UNIVERSITY OF OSLO
Department of Informatics

Exploration of UML State Machine implementations in Java

Master thesis

Morten Olav Hansen

February 15, 2011

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis will explore the possibilities for implementing the UML state machine specification [14] into the Java language. Several approaches will be explored, from going the software route with state patterns and the creation of a framework, to extending the language with new keywords.

State machines are used to describe the *reactive* properties of a system, a reactive system is a system that responds to internal and external events.

One of the recurring examples used in this thesis is a simple switch, this switch has two possible modes of operation, it is `on` or `off`. The events in this system would be the external influences that causes the switch to change mode.

1.1 Motivation

The motivation for this thesis was the lack of exploration when it comes to the integration of UML state machines into the Java language. While there have been several articles [9, 18] about extending languages with the state pattern [5], the language of choice was not Java, and while the state pattern shares some properties with state machines, it can at best be called a poor mans state machine.

1.2 Methods

Three approaches has been selected, and can be described as evolutionary exploration.

The exploration will start with the *state pattern* even if it (as mentioned before) can at best be called a subset of UML state machines, it is still a widely used pattern for creating reactive systems in Java.

Next up is the development of a *Java framework* that strictly follows the meta-model for state machines as defined by UML. Some simplification has been made due to time constraints, but it will create a solid foundation which can be built upon.

The last approach is *extending the Java language*, a special pre-processor will be built to allow for adding new keywords to the language, the output from this tool will be code that runs on the framework from the last method. It will not be built upon the work done by [9, 18] since it was decided that these approaches does not bring enough features from UML state machines into the language.

1.3 Chapter overview

Chapter 2, Introduction to UML State Machines will give a short introduction to UML state machines with a series of examples. In *Chapter 3, Related Work* gives a look into what kind of work has already been done in this area. *Chapter 4, The State Pattern* takes a look at state patterns and how they can be extended to be more like UML state machines. In *Chapter 5, A Java Framework for UML State Machines* introduces a framework for creating UML state machines in code. *Chapter 6, Extended Java* shows how the Java language can be extended to directly support UML state machines. This thesis will end in *Chapter 7, Conclusion* with an conclusion.

Chapter 2

UML State Machines

2.1 Introduction

The UML specification defines a meta-model for state machines, this meta-model[1] can be seen in Figure 2.1 on the facing page. The meta-model consist of a set of classes for describing the nodes, edges and properties that makes up the state machine graph. The state machine graph is a higraph [7] which adds the notion of depth and orthogonality.

The examples introduced in this chapter will be reused in later chapters. Every example will bring something new to the state machine graph.

The only kind of state machines that will be considered are behavioral state machines, protocol state machines are outside the scope of this thesis.

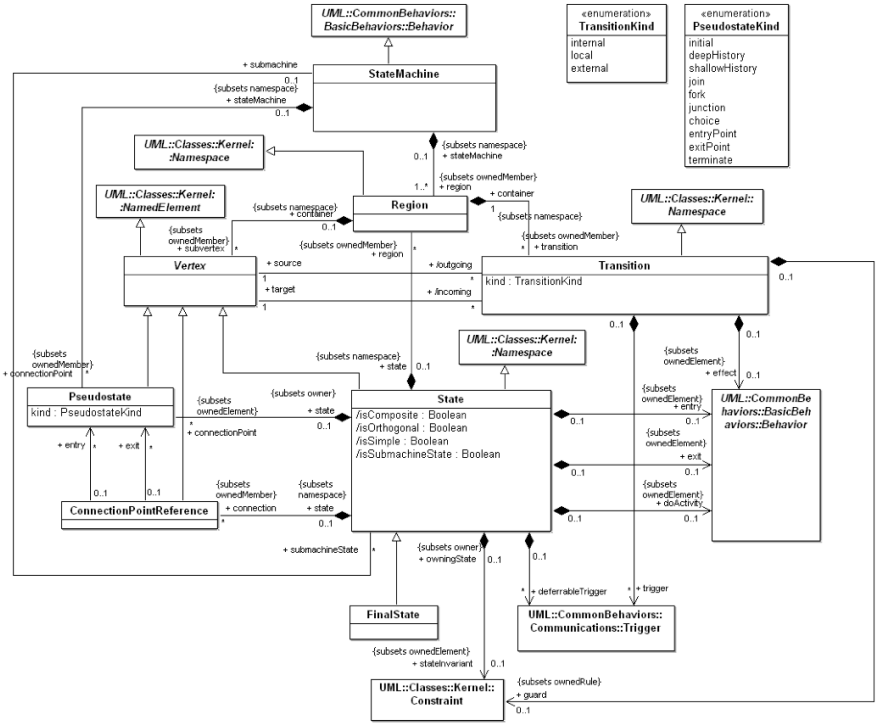
For a more thorough explanation of UML state machines, there are several UML books [17, 3, 15, 16, 12] with their own state machine chapter. The specification can be considered the authoritative source for the standard, but it is not a good starting point since knowledge of state machines are assumed and there are very few examples.

2.2 The meta-model classes

This section will describe all the meta-model classes that are used by the state machine graphs made in this chapter.

[1] For the remainder of this chapter, a simpler version of this meta-model will be used

Figure 2.1: Meta-model for UML state machines



2.2.1 StateMachine

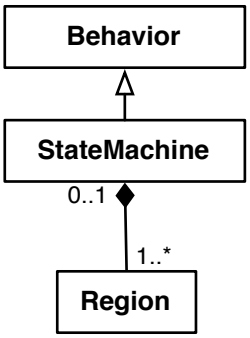
A state machine models the behavior of a system, or a part of a system. A state machine that is not used as a submachine should be seen as the top-level element of the state machine graph.

The most important fields are listed in Table 2.1 and the class relationship is shown in Figure 2.2 on the following page.

Table 2.1: Important properties of StateMachine

`region`	The regions that are owned by the state machine.

Figure 2.2: Cutout of the StateMachine class



2.2.2 Region

A region represents a containment of vertices and the transitions between them.

The most important fields and restrictions are listed in Table 2.2 on the next page and the class relationship is shown in Figure 2.3.

Figure 2.3: Cutout of the Region class

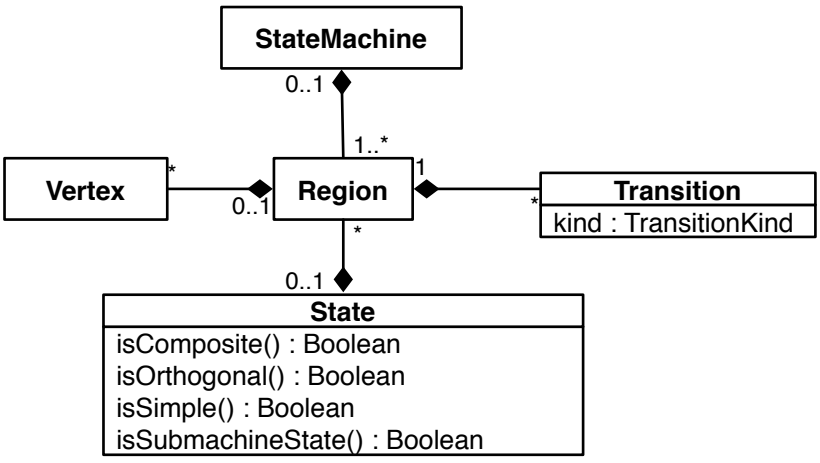


Table 2.2: Important properties of Region

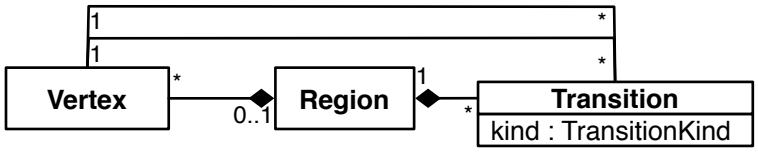
`transition`	The set of transitions owned by the region.
`subvertex`	The set of vertices owned by the region.
`statemachine`	The StateMachine that owns the region.
`state`	The state that owns the region.
restrictions	• There can be at most one initial pseudostate. • There can be at most one deep history pseudostate. • There can be at most one shallow history pseudostate. • It can not be owned by both a state and a state machine, only one of them.

2.2.3 Vertex

A vertex is the common superclass for all classes that acts as targets for transitions.

The most important fields are listed in Table 2.3 on the next page and the class relationship is shown in Figure 2.4.

Figure 2.4: Cutout of the Vertex class



2.2.4 State

A state represents a condition of an object at some point in time. A typical example could be a simple switch, it has two possible conditions,

Table 2.3: Important properties on Vertex

`incoming`	The set of transitions incoming from the vertex.
`outgoing`	The set of transitions going out from the vertex.
`container`	The owner of the vertex.

it is either on or off, but it can never be both at the same time.

There are three kinds of states; *simple*, *composite*, and *submachine* states. Unlike transitions and pseudostates there is no kind attribute here. The kind of state is determined by the configuration of the state itself.

simple A simple state is a state that have no regions, and no submachine connected to it.

composite A composite state is a state with one or more regions. If it has at least two regions, it can also be called an orthogonal state (since the regions are orthogonal).

When more than two regions is at play in the same state, the regions are considered concurrent to each other.

submachine A submachine state is a state witch contains another state machine, communication with the submachine is done using ConnectionPointReferences and exit / entry pseudostates.

All states have optional behaviors executed when entered, exited and while active.

The most important fields are listed in Table 2.4 on the next page and the class relationship is shown in Figure 2.5 on the facing page.

2.2.5 Transition

A transition is responsible for moving the state machine from a source vertex to a target vertex. Transitions can be said to be compound transition if transition moves the state machine from one complete configu-

Figure 2.5: Cutout of the State class

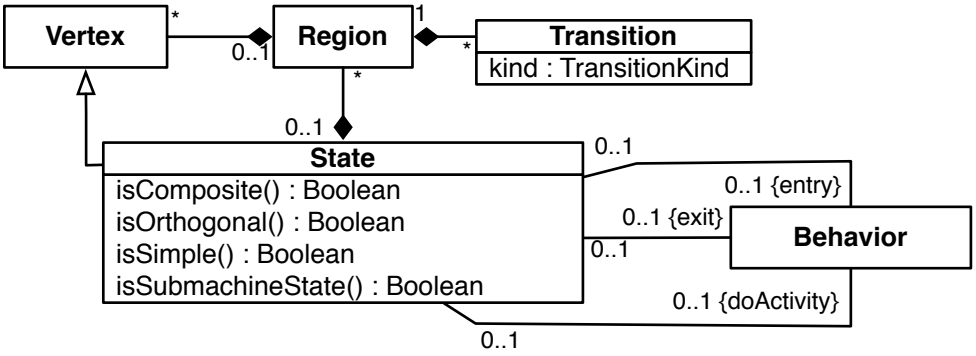


Table 2.4: Important properties of State

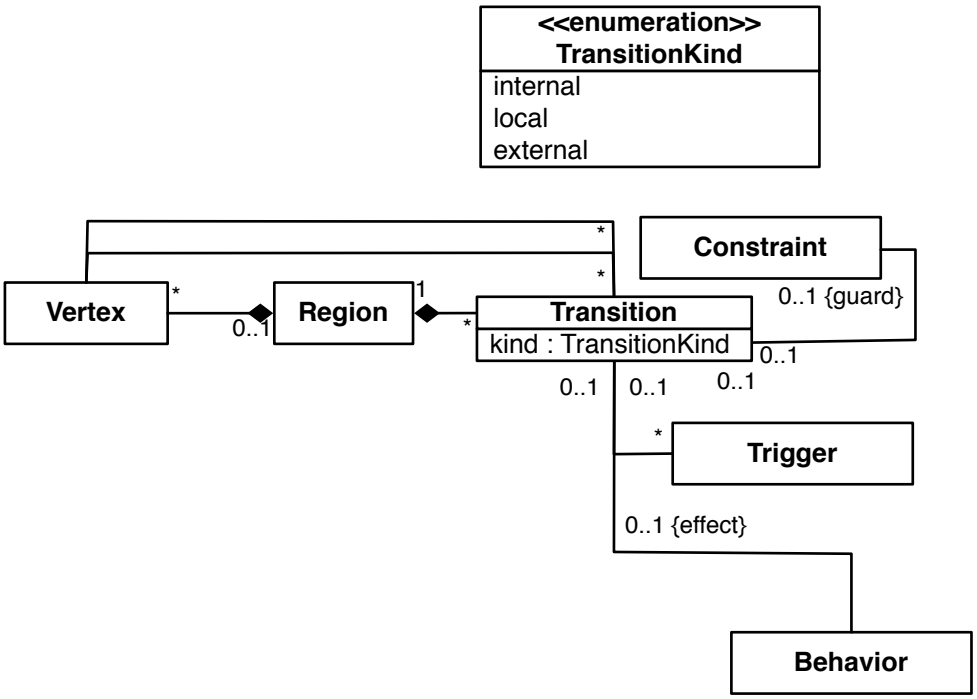
`region`	The regions owned by the state.
`doActivity`	An optional behavior that is executed while being in the state.
`entry`	An optional behavior that is executed whenever this state is entered regardless of the transition taken to reach the state.
`exit`	An optional behavior that is executed whenever this state is exited regardless of which transition was taken out of the state.

ration to another one, e.g. a transition from one state to another state is a compound transition, a transition from a state to a pseudostate is not.

