of an association.

To conclude this chapter, trace directives will result in the injection of appropriate tracepoints into generated code. We provided a mapping between traced models and Umple generated API. In the next chapter, we will discuss how injected tracepoints can be handled by several different MOTL-supported tracing technologies.

# Chapter 5. Tracer Technology: Syntax and Semantics of a Tracer Directive

As discussed earlier, trace specification at the model level results in the injection of tracepoints into the code generated from a model. Invocation of injected tracepoints triggers the recording of a wide range of information related to the traced circumstances in the form of trace messages. These trace messages are passed to the chosen tracing technology. Output might be on the console (standard error), in a trace log file, or via some other more sophisticated mechanism, specific to the chosen tracer technology. In this chapter, we discuss and investigate how MOTL should handle abstract tracing technology specification at the model level.

We have identified a set of requirements that needs to be addressed for MOTL tracer specification:

I. **Tracing technology tools**: MOTL should support different tracing technologies that can be used in the generated code. Modelers should have the ability to choose from a simple and straightforward tracing method (e.g. console tracing) to a more sophisticated tracing technologies (e.g. log4j) based on their tracing motives.

II. **Contents of Trace output**: MOTL should provide a wide range of trace message components that capture various aspects of triggered trace circumstances. To produce a meaningful trace output, components must report basic information such as system time, and relate trace output to the traced model entity. After defining possible trace output components, modelers should have the capability of including desired or excluding undesired trace output components.

III. **Abstract tracer customization**: In addition to selecting the tracer, modelers should have the option to state basic tracer customization at the model level. By this, we mean that special settings unique to a given tracing technology can be maintained at the model level.

IV. **Transparently integrated language**: Similar to trace directives, the tracer directive needs to be written in a format that blends well with the textual representation of the model.

Each Umple system may have a tracer directive written outside the scope of Umple classes. Modelers use a tracer directive to define and control the tracing technology being injected into model-generated source code. If a tracer directive is absent in the model, 'Console' is used as the default.

A tracer directive is specified using the 'tracer' keyword followed by the tracer technology name. Modelers have the ability to adjust the tracer further by providing additional tracer options. These options are specific to each tracer and may contain settings only a regular user of that tracing tool would understand. The idea is to allow modelers to maintain these settings at the model level and not modify the configuration files every time the system is regenerated. However, in cases where no options are specified, default settings are used.

The generalized syntax is:

**tracer** <Tracer_Technology> <Tracer_Options> **;**

In this chapter, first we explore MOTL tracer directive syntax and semantics by defining the grammar and metamodel. Then, we outline the tracer technologies supported by MOTL. After that, we define the components of the trace message that are passed to the tracing technology when a tracepoint is invoked. Finally, we show generated files resulting from MOTL tracer directives.

## 5.1. Architecture

We extended the Umple architecture to allow the textual specification of tracing technology at the model level, by extending the following Umple architecture components:

- **Parser**: Grammar was created to specify the syntactic structure of tracing technology specification using a tracer directive.
- **Analyzer**: Tracer directive metamodel was designed to handle tracer technology description.

- **Code generator**: We extended Umple JET templates to generate tracepoints in instrumented model-generated code with respect to selected tracer technology. In addition, any other related files will be automatically generated (e.g. XML configuration file).

### 5.1.1 Grammar

Table 5.1 contains grammar rules created to specify tracer technology at the model level in MOTL. There are four rules with their names highlighted in blue. Keywords are highlighted in red, and terminals in green. R1 defines the overall structure of a MOTL tracer directive, starting with the 'tracer' keyword and ending with a semicolon ';'.

The tracerType is a string that holds the selected tracer technology name in MOTL, which can be any of supported tracing technologies; details on how to select tracers are explained in the next section.

The tracerOptions is a subrule that handles the options passed to the tracer as defined in R2. It can have zero-or-more log configuration (as indicated by the *), an optional trace message switch controller, or an optional tracerArgument string passed unchanged to the tracer technology.

R3 is a second-level rule to manage the adjustment of logging tracers (i.e. log4j and Java logging API) by explicitly stating specific settings. Likewise, R4 is a second-level rule that controls trace output component inclusion or exclusion in trace messages. Explanation of usage is provided in the next section.

Details of our tracer directive grammar rules are below.

**Table 5.1: Tracer directive grammar rules**

| Rule | Umple Grammar |
|---|---|
| *R1* | tracerDirective : tracer [tracerType] [[tracerOptions]] ; |
| *R2* | tracerOptions- : [[logConfig]]* [[traceMessageHeader]]? [tracerArgument]? |
| *R3* | logConfig : (root = [rootLevel]) \| (monitorInterval = [monitorInterval])<br>\| [logLevel] ( , [logLevel] )* = [logAppender] ( , [logAppender] )* |
| *R4* | traceMessageHeader : [=switch:on\|off] : [option] ( , [option] )* |

Visualization of MOTL tracer directive grammar is provided as a syntax diagram in Figure 5.1, where dark blue rounded rectangles are keywords and terminal symbols and light blue rectangles are terminals. The following syntax diagram was generated based on the grammar using Railroad Diagram Generator [59].

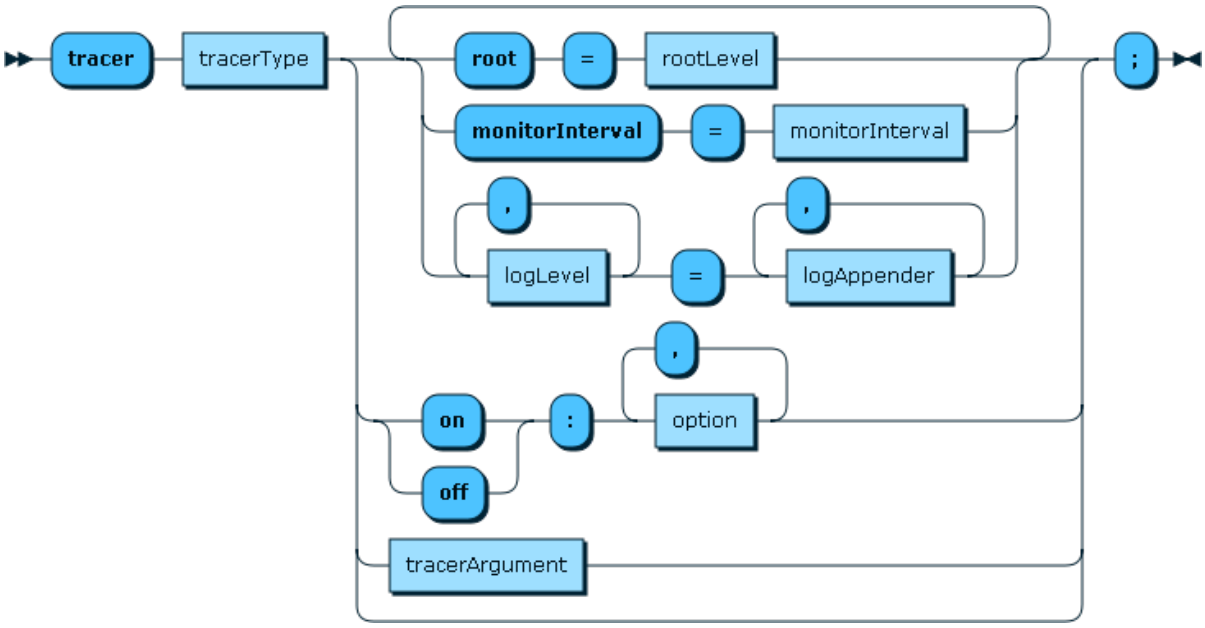


**Figure 5.1: Tracer directive grammar railroad diagram**

### 5.1.2 Metamodel

Figure 5.2 presents the tracer directive metamodel; the parser instantiates these classes:

- *TracerDirective* class: In MOTL metamodel, this class holds the responsibility of handling the tracing technology specification at the model level. An Umple model may have a tracer uniquely identified by the name of the technology being used (e.g. 'File'). Further customization can be added to the tracer specification by passing arguments to the tracer technology, controlling trace output, and/or adding more detailed configuration settings in case of logging tracer technologies was selected.
- *TracerArgument* class: Handles arguments passed to the selected tracer technology. It has an argument attribute that stores any parameter passed to the tracer (e.g. the trace file name to which to send output in the case of a 'File' tracer).
- *TraceMessageSwitch* class: Handles the inclusion or exclusion of trace output components passed to the tracer technology. It has two Boolean attributes acting as flags

for the selection of either inclusion or exclusion criteria. MessageComponent class stores the selected trace output components selected for the switch. If Boolean attribute ‘on’ was set to true, then only components held by instances of class MessageComponent will be included as components in the trace output. In the other situation when Boolean attribute ‘off’ is true, components held by instances of class MessageComponent will be removed from the trace output components.

- *LogConfiguration* class: the chosen tracer technology can be further tuned in case a logging tool has been selected (e.g. log4j). In particular the root logger can be defined, and a monitor interval value can be indicated to monitor any logging configuration changes. In case of the log4j tracer, by default, an xml configuration file is output as part of the generated files by setting Boolean attribute ‘generateConfig’ to true. However, if it is false, no configuration file will be generated; we encountered a situation where a user preferred to keep and use his customized configuration file. The LoggerLevelToAppender class manages the assignment of appenders (e.g. ‘Console’) to different logging levels.

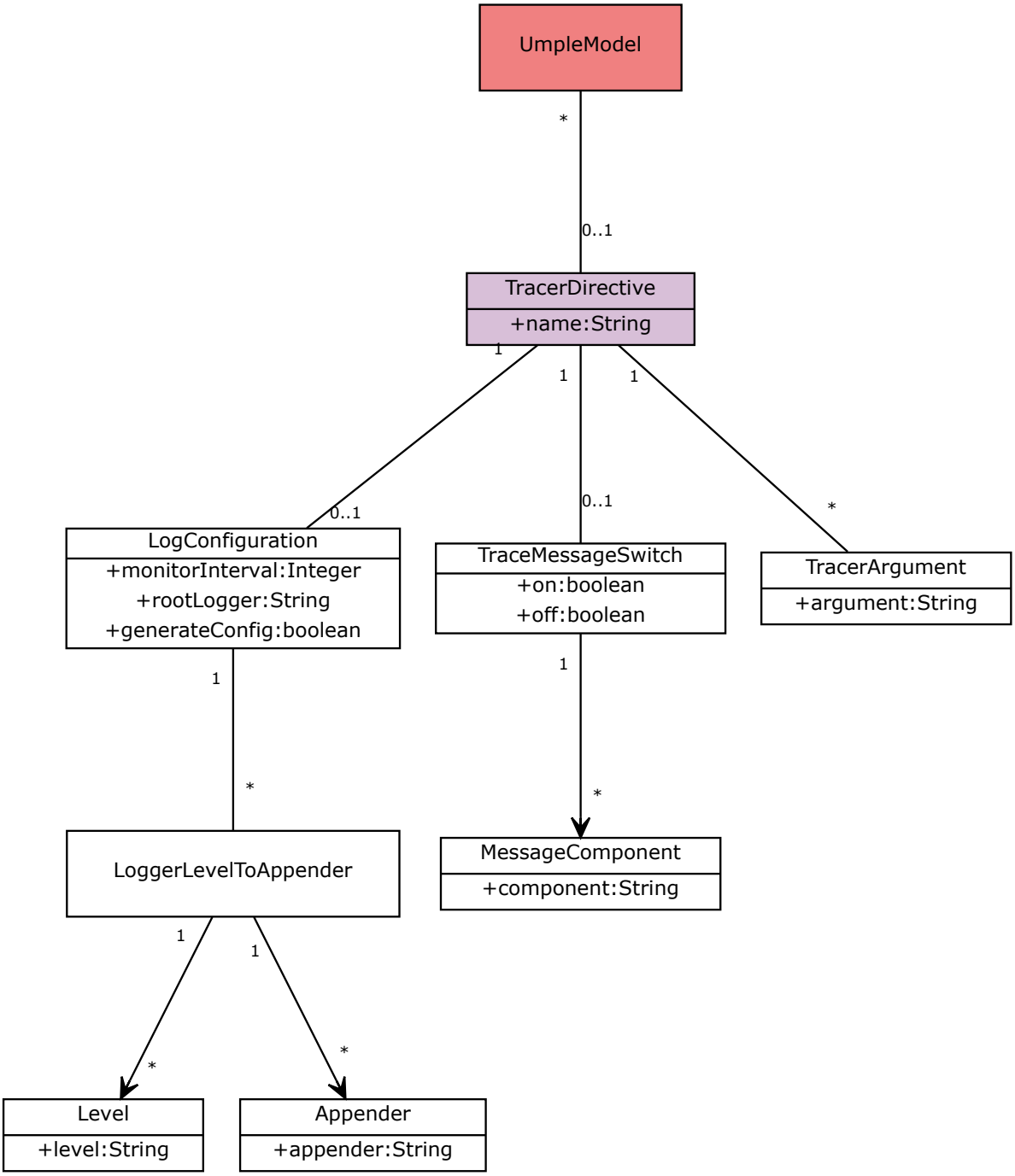


**Figure 5.2: Tracer directive metamodel**

The tracer directive metamodel was textually modeled using Umple as shown in Listing 5.1.

**Listing 5.1: Tracer directive metamodel written in Umple**

Umple

```
class UmpleModel {
  * -- 0..1 TracerDirective;
}

class TracerDirective {
  String name;
  1 -- * TracerArgument;
  1 -- 0..1 TraceMessageSwitch;
  1 -- 0..1 LogConfiguration;
}

class TracerArgument {
  String argument;
}

class TraceMessageSwitch {
  Boolean on = false;
  Boolean off = false;
  1 -> * MessageComponent;
}

class MessageComponent {
  String component;
}

class LogConfiguration {
  Integer monitorInterval = 0;
  String rootLogger = "info";
  Boolean generateConfig = true;
  1 -- * LoggerLevelToAppender;
}

class LoggerLevelToAppender {
  1 -> * Appender;
  1 -> * Level;
}

class Level {
  String level;
}
class Appender  {
  String appender;
}
```

## 5.2. Supported Tracers

MOTL supports a set of tracing technologies previously discussed in Chapter 3. They can be classified into two main categories: built-in tracers and third-party tracers. The difference between the two categories is that built-in tracers do not require any additional libraries to be imported when executing code generated from traced models, whereas the latter do. In MOTL, the built-in tracing technologies are Console, File and Java Native Logging API. The third-party tracers available include Log4j and LTTng.

In this section, we explore injected tracepoints with respect to different supported tracer technologies, and what files are generated from the traced model for each selected tracer. For illustration purposes, we consider tracing the attribute id as in Listing 5.2 and show the tracepoint injected for each supported tracer.

**Listing 5.2: Illustrative example to show generated tracepoints**

Umple

```
1 class Student {
2   Integer id;
3   trace id;
4 }
```

### 5.2.1 Console

Console tracing is the basic and 'primitive' option, but this does not necessarily mean that it's of no importance. When print-statement code is generated, as in MOTL, it can be the easiest approach to use. It can be used for educational purposes or in situations where no deep tracing is needed, and issues such as performance are not dominant. Using the MOTL console tracer will write generated trace messages to the system error stream, so they do not get mingled with standard output, and, if desired, the error stream can be intercepted by external tools for further processing.

Console tracing is the default in MOTL and hence is used in the case where a tracer directive was absent in the textual model specification, or it can be chosen as indicated below.

```
tracer console;
```

When a model has a console tracer directive, MOTL generates files with an additional (ConsoleTracer.java) file placed in a subdirectory (cruise/util). Console tracer, shown in Listing 5.3, is a singleton class to ensure that only one object is created at runtime, and it has a static method that handles any received tracing message by passing it to standard error. The constructor is called upon class instantiation to print the trace message component header.

**Listing 5.3: Console tracer class**

Java

```
1  package cruise.util;
2
3  public class ConsoleTracer
4  {
5     //-------------------------
6     // STATIC VARIABLES
7     //-------------------------
8
9     private static ConsoleTracer theInstance = null;
10
11    //-------------------------
12    // CONSTRUCTOR
13    //-------------------------
14
15    private ConsoleTracer()
16    {
17      handle("Time,Thread,UmpleFile,LineNumber,Class,"
18              + "Object,Operation,Name,Value");
19    }
20
21    public static ConsoleTracer getInstance()
22    {
23      if(theInstance == null)
24      {
25        theInstance = new ConsoleTracer();
26      }
27      return theInstance;
28    }
29
30    //-------------------------
31    // DEVELOPER CODE - PROVIDED AS-IS
32    //-------------------------
33
34    public static void handle(String message)
35    {
36       System.err.println(message);
37    }
38    static{getInstance();}
39 }
```

In Listing 5.4, injecting tracepoints is accomplished by a call to the 'handle' static method inside console tracer class, as seen in line 26. Trace message components are passed as a string where the actual handling of the trace message is delegated to the console tracer static method. The console tracer class is imported in every class that contains a tracepoint injection.

**Listing 5.4: Injected tracepoints with console tracer**

Java

```
1  import cruise.util.ConsoleTracer;
2  public class Student
3  {
4     //------------------------
5     // MEMBER VARIABLES
6     //------------------------
7
8     //Student Attributes
9     private int id;
10
11    //------------------------
12    // CONSTRUCTOR
13    //------------------------
14    public Student(int aId)
15    {
16      id = aId;
17    }
18
19    //------------------------
20    // INTERFACE
21    //------------------------
22
23    public boolean setId(int aId)
24    {
25      boolean wasSet = false;
26      ConsoleTracer.handle("Trace Message Components");
27      id = aId;
28      wasSet = true;
29      return wasSet;
30    }
31
32    public int getId()
33    {
34      return id;
35    }
36 }
```

### 5.2.2 File

Similar to console tracing, tracing to a file is a simple built-in tracing method that stores trace information into a text file. Such log files are stored for later analysis. This approach can result in heavy overhead due to file handling and the potential large size of final file produced. Console tracing would produce the same amount of output, but the user can instantly filter and ignore what they do not want; with file output this has generally to wait until later. However, file tracing avoids cluttering the console.

The tracer directive below states that 'file' is selected to be the tracer that will be used in the generated code. Accordingly, tracepoints will direct all trace output to a trace file.

```
tracer file;
```

The desired name of the trace file can be passed as an argument to the tracer, as shown in the tracer directive below.

```
tracer file traceLog.txt;
```

Similar to console tracer, when a model has a file tracer directive, MOTL generates the model source files with an additional (FileTracer.java) file placed in a subdirectory (cruise/util). The file tracer class is a singleton class, as shown below, that writes trace messages to a trace log file. Modelers can name the trace log file in the tracer directive; otherwise the file name will default to 'trace.txt'.

**Listing 5.5: File tracer class**

Java

```
1  package cruise.util;
2  import java.io.*;
3  public class FileTracer
4  {
5       //------------------------
6       // STATIC VARIABLES
7       //------------------------
8
9       public static final String filename = "trace.txt";
10      private static FileTracer theInstance = null;
11      private static FileOutputStream fout = null;
```

```
12      private static PrintStream stream = null;
13      private static boolean open = false;
14
15      //------------------------
16      // CONSTRUCTOR
17      //------------------------
18
19      private FileTracer()
20      {
21          handle("Time,Thread,UmpleFile,LineNumber,Class,"
22                  + "Object,Operation,Name,Value");
23      }
24
25      public static FileTracer getInstance()
26      {
27          if(theInstance == null)
28          {
29              theInstance = new FileTracer();
30          }
31          return theInstance;
32      }
33
34      //------------------------
35      // INTERFACE
36      //------------------------
37
38      public void delete()
39      {}
40      public String toString()
41      {
42          String outputString = "";
43          return super.toString() + "["+ "]"
44          + outputString;
45      }
46      //------------------------
47      // DEVELOPER CODE - PROVIDED AS-IS
48      //------------------------
49
50      public static void handle(String message){
51          if(!open){
52              try
53              {
54                  // Open an output stream
55                  fout = new FileOutputStream (filename,true);
56                  stream = new PrintStream(fout);
57              }
58              catch (IOException e)
59              {
60                  System.err.println ("Unable to write");
61                  System.exit(-1);
62              }
63              open = true;
64          }
```

```
65            // Write traced Item information
66            stream.println(message);
67        }
68        static{getInstance();}
69    }
```

In similar manner to ConsoleTracer, the static method 'handle' in the FileTracer class passes the trace message components as a string where the actual file operations (i.e. file opening and file writing) are delegated. In Listing 5.6, the first line shows the import statement and line 27 shows a tracepoint injection.

**Listing 5.6: Injected tracepoints with file tracer**

Java

```
1  import cruise.util.FileTracer;
2  public class Student
3  {
4     //------------------------
5     // MEMBER VARIABLES
6     //------------------------
7
8     //Student Attributes
9     private int id;
10
11    //------------------------
12    // CONSTRUCTOR
13    //------------------------
14
15    public Student(int aId)
16    {
17      id = aId;
18    }
19
20    //------------------------
21    // INTERFACE
22    //------------------------
23
24    public boolean setId(int aId)
25    {
26      boolean wasSet = false;
27      FileTracer.handle("Trace Message Components");
28      id = aId;
29      wasSet = true;
30      return wasSet;
31    }
32 }
```

### 5.2.3 Java native logging API

Previous sections explored primitive tracers supported by MOTL, but for modelers aiming for a more mature widely used tracing technology, Java native logging API can be selected. In MOTL, the 'JavaAPI' option, case insensitive, is used in the tracer directive to cause the Java native logging framework to be used in instrumented model-generated code. Clearly this kind of tracing subsumes the capabilities of the Console and File tracer. However, modelers may use the latter two when they favour simplicity.

```
tracer javaAPI;
```

As shown in Listing 5.7, selecting the Java built-in logging API as the tracer will require the following code manipulation:

- Import of Java logging API (line 1), which resides in the utility package as part of the Java Development Kit (JDK); thus no import is required for external JARs (i.e. external dependencies).
- Declaration of a static logger instance (line 8) for any class containing a tracepoint injection. All objects of the same class will use the same logger instance.
- Passing trace messages at the desired log level through the logger instance (line 34).

**Listing 5.7: Injected tracepoints with Java logging API tracer**

Java

```
1  import java.util.logging.*;
2  public class Student
3  {
4     //------------------------
5     // STATIC VARIABLES
6     //------------------------
7
8     public static final Logger logger =
9                     Logger.getLogger(Student.class.getName());
10
11    //------------------------
12    // MEMBER VARIABLES
13    //------------------------
14
15    //Student Attributes
16    private int id;
17
18    //------------------------
```

```
19      // CONSTRUCTOR
20      //-------------------------
21
22      public Student(int aId)
23      {
24        id = aId;
25      }
26
27      //-------------------------
28      // INTERFACE
29      //-------------------------
30
31      public boolean setId(int aId)
32      {
33        boolean wasSet = false;
34        logger.info("Trace Message Components");
35        id = aId;
36        wasSet = true;
37        return wasSet;
38      }
39  }
```

The tracepoint will pass the trace message to the predefined log level in the Java logging API named 'info'. Recall in Chapter 3 there were different predefined logging levels based on the message severity. Choosing the log level can be done in the trace specification using a postfix statement in a trace directive. As shown below, keyword 'logLevel' is used to select the desired logging level. However, if it is not specified in the trace directive and log4j was selected as the tracer, then 'info' is defaulted as the log level.

```
trace id logLevel debug;
```

### 5.2.4 log4j

In addition to the Java logging API, MOTL supports the well-known log4j2 logging framework. It can be selected in a tracer directive using 'log4j' as an option.

```
tracer log4j;
```

The listing below shows a model-generated system instrumented with log4j tracepoints. Selecting log4j as the tracer resulted in:

- Importing statements of required log4j packages residing in two log4j jar files (line 1 and 2).
- Declaration of a static logger attribute in every class that contains a tracepoint injection (line 14).
- Passing of a trace message to the selected log level through the logger instance (line 41). If no logging level was specified in the trace directive, then the default is 'info' level.

**Listing 5.8: Injected tracepoints with log4j tracer**

Java

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
3
4  public class Student
5  {
6     //-------------------------
7     // STATIC VARIABLES
8     //-------------------------
9     /**
10     * log4j version 2
11     * requires jars (log4j-api-2.0.1.jar)
12     * and (log4j-core-2.0.1.jar)
13     */
14    public static final Logger logger =
15                          LogManager.getLogger(Student.class);
16
17    //-------------------------
18    // MEMBER VARIABLES
19    //-------------------------
20
21    //Student Attributes
22    private int id;
23
24    //-------------------------
25    // CONSTRUCTOR
26    //-------------------------
27
28    public Student(int aId, String aName)
29    {
30      id = aId;
31      name = aName;
32    }
33
34    //-------------------------
35    // INTERFACE
36    //-------------------------
```

```
37
38      public boolean setId(int aId)
39      {
40        boolean wasSet = false;
41        logger.info("Trace Message Components");
42        id = aId;
43        wasSet = true;
44        return wasSet;
45      }
46  }
```

In addition to source code files generated from the model, a default log4j XML configuration file will be generated as shown in Listing 5.9.

**Listing 5.9: Log4j generated default configuration xml file**

XML

```
<?xml version="1.0" encoding="UTF-8"?>
 <Configuration>
   <Appenders>
     <Console name="console" target="SYSTEM_OUT">
       <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
     </Console>
     <File name="file" fileName="logs/trace.log" immedi-
ateFlush="false" append="false">
       <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
     </File>
   </Appenders>
   <Loggers>
     <Root level="info">
       <AppenderRef ref="console"/>
     </Root>
   </Loggers>
 </Configuration>
```

This configuration file can be customized at the model level by the tracer options in a tracer directive. The tracer directive below has options resulting in the generation of a customized log4j xml configuration file (log4j2.xml) with root logger level set to the debug level, info logging level directing its output to console, and error logging level directing its output to file.

```
tracer log4j root=debug info=console error=file;
```

However if generation of the configuration file is not desired, then we indicate that by passing the option 'noconfig'. Below is an example of a log4j tracer directive with an argument indicating that generation of log4j configuration file is undesired (i.e. the user prefers to write his own).

```
tracer log4j noconfig;
```

As in the previously discussed Java logging API, log4j has its own predefined logging levels, and the desired log level can be selected through the trace directive. The trace directive below indicates that when a tracepoint is triggered (i.e. condition satisfied), the trace message will be passed to log level 'severe' in log4j.

```
trace id where [id<1] logLevel severe;
```

### 5.2.5 LTTng

MOTL supports tracepoints injection with respect to LTTng's sophisticated tracing technology. LTTng is a continually improving tool capable of tracing C/C++ programs and performing kernel and userspace tracing in Linux systems. In 2015 the support for Java programs tracing reached a stable stage. We decided to include it in the list of MOTL supported tracers. Choosing LTTng as the desired tracer in MOTL by using the 'lttng' option in a tracer directive will generate tracepoints that collect traces in compliance with the Common Trace Format (CTF). LTTng tracing is supported by MOTL for the tracing of Java programs though the usage of its Jar file.

```
tracer lttng;
```

In Listing 5.10, selecting LTTng as the tracer will result in:

- Import of LTTng-UST java agent (line 2).
- Instantiation of the LTTng tracer agent in the constructor (line 21).
- LTTng-UST java agent handles any trace message passed by the logger object to produce LTTng events (line 31).

**Listing 5.10: Injected tracepoints with LTTng tracer**

Java

```
1  import java.util.logging.Logger;
2  import org.lttng.ust.agent.LTTngAgent;
3  public class Student
4  {
5     //------------------------
6     // MEMBER VARIABLES
7     //------------------------
8
9     //Student Attributes
10    private int id;
11    public static final Logger logger =
12              Logger.getLogger(Student.class.getName());
13
14    //------------------------
15    // CONSTRUCTOR
16    //------------------------
17
18    public Student(int aId)
19    {
20      id = aId;
21      LTTngAgent lttngAgent = LTTngAgent.getLTTngAgent();
22    }
23
24    //------------------------
25    // INTERFACE
26    //------------------------
27
28    public boolean setId(int aId)
29    {
30      boolean wasSet = false;
31      logger.info("Trace Message Components");
32      id = aId;
33      wasSet = true;
34      return wasSet;
35    }
36 }
```

## 5.3. Trace Output

Tracepoint invocation, during run time, results in the creation of a trace message string. Each trace message consists of the components shown in Table 5.2. These will be emitted via the designated tracer, when a tracepoint is triggered in the instrumented model-generated code. Each component is essential to monitor the traced system; for instance, a state machine with do-activities or a 'queued' state machine [60] will have multiple threads, and thus information about the thread ID is needed to properly understand the trace.

**Table 5.2: Trace output components**

| Output component | Value Type | Description |
|---|---|---|
| Timestamp | Dynamic | Current system time in milliseconds when a trace is triggered. |
| Thread | Dynamic | The unique thread identification number; needed since most Umple systems will result in traces being generated from many threads. |
| Name of Umple File | Static | Name of the file that contains the UML textual model/code containing the trace directive that has resulted in this output. |
| Line number | Static | Trace directive line number in the Umple file (as above). This relates the trace output to the exact trace directive that requested it. |
| Class name | Static | Class name of the object in which the trace was triggered. This may be a subclass of the class that actually contains the directive. |
| Object hash code | Dynamic | Reports the hash code for traced object so each of many traced objects can be tracked |
| Operation | Hard Coded | One of a set of predefined codes describing the type of trace, such as a state transition or a link being added to an association. See Table 5.3. |
| Name | Static | Name of the modeling construct being traced (attribute name, state name, association end label, etc.) |
| Value | Dynamic | A value dependent on the traced model entity |

The type of information emitted at run time consists of static, dynamic, and hard coded values. Dynamic values differ each time a tracepoint is triggered, and have to be determined in real-time. Static values can be determined by the code generator, but may change if the code is regenerated due to changes in the model. Hard coded values are fixed. Currently the only hard coded value is 'Operation', that indicates the context that triggered this trace

(e.g. at_s means the trace has been triggered by the setting of an attribute). Table 5.3 gives the current set of Operation codes.

**Table 5.3: Operation component code variations**

| Model Element | Operation | Description |
|---|---|---|
| *Attributes* | at_c | Attribute has been initialized in constructor |
| | at_s | Attribute value has been changed |
| | at_g | Attribute value has been accessed |
| | at_t | Attribute value has been reset to default value |
| | at_a | A value has been added to a multivalued attribute |
| | at_r | A value has been removed from a multivalued attribute |
| *State Machine* | sm_t | An event has been triggered in a state machine resulting in a transition |
| | sm_e | A state has been entered |
| | sm_x | A state has been exited |
| | sm_do | A do-activity in a state has started |
| | sm_di | A do-activity in a state has been interrupted |
| *Associations* | as_a | An association link has been added |
| | as_r | An association link has been removed |
| | as_d | All association links are removed due to object deletion. |
| | as_c | Association link has been created in constructor |

Every time trace output is generated it will include all previously mentioned components in a form of a trace message. However, a modeler has the capability of excluding any unwanted trace output components using the switch option in the tracer directive. A switch option starts with either 'on' or 'off' keyword followed the trace output components. In case of 'on' switch, the provided options will only be included as components of the trace message. But, in case of 'off' switch, the provided options will be excluded from the list of output components in a trace message. Listing 5.11 shows a tracer directive indicating console as the desired tracer, with the inclusion of only two output components (Time and Operation) to form the trace message in the injected tracepoints.

**Listing 5.11: Tracer directive with trace message switch option**

Umple

```
1 tracer console on : Time,Operation;
2
3 class Student {
4   Integer id;
5   trace id;
6 }
```

## 5.4. Execution trace

When instrumented model-generated code is executed, execution traces are produced when tracepoints are triggered. Accordingly, execution traces consist of a set of entries with each entry representing the triggering of a single tracepoint in executed code. Figure 5.3 illustrates our Umple model instrumentation architecture as described so far, where the inputs are the model and trace directives, and the output is an execution trace resulting from executing instrumented generated code from models.

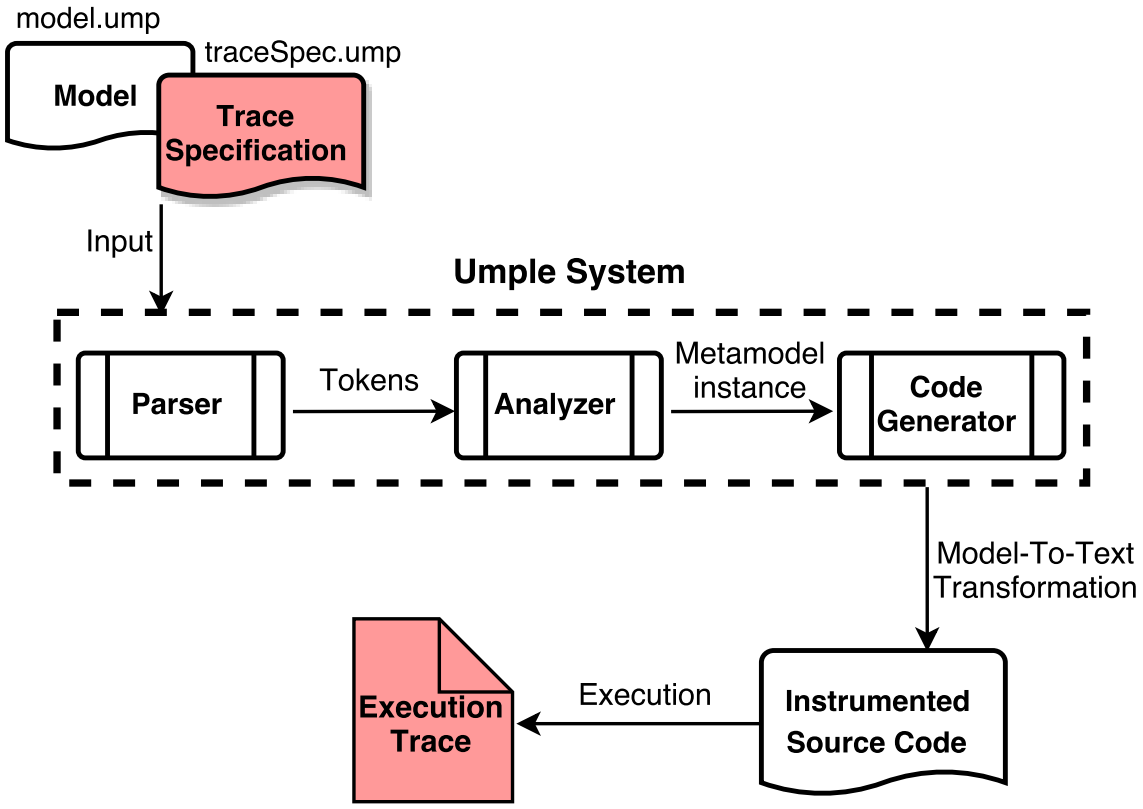


**Figure 5.3: Producing execution traces process with MOTL**

### 5.4.1 Executing Models using UmpleRun

UmpleRun [20, 61] is a command line tool used to drive the execution of Umple model through a set of execution scenarios. Currently, it is in an experimental stage and not released to the public. The tool will be used to execute Umple models and generate execution traces.

UmpleRun runs a set of execution scenarios against a targeted model. A template scenario is shown in Listing 5.12. Line 1 of any scenario (is the title line) has the keyword

'command'. Lines 2 and onwards are of a set of commands to be executed to drive the scenario. The commands can be object constructor invocations, or method calls such as state machine event calls.

**Listing 5.12: Execution scenario template**

```
command
command_1
command_2
...
command_n
```

To execute the execution scenario against the Umple model using the UmpleRun jar is done in the command line as follows:

```
> java -jar umplerun.jar model.ump ExeScenario.cmd
```

### 5.4.2 Car transmission Model

The car transmission model was inspired by a similar model in Lethbridge and Laganière's book [62]. The model consists of one class with car transmission behavior captured by the state machine shown in Figure 5.4. The state machine consists of three states: two simple states ('neutral' & 'reverse') and one composite state ('drive'). The initial state for state machine is 'neutral' state. The composite 'drive' state has three substates for transmission levels (i.e. 'first', 'second', and 'third'). There are events to trigger transitions between states and some are guarded as in [driveSelected], where event reachSecondSpeed will not cause a transition unless the Boolean guard is evaluated to true.

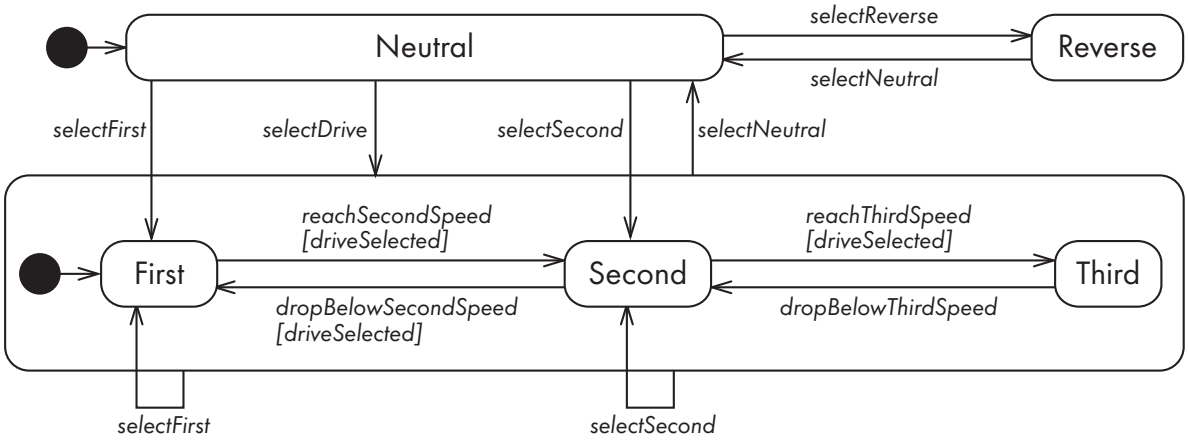


**Figure 5.4: Car transmission state machine** [62]

The Car transmission system behavior was textually modeled using Umple as shown in Listing 5.13. Lines 3 to 29 represent the car transmission status state machine. Line 4 shows the state name, while line 14 declares that there is an exit action associated with this state. Line 19 is an example of a guarded event. In line 30 we are doing a code injection to event selectDrive to set the Boolean attribute 'driveSelected' to true to differentiate it from the manual triggering caused by event 'selectFirst' which indicates the manual driving mode.

**Listing 5.13: Car transmission Umple code**

Umple

```
1  class CarTransmission {
2    Boolean driveSelected = false;
3    status {
4      neutral {
5        selectReverse -> reverse;
6        selectDrive -> drive;
7        selectFirst -> first;
8        selectSecond -> second;
9      }
10     reverse {
11       selectNeutral -> neutral;
12     }
13     drive {
14       exit / { driveSelected = false;}
15       selectNeutral -> neutral;
16       selectFirst -> first;
17       selectSecond -> second;
18       first {
19       reachSecondSpeed [driveSelected] -> second;
20       }
21       second {
22       reachThirdSpeed [driveSelected] -> third;
23       dropBelowSecondSpeed [driveSelected] -> first;
24       }
25       third {
26         dropBelowThirdSpeed -> second;
27       }
28     }
29   }
30   before selectDrive {
31     driveSelected = true;
32   }
33 }
```

We created an execution scenario to execute the Car transmission model behavior as seen in Listing 5.14. Each line represents a command executed against the model. The first line shows the title and each subsequent line from 2 to 8 begins with a command to be executed. The command in Line 2 creates a new Car transmission object. Line 3 executes event 'selectReverse', etc.

**Listing 5.14: Execution scenario for Car transmission model**

Execution scenario

```
1 command
2 new CarTransmission
3 selectReverse
4 selectNeutral
5 selectDrive
6 reachSecondSpeed
7 reachThirdSpeed
8 selectNeutral
```

Then, we wrote a trace directive to examine the value of the Boolean attribute 'driveSelected' as we transit to state 'drive' as presented in Listing 5.15. The trace directive will trace composite state 'drive' and record the value of Boolean attribute 'driveSelected' at the same time.

**Listing 5.15: Trace directive for Car transmission model**

Umple

```
1 class CarTransmission {
2   trace drive record driveSelected;
3 }
```

Adding the above trace specification to the model, and then executing the model using UmpleRun, we obtained the execution trace in comma-separated-value (CSV) form as shown in Listing 5.16 (we have replaced the system time and object hash code with * to save space). The operation code 'sm_t' in line 2 shows that this trace was recorded when a state event was triggered named 'selectDrive' that made a transition from state 'neutral' to state 'drive', and reported the value of the Boolean attribute was true. This indicates the appropriate setting of Boolean attribute from false to true, when we transited into composite state 'drive'. The next two lines in the execution trace (line 3 and 4) are tracing transition occurr inside the composite state 'drive' caused by events 'reachSecondSpeed' and 'reachThird-

Speed' respectively. The last line caused a transition exiting the composite state 'drive' to simple state 'neutral', with the value of 'driveSelected' changing to false.

**Listing 5.16: Execution trace for Car transmission model**

Execution trace

```
1 Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
2 *,1,CarTrans.ump,6, CarTransmission,*,sm_t,neutral,selectDrive,drive,true
3 *,1,CarTrans.ump,6, CarTransmission,*,sm_t,first,reachSecondSpeed,second,true
4 *,1,CarTrans.ump,6,CarTransmission,*,sm_t,second,reachThirdSpeed,third,true
5 *,1,CarTrans.ump,6,CarTransmission,*,sm_t,drive,selectNeutral,neutral,false
```

To explore the tracing further, we replaced the previous trace directive with Listing 5.17. In this trace directive, we trace the composite state 'drive' but without tracing nested states, and record the provided string.

**Listing 5.17: Another trace directive for Car transmission model**

Umple

```
1 class CarTransmission {
2   trace drive level 0 record "Tracing is not nested";
3 }
```

A rerun of the previous execution scenario with the new trace directive will generate the following execution trace. Tracing will be injected at the top level of the composite state 'drive' and nested states will not be traced, due to the 'level' postfix constraint in the trace directive.

**Listing 5.18: Second execution trace for Car transmission model**

Execution trace

```
1 Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
2 *,1,CarTrans.ump,6,CarTransmission,*,sm_t,neutral,selectDrive,drive,Tracing is not nested
3 *,1,CarTrans.ump,6,CarTransmission,*,sm_t,drive,selectNeutral,neutral,Tracing is not nested
```

### 5.4.3 Airline Model

An airline system has been modeled consisting of 7 classes and associations to capture the relationship between airline system classes, as shown in Figure 5.5. The Airline system was inspired by a previous modeled Airline system in [62]. Each airplane can be assigned to multiple flights uniquely characterized by a flight number and date. The Player-Role pattern is used to allow a person to be both a passenger and an employee. A passenger can make multiple bookings; with each booking having its own price and seat number.

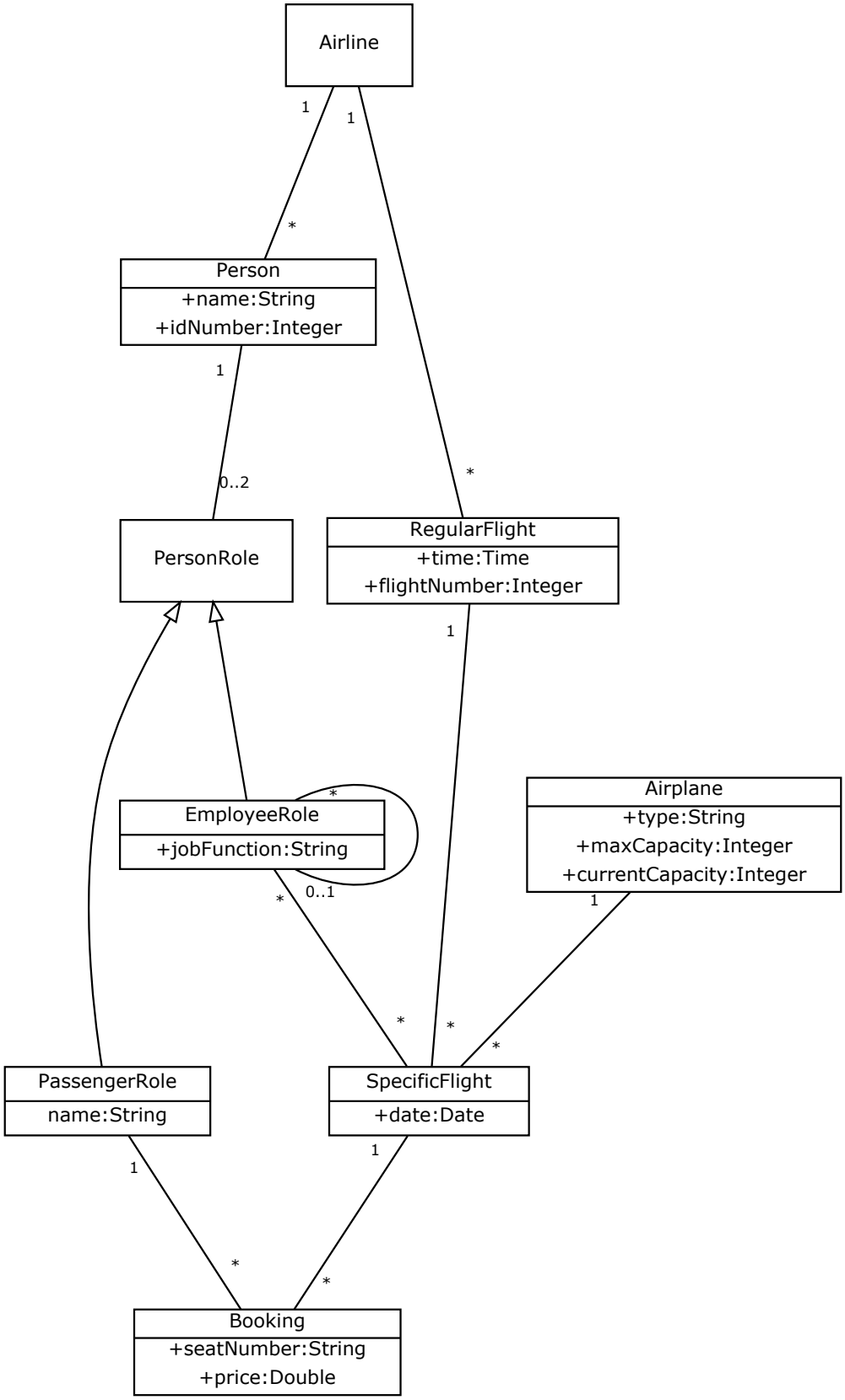


**Figure 5.5: Airline system class diagram**

The Airline system was textually modeled in Umple, as shown in Listing 5.19. Line 9 shows the bidirectional association between an airplane and a specific flight. An airplane can be assigned to many specific flights, but a specific flight will only be scheduled to a single airplane. Both lines 21 and 27 are Umple statements that represent the generalization relationship between two classes.

**Listing 5.19: Airline system static model**

Umple