There are three kinds of transitions, *external*, *local* and *internal*. The *external* kind is the default, and the semantics are quite simple, when an external transition is triggered, the source state is completely exited.

The most important fields and restrictions are listed in Table 2.5 on page 17 and the class relationship is shown in Figure 2.6 on the following page.

Figure 2.6: Cutout of the Transition class



2.2.6 Pseudostate

A pseudostate is a state that is used to control the flow between compound transitions. There are several kinds of pseudostates, and the ones that are used in the examples are described below.

initial The starting vertex in a region unless there was an explicit transition to another state. The transition going out from a initial pseudostate can have an effect, but no trigger or guard.

shallowHistory and deepHistory Unless a final state made the region completed, the most recent state configuration is saved when the region is exited.

The history pseudostates are used to re-enter a saved state configuration. E.g. if the current state is A.B.C.D (current state is D, contained in C, which is contained in B, and so on), and the next

Table 2.5: Important properties of Transition

`kind`	The type of transition this node represents.
`source`	The source vertex of the transition.
`target`	The target vertex of the transition.
`guard`	The constraint that decides if this transition is enabled or not.
`effect`	The behavior that is executed when the transition is fired.
`trigger`	The triggers that may fire this transition.
`container`	The owner of the transition.
restrictions	• A pseudostate of kind fork or join must not have guards or triggers. • A pseudostate of kind fork must always target a state. • A pseudostate of kind join must always come from a state. • No pseudostates can have triggers on their transitions, except for the initial pseudostate which can have a trigger with the "create" stereotype, but only when in the region of state machine.

transition goes from D back to state A, then a entry into a shallow history pseudostate would enter the B state, if a deep history pseudostate is entered, it would go all the way to state D.

fork Used to split one incoming transitions into two or more outgoing transitions which targets different regions in a orthogonal state. The outgoing transitions can have effects, but no triggers or guards.

join Used to merge two or more incoming transitions into one outgoing transitions. The incoming transitions are coming from different regions in the same orthogonal state. The incoming transitions can have effects, but no triggers or guards.

choice Used to represent a dynamic conditional branch, which splits one or more incoming transitions into one or more outgoing transitions. The outgoing transitions must have guards have at least one transitions that evaluated to true, if more than one guard evaluates to true, a random one is selected. If all guards evaluate to false, the state machine is considered ill-formed. An optional else guard can be supplied to be the default transition if no other guards are true.

The most important fields are listed in Table 2.6 and the class relationship is shown in Figure 2.7.

Figure 2.7: Cutout of the Pseudostate class

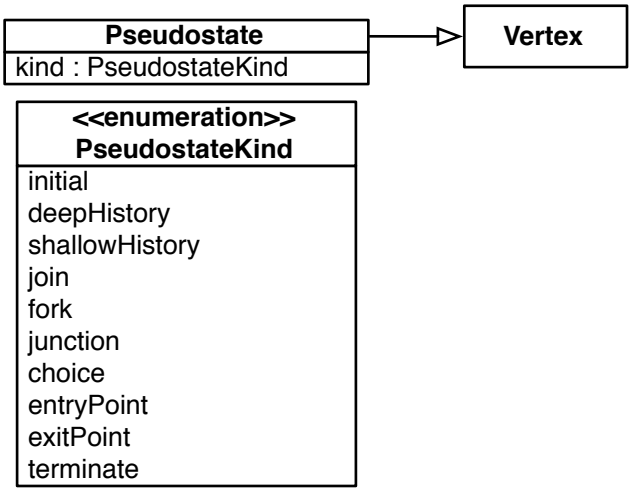


Table 2.6: Important properties on Pseudostate

`kind`	The type of pseudostate this represents.

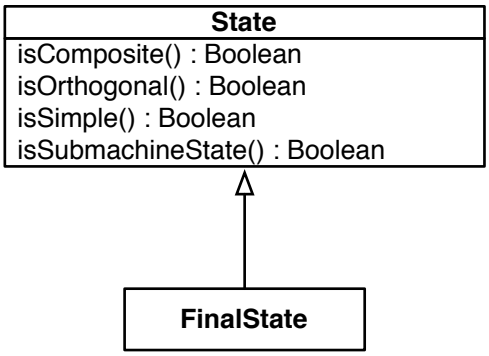
2.2.7 FinalState

This subclass of state is used to mark the containing region complete. If the region is directly contained in a state machine, and there are no other regions, the state machine is also complete.

A final state can *not* have any outgoing transitions, regions, reference a submachine, entry/exit/doActivity behavior.

The class relationship is shown in Figure 2.8.

Figure 2.8: Cutout of the FinalState class



2.3 A basic state machine

This very basic state machine captures the model of a system that has only one state.

The state is named `idle` and the machine is designed purely for demonstration of the basic building blocks of a state machine graph. It has one event called *end*, that will trigger the transitions out of the `idle` state.

The graph can be seen in Figure 2.9 on the following page and a sample run is described in section 2.3.1.

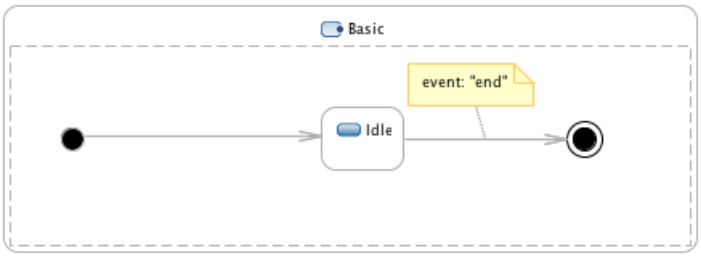
2.3.1 Sample run

A sample run of this state machine is listed below:

1. When this state machine is entered, the initial pseudostate is entered.

2. The transition out of this pseudostate does not have any triggers, so the move to the next state will happen immediately.

Figure 2.9: Diagram of a basic state machine



3. The state machine has now moved to its next complete configuration, with `idle` as its active state.

4. The event *end* is now sent to the state machine. This event reaches the `idle` state and its transitions are checked for matches to this event. The only transition matches this event.

5. The state machine now moves to the next complete configuration which is in the final state. This final state ends the region, and since this region is the only region contained by this state machine, the state machine ends.

2.4 A switch state machine

This state machines models the behavior or a simple switch. This switch has two states, `on` and `off`. The state machine starts out with the `off` state, and has three events that will trigger a change of the currently active configuration:

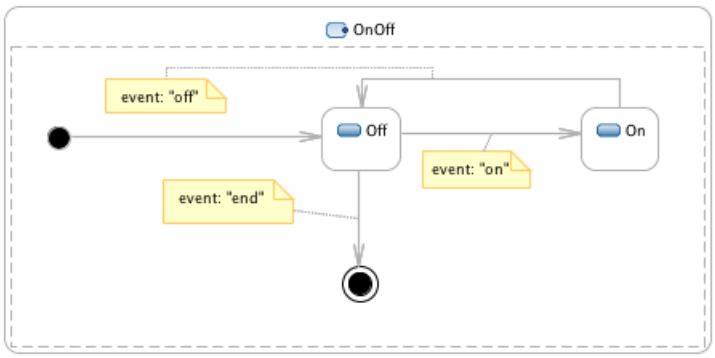
- Event *end*: This will trigger the transition to the final state and will end the state machine. This event is only valid when the current active state of the state machine is `off`.

- Event *off*: This event is only valid if the current active state is the `on` state, and will trigger the transition to the `off` state.

- Event *on*: This event is only valid if the current active state is the `off` state, and will trigger the transition to the `on` state.

The graph can be seen in Figure 2.10 and a sample run is described in section 2.4.1.

Figure 2.10: Diagram of a switch



2.4.1 Sample run

A sample run of this state machine is listed below:

1. When this state machine is entered, the initial pseudostate is entered.
2. The transition out of this pseudostate does not have any triggers, so the move to the next state will happen immediately.
3. The state machine has now moved to its next complete configuration, with `off` as its active state.
4. The event *on* is now sent to the state machine, and this will trigger the move to the complete state machine configuration with `on` as the active state.
5. The event *off* is now sent to the state machine, and this will trigger the move to the complete state machine configuration with `off` as the active state.
6. The event *end* is now sent to the state machine, and this will trigger the move to the final state of this region, and signals the end of this

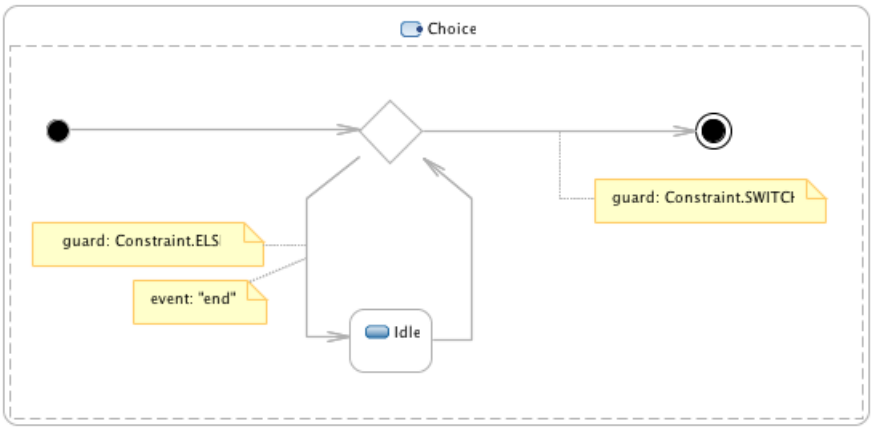
region. Since the state machine does not have any other regions, this will complete the state machine.

2.5 A choice state machine

This example models a state machine that will have its flow determined by a set of dynamic guards. The state machine have only one event called *end* which will end the state machine. The transition out of the choice pseudostate is guarded by guard that is false at the first evaluation, and then switches back and forth between true and false on every evaluation.

The graph can be seen in Figure 2.11 and a sample run is described in section 2.5.1.

Figure 2.11: Diagram of the choice state machine



2.5.1 Sample run

A sample run of this state machine is listed below:

1. When this state machine is entered, the initial pseudostate is entered.
2. The transition out of this pseudostate does not have any triggers, so it will move immediately to the next state, which is the choice pseudostate.

3. This pseudostate has two possible routes to take, and they are both guarded. Since the *else* guard has a special meaning, the only guard that is really evaluated is the *switch* guard. This guards starts out as false, so this route will not be chosen.

4. Since the *switch* guard did not pass, the *else* guard will be chosen, and this will lead to the next state machine configuration with the `idle` state as the current state.

5. The event *end* is sent to the state machine and this triggers the move to the pseudostate again. This time the *switch* guard will be true, and this path will be chosen.

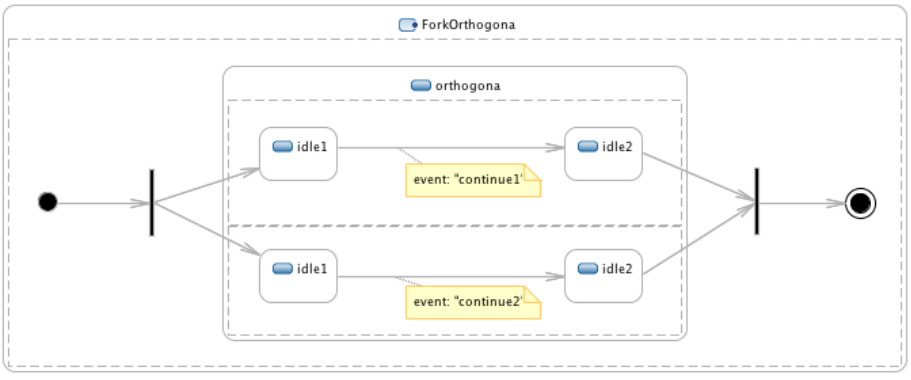
6. Since the last event triggered the move to the final state, this region is now complete, and since the state machine only have one region, the state machine is also complete.

2.6 A forking state machine

This example introduces pseudostates for splitting the flow into several concurrent regions, and then joining them back together.

The graph can be seen in Figure 2.12, and a sample run is described in section 2.6.1 on the next page.