```
1  class Airline {
2    1 -- * RegularFlight;
3    1 -- * Person;
4  }
5  class Airplane {
6    String type;
7    Integer maxCapacity;
8    Integer currentCapacity = 0;
9    1 --  * SpecificFlight;
10 }
11 class RegularFlight {
12   Time time;
13   unique Integer flightNumber;
14   1 -- * SpecificFlight;
15 }
16 class SpecificFlight {
17   unique Date date;
18   1 -- * Booking booking;
19 }
20 class PassengerRole {
21   isA PersonRole;
22   immutable String name;
23   1 -- * Booking;
24 }
25 class EmployeeRole {
26   String jobFunction;
27   isA PersonRole;
28   * -- 0..1 EmployeeRole supervisor;
29   * -- * SpecificFlight;
30 }
31 class Person {
32   String name;
33   Integer idNumber;
34   1 -- 0..2 PersonRole;
35 }
36 class PersonRole {}
37 class Booking {
38   String seatNumber;
```

```
39    Double price;
40 }
```

Dynamic behavior of a booking instance was captured by state machine shown in Figure 5.6. The Booking state machine consists of 5 states, with two as final states (‘cancelled’ and ‘completed’). Multiple events trigger transitions from one state to another. The initial state is ‘newBooking’ and event ‘assignSeat’ will do a transition from this state to ‘seatAssigned’ state. Event ‘checkIn’ will do a transition to ’checkedIn’ state. Reaching final state ‘completed’ indicates the booking was completed successfully. A cancel request can be made in any of the previous states before reaching the ‘completed’ state.

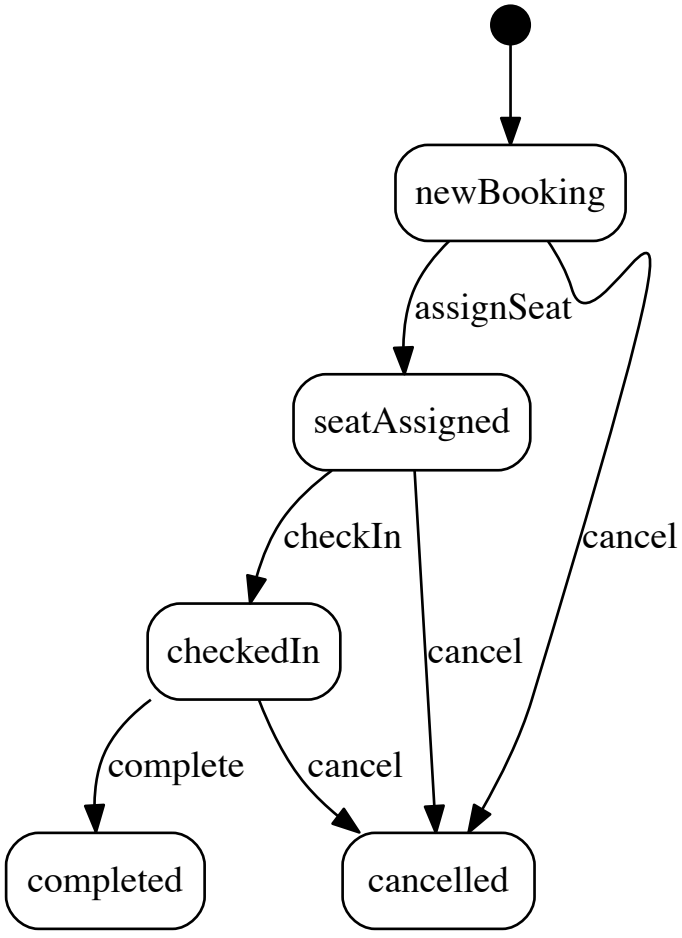


**Figure 5.6: Booking state machine**

Using the mixin capability in Umple, the state machine specification was written separately from the static model specification. In Listing 5.20, lines 2 to 17 represent the booking status state machine. Line 3 shows a state name.

**Listing 5.20: Airline system dynamic model**

Umple

```
1  class Booking {
2    state {
3      newBooking {
4        assignSeat -> seatAssigned;
5        cancel -> cancelled;
6      }
7      seatAssigned {
8        cancel -> cancelled;
9        checkIn -> checkedIn;
10     }
11     checkedIn {
12       cancel -> cancelled;
13       complete -> completed;
14     }
15     cancelled {}
16     completed {}
17   }
18 }
```

Next, we will create tracing objectives for the Airline model and use MOTL for trace specification by writing trace directives. Then, the Airline model will be executed to generate execution traces. The goals of our tracing are:

**Goal 1.** Report types of every added small-sized airplane with capacity of 50 passengers.

**Goal 2.** Trace dates of overbooked flights.

**Goal 3.** For every completed booking, record its price and seat number.

**Goal 4.** Keep a track record of cancelled bookings.

To achieve each goal, a trace directive was written in a separate trace script as shown in Listing 5.21. The first trace directive will trace any addition of an airplane with maxCapacity equal or less than 50 passengers, and record the airplane type. The second trace directive will trace any addition or deletion of association links between SpecificFlight and booking only when the number of objects linked (i.e. cardinality) of bookings is bigger than

maximum airplane capacity, meaning there is an overbooking in that flight. When this condition is satisfied, a tracepoint will be triggered and the date of that overbooked flight will be recorded along with other details. The third trace directive will trace whenever we enter the final state 'completed' and report the values booking price and seat number assigned. The last trace directive will trace every transition to state 'cancelled'. An alternative trace directive that can replace the last one is the tracing of any triggering of event cancel.

**Listing 5.21: Airline system trace directives**

Umple

```
1  class Airplane {
2    trace maxCapacity where [maxCapacity =< 50]
3                       record type;
4  }
5  class SpecificFlight{
6    trace booking
7    where[booking cardinality > airplane.getMaxCapacity()]
8    record date;
9  }
10 class Booking{
11   trace completed record price,seatNumber;
12   trace cancelled;
13   // alternative: trace cancel;
14 }
```

*Due to the limitations of UmpleRun*, that includes its current inability to create multiple objects and pass parameters with commands, we were unable to create a full execution of the Airline model to examine traced association links, however, we created a simulation using a main method, and mimicked the addition of three airplanes of different types and sizes and performed an overbooking with the purpose of examining generated execution traces. After performing the simulation, we obtain the execution trace in Listing 5.22. Line 2 and 3 shows that two small sized airplanes have been added to the system of 'Canadair CRJ100' of 50 passengers capacity and 'Embraer 120 Brasilia' of 30 passengers capacity. Execution lines 4 and 5 are related to tracing of the booking state machine. Line 4 indicates a completed booking with the price of 1000 CAD and seat assigned B12, while line 5 indicates a cancelled booking, as we were in state 'checkedIn' before the cancellation occurred. The operation code 'sm_t' in both lines (line 4 and 5) indicates a transition. Lines 6 to 9 show that

overbooking occurred between SpecificFlight and Booking classes, with the overbooked flight date reported.

**Listing 5.22: Execution trace for Airline system**

**Execution trace**

```
1 Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
2 *,1,model.ump,78,Airplane,2053965899,at_s,maxCapacity,50,Canadair CRJ100
3 *,1,model.ump,78,Airplane,1914494719,at_s,maxCapacity,30,Embraer 120 Brasilia
4 *,1,model.ump,84,Booking,614691688,sm_t,checkedIn,complete,completed,B12,1000.0
5 *,1,model.ump,85,Booking,1726858146,sm_t,checkedIn,cancel,cancelled
6 *,1,model.ump,81,SpecificFlight+Booking,2065554654,as_a,booking,101,2015-07-24
7 *,1,model.ump,81,SpecificFlight+Booking,2065554654,as_a,booking,102,2015-07-24
8 *,1,model.ump,81,SpecificFlight+Booking,2065554654,as_a,booking,103,2015-07-24
9 *,1,model.ump,81,SpecificFlight+Booking,2065554654,as_a,booking,104,2015-07-24
```

Table 5.4 performs a linking map between trace goals stated for the airline system and corresponding trace directive for each goal, then, link them to resulting execution trace lines.

**Table 5.4: Mapping link between goals, trace directives, and trace execution lines**

| Goal | Trace directive line | Execution line |
|---|---|---|
| 1 | 2 | 2-3 |
| 2 | 6 | 6-9 |
| 3 | 11 | 4 |
| 4 | 12 | 5 |

## 5.5. Summary

In this chapter, we presented the syntax and semantics of tracer directives that allows modelers to choose the tracing technology that they desire. MOTL supports a list of tracer technologies for modelers to choose from, specified in a tracer directive. The grammar and metamodel of the MOTL tracer directive were presented, and examples were provided. Trace output components forming trace messages passed to the tracing technology were outlined. Finally, we generated execution traces from executing traced modeling constructs by exploring two modeling examples.

# Chapter 6. Quality and validity of MOTL

This chapter discusses the aspects of MOTL's development process that have helped ensure its quality and validate its correctness and usefulness. We first briefly discuss the open-source nature of MOTL. Then we explain how MOTL was developed in an agile manner using both Model-Driven Development and Multi-level Test-Driven Development to ensure correctness-by-construction and the generation of high quality code. This is followed by a discussion of MOTL's documentation. Lastly, we explore MOTL's internal usage in the CRuiSE research group in two other Umple subprojects.

## 6.1. The Open-Source Nature of MOTL

An important factor for the success of university-developed research software is the decision to release the software code base to the public (open source) or reserve the right to hide all implementation details (closed source). Open source software (OSS) has established a generally positive reputation in terms of quality to compete with proprietary product [63, 64]. In Umple, the open source approach was adopted in 2012, and many positive results have arisen from that decision. Contributions to Umple were previously limited to the CRuiSE research team. However, with the open source approach, contributors from many geographic locations have been able to contribute to Umple. In particular 40 students from 21 Canadian and US universities have worked on Umple through the Undergraduate Capstone Open Source Projects (UCOSP [65]).

MOTL is fully open source. This was a natural consequence of it being implemented in Umple and incorporated as part of Umple. Some of the above-mentioned UCOSP students have worked on aspects of MOTL, such as fixing bugs. Over 90% of the work was done, however, by the author.

## 6.2. MOTL's Model-Driven, Test-Driven Agile Development Process

In this section we discuss the process we followed to ensure the quality of MOTL.

MOTL was implemented following the principles of both Test-Driven Development (TDD) [29, 30] and Model Driven Development (MDD). TDD is a style of software development where tests are written before the implementation of the system under development. The actual code is written and refactored until all tests pass. In other words, tests serve as the specification for code that needs to be written. MDD involves starting with the model of the system being developed and generating code from the model. Models are higher-level abstractions that simplify the understanding of the system [3, 66]. Using both MDD and TDD together helped make development of MOTL more rigorous, and resulted in a low level of bugs, and a high level of maintainability and development velocity.

In addition to MDD and TDD, we also employed agile processes, including developing in small increments, and continuous integration (building and releasing whenever increments pass all test cases, often several times a day).

Every change (including new features, refactoring and bug fixes) followed the process shown in Figure 6.1:

1. **Test preparation**: At the beginning, tests are created acting as the specification for the feature to be implemented.
2. **Modeling and Coding**: The feature is implemented, including updating the model and associated algorithmic code. Then a local build is fired to build the new system (including the new feature) and test whether the written code breaks any of the tests.
    a. *Successful local build*: Code changes are ready to be committed.
    b. *Failed local build*: failed tests must be investigated and required code changes are implemented.
3. **Commit**: After a successful local build, all changed aspects of the model and code are committed to the project host repository. Committing is guarded with privileges, to not allow any unknown commits from unidentified sources that would mess up the project environment. Anyone, outside the project committers' list, wishing to contribute can send a patch to one of the committers to verify it and then commit the patch.

4. **Integration on the Server**: The continuous integration server detects the commit and runs a full build, including executing all tests. The verdict can be one of the following:
    a. *Successful server build*: The change has been integrated successfully with the existing system. Other developers then incorporate the changes into their ongoing development stream; post-commit code inspection is done by other team members, which may result in further changes.
    b. *Failed server build*: In an ideal world, TDD would prevent this from happening, but it occasionally happens due to missing an item from the commit, or differences between development and server environments. Such a failure is normally followed within a few minutes by making another minor change, repeating the above process again. If the failure can not be quickly fixed (within an hour or so), a revert to the previous version is performed.

It should be noted that the model described involves a single head branch, and was based on Subversion. The trend in other projects is to use Git that has a more sophisticated versioning model. Umple and MOTL will be moving to this in the future.

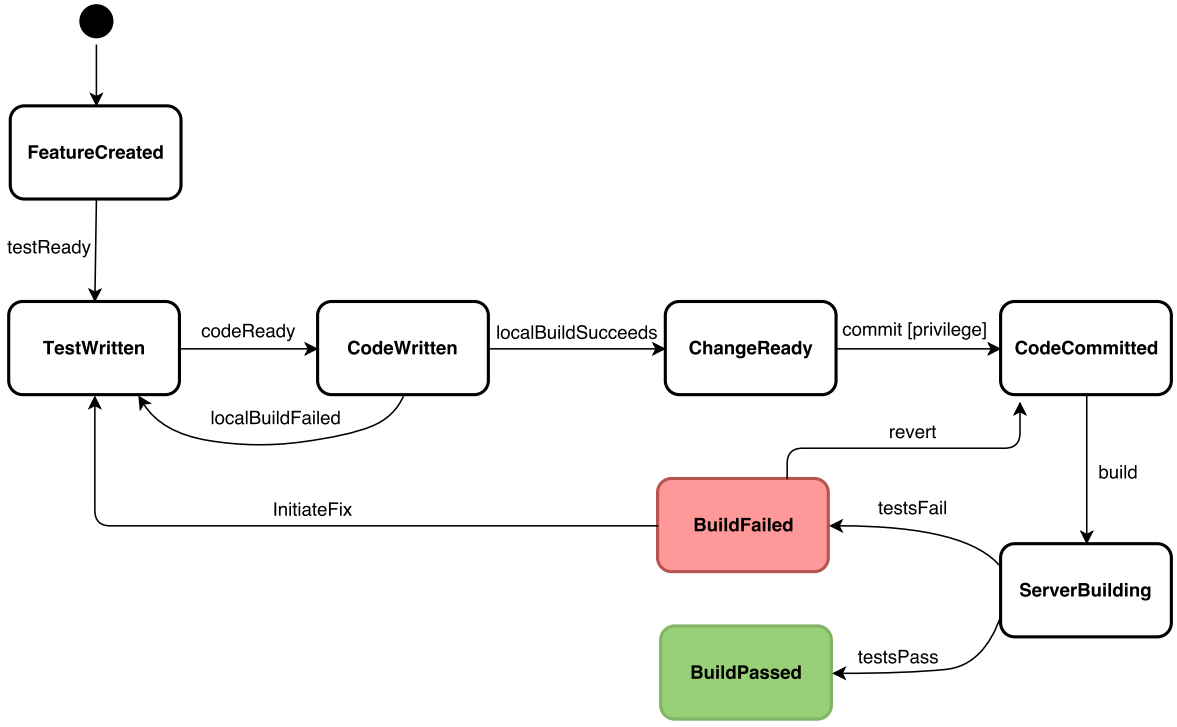


**Figure 6.1: Process of adding MOTL features**

Multi-level test-driven development (MLTDD) is achieved when the software is given a layered architecture, and every independent component of the system follows the test-driven approach. MOTL follows the MLTDD by having its own test suite for every component in the Umple architecture: Parser, Analyzer, and Code generator. In addition to these components, a semantics test suite is included to run systems *generated* by Umple/MOTL and asserting their behavior. In the next subsections, we will traverse through the different tests created for MOTL.

### 6.2.1 Tokenization Tests

Tokenization tests assert whether the Umple parser is tokenizing the Umple code as expected or not. Hence, tests in this category are based on tokenization strings. Let's consider the following code excerpt including MOTL directives:

```
tracer file log.txt;
class Tracer {
  name;
  trace get name;
}
```

The parser analyzes the above code and generates the following tokenization string. The parser tests, run under JUnit and assert that the generated tokenization string (i.e. actual) should be identical to the expected string.

```
[tracerDirective][tracerType:file][tracerArgument:log.txt]
[classDefinition][name:Tracer][attribute][name:name]
[trace][traceDirective][Prefix][option:get][traceEntity:name]
```

Over 70 JUnit tests reside in the MOTL part of the Umple parser test suite; they test tokenization strings for the full range of MOTL capabilities. We do not claim 100% coverage of all possible equivalence classes, however. Other levels of testing should help increase the coverage.

### 6.2.2 Metamodel Tests

Metamodel tests ensure that an input model after being tokenized populates the correct elements into an instance of the Umple metamodel. Considering the previous trace directive, below is the JUnit test that asserts that a correct instance of the Umple metamodel is generated.

**Listing 6.1: Metamodel test example**

JUnit

```
@Test
public void traceSingleAttributeGet()
{
  UmpleClass clazz = model.getUmpleClass("Tracer");
  TraceDirective traceDirective1 = clazz.getTraceDirective(0);
  AttributeTraceItem traceAttr1= traceDirective1.getAttributeTraceItem(0);
  Assert.assertEquals(clazz.getAttribute("name"),traceAttr1.getAttribute(0));
  Assert.assertEquals(false,traceAttr1.getTraceSet());
  Assert.assertEquals(true,traceAttr1.getTraceGet());
}
```

Metamodel tests reside in the same suite as the parser test suite and are run after the parsing tests. MOTL has metamodel tests equivalent in quantity to its parser tests.

### 6.2.3 Generated Code Syntax Tests

Syntax tests ensures that generated code's syntactical structure is as expected. In the case of MOTL, Java is currently the only supported target programming language. JUnit tests were used to assert that the expected generated code is equivalent to the model-generated code. Continuing with the above MOTL example, the expected generated code should be equivalent to:

**Listing 6.2: Expected syntax structure example**

Java

```
import cruise.util.FileTracer;
public class Tracer
{
  private String name;

  public Tracer(String aName)
  {
    name = aName;
  }
```

```
  public boolean setName(String aName)
  {
    boolean wasSet = false;
    name = aName;
    wasSet = true;
    return wasSet;
  }

  public String getName()
  {
    FileTracer.handle(
System.currentTimeMillis()+","+Thread.currentThread().getId()+
",model.ump,5,Tracer,"+System.identityHashCode(this)+",at_g,name,"
+name );
    return name;
  }

  public void delete()
  {}

  public String toString()
  {
        String outputString = "";
    return super.toString() + "["+
             "name" + ":" + getName()+ "]"
     + outputString;
  }
}
```

Equivalence means in this case ‘exactly the same as’. However there are syntax tests where certain differences are ignored. For example Umple injects comments into the generated code specifying the line numbers in the source files that gave rise to the generated output. Since these line numbers will vary over time, the equivalence testing process overlooks commented line number differences.

There are over 200 JUnit tests in the MOTL syntax test suite [31]. These tests include tracing of attributes, state machines, associations, and non-API methods with different constraints. In addition, tests are added for different supported tracers.

### 6.2.4 Semantics Tests

Semantics tests execute generated systems to assert their dynamic behavior. In MOTL, semantics test design were not as straightforward as the semantics test for other Umple features. Semantics tests for MOTL should assert whether the trace output is as expected. Thus,

we employed the console tracer for semantics test and intercepted the system error stream to verify the executed systems output.

Continuing with the MOTL code used in previous sections, we will show how the semantic test is conducted. First, a 'Tracer' class instance is created and performs some method calls. In this example, the trace directive will trace whenever an attribute name is accessed (i.e. a trace of the generated get method). As a result of the method calls, a trace output should appear on the system error stream, which will be compared against the expected trace output in array 'attrTraceExpected'. Assertion is done on part of the trace output, starting from the operation code output component, since data such as thread ID and execution timestamp will vary from run to run.

**Listing 6.3: MOTL sematic test example**

JUnit

```
@Test
public void traceConsoleAttributes()
{
  PrintStream ps = new PrintStream(System.err){
    int index=0;
    String[] attrTraceExpected = {
       "at_g,name,Steve",
       "at_g,name,John",
    };
    //------------------------------------- Prepare Console tracer
    public void println(String x){
     if(index<attrTraceExpected.length){

       String[] actualOutput = x.split(",");
       String[] expectedOutput = attrTraceExpected[index].split(",");
       int shiftIndex = 6;

       // header skip
       if( index != 0 )
         for( int i = 0 ; i < expectedOutput.length ; ++i )
           Assert.assertEquals(expectedOutput[i],actualOutput[i+shiftIndex]);

       index++;
     }
     else
       Assert.assertTrue(false);
    }
  };
  System.setErr(ps);
```

```
  //------------------------------------ invoke attributes setters to generate traces
  Tracer aTest = new Tracer(null);
  aTest.setName("Steve");
  aTest.getName();
  aTest.setName("John");
  aTest.getName();
}
```

## 6.3. Errors and Warnings Reporting

MOTL supports the reporting of issues found in an Umple model when parsing trace and tracer directives in the form of error and warning messages. The Umple compiler will emit these error and warning messages into any of the supported development environments (Eclipse, command line, and UmpleOnline) when an issue is located. MOTL distinguishes between an error and a warning as follows:

- **Error message**: An issue is considered an error when an Umple compiler is unable to parse the Umple model or generate code. Examples of such error messages in MOTL include tracing of non-existent model constructs and incorrect trace directive structure.
- **Warning message**: Unlike errors, warnings will be raised if code can be generated from the Umple model, however, the generated system may not have the desired semantics. An example of a warning in MOTL is the use of an unidentified or unsupported tracer in a tracer directive. If a modeler uses an invalid tracer technology name in a tracer directive, a warning message will be raised and the default console tracer will be used instead to inject tracepoints.

Both types of messages follow the same reporting pattern shown below: Type of raised message (error or warning), line number of defected MOTL directive, and description of the error/warning.

```
[Bug type] [Line number] : [Description]
```

## 6.4. Documentation

Documentation is a well-recognized artifact and activity of software development, however it is often neglected and not given sufficient priority [67]. MOTL provides documentation for both developers and users. Developers wishing to contribute or merely for informative motives have access for the following MOTL documentation:

1. **Code comments**: all MOTL infrastructure source files are commented.
2. **Generated documentation** [28]**:** Umple has built-in self-documentation features that generate UML diagrams after every commit as well as JavaDoc pages documenting every Umple file and Java class
3. **Development Wiki** [68]: In the Umple project website wiki, developers can access MOTL architectural overviews to help with code navigation, moreover, information about how to set up and manage the development environment. Many developers also leave logs of their experiences in the wiki pages.

MOTL end users are also provided with a simplified and readable user manual [69] to explore its usage. Different aspects of MOTL are explained with examples that are linked to UmpleOnline. This manual is generated from pages specified in a special Domain Specific Language (DSL), and includes code examples that are tested to ensure their validity. Changes to these pages are performed in the same manner as changes to any other aspects of the code.

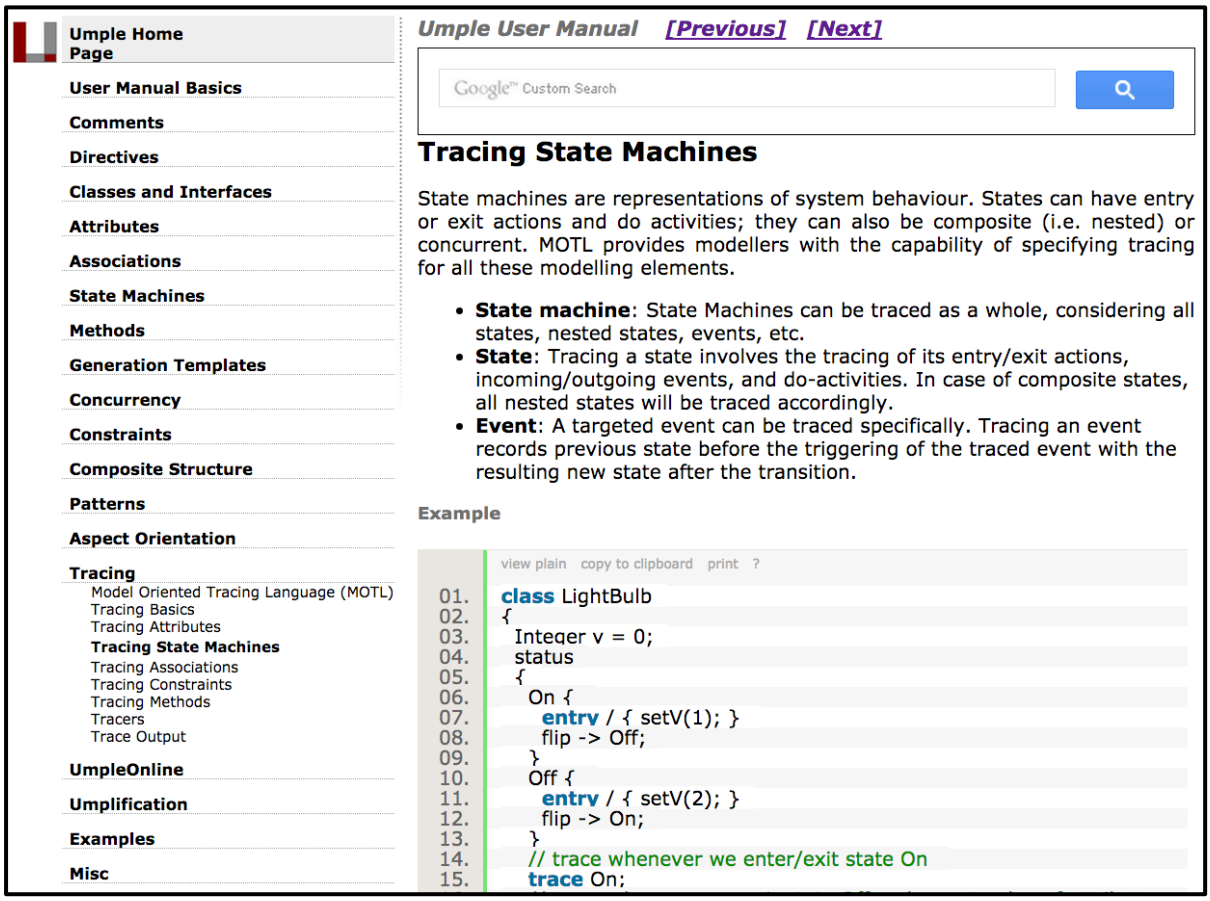


**Figure 6.2: MOTL user manual**

## 6.5. Usage of MOTL by Others

MOTL has been used internally in the CRuiSE research team by two Umple related projects: Umplificator and PSM/QSM. In this section, we will explore MOTL utilization in both projects.

### 6.5.1 Umplificator

A subproject of the CRuiSE research team, the Umplificator [34, 70] is a reverse engineering tool used to retrieve Umple models from object-oriented source code. It generates complete systems equivalent to the original system, since all algorithmic methods become part of the Umple model as well.

The Umplificator itself was developed using the Umple language with modeling constructs and Java native code to handle the Umplificator algorithmic operations. Umplificator

design consists of different components: Parser(s), Extractor, Transformer, and Generator(s). During the development, there was a demand to inject traces in different parts of the components. Consequently, MOTL was employed for the Umplificator trace specification. MOTL trace specification was modeled independently from the rest of the Umplificator components as a trace script named (Umplificator_traces.ump) [70].

Listing 6.4 presents the Umplificator trace specification using MOTL. The first line indicates that log4j is the desired tracer technology to be used in the Umplificator, with an option to exclude the generation of XML configuration file. Trace directives were written for four Umplificator classes, and the majority of them trace non-API methods, which comprise most of the Umplificator model.

In the Umplificator class (main class), the first trace directive in line 4, traces any invalid output folder directory and will direct the trace output to 'fatal' level. The second trace directive traces names of any created umplified files, and directs the output to the default level info (log level is not specified). The last trace directive in this class (line 7), logs the entry of non-API method 'umplifyCPP' with a provided message that both will be directed to the 'error' level. In the remaining classes, all trace directives log the entry of methods. In addition the exit of methods is logged in class JavaParser.

**Listing 6.4: Umplificator trace specification**

Umple

```
1  tracer log4j noconfig;
2
3  class Umplificator {
4    trace outputFolder where [outputFolder == null]
5                        logLevel fatal;
6    trace filesUmplified;
7    trace umplifyCPP record "not implemented yet"
8                      logLevel error;
9  }
10
11 class JavaParser {
12   trace parseUnit logLevel debug
13                   record "Parsing Compilation Unit";
14   trace exit parseUnit logLevel debug record
15   "Initializing Java Visitor for Compilation Unit";
16   trace parseBodyDeclarations logLevel debug record
17   "Parsing Body Declarations from source";
18   trace exit parseBodyDeclarations logLevel debug record
```

```
19    "Initializing Java Visitor for Body Declarations";
20  }
21
22  class RuleService {
23    trace startRuleEngine logLevel debug
24          record "RuleService.startRuleEngine";
25  }
26
27  class RuleRunner {
28    trace RuleRunner logLevel debug record
29    "Instantiate RuleRunner-Res, FileSys, Repo created";
30    trace addRuleFile logLevel debug record
31    "Add Rule Files to Session";
32    trace insertJavaElements logLevel debug record
33    "Insert Java elements into working memory";
34    trace insertUmpleClass logLevel debug record
35    "Insert uClass into working memory";
36    trace insertUmpleModel logLevel debug record
37    "Insert Umple Model into working memory";
38    trace insertUmpleInterface logLevel debug record
39    "Insert uInterface into working memory";
40    trace fireAllRules logLevel debug record
41    "Fire rules";
42  }
```

At the early stages of the Umplificator development, the following tracer directive was used to choose the log4j tracer and also generate a configuration file (log4j2.xml) taking into consideration certain customizations requested. In particular, the root logger was set to 'debug' level, and file and console appenders were used to log trace output.

```
tracer log4j root=debug info=file,console;
```

However, as the development of the tool was advancing, further specialized appenders (e.g. RollingFile) were required as a way to control the size of the log files (split the log files when they reached 2GB) and add additional logging levels. A more extensively customized XML configuration file was therefore manually written for the Umplificator. The tracer directive was changed to the following, indicating that the generation of the configuration file is not required since the (manually created) customized file would be used instead:

```
tracer log4j noconfig;
```

### 6.5.2 PSM and QSM

As an extension to Umple state machines, pooled state machines (PSM) and queued state machines (QSM) were implemented as a Masters thesis project [60]. Both of these state machines were designed to satisfy requirements of multi-threaded environments, with the main idea being to create a thread to handle event queues. However, PSM and QSM differ in the order in which events are handled. It was reported in the thesis that MOTL was used to trace state machines and was found beneficial to understand their behavior. In fact, the tracing of PSM and QSM is supported by MOTL, but it has not been tested thoroughly yet.

# Chapter 7. MOTL Empirical Study

As discussed in previous chapters, MOTL is a textual language with structured syntax to perform tracing at the model level. Modelers will write trace directives to trace different model constructs. To assess whether we have made reasonable decisions, we performed an empirical study of users to assess how well they could use MOTL. The general objective of this study was to measure MOTL syntax usability and discover potential language improvements. A group of participants were invited to learn MOTL and perform tracing tasks with it. In particular, we wanted to assess:

I. **Learnability**: whether new MOTL users were able to grasp the language concepts and learn the syntax and semantics well enough to be able to write correct MOTL snippets.
II. **Subjective appreciation**: whether participants liked the language or not.

## 7.1. Study planning

In this section, we overview our study phases and describe the study setup.

### 7.1.1 Phases

Every participant undertook each of the study phases as shown in Figure 7.1 and listed below:

1. **Consent form**: every participant agreed to a consent form before proceeding.
2. **Background questions**: A series of background questions were answered. These questions were used to study the nature of participants; had any participants not met certain minimum knowledge requirements, they would not have been asked to continue. These will be discussed below in section 7.2.1.
3. **Tutorial session**: Participants followed a two-part tutorial. The first was a mandatory five minute video tutorial; in the second part, participants were directed to read the sections about MOTL in the Umple online user manual [71]. The manual is organized

so that users can explore MOTL syntax alongside modeling examples. Further details are in section 7.3.

4. **Models with trace tasks**: Participants were shown Umple models in textual/visual forms, and then were asked a series of trace questions that require the writing of MOTL trace directives. These are discussed in section 7.4.1.
5. **Survey questionnaire**: At the end of the study and after participants were exposed to MOTL syntax, they were asked to answer a questionnaire to obtain feedback about MOTL, and the study itself, for future improvements. The questions and results obtained are discussed in section 7.5.

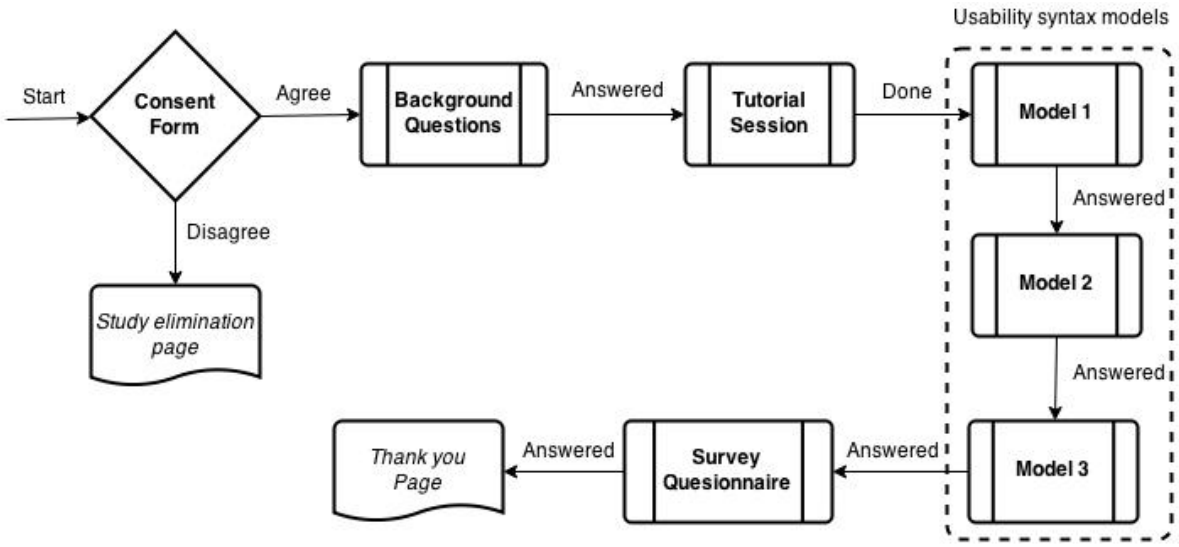


**Figure 7.1: Empirical study execution phases**

### 7.1.2 Setup

The tasks mentioned above were placed online as a survey in the SurveyMonkey website to gather responses from participants. Using this approach allowed us to receive complete answers from each participant, and ensured that each participant followed the same execution path.

## 7.2. Demographics

A total of 18 individuals participated in our study. Participants were recruited through personal invitation emails, including an invitation text and the link they can select if they wish to participate. Participation was conducted on an individual basis, in contrast to group participation setting where a group of participants are gathered in a controlled environment. Study participants were treated anonymously, and each participant was assigned a number.

As mentioned in section 7.1.1, participants were first asked four mandatory background questions. These questions were intended to categorize participants based on their background knowledge and were not intended to reveal the identity of participants. In the next subsection, we outline these background questions and then provide a summary of the answers.

### 7.2.1 Background Questions

The following four background questions were asked. The answer choices for each question are given in Table 7.1.

**Background question 1)** *What is the highest level of education you have completed or in the progress of completing?*

**Background question 2)** *Rate your knowledge of UML*

**Background question 3)** *Rate your knowledge of Umple*

**Background question 4)** *Rate your experience in Java programming*

### 7.2.2 Participant Categories

Table 7.1 presents the answers distribution over different background question. Total answers for each question sum to the total number of study participants, meaning that it was required to answer each background question.

Most of the participants (83%) either hold or are working on a graduate degree, while the remaining (17%) hold an undergraduate degree.

The second background question was designed to allow us to eliminate participants with low or no knowledge of UML, since they would have presented a validity threat. However, all of our participants had previous knowledge of UML.

The third background question was to classify participants based on their knowledge of Umple. Our participants were almost evenly split with (55%) having minimal or no previous knowledge of Umple and (45%) previously having been introduced to Umple.

The last background question was created to ensure participants had some prior Java programming knowledge. Almost all of our participants had a desired level of Java background.

**Table 7.1:Background questions answers distribution**

| Background Question | Answer Options | Response Count | Response % |
|---|---|---|---|
| *Q1 (highest education)* | BSc (Bachelor) / Working on BSc | 3 | 16.7% |
| | MSc (Masters) / Working on MSc | 5 | 27.8% |
| | PhD (Doctorate) / Working on PhD | 10 | 55.6% |
| *Q2 (UML knowledge)* | High | 7 | 38.9% |
| | Medium | 11 | 61.1% |
| | Low | 0 | 0.0% |
| | Never heard of UML | 0 | 0.0% |
| *Q3 (Umple knowledge)* | High | 6 | 33.3% |
| | Medium | 2 | 11.1% |
| | Low | 8 | 44.4% |
| | Never heard of Umple | 2 | 11.1% |
| *Q4 (Java experience)* | Professional | 10 | 55.6% |
| | Moderate | 7 | 38.9% |
| | Novice | 1 | 5.6% |
| | No Java Experience | 0 | 0.0% |

Based on data gathered from the background question, we divide our participants into two groups as listed in Table 7.2:

**Table 7.2: Usability study participants' groups**

| Participants group | Common characteristics | Sample size |
|---|---|---|
| Umple | Participants in Umple group have knowledge of UML and Umple. | 8 |
| Non-Umple | Participants in Non Umple group have knowledge of UML but has minimal or no knowledge of Umple. | 10 |

## 7.3. Tutorial

The tutorial session was held to familiarize study participants with MOTL syntax. It consisted of two parts: a video demonstration and a user manual. At first, participants were directed to a 5-minute video clip created by the principal investigator to introduce MOTL syntax and explain how it can be used to perform trace tasks at the model level. After that, participants were given the freedom to browse the MOTL online user manual [71] to further explore the language features with small examples at their own pace. Once they were satisfied, the tutorial session was concluded and they were guided to the next step.

## 7.4. Trace Tasks

After completing the tutorial session, participants were presented with three models and given a number of trace objectives that requires the writing of trace directives.

### 7.4.1 Umple Models

Study participants were shown a number of Umple models to study, and then were asked to answer a series of questions requiring the writing of trace directives. Three Umple models were selected as inspired by [26, 62]. Model complexity increases from model one to model three in terms of number of classes and the existence of a state machine. In next subsections, we will describe each model and outline the trace questions asked, representing trace tasks for each model; we also provide the required trace directives that satisfy each trace task.

## Model 1: Insurance Claim

The Insurance claim model consists of two classes capturing the relationship between a claimant and submitted claims. Each new claim has a sequence number, description and total amount claimed. A claimant is identified by a name and an insurance policy number, and can submit unlimited number of insurance claims. The insurance model was modeled textually using Umple in Listing 7.1.

**Listing 7.1: Model 1 Insurance claim Umple code**

**Umple**

```
1  class Claim {
2    String sequenceNumber;
3    String description;
4    Double amountClaimed;
5  }
6  class Claimant {
7    String name;
8    Integer policyNumber;
9    1 -- * Claim request;
10 }
```

Table 7.3 provides modeling measures for insurance claim system classes. The system has 5 attributes and a single association. Insurance claim class diagram is shown in Figure 7.2.

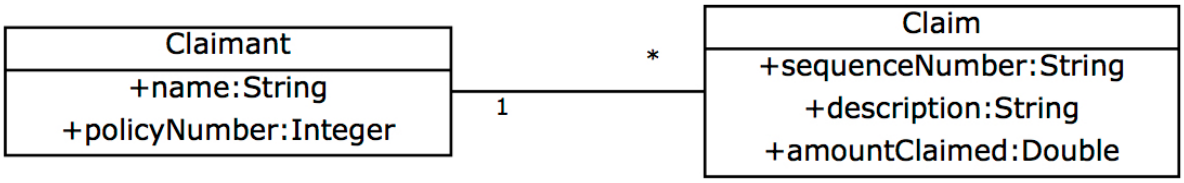


**Figure 7.2: Model 1 Insurance claim class diagram**

**Table 7.3: Model 1 classes' summary**

| Class | Number of Attributes | Number of Associations |
|---|---|---|
| Claim | 3 | 1 |
| Claimant | 2 | 0 |
| Total | 5 | 1 |

Tracing tasks were created in the form of questions as listed below:

**Trace Task 1.** A manager is interested to keep track of the sequence numbers of insurance claims with the amount claimed above 1000. (class: Claim).

**Trace Task 2.** For every claimant added to the insurance system, a manager would like to record his name and policyNumber. (class: Claimant)

**Trace Task 3.** Track whenever a claimant has requested a claim. (class: Claimant)

**Trace Task 4.** Track name of claimants that requested more than 10 claims. (class: Claimant)

Trace directives that satisfy each previously stated trace task are given in Table 7.4. These are the expected answers to the questions.

**Table 7.4: Model 1 Insurance claim trace directive answers**

| Trace Question | Correct answers: Expected trace directive(s) |
|---|---|
| 1 | trace sequenceNumber where [amountClaimed>1000]; |
| 2 | trace name, policyNumber; |
| 3 | trace request; |
| 4 | trace request where [request cardinality>10] record name; |

## Model 2: Student Registration System

The student registration system consists of four classes with attributes and associations between them. In addition, a state machine is defined inside class CourseSection to handle and capture the desired dynamic behaviour of our system. Umple code for the student registration system's static structure is provided in Listing 7.2. Each course has a unique course code with course description. Each course can have zero to many course sections, but a course section can be assigned to one course. Each course section has attributes to hold the information related to a course section such as minimum and maximum number of students allowed in the class if it is to be taught, the current class size, and the section id. A course section can have zero to many registrations with a single student assigned to each registration.

**Listing 7.2: Model 2 Student registration system static elements in Umple**

Umple