Figure 2.12: Diagram of the forking orthogonal



2.6.1 Sample run

A sample run of this state machine is listed below:

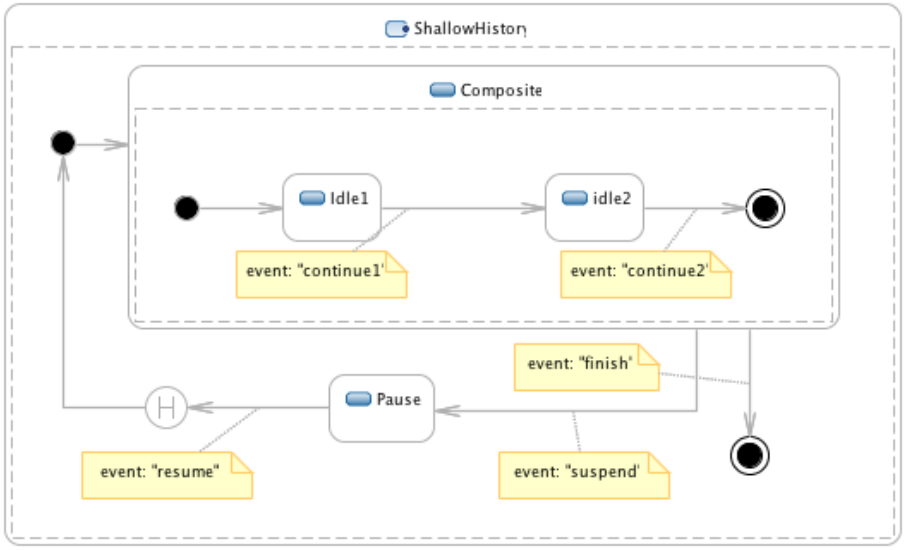
1. When this state machine is entered, the initial pseudostate is entered.
2. The transition out of this pseudostate does not have any triggers, so the move to the next state will happen immediately.
3. The state machine has now moved into the fork pseudostate, and since there can not be any guards or triggers on the transitions out of the fork, the pathways are simply followed. This means that we now have two active states, state `idle1` on region 0, and state `idle1` on region 1 (both on the `orthogonal` state).
4. The event *continue1* is now sent to the state machine, and `idle1` from region 0 triggers on this event, and moves to *idle2*.
5. The event *continue2* is now sent to the state machine, and both states triggers on this. The region that is currently in state `idle2` triggers to the join pseudostate and waits there for the rest of the incomming transitions. The region that is currently in state `idle1` moves to state *idle2*.
6. The event *continue3* is now sent to the state machine, and this triggers the move from *idle2* to the join pseudostate. This pseudostate is now complete, and it moves to the final state. This final state completes the region, and since this state machine only have one region, it will complete the state machine.

2.7 A deep history based state machine

This example introduces pseudostates for resuming a state machine configuration after a transition makes the state machine move out of its current configuration. The machine will use a deep history pseudostate for this, which means it will recursively enter the states until the exact same configuration is resumed.

The graph can be seen in Figure 2.13 on the facing page, and a sample run is described in section 2.7.1 on the next page.

Figure 2.13: Diagram of shallow history



2.7.1 Sample run

A sample run of this state machine is listed below:

1. When this state machine is entered, the initial pseudostate is entered.
2. The transition out of this pseudostate does not have any triggers, so the move to the next state will happen immediately.
3. Since this state is a composite state, the region will be entered, and the initial pseudostate will start. The pseudostate will then initiate the move to the state `idle1`.
4. The event *continue1* is sent to the state machine, and the machine moves to state `idle2`.
5. The event *suspend* is sent to the state machine, and this triggers the move to the state `pause`, the state `idle2` is marked as the last active state on the `composite` state.
6. The event *resume* is now sent to the state machine, and this triggers the move to the shallow history pseudostate.

7. This pseudostate will make the move to the last active state in the `composite` state, and this means that that the current active state machine configuration is the `idle2` state. If this would have been shallow history, it would have only entered the `composite` state.

8. The event *continue2* is now sent to the state machine, and this triggers the move to the final state of the `composite` state. This marks the region as complete, but since the containing state has two transitions out, it will not auto-complete this state.

9. The event *finish* is now sent to the state machine, and this triggers the move to the final state of the state machine. This marks the region as complete, and since this state machine does not have any more regions, it will also be complete.

Chapter 3

Related Work

3.1 Introduction

This section will introduce some of the papers that have been influential to the development of the UML state machine specification [14] and also serves as the background material for this thesis.

3.1.1 Statecharts

David Harel introduced in 1984 a paper called "Statecharts: A visual formalism for complex systems." [6].

This paper introduces a visual notation for describing *reactive* systems. A reactive system, is a system that reacts to events from both outside and inside stimuli.

Some of the more important points of this paper was:

Hierarchical machines Harel introduced the concept of hierarchical state machines. An hierarchical state machine is composed of several sub-states, which also can contain more sub-states.

AND/XOR Harel also introduced orthogonal state machines, and with that the possiblity of a state machine to be in two or more states at the same time. This relationship is described using the boolean properties AND/XOR.

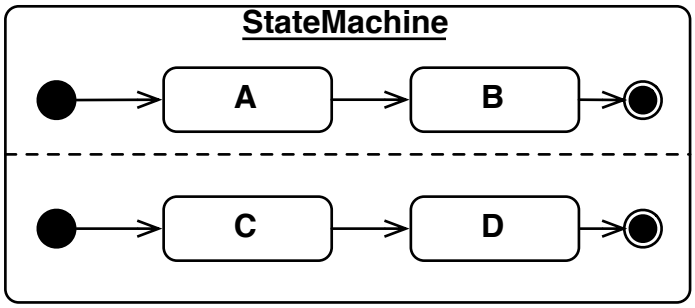
In a non-hierarchical state machine, the operator XOR would make sure that in the same grouping of states, there would only be one

active state at any point in time. E.g. in a group with states A, B only one of them could be active at once.

The AND operator introduced the possibility of having more than one active state at once, but not in the same hierarchical group.

This can be seen more clearly in Figure 3.1, in this figure there are four possible configurations, (A, C), (A, D), (B, C), (B, D).

Figure 3.1: Example of AND/XOR



This can also be expressed as $(A, B)x(C, D)$, i.e. the cartesian product of the two sets of states.

Actions and activities Another concept that was formalized was the addition of executable actions and activities that was based on work by Mealy [11], and Moore [13].

This addition allowed the user to add actions to not only transitions, but also when a state was entered, exited, and while it remained active.

3.1.2 UML state machines

State machines as defined by the UML specification [14] is an object based variant of Harel's statecharts. State machines has been part of the specification since 1.x, but only the somewhat revised 2.2 version is used in this thesis.

3.2 State machines at runtime

3.2.1 Language extension

While there have been written several books about patterns [5, 4], there have not been a lot of work done in the integration of state patterns and programming languages.

Taivalsaari [18] introduced the notion of modes (which are basically the same as states) which had automatic transitions between them.

Taivalsaari used something called *dynamic inheritance* that would allow the actual implementation part of an object to be swapped out during runtime. This is similar to how prototypal inheritance works in e.g. JavaScript.

Taivalsaari also introduced a simple system for doing automatic transitions, e.g. a transitions could be made automatically if a certain property had a certain value, see Listing 3.1 for an example of how this looks in the language presented. The full example can be seen in [18] on page 28.

Listing 3.1: Simple example of automatic transitions

```
1 TRANSITION IS
2   empty WHEN i = 0;
3   full WHEN i = 10;
4   non-empty OTHERWISE;
5 ENDTRANS;
```

In '99 Madsen published the article "Towards integration of state machines and object-oriented languages" [9] which describes at method for integrating the BETA language [10] with the state pattern. While the goal of the article is stated as:

> The goal of this paper is to obtain a one-to-one correspondence between state machines as e.g. used in UML and object-oriented programming languages.

The actual end result is the integration of state patterns and BETA using dynamic inheritance (as in Taivalsaari) which allows for changing the implementation of an object at runtime (it should be said that inheritance and composition is also explored). In this paper, all the transitions were explicit.

While both of these papers took an interesting approach to extending a language with the notion of state, they have some properties that makes them less than ideal for this thesis. First, as said they are not using the UML state machine meta-model for the extension which means that a lot of features are missing, second they are not using Java which is the language chosen for this thesis.

3.2.2 Executable state machines

While there have not been a lot of work done into integration of the full UML state machine specification [14] into programming languages, there have been some work done turning state machine diagrams into executable diagrams.

In a follow up to his '84 [6] article, Harel [8] published an article about the creation of executable statecharts. They present a visual notation for class and object diagrams called `O-chart`, which together with statecharts creates a complex framework for developing executable state machines.

A tool called `O-mate` was created to allow for the development of diagrams based on statecharts and O-charts, the behaviors and constraints were then created using the C++ language. For executing the statecharts, a code-generation step was supported that would output C++ code.

There have also been some work by Barbier [1] that uses UML state machine diagrams (described in XMI) as the basis for the Java runtime. These diagrams are then transformed into executable Java code. The framework itself is tightly integrated into the Topcased MDA platform, which makes it less than ideal for the purposes of this thesis.

3.2.3 W3C State Chart XML

W3C[1] delivered the first draft of the SCXML specification in July 2005 [2] and is a XML format for specifying state machines.

The specification says it is based on statecharts as described by Harel [6] rather than using the UML specification, this is clearly visible in the structure of the XML tags (which are not based on the UML meta-model).

[1] `http://www.w3c.org`
[2] `http://www.w3.org/TR/2005/WD-scxml-20050705/`

E.g. see Listing 3.2 for an example of a simple state, in this example the `hello` state is both the initial state, the actual state of the context, and also serves as the final state[3].

Listing 3.2: SCXML example

```
1 <scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
      initialstate="hello">
2 <state id="hello" final="true">
3   <onentry>
4     <log expr="'hello world'" />
5   </onentry>
6 </state>
7 </scxml>
```

The latest draft was released 16 december 2010, and the standard is still under development.

A couple of known runtime implementations exist for SXCML, Commons SXCML[4] for Java, and Qt State Machine Framework[5] for C++.

3.2.4 Northstate Framework

NSF (Northstate Framework) is a proprietary framework for state machines, and it describes itself as:

> The North State Framework (NSF) is a robust, Microsoft .Net class library that simplifies the process of creating highly-extensible, object-oriented code from a UML State Machine diagram.

And although the development seemed to have stopped in 2008, the framework is the most complete state machine framework out there. One big caveat of the framework is that it does not use the classes from the UML specification, e.g. the initial pseudostate is created using the `NSFInitialState` class and not a `Pseudostate` class with the *initial* kind.

[3] This example was taken from `http://commons.apache.org/scxml/guide/scxml-documents.html`

[4] `http://commons.apache.org/scxml/`

[5] `http://doc.qt.nokia.com/4.7/statemachine-api.html`

3.3 Conclusion

While there have been plenty of research on executable state machines, the focus has been on either frameworks together with tight tool-integration, or simple language extensions using state patterns. There have been little or no work looking into the expansion of the Java language to allow for the full specification of UML state machines.

Chapter 4

The State Pattern

4.1 Introduction

Design patterns have a long history in the art of software engineering, and can be conceptualized as recipes for software engineering. The best known book about design patterns, and also the book where most of them are formalized is the book *Design Patterns: Elements of Reusable Object-Oriented Software* [5]. This book, released in '94 was the result of several years of gathering solutions to common problems in software engineering.

The pattern explained and extended in this chapter is the pattern known as the state pattern, the book defines the intent of this pattern as:

> "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

This makes it fit well within the definition of UML State Machines, although simplified.

4.2 Overview

The state pattern defines three major participants in the state pattern:

context
: This is the interface for the user, and all possible interaction of

interest with the state should be managed through this interface. All interactions that are based on the current state, will be sent to the state object, if the state object needs to make modifications to the context, it can send itself as a parameter.

state

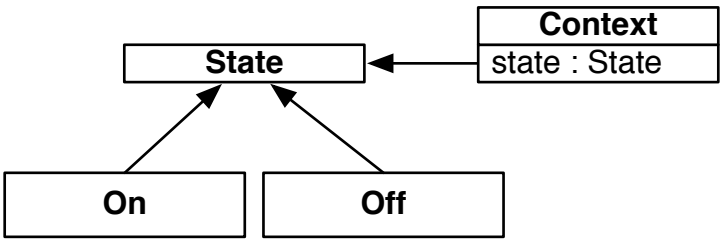
This is the catch-all interface for possible state interactions. A catch-all interface is a interface where all possible state changes (methods) are included, so in our case (which will be described later) we have two possible state changes, `on` and `off`, and so the catch-all interface would have `on` and `off` methods. This will be clearer when the actual code is described.

concrete sub-classes of state

This is the concrete definition of a state, and is based on the *state* interface.

The relation can also be seen in Figure 4.1.

Figure 4.1: Connection between the main classes in the state pattern



The book defines several possible variations on the state pattern theme, but some simplification has been made to the pattern (and to better fit with the extensions later on).

So the rules for the state pattern are as follows:

1. No external libraries or frameworks may be used, only standard Java.
2. Annotations and reflection are not allowed (these are better suited for a framework implementation). The only exception to this rule is when implementing the singleton pattern later on.

3. States are created on demand, but are using the singleton pattern [5] so there will always be at most one instance of the particular state class. This implies that all states are stateful (but there will be introduced ways to remedy this later).

4. The context will send itself as a parameter for every interaction with the state methods.

5. Except for the initial construction of the context object, it is the responsibility of the state object to change the state of the context.

6. There will be no context / state interfaces, instead the context will be directly implemented, and states will have a common abstract superclass.

The example that will be used is a simplified version of the switch from section 2.4 on page 20, see Figure 2.10 on page 21 for a graphical representation.

The switch has two possible states, `on` and `off`, and the semantics:

1. The switch always starts in the `off` state.

2. When the switch is `off`, the only legal transition is to the `on` state.

3. When the switch is `on`, the only legal transition is to the `off` state.

The first implementation will be based on the basic state pattern [5], and the following examples will extend with features from UML State Machines.

4.3 Basic implementation of a switch

In this first example, all the rules from list 4.2 on the facing page will be applied.