```
1  class Course {
2    code;
3    description;
4    1 -- * CourseSection;
5  }
6  class CourseSection {
7    sectionId;
8    Integer classSize = 0;
9    Integer minimumClassSize = 10;
10   Integer maximumClassSize = 100;
11 }
12 class Student {}
13 class Registration {
14   grade;
15   * -- 1 CourseSection;
16   * -- 1 Student;
17 }
```

Table 7.5 provides the static structure summary for student registration. There are seven attributes and 3 associations. Figure 7.3 shows the class diagram for the student registration system.

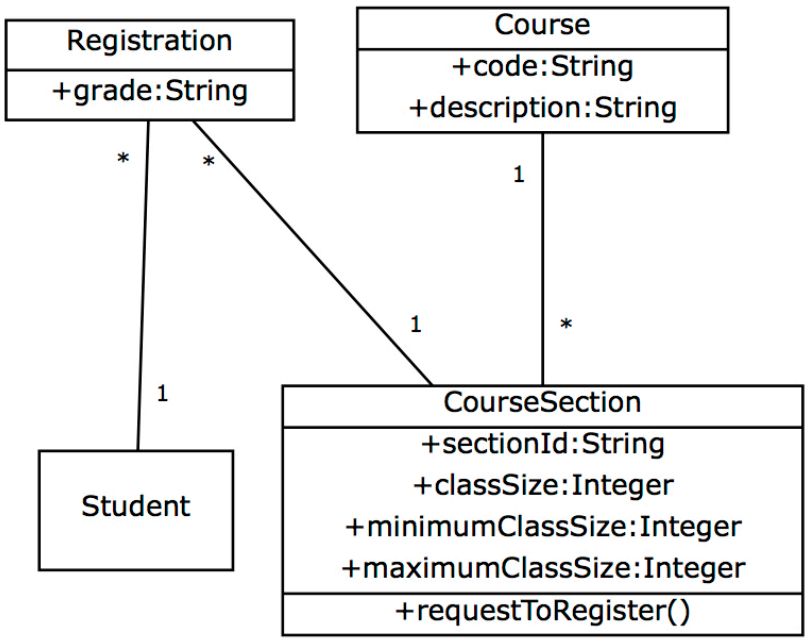


**Figure 7.3: Model 2 Student registration system class diagram**

**Table 7.5: Model 2 static structure summary**

| Class | Number of Attributes | Number of Associations |
|---|---|---|
| Course | 2 | 0 |
| CourseSection | 4 | 2 |
| Student | 0 | 1 |
| Registration | 1 | 0 |
| Total | 7 | 3 |

Dynamic behavior for student registration is captured by state machine in class CourseSection as shown in Listing 7.3. There are five states with different events to transit between states. For example, the initial state is the Planned state and the event openRegistration triggers a transition from state Planned to state OpenNotEnoughStudents. Some events are guarded, such as the register event from state OpenNotEnoughStudents to state OpenEnoughStudents.

**Listing 7.3: Model 2 Student registration system state machine in Umple**

Umple

```
class CourseSection{
  status {
    Planned {
      openRegistration -> OpenNotEnoughStudents;
    }
    OpenNotEnoughStudents {
      closeRegistration -> Cancelled;
      cancel -> Cancelled;
      register [getClassSize() > getMinimumClassSize()]
               -> OpenEnoughStudents;
    }
    OpenEnoughStudents {
      closeRegistration -> Closed;
      cancel -> Cancelled;
      register [getClassSize() > getMaximumClassSize()]
                -> Closed;
     }
     Cancelled {}
     Closed {}
  }
  // Code mixins
  boolean requestToRegister(Student aStudent) {
    register();
    setClassSize(getClassSize() + 1);
  }
}
```

Student registration system state machine visualization is shown in Figure 7.4.

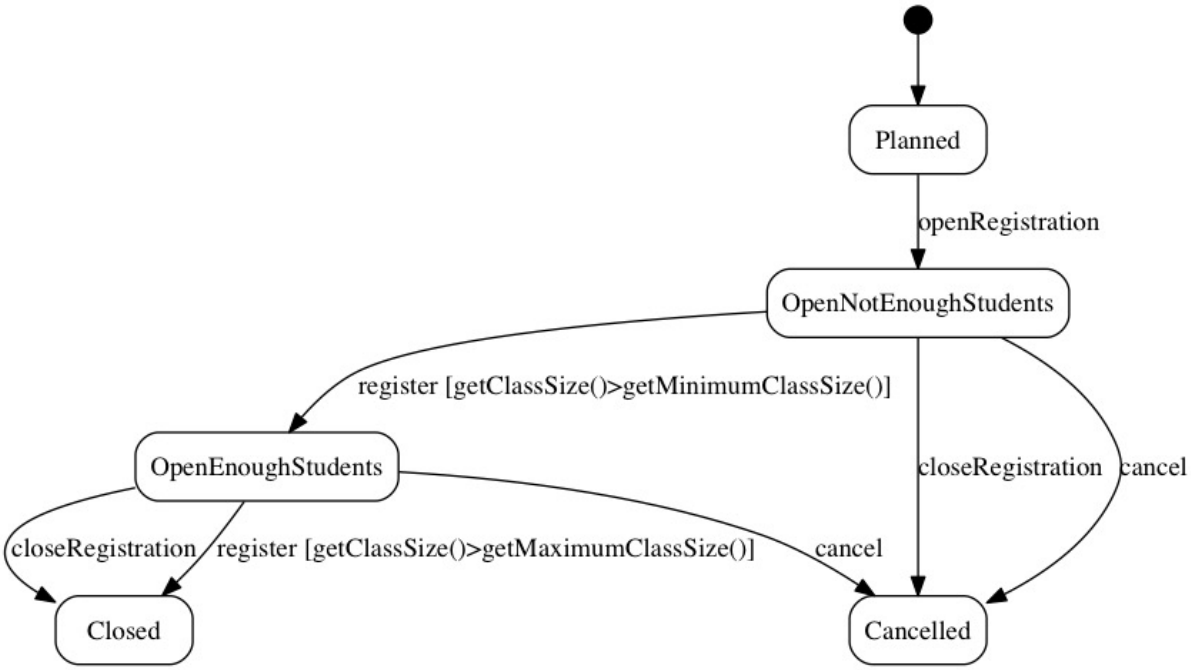


**Figure 7.4: Model 2 Student registration system state machine**

Table 7.6 presents Model 2 state machine 'status' statistics.

**Table 7.6: Model 2 state machine statistics**

| # States | # Transitions | # Guards | # Events |
|---|---|---|---|
| 5 | 7 | 2 | 7 |

As with Model 1, tracing tasks were created to trace different modeling constructs. In the context of CourseSection class, the following list of tracing tasks were created:

**Trace Task 1.** When the class size changes, trace the class size and course section id.

**Trace Task 2.** Track ids of course sections with their class size reaching the maximum capacity.

**Trace Task 3.** Capture all behavior specified by the state machine.

**Trace Task 4.** Check the class size of cancelled sections.

**Trace Task 5.** Track the class size of closed sections.

Trace directives that satisfy each previously stated trace task is given in Table 7.7. These are the expected answers.

**Table 7.7: Model 2 Student registration system tracing directives**

| Trace Question | Correct answers: Expected trace directive(s) |
|---|---|
| 1 | trace classSize record sectionId; |
| 2 | trace classSize where [classSize>=maximumClassSize] record sectionId; |
| 3 | trace CourseSectionStm; |
| 4 | trace entry Cancelled record classSize; |
| 5 | trace Closed record classSize; |

## Model 3: Airline System

The last model is a charter airline with no regular schedules. A charter airline can have multiple airplanes, and each airplane can be assigned to multiple flights uniquely characterized by its flight number and date. A passenger can make multiple bookings; where each booking has its own price and seat number. The behavior of the booking instance was captured by a state machine.

**Listing 7.4: Model 3 Airline system static elements in Umple**

Umple

```
1  class Airline {
2    1 -- * Airplane airplane;
3  }
4  class Airplane {
5    type;
6    Integer capacity;
7    1 --  * Flight flight;
8  }
9  class Flight {
10   String flightNumber;
11   unique Date date;
12   1 -- * Booking booking;
13 }
14 class Passenger {
15   String name;
16   Integer age;
17 }
18 class Booking {
19   String seatNumber;
20   Double price;
21   * -- 1 Passenger passenger;
22 }
```

Figure 7.5 shows airline system class diagram.

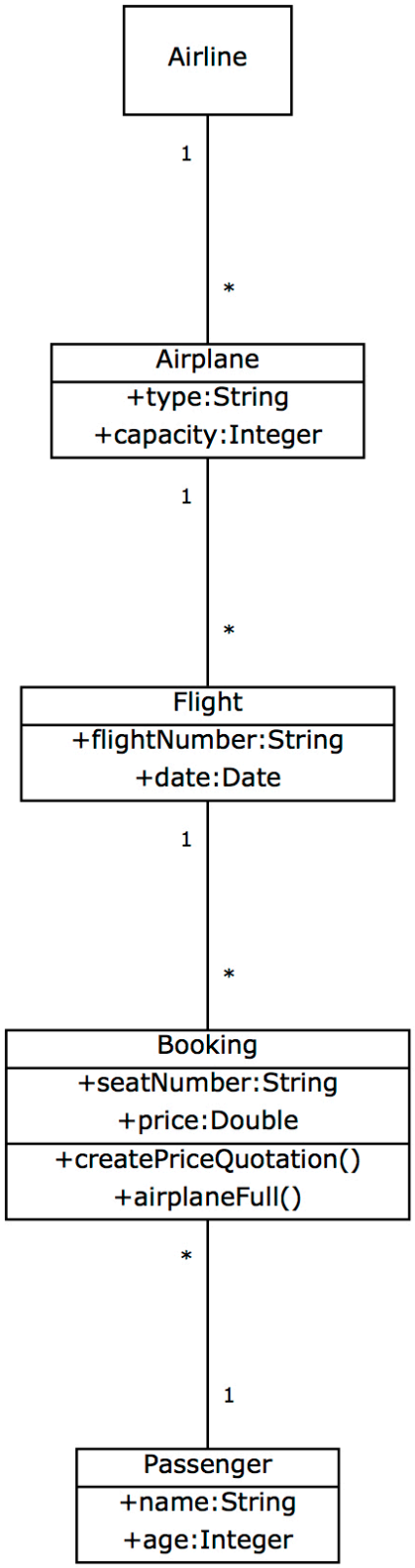


**Figure 7.5: Model 3 Airline system class diagram**

Table 7.8 summarizes the Airline system static structure statistics. It consists of 8 attributes and 4 associations.

**Table 7.8: Model 3 classes' summary**

| Class | Number of Attributes | Number of Associations |
|---|---|---|
| Airline | 0 | 0 |
| Airplane | 2 | 1 |
| Flight | 2 | 1 |
| Passenger | 2 | 1 |
| Booking | 2 | 1 |
| Total | 8 | 4 |

The behavior of the Booking instance was captured by state machine in Listing 7.5.

**Listing 7.5: Model 3 Airline system Booking state machine in Umple**

Umple