Listing 4.1: SwitchState

```
1 abstract class SwitchState {
2   public void on(SwitchContext ctx) { }
3   public void off(SwitchContext ctx) { }
4 }
```

We define the *SwitchState* as an abstract class so that it can not be instantiated. The `on` and `off` methods are the two possible state triggers available for the context.

Listing 4.2: OnState and OffState

```
1 class OffState extends SwitchState {
2   static private SwitchState self;
3
4   static public SwitchState instance() {
5     if (null == self) {
6       self = new OffState();
7     }
8
9     return self;
10  }
11
12  @Override
13  public void on(SwitchContext ctx) {
14    ctx.changeState(OnState.instance());
15  }
16 }
17
18 class OnState extends SwitchState {
19  static private SwitchState self;
20
21  static public SwitchState instance() {
22    if (null == self) {
23      self = new OnState();
24    }
25
26    return self;
27  }
28
29  @Override
30  public void off(SwitchContext ctx) {
31    ctx.changeState(OffState.instance());
32  }
33 }
```

Both of these classes are extensions of the *SwitchState* ABC[1] defined earlier. The classes implement the methods that they can handle, and is responsible for changing the state object on the context object.

[1] Abstract base class.

Another responsibility is returning a singleton object for this class, this is handled through the `instance` method, and is a very simple (and non thread-safe) implementation of the pattern.

Listing 4.3: SwitchContext

```
1 class SwitchContext {
2   public SwitchContext() {
3     changeState(OffState.instance());
4   }
5
6   public void on() {
7     state().on(this);
8   }
9
10  public void off() {
11    state().off(this);
12  }
13
14  public void changeState(SwitchState state) {
15    this._state = state;
16  }
17
18  public SwitchState state() {
19    return this._state;
20  }
21
22  private SwitchState _state;
23 }
```

The last participant in the switch is the actual context, this class implements the same state triggers as the *SwitchState* ABC, but also has a couple of its own. For changing the current state, the class has the `changeState` method (this will be extended later), and a simple getter is available through the `state` method.

4.4 Extending the switch with behaviors

In this first extension, the pattern will be extended using the behaviors from a simple state (from UML State Machine). What this means, is that we will add optional `entry` and `exit` methods to the pattern. This makes the state objects optionally stateless (if the programmer cleans up on

exit).

The singleton pattern will also be extended, making the state implementation easier (by moving the singleton from the state object to the context object), this introduces elements from the reflection API, but nothing that should cause the programmer any headache.

Listing 4.4: SwitchState

```
1 abstract class SwitchState {
2   public void on(SwitchContext ctx) { }
3   public void off(SwitchContext ctx) { }
4
5   public void entry() { }
6   public void exit() { }
7 }
```

The `entry` and `exit` methods have been added here with their default implementations. It is up to the implementor if he wants to implement these.

Listing 4.5: OnState

```
1 class OnState extends SwitchState {
2   @Override public void entry() { }
3   @Override public void exit() { }
4
5   @Override
6   public void off(SwitchContext ctx) {
7     ctx.changeState(OffState.class);
8   }
9 }
```

Here the *OnState* class have added the two methods defined above. This only serves as an example, and they do nothing useful.

Listing 4.6: SwitchContext

```
1 class SwitchContext {
2   private Map<Class<? extends SwitchState>, SwitchState> states
3     = new HashMap<Class<? extends SwitchState>, SwitchState>();
4
5   public SwitchContext() {
6     changeState(OffState.class);
7   }
8
```

```
 9   public void on() {
10     state().on(this);
11   }
12
13   public void off() {
14     state().off(this);
15   }
16
17   public void changeState(Class<? extends SwitchState> c) {
18     if (null != _state) _state.exit();
19
20     if (states.containsKey(c)) {
21       _state = (SwitchState) states.get(c);
22     } else {
23       try {
24         _state = (SwitchState) c.newInstance();
25       } catch(Exception e) { }
26
27       states.put(c, _state);
28     }
29
30     _state.entry();
31   }
32
33   public SwitchState state() { return this._state; }
34
35   private SwitchState _state;
36 }
```

The main focus of the new implementation is in `changeState`. Using a simple *Map*, we can directly connect classes with their singleton instances. This takes all the singleton work out of the state object, and makes implementing new states much easier. When a state change occurs, this method will make sure that the `exit` method is called on the last state, and that the `entry` method is run on the new state instance.

4.5 Extending the switch with guards

Building on the code from the last section, in this section there will be added guards to our state changes.

There will be two kinds of guards introduced:

state transition guard

This guard is for guarding transitions that are legal, just not for this particular state configuration. This is implemented as a simple `guard` method on the state object, and the `changeState` method will check for this value, and only change the state object if the guard passes.

illegal transition guard

This guard is for guarding transitions that are not legal, a simple example would be calling the `off` method on a *OffState* object. This is implemented as a extension of the *Exception* class.

Listing 4.7: IllegalStateTransitionException

```
1 class IllegalStateTransitionException extends Exception { }
```

This is the exception class for when illegal transitions are called.

Listing 4.8: SwitchState

```
1  abstract class SwitchState {
2    public void on(SwitchContext ctx) throws
         IllegalStateTransitionException {
3      throw new IllegalStateTransitionException();
4    }
5
6    public void off(SwitchContext ctx) throws
         IllegalStateTransitionException {
7      throw new IllegalStateTransitionException();
8    }
9
10   public void entry(SwitchContext ctx) { }
11   public void exit(SwitchContext ctx) { }
12
13   public boolean guard(SwitchContext ctx) { return true; }
14 }
```

These are the default implementations of `on` and `off` which both are declared to throw the newly defined *IllegalStateTransitionException* exception by default.

A `guard` method that always returns true has also been added.

Listing 4.9: OnState

```
1 class OnState extends SwitchState {
2   private boolean sw = true;
3
4   @Override public void entry(SwitchContext ctx) { }
5   @Override public void exit(SwitchContext ctx) { sw = true; }
6
7   @Override public boolean guard(SwitchContext ctx) {
8     sw = !sw;
9     return sw;
10  }
11
12  @Override
13  public void off(SwitchContext ctx) {
14    ctx.changeState(OffState.class);
15  }
16 }
```

This is a simple implementation of a guard in the *OnState* class, it will return false at first try, and then true at the second try. Since this is a stateful class, we reset the affected values when the `exit` method is run.

Listing 4.10: SwitchContext

```
1 class SwitchContext {
2   private Map<Class<? extends SwitchState>, SwitchState> states
3     = new HashMap<Class<? extends SwitchState>, SwitchState>();
4
5   public SwitchContext() { changeState(OffState.class); }
6   public void on() throws IllegalStateTransitionException {
7     state().on(this);
8   }
9
10  public void off() throws IllegalStateTransitionException {
11    state().off(this);
12  }
13
14  public void changeState(Class<? extends SwitchState> c) {
15    SwitchState tmpState = null;
16
17    if (states.containsKey(c)) {
18      tmpState = (SwitchState) states.get(c);
19    } else {
20      try {
21        tmpState = (SwitchState) c.newInstance();
```

```
22        } catch(Exception e) { }
23
24        states.put(c, tmpState);
25      }
26
27      if(!tmpState.guard(this)) {
28        return;
29      }
30
31      if (null != _state) _state.exit(this);
32
33      _state = tmpState;
34      _state.entry(this);
35    }
36
37    public SwitchState state() { return this._state; }
38
39    private SwitchState _state;
40  }
```

In this new implementation of `changeState` we now get the wanted state object first, and then run the guard on it. If it passes, we continue as normal. If it does not pass, we just return from the method.

We also continue mimicking the *SwitchState* class, and adds the throws clause to our two state-changing methods `on` and `off`.

4.6 Conclusion

The use of the state pattern might seem like a good solution at first, it has all the characteristics that would make it seem like the ideal candidate:

1. It is well known in the software engineering community, and has implementations in most (if not all) programming languages.
2. The pattern itself is documented in several books [5, 4].
3. It's simple, so implementation should never lead to any problems.

The real problem here is point number 3. It's maybe too simple, as it only gives the very basic features of what you would want, and leaves all the details up to the implementer. So when we start implementing

features that expand the pattern, it has already given you what it can, and you are on your own implementing the rest.

The pattern could of course be expanded as it has already been shown, state behaviors, guards, etc. all could be described in the pattern, but that would work against the pattern, and not for it, since it would end up too complex and nobody would use it.

Another problem with using this approach (and this is a problem with patterns in general), is that unless you make the pattern into a framework you end up mixing the pattern and the other parts of your code that is not really relevant to the logic of the state pattern. An important principle in software engineering is the separation of concerns [2], and using a pattern for anything but the very basic building blocks of a program works against this principle.

Chapter 5

A Java Framework for UML State Machines

5.1 Introduction

The framework presented here is an implementation in Java of the state machine meta-model, semantics and constraints from the UML 2.2 specification [14].

The API and the runtime of the framework has gone through a complete rewrite from the first version. The first version used threading and stacks (for level-management) which in hindsight was a bit overkill for a prototype implementation.

The current runtime system uses multiplexing instead of threads, which means that there is no true concurrency in the system (but for our purposes that is not important). A simple integer-based level system instead of stacks, these will both be more explained later.

This chapter starts with a introduction to the meta-model based API, and then continues with the runtime system.

For more information about the semantics, constraints and a general overview of UML state machines please see chapter 2 on page 10.

5.2 The state machine classes

This section will explain all the important fields and methods on the classes that are related to the building of the state machine graph.

All the classes are located in the *smlib.uml2* package, and are documented with information from the specification itself.

The classes in the API are implemented with compliance in mind, so all the constructors will do their best to make it impossible to create a non-conforming state machine graph. It is still possible to create state machines that are not conformant, so a validator based on the constrains and static semantics from the specification has been implemented. The run of the validator is forced in the runtime system, but it is optional if only using the graph API.

In the *smlib.uml2.xmi* package, an importer for XMI 2.1 files has been implemented. This importer allows the user to create a state machine either in handwritten XMI, or using a modeling tool like Eclipse. This allows for rapid development, and rapid changes to a state machine graph. Using Eclipse will also result in state machines that are always compliant.

5.2.1 Semantic and SemanticException

For validation of the static semantics of the current state machine setup, all the classes in the runtime system implements an interface called *Semantic*. This interface contains a `validate` method which checks the current setup, and reports on the errors it occurs.

When a semantic error is found, an instance of the *SemanticException* class is thrown with a description of the error.

5.2.2 Node

This class is the super class for all nodes in the class hierarchy (except the classes dealing with constraints and semantics). Its main responsibility is making sure every node has an id and a name.

The important fields are listed in Table 5.1 on the following page.

5.2.3 Behavior

This class represents an executable behavior specification, there are four places in the system where these are needed, for *transition* effects and for all actions associated with state change (enter, do, exit).

Table 5.1: Table of the fields on Node

`id`	A unique ID for this node. If one is not given when creating this node, a unique ID will be generated using UUID[1].
`name`	A name for this node (not unique). If one is not given, it will get one using `getClass().getName()`.
`REGISTER`	Provides a register of all the IDs in the system, and their mapping to node objects.

Since Java do not support closures yet, the class implements the command pattern [5] and has one method called `run` that must be implemented in all sub-classes. This method is run when the behavior is entered.

There is no support for more advanced behaviors in the state machine graph, like state machines or activity diagrams as behaviors.

5.2.4 Vertex

The abstract *Vertex* class is a super-class for everything in the state machine that needs to be connected to each other. It has lists for incoming / outgoing transitions and is also responsible for the container (region).

See Table 5.2 for the important fields in this class.

Table 5.2: Table of the fields on Vertex

`incoming`	All edges (transitions) entering this vertex.
`outgoing`	All edges (transitions) departing from this vertex.
`region`	The regions that owns this vertex.

5.2.5 ConnectionPointReference

This class represents entries and exits into a submachine. Support for this is not implemented in the runtime system, but the API has support

for it.

See Table 5.3 for important fields on this class.

Table 5.3: Table of the fields on ConnectionPointReference

`state`	The owner of the connection point reference.
`entries`	Entries into the sub-machine.
`exits`	Exits out of the sub-machine.

5.2.6 Constraint

This class represents a constraint (guard) in the state machine graph.

Since there is no closure support in Java, a command pattern has been implemented. The class demands the implementation of a method called `check` which is a boolean method.

Both dynamic and static constraints are represented in the runtime system, but the graph API treats them alike.

A set of contraints have already been defined for usage in the graph, these constraints are listed in Table 5.4.

Table 5.4: Table of the included constraints

`Constraint.TRUE`	Constraint that always passes.
`Constraint.FALSE`	Constraint that never passes.
`Constraint.ELSE`	Constraint for when no other guards passes.
`Constraint.SWITCH`	Constraint that switches between being true and false (start outs as false).

5.2.7 Event

This interface is for defining custom events in the state machine. The interface is very simple and it extends *Comparable*[2].

The interface it extends has only one method, and it is used for comparing two classes which both implement this interface.

[2] `http://download.oracle.com/javase/6/docs/api/java/lang/Comparable.html`

The framework supplies one sub-class of this interface, the `StringEvent` class which allows for events based on strings.

See Table 5.5 for details.