```
1  class Booking {
2    status {
3      InitiateReservation {
4        makeReservation -> ReservationCreated;
5      }
6      ReservationCreated {
7        entry / { createPriceQuotation(); };
8        issueETicket [!airplaneFull() ] -> EticketIssued;
9        cancel -> ReservationCanceled;
10     }
11     EticketIssued {}
12     ReservationCanceled {
13       startSearch -> InitiateReservation;
14     }
15   }
16 }
```

Airline system state machine visualization is shown in Figure 7.6.

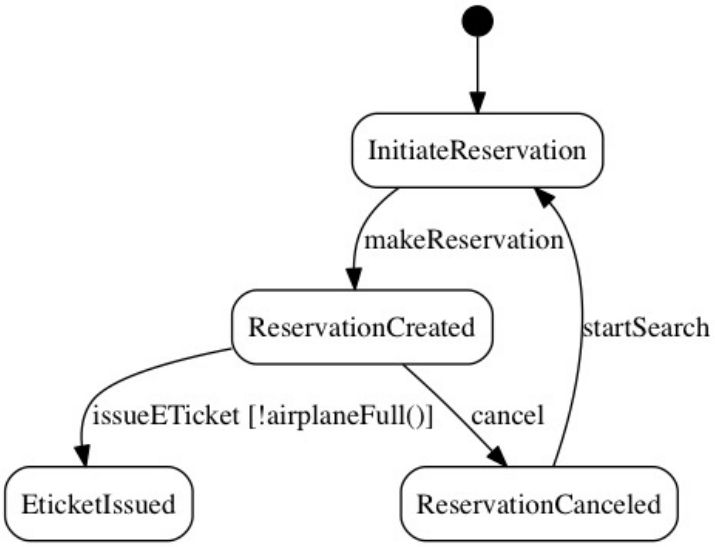


**Figure 7.6: Model 3 Booking state machine diagram**

Table 7.9 presents Model 3 state machine 'status' statistics.

**Table 7.9: Model 3 state machine statistics**

| # States | # Transitions | # Guards | # Events | # Actions |
|---|---|---|---|---|
| 4 | 4 | 1 | 4 | 1 |

Tracing tasks were created in the form of questions, as follows:

**Trace Task 1.** Track whenever a new airplane has been added or removed from the system.

**Trace Task 2.** Report flight numbers of flights that have an overbooking.

**Trace Task 3.** For every e-ticket issued to passengers, record its price and seat number.

**Trace Task 4.** Track cancelled reservations.

Trace directives that satisfy each previously stated trace task are given in Table 7.10. These are the expected answers.

**Table 7.10: Model 3 Airline model tracing directives**

| Trace Question | Correct answers: Expected trace directive(s) |
|---|---|
| 1 | trace airplane; |
| 2 | trace booking where [airplane.capacity < booking cardinality] record flightNumber; |
| 3 | trace EticketIssued record price,seatNumber; |
| 4 | trace entry ReservationCanceled; |

### 7.4.2 Rating of the Responses

After we executed the study and gathered data, we computed the error rate to assess the number of incorrect trace directives written by participants. Our marking scheme was inspired by a previous marking scheme proposed to compute error rates of concurrent Java programs [72]. Each trace directive in our study data was given an error score on the scale [0,1] with the lower bound (i.e. zero) indicating a correct trace directive and the upper bound (i.e. one) indicating a fully erroneous trace directive. Partial points were given based on the error severity as follows:

- **High (0.75)**: High-severity errors are incomplete or largely invalid trace directives. For instance, in the circumstance where the required trace directive should have a condition and a record clause, it would be judged as a high-severity error if the provided trace directive was lacking both sub directives. A total of (0.75) point is given to trace directives at this level.
- **Medium (0.5)**: Medium severity errors cover partially valid trace directives that need further refinement to accomplish the tracing task. Half a point (0.5) was given to this category. An example is a trace directive with the appropriate traced model constructs but lacking a constraint postfix or vice versa where a constraint is right but the traced model construct is invalid.
- **Low (0.25)**: Low severity errors cover simple grammatical errors that users make such as spelling mistakes in MOTL keywords and model construct names. Another type of low severity error is the use of parentheses instead of square brackets. A quarter of a point is given to any trace directive that has errors in this category.

In addition to the errors mentioned above, we encountered situations that were not counted as errors:

- **Semicolons**: Each trace directive must be ended with a semicolon, and the absence of a semicolon will produce a syntax error. We encountered a single participant who answered all trace tasks with trace directives missing a semicolon. We decided to include his answers and ignore the absence of semicolons in his responses. The rationale for that is that he might have assumed the question was asking for the core content of the directive prior to the semicolon.

- **Alternative solutions**: When we designed the study, we wrote trace directives to accomplish every tracing task in each model. However, in certain questions, we received responses containing trace directives that would accomplish the stated tracing objective in a different way. For example, in question 4 in model 3, we stated that the tracing task can be accomplished when we trace entry into state ReservationCanceled; however, we received responses that trace any triggering of event cancel, which is correct too.
- **Multiple directives rather than a single directive**: There are tracing tasks that can be answered in a single trace directive, however, some provided multiple directives that is equivalent to a single directive. For example, in model 1, question 2, the trace task can be answered in a trace directive ("trace name, policyNumber;"), but we encountered participants who answered this question in two trace directives ("trace name; trace policyNumber;"). Both answers are correct both syntactically and semantically. Thus, any responses that provide multiple directives that are correct as in a single directive will be counted valid.

### 7.4.3 Results and Discussion of Error Rate

In this section, we present and discuss the error rate data calculated from participants' responses by following the error rate measurement criteria explained in the previous section. Every trace directive was given a score from zero to one based on the severity of the error. In the optimal situation, where all participants provided correct trace directives as responses to given tracing tasks, the sum of the errors would be zero; because each trace directive would be given a zero score. In the opposite case, in the situation where all participants provided fully erroneous trace directives, the sum of the errors would be equal to 234 in the context of our study (since there are 18 participants * 13 Questions).

Table 7.11 presents the sum of the errors for all study participants in the three models. Participants' answers (i.e. trace directives) for model 1 tasks received a score of (28.75) as sum of errors (out of a maximum possible of 72 error score for the 4 questions). In other words, responses were 60% correct. As for the second model, participants' responses scored an error of 30 (out of a maximum of 90 error score for the 5 questions), representing 77%

correct responses. In last model, responses had the lowest sum of errors, at 19.5, and a 27% error rate.

It was observed that the error rate decreased as participants answered the trace tasks from the first model to the last model. This might be due to the learnability factor where participants became more familiar with MOTL syntax and the presented study tracing tasks.

**Table 7.11: Error rates for three models**

| | ***Sum of Errors*** | ***Error rate*** = (Sum of Error scores) / (#Questions * 18 Participants) |
|---|---|---|
| ***Model 1*** | 28.75 | 40% |
| ***Model 2*** | 30 | 33% |
| ***Model 3*** | 19.5 | 27% |
| ***All Models*** | 78.25 | 33% |

### 7.4.4 Differential Analysis

One of study goals was to learn to what extent software modelers who are not familiar with textual modeling using Umple would be able to perform tracing tasks as compared to modelers familiar with Umple syntax.

For this differential analysis, we employed the following hypothesis.

**H1**: Software modelers who are familiar with Umple syntax will produce less error-prone trace directives than software modelers who are not familiar with Umple.

The corresponding null hypothesis is:

**$H1_0$**: Umple and non-Umple modelers trace directive error rates do not differ.

In the first model, software modelers who were not familiar with Umple produced answers with an overall higher error rate compared to Umple modelers. Figure 7.7 shows the error rate comparison of both groups over first model trace questions. The difference was marginal in both question 1 and 3. However, the difference in question 4 was notable. Errors committed by participants who are not familiar with Umple in response to question 4 were related to the usage of role names to trace an association link and how role names can be used in trace directive constraints.

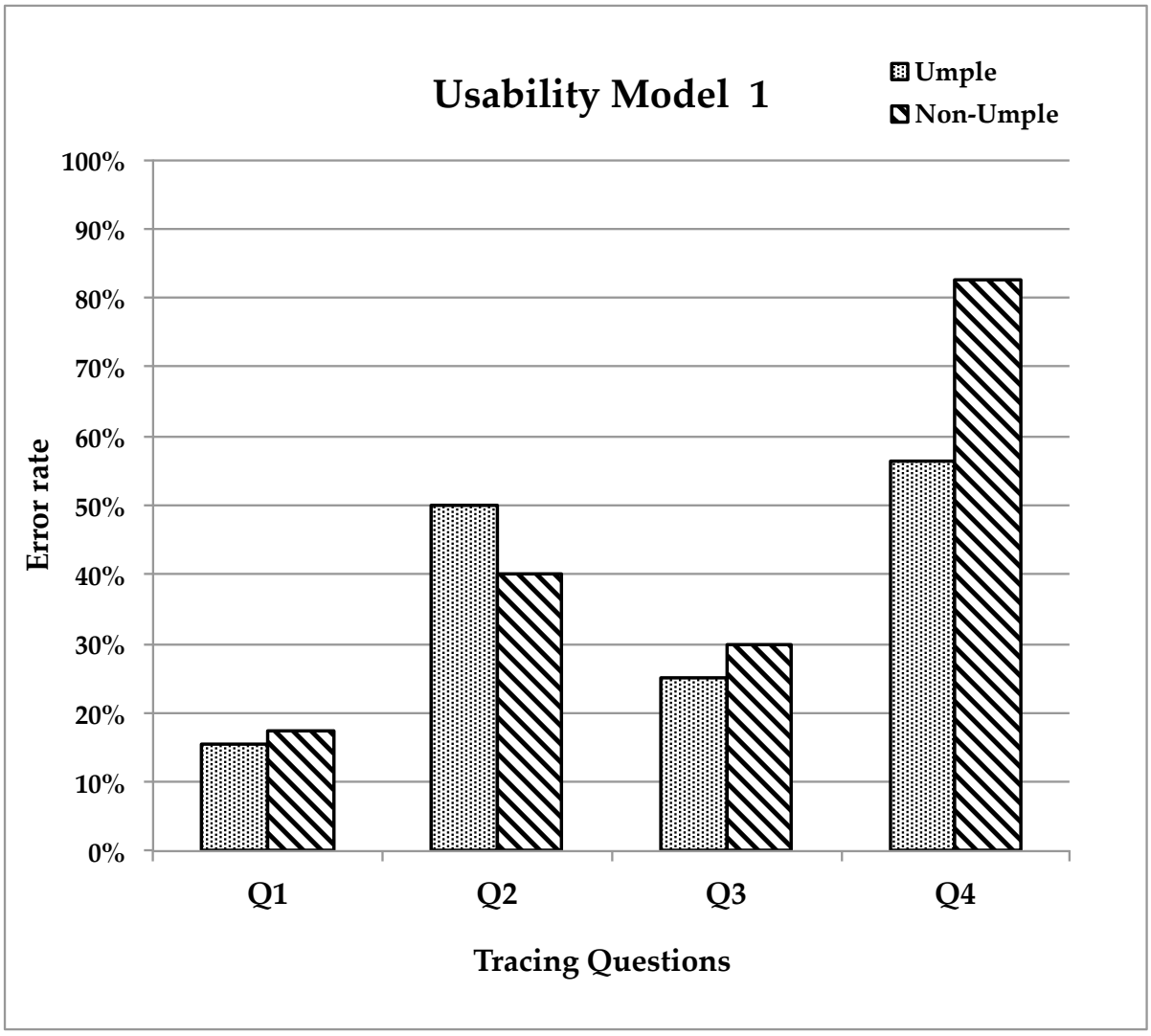


**Figure 7.7: Error rate comparison between the two groups in Model 1**

In the second model, Umple modelers also outperformed the non-Umple modelers. The highest error for both groups was reported in question 6. The common mistake in both groups' responses was the lack of a 'record' clause subdirective to report the value of section id in addition to the traced entity. Thus, most trace directives were incomplete resulting in higher error rates.

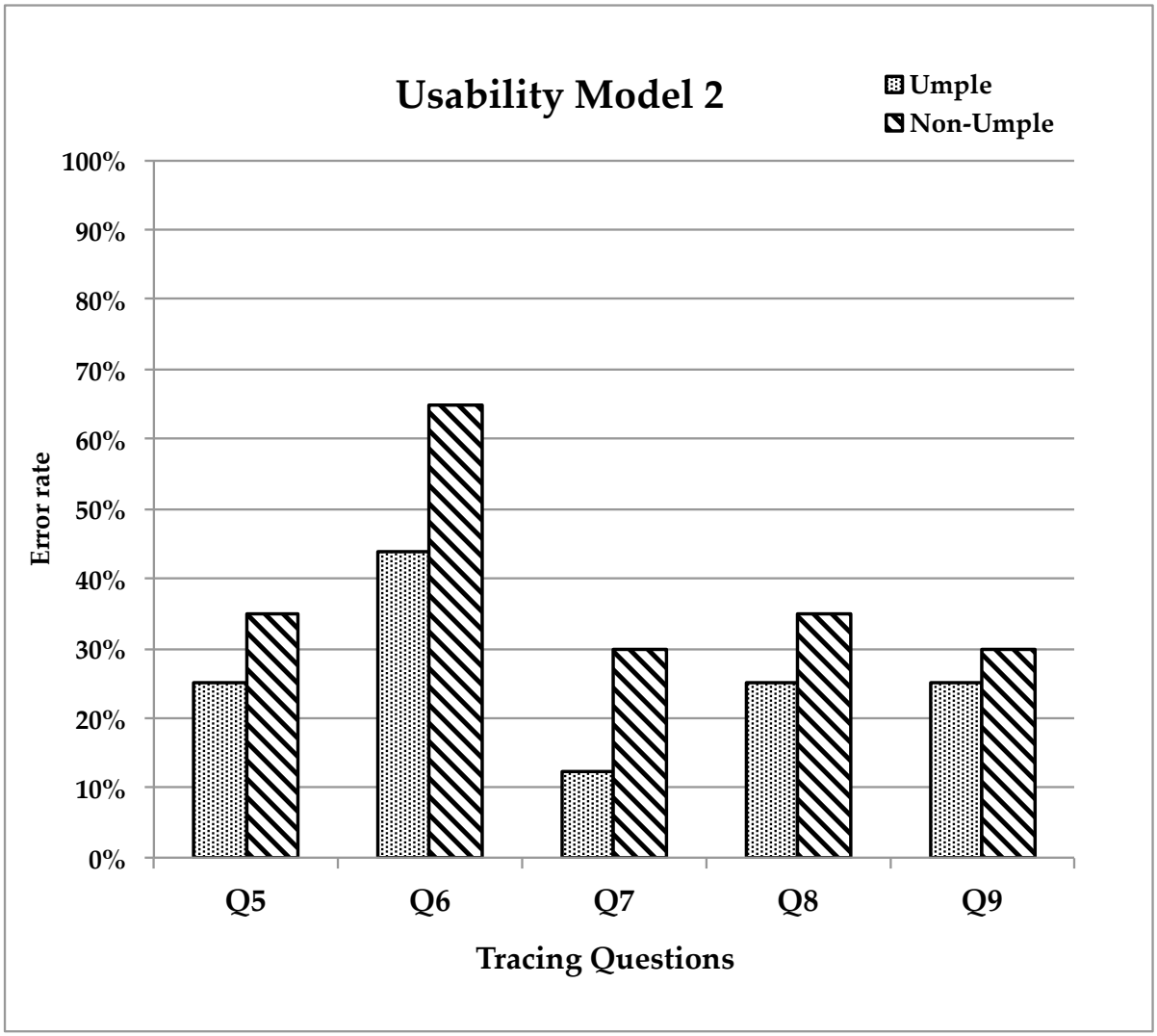


**Figure 7.8: Error rate comparison between the two groups in Model 2**

In the last model, both groups performed equally in terms of error rate or with slight differences. Both groups achieved a zero error rate in question 10, which requires the tracing of an association link. The highest error rate for both groups was reported in question 11, due to the complexity of the trace directive required to answer this question.

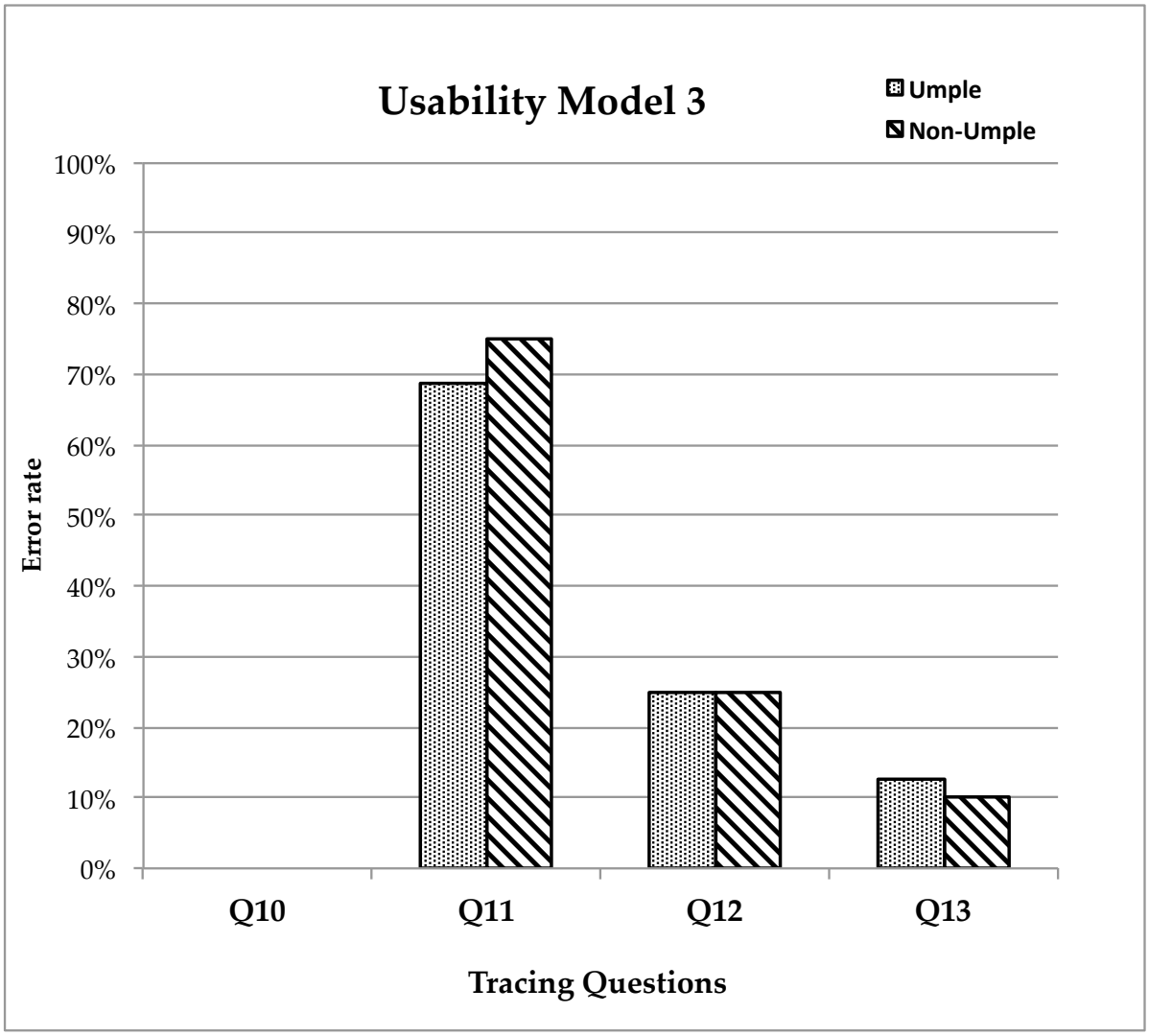


**Figure 7.9: Error rate comparison between the two groups in Model 3**

Table 7.12 presents the two groups' error rate on all the models. Overall, Umple modelers had a lower error rate compared to non-Umple modelers. In Model 1 and 2, non-Umple modelers had respectively (6%) and (13%) error rate difference over Umple modelers, while the difference was minimal in model 3 with (1%). Combined error rates for all study models indicate an overall (7%) error rate difference for non-Umple modelers over Umple modelers.

**Table 7.12: Error rate between the two groups**

| | **Umple** | **Non-Umple** | **Difference** |
|---|---|---|---|
| ***Model 1*** | 37% | 43% | +6% |
| ***Model 2*** | 26% | 39% | +13% |
| ***Model 3*** | 27% | 28% | +1% |
| ***All Models*** | 30% | 37% | +7% |

We present Boxplots to visualize the error rate distribution for the two groups in Figure 7.10. The Umple group participants had a lower error rate median of (0.25) than non-Umple group median (0.3), and were equal to the first quartile of non-Umple box. There were many outliers in the non-Umple error rate distribution.

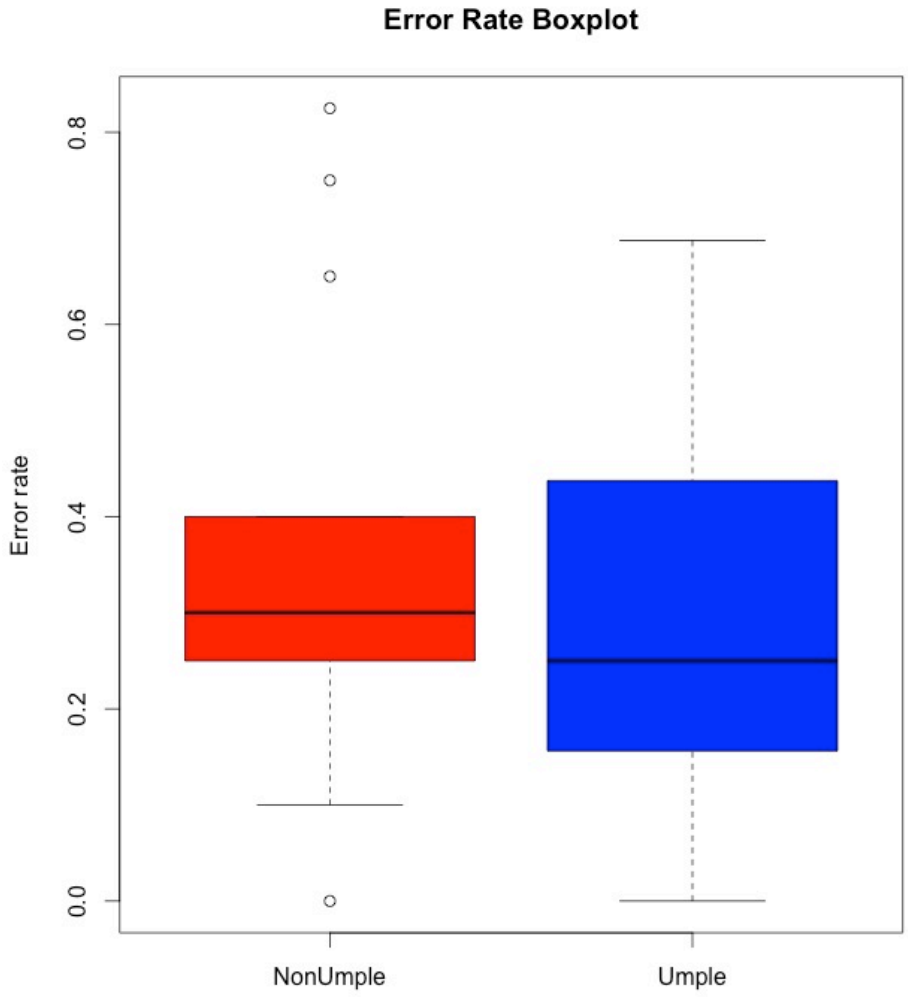


**Figure 7.10: Error rate boxplot for Umple and Non-Umple groups**

To decide whether the error difference between the two groups (Umple and non-Umple) was significant or not, we employed the non-parametric Mann-Whitney U test due to the sample size, and lack of normal distribution. Based on the sample size, the critical value of U is (17). Results obtained from running the test shows that at ($\alpha$ =0.05), the U value = 104 and computed p-value = 0.3264. Thus, the error rate difference was not found to be statistically significant and we accept the null hypothesis. Nonetheless, we have reported the data, as is considered best practice, as suppression of null results in science can cause distortions to the scientific process [73, 74].

## 7.5. Satisfaction Questionnaire

A series of survey questions was created to measure participants' subjective satisfaction with MOTL syntax and purpose. Questions cover categories such as the extent to which they like the syntax, and the ease of language usage. The questionnaire consisted of a total of 12 questions, with eight questions answered on a five-point Likert scale (Strongly agree, Agree, Neutral, Disagree, Strongly disagree), and four open-ended questions. Table 7.13 lists the usability study survey questions.

**Table 7.13: MOTL satisfaction questionnaire**

| Questions type | Number | Question |
|---|---|---|
| ***Likert scale*** | Q1-8 | See Table 7.14 |
| ***Open ended*** | 9 | Please indicate what you like about the Umple (MOTL) tracing syntax |
| | 10 | Please indicate what you dislike about the tracing syntax |
| | 11 | Please provide any suggestions to improve the MOTL syntax |
| | 12 | Please provide us with any feedback about this study itself, so we may improve it |

### 7.5.1 Responses and Discussion for the Satisfaction Questionnaire

At the end of the study participation, the 18 participants answered the satisfaction questionnaire discussed in the last subsection.

The first eight questions, as shown in Table 7.14, enquired about MOTL capabilities and asked participants to rate their opinions on a five-point Likert scale. The scale allowed participants to provide positive feedback (strongly agree and agree), negative feedback (strongly disagree and disagree), or a neutral position.

Table 7.14 presents the response percentages for each question. The first two questions aimed at measuring participants' satisfaction with MOTL syntax learnability and memorability. Most of study participants (89%) found MOTL syntax was easy to learn, and over two thirds (70%) found MOTL syntax was easy to remember.

Questions 3 to 7 were created to measure participants' satisfaction with different MOTL capabilities that they were introduced to participants in this study. Most of the partic-

ipants (89%) indicated that they perceived tracing attributes specification using MOTL to have a clear syntax, and highly agreed (84%) that attributes tracing should have the possibility to be limited to setters and/or getters. Over (75%) of study participants showed agreement that MOTL syntax is expressive enough to trace any state machine element. However, (11%) disagreed with this statement. Over half of study participants (55%) shown a strong agreement that the MOTL constraints capabilities allow them to limit the scope of desired tracing in the manner they would want.

The last question (question 8) answered on the Likert scale was a general-purpose question to measure satisfaction with MOTL's ease of use. Almost all of participants (94%) provided a positive feedback regarding this.

**Table 7.14: Satisfaction questionnaire Likert scale percentages**

| Questions | Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree |
|---|---|---|---|---|---|
| 1. MOTL syntax was easy to learn | 33.3% | 55.6% | 11.1% | 0.0% | 0% |
| 2. MOTL syntax was easy to remember | 33.3% | 38.9% | 27.8% | 0.0% | 0% |
| 3. MOTL attributes tracing syntax was clear | 38.9% | 50.0% | 5.6% | 5.6% | 0% |
| 4. MOTL capability of limiting attributes tracing to setters and getters is important | 38.9% | 44.4% | 16.7% | 0.0% | 0% |
| 5. MOTL syntax allows me to trace any state machine element (e.g. events, substates) | 33.3% | 44.4% | 11.1% | 11.1% | 0% |
| 6. MOTL made associations tracing intuitive and easy to accomplish | 33.3% | 61.1% | 5.6% | 0.0% | 0% |
| 7. MOTL constraints allows me to limit tracing as I desire | 55.6% | 38.9% | 0.0% | 5.6% | 0% |
| 8. MOTL is an easy to use language to specify tracing at the model level | 33.3% | 61.1% | 5.6% | 0.0% | 0% |

Figure 7.11 adds another view of previous data showing the count for each Likert scale response. For example, it can be seen that in the first question, 16 participants agreed or strongly agreed that MOTL syntax was easy to learn, while two participants were neutral. There is a slight negative feedback was received as in question 5. Two participants showed their disagreement with the statement that states MOTL allows them to trace any state machine element.

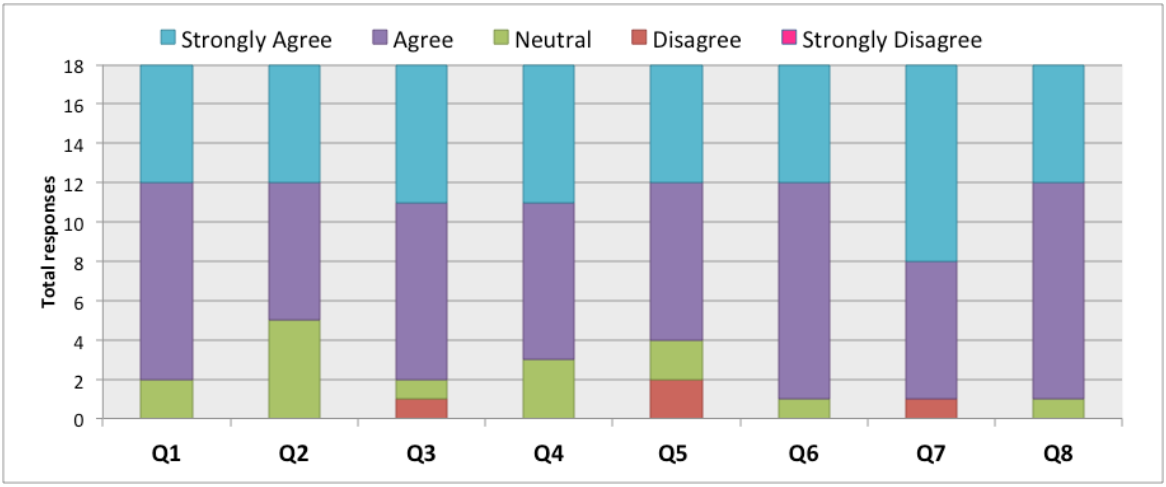


**Figure 7.11: Participants answers distribution over satisfaction questionnaire**

Level of agreement is another way of viewing the above data. It is a summarized measurement that gives a median score for each subjective question. It is derived from the same Likert scale data, ranging from strongly disagree (1) to strongly agree (5). Table 7.15 presents participants level of agreement for each question. High level of agreement was obtained for each question, which indicates the overall positive feedback for MOTL syntax.

**Table 7.15: Participants level of agreement results**

| Question | Level of agreement |
|---|---|
| Q1 | 4.2 |
| Q2 | 4.1 |
| Q3 | 4.2 |
| Q4 | 4.2 |
| Q5 | 4.0 |
| Q6 | 4.3 |
| Q7 | 4.4 |
| Q8 | 4.3 |

Finally in this section, we will discuss the answers received from the study participants in response to our open-ended questions. The first open-ended question asked participants to state what they liked in MOTL syntax. We noticed a repeated word "intuitive" in many of the responses to describe MOTL syntax that made it easy to learn and use. Sample of the answers are the following:

*Feedback #1*

*"Its intuitiveness. Its syntax resembles most normally used coding and scripting languages, so it is intuitive and easy to learn"*

*Feedback #2*

*"It seems intuitive. I like the way prefix and postfix operators work."*

*Feedback #3*

*"Intuitive, easy to use and learn with great outcome. The time saved by writing less code to trace UML elements is significant."*

In addition, participants indicated that MOTL syntax was easy to grasp if you had previous programming experience, and moreover, syntax structure was a natural translation to what modeler would like to achieve. Participants' responses included the following:

*Feedback #4*

*"I think the syntax is pretty easy to pick up, especially if you're used to pretty much any other programming language (Java, Python, etc.)."*

*Feedback #5*

*"It feels like a fairly natural language representation of what I'd like to do."*

In the side of MOTL capabilities, limiting the scope of tracing using constraints was liked by the study participants, in addition to the ability to trace different model constructs. Participants' responses were the following:

*Feedback #6*

*"I liked the flexibility of the tool in allowing me to trace attributes based on conditions"*

*Feedback #7*

*"* Easy to learn and easy to embed to code. * Capability to trace associations and state machine elements. * Easy to add constraint to limit the tracing condition."*

One participant showed support for MOTL's purpose and indicated a resemblance between MOTL syntax and SQL queries:

*Feedback #8*

*"I like that it uses SQL similar syntax for queries at the model level which opens new applications for debugging and profiling software code."*

The second open-ended question asked participants to state what they disliked in MOTL syntax. There is an observed confusion based on participants responses related to the condition types, and the keywords selected for pre/post conditions. Participants' responses were the following:

*Feedback #9*

*"Basically almost everything is good, except as just a personal preference, I liked having instead of giving much better."*

*Feedback #10*

*"I'm still not really sure of the difference between "where" and "giving" (though I think I might get it?)"*

*Feedback #11*

*" 'giving' for post conditions sounds weird."*

Based on this feedback, a further analysis ought to be conducted to test whether using the 'giving' keyword is better than other possible keywords alternatives such as 'having' or 'postcondition'. Such a study is deferred to future work.

The third open-ended question asked participants to indicate whether they have any suggestions to improve MOTL syntax. An informative suggestion in form of questions was given by one of the study participants:

*Suggestion #1*

*"One question about separation of concerns: Do we want to pollute the Umple code with tracing information, or shouldn't this be kept separate? Can I tell the model that I want to trace all method entries/exit easily, without doing it explicitly for all methods?"*

Using mixin capability, MOTL directives *can* indeed be written in an independent file from the model code, enabling separation of concerns. For the second part of the suggestion, indeed MOTL allows all model attributes or native code methods to be traced in a single directive as shown in 4.7. The tutorial session and user manual do not cover these aspects. This suggestion therefore will be used to update the documentation.

The final open-ended question asked participants to provide feedback to improve the empirical study itself. Most feedback concentrated on improving the tutorial session by making the video longer and improving the user manual examples. Participants' feedback included the following:

*Feedback #1*

*"The study is well-designed. The longer video would have been better so it was not rushed. Other than that, it was intuitive and easy to do."*

*Feedback #2*

*"Add to the video a sample run and resulting output of using trace in action."*

*Feedback #3*

*"Update usermanual with more examples syntax highlighting for given examples needs to be revised."*

*Feedback #4*

*"The video was short but too dense/quick I think. I wish I had an easy way to view it again."*

*Feedback #5*

*"More examples would be beneficial in improving the learning experience especially for someone who tries to learn MOTL for the first time. This can be done by adding a Module example where the user tries to use MOTL and observe the proper way to trace."*

This feedback suggests that future participants should be introduced to a model example and be asked to write traces using MOTL and compare them with the correct answers as part of the tutorial session before commencing the actual study. Doing this would, however, perhaps reduce the value of the raw data gathered about intuitiveness and learnability.

### 7.5.2 Differential Analysis for the Questionnaire Data

The previous section discussed and analyzed the subjective satisfaction of all of our participants. In this section, we will investigate whether there is a significant difference in subjective satisfaction towards MOTL between software modelers who are familiar with Umple as compared to modelers not familiar with Umple.

For this differential analysis, we employed the hypotheses grouped in Table 7.16.

**Table 7.16: Satisfaction hypotheses**

| ***Hypothesis 1*** |
|---|
| **H1**: Software modelers who are familiar with Umple syntax found MOTL syntax easy to learn than software modelers who are not familiar with Umple.<br><br>**$H1_0$**: *Both of Umple and non-Umple modelers found MOTL syntax easy to learn.* |
| ***Hypothesis 2*** |
| **H2**: Software modelers who are familiar with Umple syntax found MOTL syntax easier to remember than software modelers who are not familiar with Umple.<br><br>**$H2_0$**: *Both of Umple and non-Umple modelers found MOTL syntax easy to remember.* |
| ***Hypothesis 3*** |
| **H3**: Software modelers who are familiar with Umple syntax found MOTL attributes tracing syntax more clear than software modelers who are not familiar with Umple.<br><br>**$H3_0$**: *Both of Umple and non-Umple modelers found MOTL attributes tracing syntax clear.* |
| ***Hypothesis 4*** |
| **H4**: Software modelers who are familiar with Umple syntax beliefs of the importance of limiting attributes tracing to setters and getters more than software modelers who are not familiar with Umple.<br><br>**$H4_0$**: *Both of Umple and non-Umple modelers believes in the importance of limiting attributes tracing to setters and getters.* |
| ***Hypothesis 5*** |