Table 5.5: Table of the Event methods

`compareTo`	A compare method that takes an *Event* instance as a parameter. The semantics of the return value is as follow: **return 0** If the event instance is a match. **return -1** If the value of `this` is smaller than the event instance. **return 1** If the value of `this` is larger than the event instance. Any value besides 0 indicates that the the event in testing did not match the event it was tested against.

5.2.8 Trigger

This class represents a possible event trigger for an edge (transition) in the graph. Every edge can have multiple triggers, and it is wise to choose triggers that have different events (there is no well defined semantics if an event matches the triggers of multiple transitions).

The fields on *Trigger* are all explained in Table 5.6.

Table 5.6: Table of the fields on Trigger

`targetState`	The target of the transition if this trigger is triggered.
`event`	The event for this trigger (explained in the previous section).

5.2.9 FinalState

This sub-class of *State* offers nothing new in features, but is more constrained than the normal *State*. These new constraints are checked for in the validator.

When this class is reached it means that the enclosing region is finished, and if this region is contained in a state machine, the machine is also finished.

5.2.10 PseudoState and PseudoStateKind

The *PseudoState* class is used to represent all the different kinds of pseudostates in the state machine graph. If you give no *PseudoStateKind* when creating the node, the default is "initial", but this can be reconfigured to using another kind or you can give the kind at construction.

All the different kinds of pseudostates are listed in Table 5.7 and all the important fields on *PseudoState* is explained in Table 5.8 on the next page.

Table 5.7: Table of the different kinds of PseudoStateKind

`PseudoStateKind.initial`	Represents a "initial" vertex node.
`PseudoStateKind.deepHistory`	Represents a "deepHistory" vertex node.
`PseudoStateKind.shallowHistory`	Represents a "shallowHistory" vertex node.
`PseudoStateKind.join`	Represents a "join" vertex node.
`PseudoStateKind.fork`	Represents a "fork" vertex node.
`PseudoStateKind.junction`	Represents a "junction" vertex node.
`PseudoStateKind.choice`	Represents a "choice" vertex node.
`PseudoStateKind.entryPoint`	Represents a "entryPoint" vertex node.
`PseudoStateKind.exitPoint`	Represents a "exitPoint" vertex node.
`PseudoStateKind.terminate`	Represents a "terminate" vertex node.

Table 5.8: Table of the fields on PseudoState

`kind`	The kind of pseudostate this instance represents. Default is *PseudoStateKind.initial*
`stateMachine`	The containing state machine, if this pseudostate represents kind entryPoint or exitPoint.
`state`	The owning state.

5.2.11 Region

This class represents an containment of nodes (vertices) and edges (transitions). There might be sibling regions, but that part is further explained in the runtime section.

The most important fields are explained in Table 5.9.

Table 5.9: Table of the fields on Region

`stateMachine`	If this is not `null` then the region is owned by a state machine.
`state`	If this is not `null` then the region is owned by a state.
`subVertices`	The list of sub-vertices owned by this region.
`transitions`	The list of edges connection the sub-vertices in this region.

5.2.12 State

An object of the *State* class represents a state in the state machine graph, it can be any of the three types of possible states (simple, composite, orthogonal, see Table 5.10 on the next page).

It uses *Behavior* for entry, exit and do actions.

The most important fields are explained in Table 5.11 on the facing page.

Table 5.10: Table of the types of states

`simple`	This is a state with no regions. Use the `isSimple` method to test for this.
`composite`	This is a state with at least one region. Use the `isComposite` method to test for this.
`orthogonal`	This is a state with more than one concurrent regions. Use the `isOrthogonal` method to test for this.

Table 5.11: Table of the fields on State

`regions`	List of regions owned by this state.
`connections`	Connection points in and out of the submachine state.
`connectionPoints`	Connection points in and out of the composite state.
`submachine`	The state machine that this state represents.
`entry`	The *Behavior* to run when this state is entered.
`exit`	The *Behavior* to run when this state is exited.
`doActivity`	The *Behavior* to run while this state is active (this is a blocking method, use with caution).

5.2.13 StateMachine

This class represents a containment of regions. The graph API supports state machines both as a standalone machine, and when being used as a sub-machine. A default region will always be created when creating a state machine, and as such, the state machine can be seen as a composite state.

The most important fields on *State* is explained in Table 5.12 on the next page.

5.2.14 Transition and TransitionKind

This class represents edges (transitions) in the graph and are responsible for connecting the nodes in the graph. A transition always have a

Table 5.12: Table of the fields on StateMachine

`regions`	The regions owned by this state machine.
`connectionPoints`	When used as a submachine, these represents the point in and out the the state machine.
`submachineStates`	List of states where this machine is used as a submachine.

source and a target vertex, and together with triggers handles the transitions between them.

The most important fields are explained in Table 5.13 and the different kinds of *TransitionKind* are listed on Table 5.14.

Table 5.13: Table of the fields on Transition

`kind`	The kind of transition this is. The default is *TransitionKind.external*.
`source`	The source vertex for this transition.
`target`	The target vertex for this transition.
`container`	The containing region.
`guard`	The guard for this transition. Can be `null`.
`effect`	The effect for this transition. Can be `null`.
`triggers`	List of triggers for this transition.

Table 5.14: Table of the different kinds of TransitionKind

`TransitionKind.internal`	Represents a “internal” transition.
`TransitionKind.local`	Represents a “local” transition.
`TransitionKind.external`	Represents a “external” transition.

5.2.15 The XMI importer

To support rapid development of state machines, an XMI importer has been added to the framework. The importer supports most of the fea-

tures of XMI 2.1.1[3] (related to state machines), but have only been tested in Eclipse (Galileo) using the UML 2 Tools project. In theory, XMI files exported from other tools should work just as well, but that depends on how well they are following the standard.

The importer uses what it can from the XMI file, which means both id and name are taken from there. No support for sub-machines exists in the importer (in XMI these will span several files, and this is not currently supported).

The usage of the importer was made to be very simple, and should cause no problem as long as the XMI is standards compliant.

Listing 5.1: Example of Import class

```
1 StateMachine stateMachine = new Import().parse("/path/to/model.uml");
```

For a more elaborate example, please see the examples section 5.4 on page 60.

5.3 Runtime system

This section will explain all the runtime classes that are located in the *smlib.runtime.v2* package. These are all the classes that are responsible for traversing the state machine graph, and executing transitions, concurrency (emulated), guards, events, etc.

5.3.1 RTNode

This abstract class is the super class of every node in the runtime system. It's important to not confuse these nodes with nodes in the state machine graph, these nodes are only for the runtime system and so they do not follow any rules from the state machine specification.

It has several responsibilities which will be explained below:

modes

All nodes in the runtime system starts out as inactive, and the each individual nodes are themselves responsible for changing its mode.

[3] http://www.omg.org/spec/XMI/2.1.1/

This should not be confused with modes in a state machine, in the runtime system; all nodes (edges and nodes) can be active or not. In a state machine, only the states are active.

All the available modes are listed in table 5.15 on the next page.

stepping

The runtime system is using the method `step` for running the machine. This method is implemented in the *RTNode* class, and is responsible for running the correct methods in the node. If the current mode is inactive then `enter` will be run, if its active then `active` will be run, and if its leaving then `leave` will be run.

It should be noted that this does not represent a RTC step in the state machine, one RTC step can be made up of many runtime steps. Also, the actions `enter`, `active` and `leave` does not represent state behaviors, but rather the lifecycle of a node in the runtime system.

leveling

For supporting hierarchical machines, a simple level system has been implemented. The root machine is at level 0, and its region is at level 1. For every new node on top of this, the level indicator is incremented. The leveling is not something the user should need to worry about (but the current level can be queried if needed), but it is heavily used by the two leveling methods in *RTNode*.

levelUp This takes care of going from one level, and up to another level. The important part here is the stepping, which will make sure that every new level entered will have the correct mode. It will recurse until the current level equals the wanted level. This is often used for jumping into the wanted `resumeState`, which is used for history pseudostates.

levelDown This is the opposite of `levelUp`, and the main difference is that this function forcibly sets the mode on the node. This is for making sure that levels that are higher than the wanted level are made inactive.

For an example of levels, see Figure 5.1 on the facing page. In this

figure the state machine is at level 0, the two regions at level 1, and everything else at level 2.

Figure 5.1: Example for describing levels

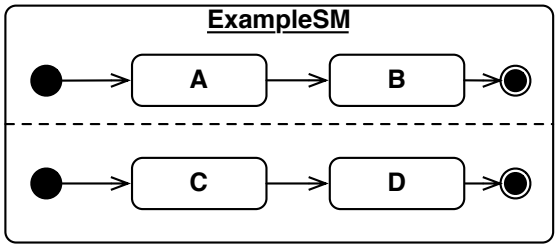


Table 5.15: Table of the possible modes of a Node

`RTNode.Mode.INACTIVE`	The node is currently not active. No stepping will be performed on this.
`RTNode.Mode.ACTIVE`	The node is currently active. When the machine is stepped, this node will be invoked.
`RTNode.Mode.LEAVING`	The node is currently leaving and the next step will make sure everything is cleaned up before entering state inactive.

5.3.2 RT

The *RT* class is the main runtime class. It implements the *Runnable* interface and is intended to be run in its own thread.

This class has three main responsibilities:

preparation / caching

When the state machine graph has been created and the state machine is validated, the next step is calling the `prepare` method. This method calls the method on every class in the runtime graph, and the responsibility of this method is making sure that every runtime class has been instantiated, and that all the static properties of the runtime has been cached. An example of this is the static decisions for a *Junction* node.

If the developer does not call this method explicitly it will be run when the runtime is started.

events
The runtime has a very simple system of events, and the *RT* class is responsible for this system. The queue itself is implemented as a *BlockingQueue* and is accessible for the user through the method `getEventQueue`.

main loop
As explained before, this class implements the *Runnable* interface and therefore has the method `run`. This method is responsible for the main loop for the system.

The main loop has three main concerns:

1. Make sure the preparation stage has been run, if not, run it.
2. Step through the machine until the machine is inactive or if the machine is terminating.
3. Check for new events in the event queue, and push these through the machine.

For an example of how to use this class, please see the example 5.4.1 on page 60.

5.3.3 RTConnectionPointReference

Placeholder class for future support of *ConnectionPointReference*.

5.3.4 RTFinalState

This marks the end of the region, and the implementation itself is mostly empty. When node is leaving, it will make sure that the `activeNodes` and the `resumeState` of the parent region is empty.

5.3.5 RTPseudoStateChoice

This pseudostate class will choose where to go in the state machine based on running the guards that are connected with the *Transition*

class. If the guard is one of the ones pre-defined in Table 5.4 on page 47 then the actual guard is not run (except for *Constraint.SWITCH* which must be evaluated on every turn).

5.3.6 RTPseudoStateDeepHistory

This class is responsible for resuming a suspended node, or if there are no suspended nodes available, it will find the initial pseudostate for the region and start this.

5.3.7 RTPseudoStateEntryPoint

Placeholder class for future support of *PseudoStateKind.entryPoint*.

5.3.8 RTPseudoStateExitPoint

Placeholder class for future support of *PseudoStateKind.exitPoint*.

5.3.9 RTPseudoStateFork

This class is responsible for starting several concurrent regions. In the `active` part, it will create the list of target states, and these will be set as targets when the `leave` method runs.

5.3.10 RTPseudoStateInitial

Since only one transition out is legal from this, when the method enter the `leave` method it will set this as the `activeNode`.

5.3.11 RTPseudoStateJoin

This class sits and waits for all incoming transitions from two or more concurrent regions (started with *RTPseudoStateFork*) before continuing.

5.3.12 RTPseudoStateJunction

This class decides (statically) which transition to use during the `prepare` method.

5.3.13 RTPseudoStateShallowHistory

This class is responsible for resuming a suspended node, or if there are no suspended nodes available, it will find the initial pseudostate for the region and start this.

5.3.14 RTPseudoStateTerminate

This class has one responsibility, and that is setting the `shouldTerminate` flag on the *RTNode* class. This will make the main loop exit during the next iteration.

5.3.15 RTRegion

The *RTRegion* class is one of the main classes in the runtime system, and has several responsibilities:

prepare
: In the preparation stage, this class is responsible for creating all the nodes that it contains, and also calling the `prepare` method on these nodes.

active
: In this stage, the class makes sure that every active node in the region is stepped.

leave
: This means that the region is finished, so all active nodes are cleared.

resumeState
: The `resumeState` is always pointing to the last touched node. If this is `null` then the final state has been reached, and there is nothing to resume.

5.3.16 RTStateComposite

This class is responsible for handling of concurrent regions. During the `enter` phase it will make sure that all regions are have their `targetStates` correctly set. In the `active` phase, it will step through all the regions,

and also make sure that if all the regions have finished, then exit out of this state. This class is used for all states that have one or more regions.

5.3.17 RTStateMachine

This class is responsible for stepping one or more regions that are directly contained in the *StateMachine*.

5.3.18 RTStateSimple

This class is responsible for executing the behavior that is associated with the state. It also checks the guards on the outgoing transitions, and will block the transition if the guard does not pass.

One more important feature of this class is that during the `leave` stage, it will set itself as the `resumeState`. This makes sure that if the transition crosses the level boundary, it will resume the correct state if one of the history pseudostate nodes are entered.

5.3.19 RTStateSubmachine

Placeholder class for future support of *State* of type submachine.

5.3.20 RTTransitionExternal

The main responsibilities of this class is to take care of leveling when the transition is from one level to another level, it does this by checking the current and the target level, and then using the `levelUp` / `levelDown` methods (from *RTNode*) to do the actual leveling.

If the target of the transition is a *RTPseudoStateJoin* then it will increase the incoming count on this node, since this node waits for all incoming before continuing.

5.3.21 RTTransitionInternal

Placeholder class for future support of *TransitionKind.internal*.

5.3.22 RTTransitionLocal

Placeholder class for future support of *TransitionKind.local*.

5.4 Examples

This section will consist of a series of examples of creating UML State Machines by code. We will also look into using the XMI 2.1 importer.

5.4.1 Standard setup of the runtime system

Before showing how to create state machines using the framework, a basic setup for running the examples will be shown. All the examples are built around the same *Event* called `StringEvent` which is just a simple sub-class of *Event* which uses strings as the input event-change. The events themselves will be pushed into the system by using a simple `Console`[4].

Listing 5.2: Basic runtime setup

```
1 RT runtime = new RT(stateMachine);
2 runtime.prepare();
3
4 Thread runtimeThread = new Thread(runtime);
5 runtimeThread.start();
6
7 Console c = System.console();
8 BlockingQueue<Event> eventQue = runtime.getEventQue();
9
10 while (runtimeThread.isAlive()) {
11   Thread.sleep(200);
12
13   System.out.print("> ");
14   String e = (c == null ? "" : c.readLine());
15
16   if (e.isEmpty()) {
17     continue;
18   }
19
20   eventQue.add(new StringEvent(e));
21 }
```

The basic steps for using the runtime system are simple and always follows the same pattern.

1. Create an instance of *smlib.runtime.v2.RT*

[4] `http://download.oracle.com/javase/6/docs/api/java/io/Console.html`

2. Create a *Thread* [5] for the runtime instance.

3. Start this thread. The machine will now run until it needs an event input.

4. In your main thread, add code for handling events from your system (in our case it will always be from the console, and always strings). Get the event queue, and start giving it your events.

5. When your runtime thread is dead, your machine has now reached a *FinalState* from where it can now longer continue.

5.4.2 A basic state machine

Please see section 2.3 on page 19 for a description of the semantics and the constraints for this state machine, and see Figure 2.9 on page 20 for a graphical representation.

Listing 5.3: BasicSM.java

```
1 StateMachine stateMachine = new StateMachine("Basic");
2 Region r = stateMachine.getRegions().get(0);
3
4 PseudoState start = new PseudoState(r);
5 State idle = new State(r);
6 FinalState end = new FinalState(r);
7
8 Transition t0 = new Transition(start, idle, r);
9 Transition t1 = new Transition(idle, end, r);
10
11 new Trigger(t1, new StringEvent("end"));
12
13 stateMachine.validate();
```

Line 1 Creates a new empty state machine with one region.

Line 2 Sets a local variable to point to region 0.

Lines 4 - 6 Creates the initial state, idle state, and final state with region 0 as the container.

[5] http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html

Lines 8 - 9 Sets up connection from start to idle and from idle to end, both using region 0 as their container.

Line 11 Creates a new trigger for `StringEvent` “end” that is connected to transition t1 (from idle state to end state).

5.4.3 A switch state machine

Please see section 2.4 on page 20 for a description of the semantics and the constraints for this state machine, and see Figure 2.10 on page 21 for a graphical representation.

Listing 5.4: OnOffSM.java

```
1 StateMachine stateMachine = new StateMachine("OnOff");
2 Region r = stateMachine.getRegions().get(0);
3
4 PseudoState start = new PseudoState(r);
5 State on = new State(r);
6 State off = new State(r);
7 FinalState end = new FinalState(r);
8
9 Transition t0 = new Transition(start, off, r);
10 Transition t1 = new Transition(off, on, r);
11 Transition t2 = new Transition(on, off, r);
12 Transition t3 = new Transition(off, end, r);
13
14 new Trigger(t1, new StringEvent("on"));
15 new Trigger(t2, new StringEvent("off"));
16 new Trigger(t3, new StringEvent("end"));
17
18 stateMachine.validate();
```

Line 1 Creates a new empty state machine with one region.

Line 2 Sets a local variable to point to region 0.

Lines 4 - 7 Creates the initial state, on state, off state, and the final state with region 0 as their container.

Lines 9 - 12 Setup connections from initial state to off state, off state to on state, on state to off state, and off state to final state.

Line 14 Creates a new trigger for `StringEvent` “on” that is connected to transition t1 (from off state to on state).

Line 15 Creates a new trigger for `StringEvent` “off” that is connected to transition t2 (from on state to off state).

Line 16 Creates a new trigger for `StringEvent` “end” that is connected to transition t3 (from off state to end state).

5.4.4 A choice state machine

Please see section 2.5 on page 22 for a description of the semantics and the constraints for this state machine, and see Figure 2.11 on page 22 for a graphical representation.

Listing 5.5: ChoiceSM.java

```
1 StateMachine stateMachine = new StateMachine("Choice");
2 Region r = stateMachine.getRegions().get(0);
3
4 PseudoState start = new PseudoState(r);
5 PseudoState choice = new PseudoState(PseudoStateKind.choice, r);
6 State idle = new State(r);
7 FinalState end = new FinalState(r);
8
9 Transition t0 = new Transition(start, choice, r);
10 Transition t1 = new Transition(choice, end, r);
11 Transition t2 = new Transition(choice, idle, r);
12 Transition t3 = new Transition(idle, choice, r);
13
14 new Trigger(t3, new StringEvent("end"));
15
16 t2.setGuard(Constraint.SWITCH);
17 t1.setGuard(Constraint.ELSE);
18
19 stateMachine.validate();
```

Line 1 Creates a new empty state machine with one region.

Line 2 Sets a local variable to point to region 0.

Lines 4 - 7 Creates the initial pseudostate, choice pseudostate, idle state, and the final state.

Lines 9 - 12 Setup connections from initial pseudostate to choice pseudostate,choice pseudostate to final state, choice pseudostate to idle state, and from idle state to choice pseudostate.

Line 14 Creates a new trigger for `StringEvent` “end” that is connected to transition t3 (from idle state to choice pseudostate).

Lines 16 - 17 Setup a guard on t1 and t2, the *Constraint.SWITCH* is false initially and then turns true, the *Constraint.ELSE* pathway is chosen when the switch is false.

5.4.5 A forking state machine

Please see section 2.6 on page 23 for a description of the semantics and the constraints for this state machine, and see Figure 2.12 on page 23 for a graphical representation.

Listing 5.6: ForkOrthogonalSM.java

```
1  StateMachine stateMachine = new StateMachine("ForkOrthogonal");
2  Region r = stateMachine.getRegions().get(0);
3
4  PseudoState start = new PseudoState(r);
5  State orthogonal = new State(r);
6  FinalState end = new FinalState(r);
7
8  PseudoState fork = new PseudoState(PseudoStateKind.fork, r);
9  PseudoState join = new PseudoState(PseudoStateKind.join, r);
10
11 Region c_r0 = new Region(orthogonal);
12 State c_r0_idle1 = new State(c_r0);
13 State c_r0_idle2 = new State(c_r0);
14
15 Transition c_r0_t2 = new Transition(c_r0_idle1, c_r0_idle2, c_r0);
16 c_r0_t2.getTriggers().add(new Trigger(new StringEvent("continue1")));
17
18 Region c_r1 = new Region(orthogonal);
19 State c_r1_idle1 = new State(c_r1);
20 State c_r1_idle2 = new State(c_r1);
21
22 Transition c_r1_t2 = new Transition(c_r1_idle1, c_r1_idle2, c_r1);
23 c_r1_t2.getTriggers().add(new Trigger(new StringEvent("continue2")));
24
```

```
25 Transition t1 = new Transition(start, fork, r);
26 Transition t2 = new Transition(join, end, r);
27
28 Transition t3 = new Transition(fork, c_r0_idle1, c_r0);
29 Transition t4 = new Transition(fork, c_r1_idle1, c_r1);
30
31 Transition t5 = new Transition(c_r0_idle2, join, r);
32 Transition t6 = new Transition(c_r1_idle2, join, r);
33
34 stateMachine.validate();
```

Line 1 Creates a new empty state machine with one region.

Line 2 Sets a local variable to point to region 0.

Line 4 Defines the starting node for this state machine.

Line 5 Defines the state that will later be expanded into an orthogonal state.

Line 6 Defines the ending node for this state machine.

Line 8 Defines a forking node.

Line 9 Defines a join node.

Lines 11 - 23 Defines the two regions that will make up the orthogonal state. Both regions is made up of two states, and the transition between them is triggered by the `StringEvent` "continue1" for region 0, and "continue2" for region 1.

Lines 25 - 26 Connects the initial node with the forking node, and the join node with the final node.

Lines 28 - 29 Connects the two outgoing transitions comming out of the fork, and into their places in the orthogonal state.

Lines 31 - 32 Connects the two transitions comming from the two states in the orthogonal state.

5.4.6 A deep history based state machine

Please see section 2.7 on page 24 for a description of the semantics and the constraints for this state machine, and see Figure 2.13 on page 25 for a graphical representation.

Listing 5.7: DeepHistorySM.java

```
1 StateMachine stateMachine = new StateMachine("DeepHistory");
2 Region r = stateMachine.getRegions().get(0);
3
4 PseudoState start = new PseudoState(r);
5 State composite = new State(r);
6 State pause = new State(r);
7 PseudoState history = new PseudoState(PseudoStateKind.deepHistory, r)
      ;
8 FinalState end = new FinalState(r);
9
10 Transition t0 = new Transition(start, composite, r);
11 Transition t1 = new Transition(composite, end, r);
12 Transition t2 = new Transition(composite, pause, r);
13 Transition t3 = new Transition(pause, history, r);
14 Transition t4 = new Transition(history, start, r);
15
16 Region c_region = new Region(composite);
17
18 PseudoState c_start = new PseudoState(c_region);
19 State c_idle1 = new State(c_region);
20 State c_idle2 = new State(c_region);
21 FinalState c_end = new FinalState(c_region);
22
23 Transition c_t0 = new Transition(c_start, c_idle1, c_region);
24 Transition c_t1 = new Transition(c_idle1, c_idle2, c_region);
25 Transition c_t2 = new Transition(c_idle2, c_end, c_region);
26
27 new Trigger(c_t1, new StringEvent("continue1"));
28 new Trigger(c_t2, new StringEvent("continue2"));
29
30 new Trigger(t1, new StringEvent("finish"));
31 new Trigger(t2, new StringEvent("suspend"));
32 new Trigger(t3, new StringEvent("resume"));
33
34 stateMachine.validate();
```

Line 1 Creates a new empty state machine with one region.

Line 2 Sets a local variable to point to region 0.

Lines 4 - 8 Creates the initial pseudostate, composite state, pause state, deep history pseudostate, and the final state.

Lines 10 - 14 Setup connections from initial pseudostate to composite state, composite state to final state, composite state to pause state, pause state to deep history pseudostate, deep history to initial pseudostate.

Line 16 Creates a new region in the "composite" state, this converts the state from a simple state to a composite state.

Lines 18 - 25 Creates and connect a simple composite state with nodes going from initial to idle1, idle2 and ending with the final state.

Line 27 Creates a new trigger for `StringEvent` "continue1" that is connected to transition c_t1 (from c_idle1 state to c_idle2 state).

Line 28 Creates a new trigger for `StringEvent` "continue2" that is connected to transition c_t2 (from c_idle2 state to c_end state).

Line 30 Creates a new trigger for `StringEvent` "finish" that is connected to transition t1 (from composite state to end state).

Line 31 Creates a new trigger for `StringEvent` "suspend" that is connected to transition t2 (from composite state to pause state).

Line 32 Creates a new trigger for `StringEvent` "resume" that is connected to transition t3 (from pause state to start pseudostate).

5.4.7 Using the XMI importer

This example will be based on Figure 2.10 on page 21 from one of the previous examples. Only this time the diagram will be loaded at runtime using the *xmi.Import* class. For convenience you should always use names on your models since this will make it easier to get nodes out of the graph. To set the name, right click on a diagram element and select "Show properties view", here you can set the wanted name.

Listing 5.8: OnOffXMI.java

```
1 StateMachine stateMachine = new Import().parse("/OnOff.uml");
2 Region r0 = stateMachine.getRegions().get(0);
3
4 Transition t1 = r0.getTransition("t1");
5 Transition t2 = r0.getTransition("t2");
6 Transition t3 = r0.getTransition("t3");
7
8 new Trigger(t1, new StringEvent("on"));
9 new Trigger(t2, new StringEvent("off"));
10 new Trigger(t3, new StringEvent("end"));
11
12 stateMachine.validate();
```

Line 1 Creates a new state machine based on the XMI model "OnOff.uml".

Line 2 Sets a local variable to point to region 0.

Line 4 - 6 Creates local variables for the transitions on the graph.

Line 8 Creates a trigger on the event "on", and adds this to the transition t1.

Line 9 Creates a trigger on the event "off", and adds this to the transition t2.