| **H5**: MOTL syntax allowed Umple modelers to trace any state machine element more than modelers who are not familiar with Umple.<br><br>**$H5_0$**: *MOTL allowed Umple and non-Umple modelers to trace any state machine element*. |
| ***Hypothesis 6*** |
| **H6**: Software modelers who are familiar with Umple syntax found MOTL association tracing intuitive compared to software modelers who are not familiar with Umple.<br><br>**$H6_0$**: *Umple and non-Umple modelers found MOTL association tracing intuitive.* |
| ***Hypothesis 7*** |
| **H7**: Software modelers who are familiar with Umple syntax found MOTL syntax easy to learn than software modelers who are not familiar with Umple.<br><br>**$H7_0$**: *Umple and non-Umple modelers found MOTL constraints limits tracing, as they desire.* |
| ***Hypothesis 8*** |
| **H8**: Umple modelers found MOTL an easy to use language to specify tracing at the model level compared to modelers who are not familiar with Umple.<br><br>**$H8_0$**: *Umple and non-Umple modelers found MOTL an easy to use language to specify tracing at the model level.* |

The nonparametric Mann-Whitney U test was used to investigate whether there is a significant difference in opinions between Umple and non-Umple modelers towards MOTL. Table 7.17 shows results obtained from running Mann-Whitney U test to test the difference between Umple and non-Umple groups.

**Table 7.17: Results obtained from Mann-Whitney U test**

| *Hypothesis* | *U* | *p-value* |
|---|---|---|
| 1 | 54 | 0.1779 |
| 2 | 49 | 0.4222 |
| 3 | 50 | 0.3508 |
| 4 | 31.5 | 0.4411 |
| 5 | 60 | 0.06401 |
| 6 | 49 | 0.3789 |
| 7 | 54.5 | 0.1568 |
| 8 | 43 | 0.7958 |

Based on the sample size, the critical value of U is (17) and it is used to decide whether the observed U supports the null hypothesis or not. If the observed value of U is less than or equal to the critical value, we reject the null hypothesis and accept the alternative hypothesis, however, if the observed value of U surpasses the critical value we do not reject the null hypothesis. Results reported in Table 7.17 indicate that there is no statistically significant difference at ($\alpha$ =0.05) between the responses received from the two groups. As a result we do not reject the null hypotheses previously formulated in this section.

## 7.6. Threats to Validity

Validity threats exist in most software engineering empirical studies, and they need to be reported and discussed [75]. There are threats of validity noted while conducting MOTL usability study. In this section, we present those threats and discuss them:

i. **Number of participants**: Sample size is an important factor that affects the significance level of empirical study findings. There is an ongoing debate among usability researchers on recommended study sizes [76–78]. J. Nielsen is a strong advocate of the having five users participate to discover 80% of an interface usability defects [78, 79]. In another article [80], J. Nielsen recommended the recruitment of 20 users as a

sample size for usability quantitative studies. Based on his analysis, collecting usability metrics, such as error rate and subjective satisfaction, from 20 participants will return a reasonably tight confidence interval.

ii. **Ambiguous tracing tasks:** Tracing questions presented after each model, where created by the principal investigator. Participants may find some of the created questions vague and difficult to answer, however, an answer is required from participants for each question, which might lead to a higher error rate due to questions ambiguity rather than an error in a trace directive. To overcome this threat, after each model, an optional question was presented to participants asking them to report any ambiguous tracing question. Due to the fact this question was optional, we received a low amount of feedback and all but one were related to how they answered the questions, and were unrelated to the question ambiguity. A single response from one participant was received regarding the questions wording in Model 2.

*Feedback on Model 2 questions ambiguity*

*"I don't think I understand the differences between 'capturing,' 'checking,' and 'tracking.' I'm not sure if there is a difference at all".*

iii. **Syntax coverage:** We do not claim that all tracing tasks given in each model cover the whole MOTL syntax. Therefore, our findings obtained from this study are limited to the aspects of MOTL syntax we focused on.

iv. **Efficiency:** MOTL usability can be assessed also by measuring the efficiency metric that reports how quickly (in terms of time units) participants were able to write trace directives to perform tracing tasks. However, we decided not to include the efficiency metric in our analysis and data gathering. Individuals vary in keyboard typing speed and time taken to read and understand the tracing tasks, resulting in individuals who are fast, while others slow. To obtain reliable results, time measurement would have needed to be averaged over a considerably larger sample size [80, 81].

v. **Learning factor:** Study models were shown to participants in sequential order rather than in a randomized order. Our motivation for this was to present models in an increasing order according to model complexity. As a consequence, this might introduce a learning factor in participants' answers as they do the experiment and move

from Model 1 to Model 3. Comparisons between the error rates of the three models are therefore not useful.

## 7.7. Related Work Regarding Empirical Studies of Modeling languages

Syntax usability and language comprehension evaluation studies have been conducted on programming languages by numerous researchers. In the area of concurrent programming languages, Nanz et al. [72] conducted a study to compare the usability of two concurrent programming languages (multithreaded Java and SCOOP) in terms of program comprehension, debugging existing programs, and writing correct new programs. Their usability metrics defined levels for error severity and outlined a marking scheme for evaluating participants programs.

In the area of MDD, the increasing number of modeling language tools have introduced different syntactical representations of UML models. Usability empirical evaluations became a necessity to improve these languages. In case of Umple, Lethbridge et al. [36] studied how Umple would effect student performance on grasping UML concepts if used in student laboratory exercises and assignments. They conducted a student satisfaction survey to measure student opinion on different Umple qualities including the availability of a textual notation to create UML models. Students' responses indicated their support for the textual notation capability of Umple. In another empirical study, Badreddin et al. [11] tested Umple's textual notation (i.e. syntax) against the same systems described using UML diagrams and Java code. Participants were presented with each of the three notations and were asked to answer a list of questions reflecting their level of comprehension. Several different systems were studied, and the systems and notations were rotated to avoid the learning effect. Empirical evidence showed that Umple performed significantly better in comprehensibility than Java and neither exceeded nor was worse than UML diagrams in terms of comprehensibility.

Following the motivation of prior studies on Umple, the empirical study presented in this chapter continues the effort to evaluate aspects of Umple, since MOTL was implemented as an extension of Umple.

## 7.8. Summary

In this chapter, we performed an empirical evaluation of MOTL by inviting a group of participants to perform tracing tasks at the model level using MOTL. The study aimed at evaluating MOTL usability by assessing the error rate for participants when completing tracing tasks and measuring their subjective satisfaction through survey questions. Each participant completed a tutorial session and then was presented with three models in both textual and visual notations, and then answered a list of tracing questions on each presented model. In the end, a questionnaire was presented and answered by each participant.

Results indicate a relatively low error rate among all participants. In terms of subjective satisfaction, the data indicate a generally high level of satisfaction towards MOTL. We further divided our participant sample into two groups: Umple modelers and modelers not previously familiar with Umple notation, and conducted a differential analysis to test whether there is a significant difference between the two groups in error rate and subjective satisfaction. We concluded that there is no significant difference in error rate between the Umple and non-Umple modelers and that there are no significant subjective satisfaction differences either.

# Chapter 8. Related work

We discussed various tracing technologies in Chapter 3. In section 1.6 we also pointed out various work that uses similar terminology, but is in fact quite distinct from ours. This chapter compares our work to the few other research projects that actually focus on generating traces from UML models, as we do.

## 8.1. Work on Object-Oriented System Tracing

The following two projects do not involve tracing of true models, but nonetheless are interesting because they show how researchers were thinking about tracing of entities other than methods.

### 8.1.1 The Work of Lange and Nakamura Including Program explorer

Lange and Nakamura [82] investigated and explored the tracing of object-oriented program execution. They identified important information related to objects that must be captured to trace OO programs. Information about object lifetime is important (i.e. object creation and object destruction) and identifying the object associated with each method invocation must also be traced. They investigated several approaches to accomplish this task:

- Program instrumentation: trace code is injected into source code, which requires ad-additional compilation.
- Metaclass programming in IBM's System Object Model (SOM): traces are created through the extension of SOM metaclasses, which allow users to capture method invocations, and gather static information using the SOM repository.
- **Program explorer** [83]: This uses debugging technology to monitor the execution of C++ programs. General structure of the tool is shown in Figure 8.1. IBM's HeapView Debugger allows the monitoring of objects created and destroyed on the heap. Trace generation can be limited based on selected tracepoints set at the trace selection GUI.

The main drawback of this approach is that it spends a large amount of time in context switching between the target program's process and Program Explorer's process.

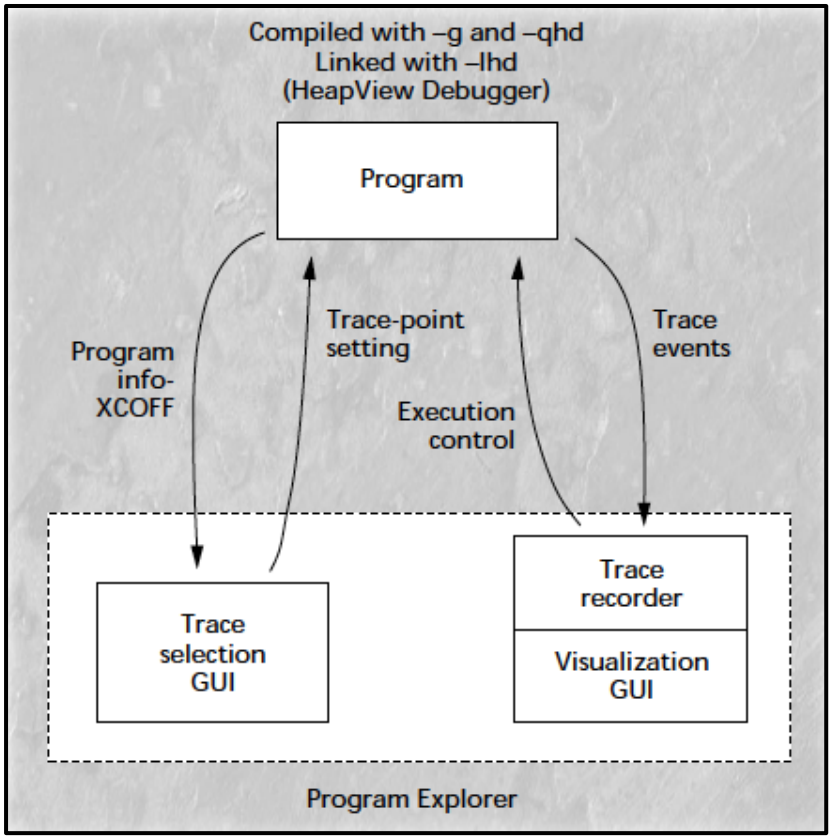


**Figure 8.1: Program Explorer architecture** [83]

### 8.1.2 JaVis

K. Mehner [84] developed the JaVis environment for visualizing and debugging concurrent and sequential Java programs. Their motivation is that debugging of concurrent Java programs is complex due to usage of threads. The JaVis environment consists of three stages: collecting traces while executing, visualizing these traces, and performs thread deadlock detection and analysis.

The tracing component of JaVis uses the Java Debug Interface (JDI) of the Java Platform Debugger Architecture to allow the collection of debugging and tracing information from a running Java program. Traces are represented in a textual format with each trace entry consisting of a single line. Each trace line contains a method entry, a method exit, object IDs, calling thread, and other information that can be used for deadlock detection. Usage of JDI

provides advantages when used for tracing such as it allows tracing of remote and already running Java programs and does not require the source code to be modified. After the generation of traces, they are visualized using UML 1.6 sequence and collaboration diagrams. These diagrams will show dependencies between different program threads to help detect deadlocks.

## 8.2. ObjecTime

Model level debugging was introduced in early research pre-dating UML. ObjecTime [85] was a software development tool introduced in the 1990's to help developers specify and create real-time concurrent applications from visual designs. Finite state machines (FSM) and message sequence chart (MSC) notations were used to specify the model behavior. Action code that corresponds to FSM transitions was generated in C/C++, and then a test harness was created to run executable application and collect execution messages. Finally, these execution messages were compared to the specified MSC to locate mismatches in the MSC specification.

## 8.3. Li's work into Instrumenting State Machines

Li et al. [86] implemented a prototype tool that does runtime verification of Java programs for UML state machines with the focus on the temporal order of the messages received by objects, as reflected in the state machine diagrams (i.e. consistency checking). They conduct this verification process in the following steps: Firstly, Java source code is hand-written according to state machine specifications. This is then manually instrumented to gather execution traces. Next, test cases are derived to execute instrumented Java programs and produce execution traces. These execution traces represent messages between objects gathered when events are triggered. Finally, the collected execution traces are checked to determine whether they are consistent with the behavior of the original state machine. The main limitation of this approach is that model-driven development does not occur: code is not generated from the state machine. Another limitation mentioned by the authors is that their tool does not cover composite state machines, actions, and other advanced features of state machines.

## 8.4. fUML Tracing

Mayerhofer et al. [87, 88] proposed extensions to the standardized fUML virtual machine to enable the debugging of models at run time. These extensions aim to overcome the limitations of fUML in monitoring the models' runtime behaviour. Three models were proposed:

- Trace model: a dedicated trace metamodel capable of recording the model execution carried out by the fUML virtual machine.
- Event model: monitors run time state and triggers events based on changes to run time state.
- Command API: a set of commands that enables the control of model execution.

No code generation is performed in this work since the fUML virtual machine was used to interpret the models directly. The trace model was illustrated using only a UML activity diagram example. Execution traces showed only the input and output relationship between activity nodes, and the order of activity node execution.

## 8.5. Eakman's Approach

Eakman [89] explored and discussed the idea of instrumenting UML models for debugging purposes. He pointed out that systems are more visible at the model level than at the code level, where it's difficult to visualize systems due to implementation details. The main objective of this idea is that model level instrumentation will provide full access to the system under test at the level of UML modeling, allowing a glass-box approach to testing with greater observability, controllability, and testability. Their proposed instrumentation will occur at the translational process where UML models are mapped to the implementation (i.e. targeted programming language) by the model compiler. Hence, the model compiler will be responsible for the insertion of instrumentation into generated code and ensuring that instrumentation will not add any additional functionality to the software, other than enhanced testability. This is similar with what we discussed in this thesis.

In Eakman's proposed instrumentation approach, important data values, attributes, inputs, and control points must be identified and appropriate instrumentation added. Instrumentation can be triggered based on data access and/or dynamic behavior. In data access situations, attribute values can be monitored and state machines' response to an event can be

recorded, while in dynamic behavior, the creation and deletion of instances and/or associations are monitored. In addition, Eakman proposed a general architecture for UML instrumentation. This architecture consists of two components:

1. Dynamic verification user interface (DVUI) that is responsible for displaying information to the user and accepting user commands.
2. Instrumentation agent providing an interface between the application and the DVUI.

The DVUI can subscribe to different incidents that might occur during the execution of the program. If any subscribed incident is triggered, a trace will be generated and passed to the DVUI by the instrumentation agent.

## 8.6. Modeling tools

In this section, we discuss tools that perform tracepoint injection into code generated from models.

### 8.6.1 FXU Tracer

Derezinska and Szczykulski [90] presented a framework for executable UML, called FXU, that performs transformation from a UML class and state machine model into a C# implementation. Their framework consists of two main components: a code generator that assumes direct model transformation to the target code, and a runtime library that contains realization of different UML meta model elements. Usage of FXU involves many steps. First, a UML model file is input to the FXU generator. Then, this model is transformed into the corresponding C# implementation code. After that, a Microsoft Visual Studio project is created with generated C# code and then compiled and linked with a runtime library. Finally, generated code is run and information about its execution is stored in a log file.

As an extension to this framework, FXU tracer [53] was designed to allow the tracing of state machine execution generated in C# code. Tracing will be specified and commence after the state machine execution using log files created in the FXU environment. The FXU tracer is intended to help increase state machine comprehension and verify state machines behavioural correctness. The FXU tracer has two main panels, as shown in Figure 8.2: the left panel shows the input UML model hierarchy and the right panel shows information related to state machine tracing indicating entry or exit of a state. The FXU tracer requires two

input files: a UML model and a log file generated from the execution of the state machines. After these files are fed to the tracer, tracing can start by following the nodes in the visualized tree or reading the textual information shown in the right panel. Moreover, tracing can be stopped when it reaches a certain node in the model tree by inserting a breakpoint.

Many disadvantages and drawbacks can be seen from this tool design:

i. Specification of traces can only be done after the execution of the application and not before application execution.
ii. The FXU tracer can only use trace logs produced by FXU environment.
iii. There is no control of information collected during state machine execution, which results in collecting irrelevant information and producing massive files.
iv. The authors indicate that not all events can be traced since the FXU environment does not log all events that occur during state machines execution.
v. Tracing of state machines cannot be specified in terms of other UML constructs.
vi. This tool is limited to C#

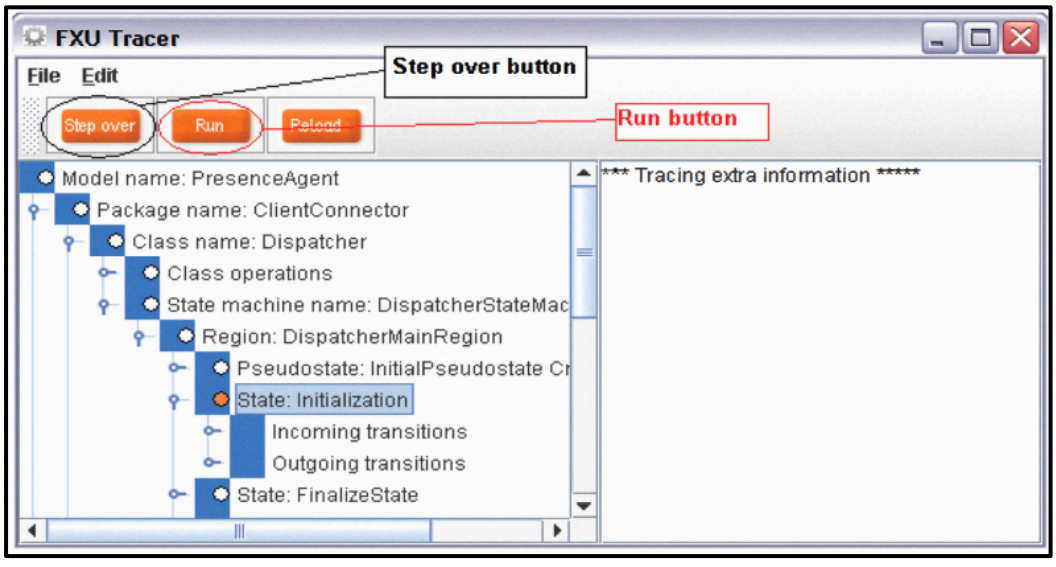


**Figure 8.2: FXU tracer** [53]

### 8.6.2 StateForge

StateForge [54] is a tool that transforms state machine models expressed in XML into C, C++, and Java source code. StateForge includes some observer classes that observe and record state machine behavior. Two types of observers are provided ObserverConsole or Ob-

serverTrace. In the case of Java code generated from state machine specification, ObserverConsole will use *System.out* to direct the trace output, and Java logging API in case of ObserverTrace. In addition to the previous two implemented observers, additional user-specified observers can be created by implementing an observer interface.

StateForge observers will trace whenever a state is entered or exited, and when an event is triggered. After, examining the observers' implementation, we noticed that the trace output components reported are as follows:

i. State entry: name of entered state with context (i.e. class name).
ii. State exit: name of exited state with context (i.e. class name).
iii. Transition: name of previous state, next state, and event that caused the transition, along with transition context (i.e. class name).

Issues with this tool are the following: Firstly, the observers are overly simple and do not permit any control over state machine tracing; secondly there is no support for other modeling constructs, since StateForge only models state machines.

### 8.6.3 Papyrus

Papyrus is an open source graphical UML editing tool [55] based on Eclipse. Papyrus also provides support for Domain Specific Languages and SysML. Papyrus is *planning* to add tracing support in its tool as a Model-Based Tracing framework. The MBT framework should allow the addition, deletion, and displaying of static tracepoints on UML model elements. The Papyrus code generator will use tracepoints to instrument the model-generated code with tracing information containing the model element on which the tracepoint has been set. The targeted programming language is C++ and LTTng will be used for tracepoints injection.

The planned UML model elements for Papyrus tracing are:

- Class diagram: tracepoints will be added to all owned properties.
- State machine: tracepoints will be added on all states, transitions, composite states (tracepoints will be added to substates and contained transitions), entry, exit and do-activities.
- Composite structure diagram: tracepoints will be added to all properties and ports.
- Activity diagram: tracepoints will be added to all owned actions and control flows.

- Sequence diagram: Tracepoints will be added on lifelines and messages.

MOTL and Papyrus share a similar objective in terms of trace specification at the model level and injecting tracepoints into code generated from models. However, in MOTL, we have implemented a textual language for tracing, with subdirectives that allow a level of control appears to be richer than what is planned for Papyrus. MOTL is ready to be used and supports a wider variety of tracing technologies.

## 8.7. Discussion

The research and tools, discussed above have been conducted to tackle the problem of specifying traces at the model level. The approaches allow developers to trace models either through executable models, or by executing generated code from models. Each approach has its advantages and disadvantages. However, none of them provided a complete specification of traces at the model level.