Line 10 Creates a trigger on the event "end", and adds this to the transition t3.

5.5 Conclusion

Creating a framework for UML state machines where you are constrained by strict adherence to the UML specification is a task that is both feasible and frustrating. Feasible since the OO design of your classes has already been made for you, and you are left with very little choice when nodes in the graph are implemented. It has also been a frustrating task, since the specification have not been designed for simplicity in mind, and the framework get unnecessary complex at several points:

Behaviors as classes

The behaviors in the system (entrys, exits, etc.) are all implemented

by means of the the command pattern [5], and clutters the user code with an unnecessary amount of classes.

In the extreme, every single transition and state in the state machine could have 1 - 3 classes associated with them, this is not a problem in a state machine with few states, but looking at more complex systems with maybe hundreds of states it very soon get unnecessarily complex.

Constraints as classes

The same problem relates to constraints, what should be a simple one-line evaluation of the current state of the state machine turns into yet another class.

Even with these added complexities the task was mostly completed, and the framework enabled the implementation of most of the UML state machine specification as shown in the examples in section 5.4 on page 60.

Another bi-product of using the UML specification directly is tool support. Using XMI files as a possible means for creating state machines, and since the XMI file format is used natively by Eclipse UML2 Tools project (other tools also export support for this format), state machines created with this tool can be directly imported into the state machine framework.

This has several advantages:

Rapid development

Using XMI files at the interchange format between the framework and the modeling tool opens up the possibility for rapid development, a composite state with several sub-states can be easily created in the tool and gives a clear overview of the state machine.

With todays tools, some parts of the state machines would still have to be hard-coded, but this is getting better all the time, and a framework plugin for Eclipse would be possible to complement the support already there.

Enabling the involvement of the domain experts

By using a more graphical approach to state machines, the domain

experts can be much more involved in the design of the machine, and can even change parts of the state machine if it is needed without ever touching any code.

Documentation

An Achilles' heel of any software frameworks is the level of focus on documentation, and in modeling this is and even bigger problem since there should always be a one-to-one relation between the model and the code.

The model serves as both a reference for implementers, and documentation for the system at large. The problem arises when the code moves away from the model, and discrepancies makes the two move more and more apart.

A solution to this has for a long time been to generate code from the model, but the same problem can exist here, what if you are not happy with the generated code and start editing it?

This problem can be solved using the model as the basis of the state machine nodes, and the developer can use his or hers time instead implementing the code that uses the state machine.

Even without using the graphical approach, the framework has shown to be a good solution to implementing state machines without getting too much in the way. Besides the obvious documentation disadvantages of not using a model, the framework is very usable and get the job done nicely.

Chapter 6

Extended Java

6.1 Introduction

This chapter will explore the possibilities for extending the Java language with state machine specific keywords related to the UML State Machine specification. It will start by looking into the tools used for actually extending the Java language, and then go on to the implementation of the state based extension.

At the end of the chapter more examples in the language will be shown, and a conclusion drawn.

6.2 Tools

For extending Java, there are several tools available, and also at least two ways of doing the actual implementation. These two approaches will be explained here together with the reasoning for the choice made.

6.2.1 Bytecode implementation

The first approach is a bytecode implementation of the state machine language. This would give us full access to every low level instruction in the Java language, and would end up in a implementation that would be tightly coupled with the Java language.

This approach was considered for a while using excellent libraries

like BCEL[1], but this would also introduce a massive complexity to the implementation, especially considering the tools that was chosen for the task (see section 6.2.3) only parses the language, and does not come with any kind of support for generating bytecode for the language. The implementation would then have to create a complete template for every part of the Java language. Even if a subset was chosen, it would take too much focus away from the actual task, which was defining a language extension for Java.

6.2.2 Source-to-source translation

Another approach to extending a language is using source-to-source approach where one would write in the extended language, and when it was time for compiling the source code, the source would first be preprocessed with a tool to create a source that is compilable with the standard language tools accompanying the language. Since a framework for state machines has already been created (see section 5 on page 44) this approach was soon considered a far better option than the bytecode implementation since it would allow for focus on the extension of the language, and not on the language itself.

Another similar approach was used when the Java language in version 1.5 implemented generics to the language. The approach they choose was a source-to-source implementation, coupled with a type erasure process [2].

6.2.3 ANTLR v3 and StringTemplate

There are several tools available for writing scanners and parsers, most of them have been around for decades and has been doing a good job at generating scanners and parsers. The tool that was chosen, was the ANTLR v3[3] scanner / parser generator which describes itself as:

> "ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recog-

[1] `http://jakarta.apache.org/bcel/`
[2] `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`
[3] `http://www.antlr.org`

> nizers, compilers, and translators from grammatical descriptions containing actions in a variety of target languages."

There are several reasons for choosing this tool, but the important parts are:

- Support for extended Backus-Naur form (EBNF), which is a notation for describing languages. EBNF has the advantage over BNF (that a lot of other tools support) that is has options and repetitions.
- Good 3rd party integration with the Eclipse IDE. The tool used for this thesis is ANTLRide[4].
- Template language that has deep integration with the ANTLR grammar file. This template language is called StringTemplate[5] and also has good documentation.
- A mode called "rewrite" which basically means that it can target specific parts of the grammar, and at the same time let everything else be just as it is. This is perfect for doing source to source translation, since only the interesting bits are targeted (every rule related to the smjava language).
- Active community, which means plenty of freely available documentation and grammar files.
- In the source distribution, a grammar file for Java 1.5 is included and this was the basis for the `smjava` implementation.

6.3 Design of the language

The design of the state-based Java language proved to be a simple task, since the actual structure of the language was already described in the specification. Given a `region` there is only a finite set of elements that can follow (transitions, states, etc.).

[4] `http://antlrv3ide.sourceforge.net/`
[5] `http://www.stringtemplate.org`

Not every feature of the UML specification has been implemented, but rather a subset that was considered essential. Further feature development should be easy to implement, considering that the foundation has already been implemented.

The name *smjava* was decided as the name of the language extension, and the filename extension `smjava` was chosen to separate it from normal Java files.

6.3.1 smjava keywords

region

A `region` block defines a region in the state machine, this is an important keyword as it declares that a class is state machine controlled. If one or more of these are present in a class, it will be considered a state machine. There are two places where a `region` can be defined, after the class block starts, and after the state block starts.

If more than two regions are defined in the same block, they will be run concurrent.

After the keyword and before the block, you are allowed to enter an identifier if desired, they will not have any affect on the operation, but can serve as documentation.

Example usage:

Listing 6.1: The two points where region can be defined

```
1    class TestRegionSM {
2      region {
3        state s { region { ... } }
4      }
5      region RegionWithID { ... }
6    }
```

state

The `state` keyword defines a state in the state machine and can be defined in every `region` block. If a `state` also has a block defined, it can also include `entry` and `exit` actions. If a `state` contains one

region, it is considered an composite state and if it has more than one regions, it is considered a orthogonal state.

Example usage:

Listing 6.2: Example usage of the state keyword

```
1   region {
2     state simpleState;
3     state simpleState2 {
4       entry { ... }
5       exit { ... }
6     }
7     state compositeState {
8       region { ... }
9     }
10    state orthogonalState {
11      region { ... }
12      region { ... }
13    }
14  }
```

entry and exit

The `entry` and `exit` keywords are used to defined methods that are run whenever a state is entered or exited. They can only be defined in a state block.

For example usage see the examples from `state` on page 74.

transition(from, to)

The `transition` keyword is used to defined a transition from one vertex, to another vertex in the state machine.

Listing 6.3: Example usage of the transition keyword

```
1   region {
2     state on; state off;
3     transition(on, off) {
4       trigger("off");
5       effect { ... }
6       guard { ... }
7     }
8     transition(off, on) { trigger("on"); }
9   }
```

The parameters `to` and `from` maps the source and target of the vertex class in UML state machines.

trigger, effect, and guard

The `trigger`, `effect` and `guard` keywords are all used to defined extended functionality in a `transition`. `trigger` for defining a triggering event for this `transition`. `effect` for a `transition` effect. `guard` for checking whether this transition should occur.

psinitial

The `psinitial` keyword is used to define a starting point in a region. It can only be defined inside a `region` block.

finalstate

The `finalstate` keyword is used to define a ending point for a region. It can only be defined inside a `region` block.

pshistory and psdeephistory

The `pshistory` and `psdeephistory` keywords are used to define shallow and deep history in the state machine. They can both be defined at most one time per region block.

6.3.2 API for interfacing with state machine based classes

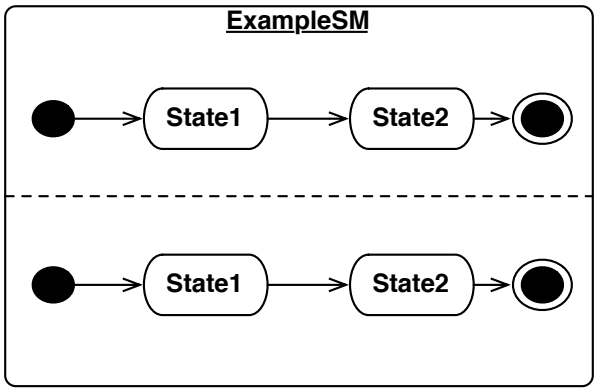
The interface that is created for state machines is very simple, and should provide all that the developer needs to care about. The interface is exposed through the `sm` field in a class that is defined as a state machine, this can be seen in Figure 6.1 on the next page.

The interface is defined as:

Listing 6.4: Interface for state machines

```
1 interface statemachine
2 {
3   public void start();
4   public boolean isAlive();
5   public void push(Event e);
6 }
```

Figure 6.1: The connection between the Context class and the statemachine interface



start

Starts the state machine. There is no `stop` method, since the state machine will run to completion.

isAlive

Is the state machine currently running? After the state machine has been started, this can be used to tell when the state machine has finished.

push(Event e)

Push an event into the state machine.

Example usage of this API can be found in section 6.5 on page 93.

6.4 Implementation

This section will describe all the grammar and tools related to the smjava implementation. A simple tool for rewriting the smjava files will also be explained.

6.4.1 SMJavaRewriter - A preprocessor tool for smjava

The *SMJavaRewriter* tool can be found in the package *net.mortenoh.smjava* package and is a tool for rewriting smjava files to their Java equivalent.

The rewriter uses the smlib framework (see chapter 5 on page 44) for creating and connecting the state machine that was described by the smjava file.

A shell-script for running the rewriter is included in the *source/smjava* directory, and is called `smjavac`. The tool is run with `./smjavac <filename>` and the output is a file with the smjava extension replaced by a Java extension.

6.4.2 Identifiers in the converted source

To avoid namespace collision, all the identifiers in the translated source is generated from UUID. So the generated parser is augmented with a `createUUID` method:

```
1 @parser::members {
2   protected String createUUID() {
3     return "id" + Math.abs(UUID.randomUUID().getMostSignificantBits()
          );
4   }
5 }
```

This method generates a random UUID and gets the most significant bits in it (since the actual UUID contains dashes). This number is then prefixed with “id”.

6.4.3 The augmented classBodyDeclaration rule

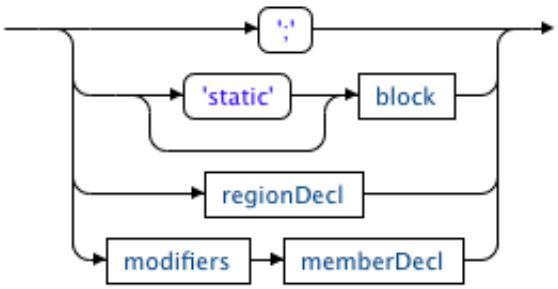
The first rule we are going to look at, is the point at where classes are declared in the grammar. This declaration is the `classBodyDeclaration`.

Listing 6.5: Augmented classBodyDeclaration

```
1 classBodyDeclaration
2     :   ';'
3     |   'static'? block
4     | regions+=regionDecl+
5       -> statemachine(regions={$regions})
6     |   modifiers memberDecl
7     ;
```

This is the augmented version where support for regions are added. Regions will be built up as a list and sent to the template as a parameter called regions.

Figure 6.2: Railroad diagram for classBodyDeclaration



The template definition for `statemachine(region)` is the largest of the templates, and is also the starting point for generating the rewritten code.

The steps in the template are:

1. Define the `statemachine` interface.
2. Define an implementation of the `statemachine` interface. This implementation is called `_impl_statemachine`.
3. This class has two important methods (described below) that are called from the constructor of the class, and also includes the methods from the `statemachine` interface. These methods will not be described since they are directly calling methods from my state machine framework.

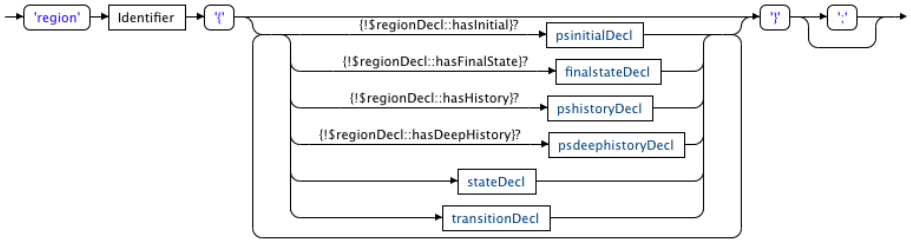
   _create_statemachine This method is responsible for using the region list (which was given as a argument) for calling the region-template in a iterative way (basically a for-loop).