Table 8.1 summarizes a few key aspects of the relevant approaches discussed above, where rows represents the comparison aspect and columns for relevant approaches. The first major observation is that none of the tools, except MOTL, provide a full tracing specification for UML modeling constructs, and none provide tracing constraints as have accomplished in MOTL. None of the other tools provide a comprehensive mechanism for detailed trace specification at the model level and none of them have explored tracing interaction possibilities between modeling constructs. Another observation is the lack of support for different tracing technologies that can be employed in code generated from models. In MOTL, we support a list of well-known tracers, and our architecture design allows us to integrate further tracing technologies for modelers. Our final observation is that Umple greatly exceeds the other tools in terms of capabilities for trace specification at the model level.

**Table 8.1: MOTL comparison with existing tools**

| ***Aspects*** | ***FXU tracer*** | ***StateForge*** | ***Papyrus*** | ***MOTL*** |
|---|---|---|---|---|
| *Attributes* | No | No | Idea | **Yes** |
| *State Machines* | **Yes** | **Yes** | Idea | **Yes** |
| *Associations* | No | No | Idea | **Yes** |
| *Trace Constraints* | No | No | No | **Yes** |
| *Code generated* | C# | Java, C++, C# | C++ planned | Java |
| *Tracer technology* | FXU environment log file | Console, Java logging API | LTTng | Console, File, Java logging API, log4j, LTTng |
| *Model specification* | Export from UML CASE | XML | Textual / Visual | Textual |
| *Trace specification* | No specification - All state machine behavior is collected | No specification - All state machine behavior is collected | Visual | Textual |

# Chapter 9. Conclusions and Contributions

In this thesis, we explored how to add model-oriented tracing capabilities in a UML model rendered textually using Model-Oriented Tracing Language (MOTL). Our work is an extension to the Umple language.

## 9.1. Answers to Research Questions

In this section we review the answers to the research questions we posed at the beginning of this thesis in Section 1.3.

**RQ 1. What should the syntax and semantics of a model-oriented tracing language be?**

We addressed this research question in both Chapter 4 and Chapter 5. In particular, Chapter 4 explored the syntax and semantics for tracing modeling constructs: attributes, state machines, and associations. Furthermore, syntax and semantics of tracing constraints were discussed in that chapter. In addition, Chapter 5 explored the syntax and semantics of the 'tracer' directive that allows modelers to choose from a list of supported tracing technologies that we discussed in detail in Chapter 3.

**RQ 2. How can such a language best be implemented?**

MOTL's architecture and implementation details for both of its 'trace' and 'tracer' directives were presented in Chapter 4 and Chapter 5 respectively. Grammar rules and metamodel classes were defined. In Chapter 6, we stated that MOTL was developed in an open source environment and explained how it was developed in an agile manner using both Model-Driven Development and Multi-level Test-Driven Development.

**RQ 3. How to link trace specification at the model level with execution trace?**

In Chapter 5, we outlined the trace output components that will allow the linking of execution traces to model constructs. Examples of execution traces were given in the context of two modeling examples. Tracepoint injection locations were discussed in detail in Chapter 4.

**RQ 4. How usable is the implemented language in practice?**

Chapter 7 address this research question by presenting an empirical study to test the language's usability. Moreover, usages of MOTL by other CRuiSE researchers were explored in Chapter 6.

## 9.2. Summary of Contributions

The key contributions of this thesis are as follows:

1. The core idea of the MOTL language: Tracing at the model level, and particularly tracing of attributes, associations and state machines in an integrated manner;
2. The detailed semantics of MOTL, including mechanisms for dealing with conditions, and the integration with modeling (we expand on this below);
3. The specific syntax of MOTL, which we have tested and found to be usable;
4. The specific implementation of MOTL in the Umple technology. This is ready to use in real systems and is open source;
5. An evaluation showing that MOTL is usable in the context of an empirical study;
6. The practical use of MOTL by others in the same research team.

The MOTL language demonstrates two key properties:

I. **Abstract trace specification**: Using MOTL, modelers do not need to manually inject tracepoints in the generated code, but can inject them into the model, which is the source of the system. Hence, our work provides an abstraction layer over traditional source code tracing. The following are aspects of model tracing that are particularly important contributions:

a. **State machine tracing:** MOTL allows tracing of entire state machines, or limiting the scope of tracing to a substate at any level of nesting, or to concurrent regions. Since concurrent regions and the do-activities embedded in these are a key mechanism for enabling easy-to-manage concurrency, MOTL reports the thread's unique identification as part of the trace output.

b. **Association tracing:** MOTL allows specifying of tracing when links are added and deleted to/from associations; tracing conditions can be specified based on the cardinalities of particular association instances.

II. **Tracer-independence**: The MOTL general architecture was designed to be tracer-independent. Modelers can chose their targeted tracer and will have the choice between different tracer technologies such as Console, File, Java logging frameworks and LTTng. This allows the addition of a level of abstraction in the usage of existing tracers and will contribute to the research area of tracing technologies.

## 9.3. Future Work

With the work presented in this thesis, we foresee a number of different future work directions:

I. **Extensibility**: work in this thesis can be extended in two directions: (a) *Supporting more tracing technologies* and (b) Adding support for more *targeted programming languages*. Currently, Umple supports a wide range of languages such as Java, C++, PhP and Ruby, however, MOTL only supports the tracing of systems generated in Java. MOTL can be extended to support all of Umple's supported languages, with tracing technologies suited for each language. To perform this extension, a targeted programming language should be selected, first and then its supported tracing technologies should be explored. In terms of implementation effort, the MOTL code generator would need to be extended to support tracepoint injection for the selected programming language.

II. **Scalability**: MOTL has been tested with small sized examples and execution traces were generated from a limited set of relatively simple modeling examples. In addition, it has been used internally in the Umple CRuiSE research team. MOTL needs to

be tested further with large-scale systems, to further support the findings of this thesis.

III. **Performance overhead**: An empirical study is suggested to examine the performance overhead introduced by MOTL when used for trace specification at the model level. A system under inspection can be annotated by MOTL trace directives and then the system can be generated and the resulted overhead can be examined.

IV. **Usability Study**: Based on the experience gained from conducting the empirical study, we can draw a set of recommendations that can be considered for future work

   a. **Keyword selection**: We noticed confusion about or failure to easily grasp certain keywords in MOTL like the 'giving' keyword. Thus, we suggest further investigation into these keywords with the possibility of choosing better alternatives. One way to accomplish this objective would be to present a list of keywords to a group of participants in the context of example usages, and ask participants to select the most appropriate keyword for each objective. The most preferred keyword would be selected.

   b. **Replication**: replication of an empirical study raises the confidence and validity in the study results and findings [91]. It can be done internally by our research group members (CRuiSE), or externally by outside researchers [92]. A replication of this study can be accomplished by taking into consideration [92, 93]: increasing participants sample size and incorporating additional models into the study with varying tracing tasks. More participants should be recruited, forming a larger sample size. More models should be designed as part of the study, and they can be further categorized into three complexity levels: low, medium and high. Randomized selection of models in each category should be done. The number of tracing tasks should be increased with answers requiring coverage of more of MOTL syntax. Tutorial sessions and the user manual would also need to be enhanced.

# References


[1] B. Selic, “Models, Software Models and UML,” in *UML for Real SE - 1*, L. Lavagno, G. Martin, and B. Selic, Eds. Springer US, 2003, pp. 1–16.

[2] B. Selic, “Model-Driven Development: Its Essence and Opportunities,” in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’06)*, pp. 313–319.

[3] B. Selic, “The pragmatics of model-driven development,” *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

[4] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[5] B. Cantrill, “Hidden in plain sight,” *Queue*, vol. 4, no. 1, p. 26, Feb. 2006.

[6] B. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *ATEC ’04 Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, p. 2.

[7] D. Toupin, “Using Tracing to Diagnose or Monitor Systems,” *IEEE Softw.*, vol. 28, no. 1, pp. 87–91, 2011.

[8] M. Desnoyers and M. R. Dagenais, “LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer,” in *Linux Foundation Collaboration Summit 2009 (LFCS 2009)*, 2009.

[9] M. A. Garzón, H. Aljamaan, and T. Lethbridge, “Umple : A Framework for Model Driven Development of Object-Oriented Systems,” in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 494–498.

[10] A. Forward, O. Badreddin, T. C. Lethbridge, and J. Solano, “Model-driven rapid prototyping with Umple,” *Softw. Pract. Exp.*, vol. 42, no. 7, pp. 781–797, Jul. 2012.

[11] O. Badreddin, A. Forward, and T. C. Lethbridge, “Model oriented programming: an empirical study of comprehension,” in *CASCON ’12 Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, 2012, pp. 73–86.

[12] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research,” *Journal of Management Information Systems*. Routledge, 08-Dec-2014.

[13] G. K. Olsen and J. Oldevik, “Scenarios of traceability in model to text transformations,” in *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, 2007, pp. 144–156.

[14] V. Guana and E. Stroulia, “ChainTracker, a Model-Transformation Trace Analysis

Tool for Code-Generation Environments," in *7th International Conference on Model Transformation, ICMT 2014*, 2014, vol. 8568, pp. 146–153.

[15] Á. Jiménez, J. M. Vara, V. A. Bollati, and E. Marcos, "MeTAGeM-Trace: Improving trace generation in model transformation by leveraging the role of transformation models," *Sci. Comput. Program.*, vol. 98, pp. 3–27, Feb. 2015.

[16] "Acceleo." [Online]. Available: http://www.eclipse.org/acceleo/. [Accessed: 24-Jul-2015].

[17] H. A. Ramadhan, "Improving the engineering of model tracing based intelligent program diagnosis," *IEE Proc. - Softw. Eng.*, vol. 144, no. 3, p. 149, Jun. 1997.

[18] V. Kodaganallur, R. R. Weitz, and D. Rosenthal, "A Comparison of Model-Tracing and Constraint-Based Intelligent Tutoring Paradigms," *Int. J. Artif. Intell. Educ.*, vol. 15, no. 2, pp. 117–144, Apr. 2005.

[19] H. Aljamaan, T. C. Lethbridge, and M. Garzón, "MOTL: a Textual Language for Trace Specification of State Machines and Associations," in *ACM proceedings of the 25th Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2015, pp. 101–110.

[20] H. Aljamaan, M. Garzón, T. C. Lethbridge, and A. Forward, "UmpleRun: a Dynamic Analysis Tool for Textually Modeled State Machines using Umple," in *Workshop on Executable Modeling at the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2015.

[21] H. Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying Trace Directives for UML Attributes and State Machines," in *2nd International Conference on Model-Driven Engineering and Software Development*, 2014, pp. 79–86.

[22] H. Aljamaan and T. C. Lethbridge, "Towards Tracing at the Model Level," in *19th Working Conference on Reverse Engineering*, 2012, pp. 495–498.

[23] N. Medvidovic, A. Egyed, and D. S. Rosenblum, "Round-trip software engineering using uml: From architecture to design and back," in *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99), Toulouse, France*, 1999, pp. 1–8.

[24] "Umple github." [Online]. Available: https://github.com/umple/Umple.

[25] O. Badreddin, "A manifestation of model-code duality: facilitating the representation of state machines in the umple model-oriented programming language," University of Ottawa, 2012.

[26] "UmpleOnline: Generate Java, C++, PHP, or Ruby code from Umple." [Online]. Available: try.umple.org. [Accessed: 10-Jun-2014].

[27] "Umple Grammar." [Online]. Available: http://cruise.eecs.uottawa.ca/umple/UmpleGrammar.html. [Accessed: 13-Apr-2015].

[28] "Umple Metamodel autogenerated diagram." [Online]. Available: metamodel.umple.org. [Accessed: 17-Apr-2015].

[29] K. Beck, *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[30] A. Tevanlinna, J. Taina, and R. Kauppinen, "Product family testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 2, p. 12, Mar. 2004.

[31] "Umple's Test Suite ." [Online]. Available: http://cruise.eecs.uottawa.ca/qa/. [Accessed: 07-Apr-2015].

[32] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development," in *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, 2006, p. 356.

[33] A. Gupta and P. Jalote, "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 285–294.

[34] M. A. Garzón, T. C. Lethbridge, H. Aljamaan, and O. Badreddin, "Reverse engineering of object-oriented code into Umple using an incremental and rule-based approach," in *24th Annual International Conference on Computer Science and Software Engineering (CASCON 14)*, 2014, pp. 91–105.

[35] T. C. Lethbridge, "Teaching modeling using Umple: Principles for the development of an effective tool," in *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T)*, 2014, pp. 23–28.

[36] T. C. Lethbridge, G. Mussbacher, A. Forward, and O. Badreddin, "Teaching UML using umple: Applying model-oriented programming in the classroom," in *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, 2011, pp. 421–428.

[37] D. Ferrari and M. Liu, "A general-purpose software measurement tool," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 3, no. 4, pp. 94–105, Dec. 1974.

[38] M. Factor, A. Schuster, and K. Shagin, "Instrumentation of standard libraries in object-oriented languages," *ACM SIGPLAN Not.*, vol. 39, no. 10, p. 288, Oct. 2004.

[39] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[40] M. Nick and G. Sevitsky, "LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications," in *ECOOP 2003 – 17th European Conference Object-Oriented Programming*, 2003, vol. 2743, pp. 351–377.

[41] "Java Profiler - JProfiler." [Online]. Available: http://www.ej-technologies.com/products/jprofiler/overview.html. [Accessed: 08-Jul-2015].

[42] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*, 2010, vol. 45, no. 6, p. 187.

[43] S. Horwitz, B. Liblit, and M. Polishchuk, "Better Debugging via Output Tracing and Callstack-Sensitive Slicing," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 7–19, Jan. 2010.

[44] “Java Logging API.” [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/logging/. [Accessed: 11-Nov-2014].

[45] “Apache Log4j2.” [Online]. Available: http://logging.apache.org/log4j/2.x/. [Accessed: 11-Nov-2014].

[46] J. Carnell and H. Rob, “Logging and Debugging,” in *Pro Apache Struts with Ajax*, K. Mittal, Ed. Berkeley, CA: Apress, 2007, pp. 317–357.

[47] “Log4j 2 Architecture.” [Online]. Available: https://logging.apache.org/log4j/2.0/manual/architecture.html. [Accessed: 12-Jul-2015].

[48] “LTTng: an open source tracing framework for Linux.” [Online]. Available: http://lttng.org/. [Accessed: 15-Mar-2015].

[49] R. W. Wisniewski, R. Azimi, M. Desnoyers, M. M. Michael, J. Moreira, D. Shiloach, and Livio Soares, “Experiences Understanding Performance in a Commercial Scale-Out Environment,” in *13th International Euro-Par Conference*, 2007, vol. 4641, pp. 139–149.

[50] A. Spear, M. Levy, and M. Desnoyers, “Using Tracing to Solve the Multicore System Debug Problem,” *Computer (Long. Beach. Calif).*, vol. 45, no. 12, pp. 60–64, Dec. 2012.

[51] “Babeltrace.” [Online]. Available: https://www.efficios.com/babeltrace. [Accessed: 12-Jul-2015].

[52] “Trace Compass.” [Online]. Available: https://projects.eclipse.org/projects/tools.tracecompass. [Accessed: 12-Jul-2015].

[53] A. Derezinska and M. Szczykulski, “Tracing of state machine execution in the model-driven development framework,” in *2nd International Conference on Information Technology (ICIT)*, 2010, pp. 109–112.

[54] “StateForge - State machine generator & state diagram editor.” .

[55] “Papyrus.” [Online]. Available: https://eclipse.org/papyrus/. [Accessed: 28-Jul-2015].

[56] A. Forward, “The Convergence of Modeling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language,” Unviersity of ottawa, 2010.

[57] O. Badreldin, “A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language,” Unviersity of ottawa, 2012.

[58] O. Badreddin, A. Forward, and T. C. Lethbridge, “A test-driven approach for developing software languages,” in *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014*, 2014, pp. 225–234.

[59] G. Rademacher, “Railroad Diagram Generator.” [Online]. Available: http://www.bottlecaps.de/rr/ui.

[60] A. Alghamdi, “Queued and Pooled Semantics for State Machines in the Umple

Model-Oriented Programming Language," University of Ottawa, 2015.

[61] CruiSE, "UmpleRun." [Online]. Available: https://github.com/umple/umple/wiki/umplerun. [Accessed: 20-Jul-2015].

[62] T. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, 2nd ed. McGraw-Hill, Inc., 2004.

[63] M. Aberdour, "Achieving Quality in Open-Source Software," *Software, IEEE*, vol. 24, no. 1, pp. 58–64, 2007.

[64] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou, "Open Source Software Development Should Strive for Even Greater Code Maintainability," *Commun.ACM*, vol. 47, no. 10, pp. 83–87, Oct. 2004.

[65] "UCOSP." [Online]. Available: http://ucosp.ca/. [Accessed: 24-Jul-2015].

[66] T. S. Markus Völter Jorn Bettin, Arno Haase, Simon Helsen, Krzysztof Czarnecki, *Model-Driven Software Development*. Wiley.

[67] "Qualities of Relevant Software Documentation: An Industrial Study." [Online]. Available: https://www.site.uottawa.ca/~tcl/gradtheses/aforward/papers/aforward_icse2003_sub.pdf. [Accessed: 23-Jul-2015].

[68] "MOTL wiki." [Online]. Available: https://github.com/umple/umple/wiki/UmpleArchitectureMOTLTracingSubsystem. [Accessed: 26-Jul-2015].

[69] "Model Oriented Tracing Language (MOTL) user manual." [Online]. Available: http://cruise.eecs.uottawa.ca/umple/ModelOrientedTracingLanguage(MOTL).html. [Accessed: 24-Jul-2015].

[70] "Umplificator - github." [Online]. Available: https://github.com/umple/Umple/tree/master/cruise.umplificator.

[71] "Umple User Manual: Model Oriented Tracing Language (MOTL)." [Online]. Available: manual.umple.org. [Accessed: 12-May-2015].

[72] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer, "Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages," in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 325–334.

[73] M. Peplow, "Social sciences suffer from severe publication bias," *Nature*, Aug. 2014.

[74] A. Franco, N. Malhotra, and G. Simonovits, "Publication bias in the social sciences: Unlocking the file drawer," *Science (80-. ).*, vol. 345, no. 6203, pp. 1502–1505, Aug. 2014.

[75] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[76] W. W. N. Ritch Macefield, Cornovian Close, "How To Specify the Participant Group Size for Usability Studies: A Practitioner's Guide," *J. Usability Stud.*, vol. 5, no. 1, pp.

34–45, 2009.

[77] L. Faulkner, “Beyond the five-user assumption: Benefits of increased sample sizes in usability testing,” *Behav. Res. Methods, Instruments, Comput.*, vol. 35, no. 3, pp. 379–383, Aug. 2003.

[78] J. Nielsen and T. K. Landauer, “A mathematical model of the finding of usability problems,” in *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI ’93*, 1993, pp. 206–213.

[79] J. Nielsen, “Why You Only Need to Test with 5 Users.” [Online]. Available: http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/. [Accessed: 07-Jun-2015].

[80] J. Nielsen, “Quantitative Studies: How Many Users to Test?,” 2006. [Online]. Available: http://www.nngroup.com/articles/quantitative-studies-how-many-users/. [Accessed: 07-Jun-2015].

[81] W. Albert and T. Tullis, *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*, 2nd ed. Morgan Kaufmann, 2013.

[82] D. B. Lange and Y. Nakamura, “Object-oriented program tracing and visualization,” *Computer (Long. Beach. Calif).*, vol. 30, no. 5, pp. 63–70, May 1997.

[83] D. B. Lange and Y. Nakamura, “Program explorer: a program visualizer for C++,” in *Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, 1995, p. 4.

[84] K. Mehner, “JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs,” in *Revised Lectures on Software Visualization, International Seminar*, 2001, pp. 163–175.

[85] A. Lyons, “Developing and debugging Real-Time Software with ObjecTime Developer,” *Real-Time Magazine*, vol. 1, pp. 17–24, 1999.

[86] X. Li, X. Qiu, L. Wang, B. Lei, and W. E. Wong, “UML state machine diagram driven runtime verification of Java programs for message interaction consistency,” in *Proceedings of the 2008 ACM symposium on Applied computing - SAC ’08*, 2008, p. 384.

[87] T. Mayerhofer, P. Langer, and G. Kappel, “A runtime model for fUML,” in *Proceedings of the 7th Workshop on Models@run.time - MRT ’12*, 2012, pp. 53–58.

[88] T. Mayerhofer, “Testing and debugging UML models based on fUML,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1579–1582.

[89] G. Eakman, “Strategies for Debugging Embedded Systems,” *Embed. Syst. Program.*, pp. 139–147, 2000.

[90] A. Derezińska and M. Szczykulski, “Towards C# Application Development Using UML State Machines: A Case Study,” in *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, vol. 151, 2013, pp. 793–803.

[91] N. Juristo and O. S. Gómez, *Replication of software engineering experiments*.

Springer-Verlag, 2012.

[92] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller, *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008.

[93] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in Empirical Software Engineering," *Empir. Softw. Eng.*, vol. 13, no. 2, pp. 211–218, Jan. 2008.