   _create_runtime This method is responsible for creating a new instance of the runtime class, and also creating a new thread (but not starting it) with the runtime as the *Runnable*.
4. Define the field `sm` which is an instance of the _impl_statemachine class described before. The field uses the `statemachine` interface as the type.

6.4.4 Rule regionDecl

The first rule we are going to dissect is the `regionDecl` rule, this rules defines the `region` keyword.

Figure 6.3: Railroad diagram for regionDecl



Listing 6.6: regionDecl

```
1 regionDecl
2   scope {
3     java.util.Map<String, String> vertexMap;
4     boolean hasInitial;
5     boolean hasFinalState;
6     boolean hasHistory;
7     boolean hasDeepHistory;
8   }
```

We start by defining 5 scope-based fields. By defining them as scope-based we make sure that the actual fields are defined in the `regionDecl` part of the generated parser.

The fields are responsible for:

Listing 6.7: regionDecl

```
1   @init {
2     $regionDecl::vertexMap = new java.util.LinkedHashMap<String,
          String>();
3     $regionDecl::hasInitial = false;
4     $regionDecl::hasFinalState = false;
5     $regionDecl::hasHistory = false;
6     $regionDecl::hasDeepHistory = false; }
```

Since the scope-based fields are used for every `regionDecl`, they need to be re-initialized every time a `regionDecl` rule is entered.

Table 6.1: Scope fields for regionDecl

vertexMap	This is used as a symbol table for making sure that the to and from arguments on transitions are defines as vertices.
hasInitial	This field is for making sure that there are at most one psinitial state in the region.
hasFinalState	This field is for making sure that there are at most one finalstate state in the region.
hasHistory	This field is for making sure that there are at most one pshistory state in the region.
hasDeepHistory	This field is for making sure that there are at most one psdeephistory state in the region.

Listing 6.8: regionDecl

```
1  : 'region' ID=Identifier? '{'
2      (
3          {!$regionDecl::hasInitial}? psinitial=psinitialDecl {
               $regionDecl::hasInitial = true;}
4        | {!$regionDecl::hasFinalState}? finalstate=finalstateDecl {
             $regionDecl::hasFinalState = true;}
5        | {!$regionDecl::hasHistory}? pshistory=pshistoryDecl {
             $regionDecl::hasHistory = true;}
6        | {!$regionDecl::hasDeepHistory}? psdeephistory=
             psdeephistoryDecl {$regionDecl::hasDeepHistory = true;}
7        | states+=stateDecl
8        | transitions+=transitionDecl
9      )*
10   '}' ';'?
11   -> template(id={createUUID()}, psinitial={$psinitial.st},
        finalstate={$finalstate.st},
12         pshistory={$pshistory.st}, psdeephistory={$psdeephistory.st
               },
13         states={$states}, transitions={$transitions}) ""
14 ;
```

This is the grammar part of the regionDecl rule. It is mostly straight forward, but some care has been made to disable the possibility of more than one of certain keywords.

Line 1 Defines the region keyword, with an optional identifier.

Lines 3 - 6 These lines are basically the same, so only one of them will be described. The line starting on line 3 uses what in ANTLR is called a predicate test, and in this case it simple tests to check whether the `hasInitial` flag is true. If it is, throw a `FailedPredicateException`. If the field is not true, go on matching the `psinitialDecl` declaration. After the match is finished, set the `hasInitial` to true. This makes sure that the keywords that are only allowed one match, are correctly rejected by the parser if more than one of them appears.

Lines 7 - 8 Builds up a list of `stateDecl` and `transitionDecl`.

Lines 10 - 14 Inline template that sends all the synthesized templates one level up (to the `classBodyDeclaration`).

Listing 6.9: regionDecl

```
1 catch [FailedPredicateException e] {
2   StringBuffer b = new StringBuffer();
3   b.append("ERR: Syntax error in line " + e.line + " at position " +
        e.charPositionInLine + ".\n");
4   b.append("ERR: Failed predicate: " + e.predicateText + ".");
5   b.append("\nTIP: Remember, psinitial, finalstate, pshistory and
        psdeephistory can only appear once per region.\n");
6   System.err.println(b.toString());
7   System.exit(-8);
8 }
```

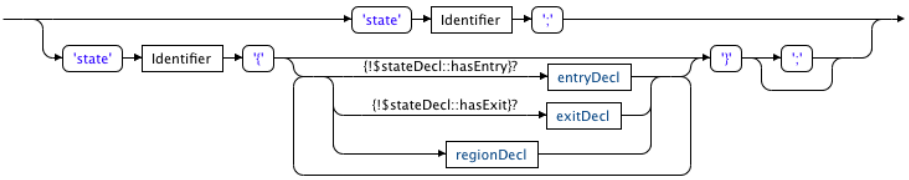
When a predicate test fails, the standard ANTLR way to handle it is to backtrack and try to continue. This behaviour doesn't work well in the rewrite mode, since we are ending up with an output that is essentially missing all the parts where the predicate failed (and the parser itself returns 'ok').

The way this is handled is by catching the *FailedPredicateException* exception, and then write some debug information out to the user, and then quit the parser. In a full featured parser this would not suffice, but the focus is on the language and not the tools.

6.4.5 Rule stateDecl

The next rule we are looking into is the `stateDecl` rule, it is the rule that handles all variants of the `state` keyword.

Figure 6.4: Railroad diagram for stateDecl



Listing 6.10: stateDecl

```
1 stateDecl
2   scope {
3     boolean hasEntry;
4     boolean hasExit;
5   }
```

These scope-based fields are used in the predicate tests to make sure that there will at most be only one of `entry` and `exit` keywords.

Listing 6.11: stateDecl

```
1   @init {
2     $stateDecl::hasEntry = false;
3     $stateDecl::hasExit = false;
4   }
```

Initialize the scope-based fields to their default value (false).

Listing 6.12: stateDecl

```
1   : 'state' ID=Identifier ';'
2     { $regionDecl::vertexMap.put($ID.text, createUUID()); }
3     -> template(id={$regionDecl::vertexMap.get($ID.text)}) ""
```

In this first `state` rule, we declare a `state` keyword that can only have an identifier. After this rule has been matched, the identifier is put into the scope-based `vertexMap`, so that it can be matched later when transitions are declared.

Listing 6.13: stateDecl

```
1 | 'state' ID=Identifier '{'
2     (
```

```
3         {!$stateDecl::hasEntry}? entry=entryDecl {$stateDecl::
              hasEntry = true;}
4       | {!$stateDecl::hasExit}? exit=exitDecl {$stateDecl::hasExit =
            true;}
5       | regions+=regionDecl
6     )*
7   '}' ';'?
8   { $regionDecl::vertexMap.put($ID.text, createUUID()); }
9   -> template(id={$regionDecl::vertexMap.get($ID.text)},
10          regions={$regions},
11          entry={$entry.st},
12          exit={$exit.st}) ""
13 ;
```

In this second rule, we add blocks to the `state` keyword.

Lines 1 - 2 Same as rule #1, the `state` keyword followed by an identifier. We also start the block here.

Line 3 Has an `entry` action already been defined? if so, throw an *FailedPredicateException* and quit. If it has not been defined, match this rule and set `hasEntry` to true.

Line 4 Has an `exit` action already been defined? if so, throw an *FailedPredicateException* and quit. If it has not been defined, match this rule and set `hasExit` to true.

Line 5 Match one or more regions, build up as a list.

Line 7 End the block here, with an optional semicolon.

Line 8 Put the identifier into the scope-based `vertexMap`.

Lines 9 - 12 Synthesize the template for this grammar rule, use all collected attributes.

Listing 6.14: stateDecl

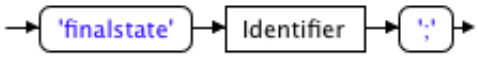
```
1 catch [FailedPredicateException e] {
2   StringBuffer b = new StringBuffer();
3   b.append("ERR: Syntax error in line " + e.line + " at position " +
        e.charPositionInLine + ".\n");
4   b.append("ERR: Failed predicate: " + e.predicateText + ".");
```

```
5   b.append("\nTIP: Remember, entry and exit can only appear once per
        state.\n");
6   System.err.println(b.toString());
7   System.exit(-8);
8 }
```

Same error situation as in the `regionDecl` rule.

6.4.6 Rule finalstateDecl

Figure 6.5: Railroad diagram for finalstateDecl



Listing 6.15: finalstateDecl

```
1 finalstateDecl
2   : 'finalstate' ID=Identifier ';'
3     {
4       $regionDecl::vertexMap.put($ID.text, createUUID());
5     }
6     -> template(id={$regionDecl::vertexMap.get($ID.text)}) ""
7   ;
```

Line 2 Declare the `finalstate` keyword with an identifier.

Line 4 Put the identifier into the `vertexMap`, mapped to an UUID.

Line 6 Synthesize the attributes for this level.

6.4.7 Rule entryDecl

Listing 6.16: entryDecl

```
1 entryDecl
2   : 'entry' b=methodBody ';'?
3     -> template(id={createUUID()},
4           body={$b.text}) ""
5   ;
```

Figure 6.6: Railroad diagram for entryDecl

Line 2 Defines the `entry` keyword with a `methodBody` and ending in an optional semicolon.

Lines 3 - 4 Synthesize attributes for this rule.

6.4.8 Rule exitDecl

Figure 6.7: Railroad diagram for exitDecl

Listing 6.17: exitDecl

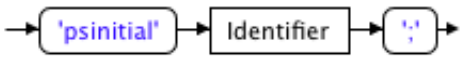
```
1 exitDecl
2   : 'exit' b=methodBody ';'?
3     -> template(id={createUUID()},
4           body={$b.text}) ""
5   ;
```

Line 2 Defines the `exit` keyword with a `methodBody` and ending in an optional semicolon.

Lines 3 - 4 Synthesize attributes for this rule.

6.4.9 Rule psinitialDecl

Figure 6.8: Railroad diagram for psinitialDecl



Listing 6.18: psinitialDecl

```
1 psinitialDecl
2   : 'psinitial' ID=Identifier ';'
3     {
4       $regionDecl::vertexMap.put($ID.text, createUUID());
5     }
6     -> template(id={$regionDecl::vertexMap.get($ID.text)}) ""
7   ;
```

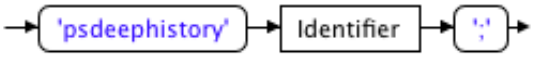
Line 2 Declare the `psinitial` keyword with an identifier and ending in a semicolon.

Line 4 Put the identifier into the `vertexMap`, mapped to an UUID.

Line 6 Synthesize the attributes for this rule.

6.4.10 Rule psdeephistoryDecl

Figure 6.9: Railroad diagram for psdeephistoryDecl



Listing 6.19: psdeephistoryDecl

```
1 psdeephistoryDecl
2   : 'psdeephistory' ID=Identifier ';'
3     {
4       $regionDecl::vertexMap.put($ID.text, createUUID());
5     }
6     -> template(id={$regionDecl::vertexMap.get($ID.text)}) ""
7   ;
```

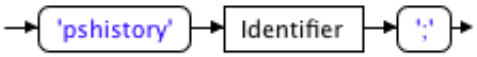
Line 2 Declare the `psdeephistory` keyword with an identifier and ending in a semicolon.

Line 4 Put the identifier into the `vertexMap`, mapped to an UUID.

Line 6 Synthesize the attributes for this rule.

6.4.11 Rule pshistoryDecl

Figure 6.10: Railroad diagram for pshistoryDecl



Listing 6.20: pshistoryDecl

```
1 pshistoryDecl
2   : 'pshistory' ID=Identifier ';'
3     {
4       $regionDecl::vertexMap.put($ID.text, createUUID());
5     }
6     -> template(id={$regionDecl::vertexMap.get($ID.text)}) ""
7   ;
```

Line 2 Declare the `pshistory` keyword with an identifier and ending in a semicolon.

Line 4 Put the identifier into the `vertexMap`, mapped to an UUID.

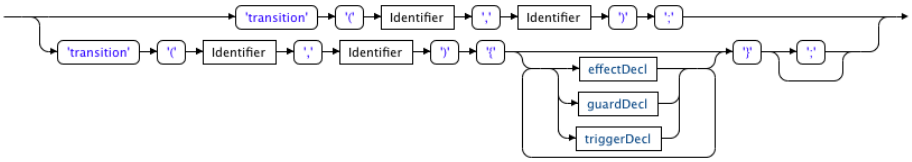
Line 6 Synthesize the attributes for this rule.

6.4.12 Rule transitionDecl

Listing 6.21: transitionDecl

```
1 scope {
2   boolean hasEffect;
3   boolean hasGuard;
4 }
```

Figure 6.11: Railroad diagram for transitionDecl



These scope-based fields are used in the predicate tests to make sure that there will at most be only one of `effect` and `guard` keywords.

Listing 6.22: transitionDecl

```
1 @init {
2   $transitionDecl::hasEffect = false;
3   $transitionDecl::hasGuard = false;
4 }
```

Initialize the scope-based fields to their default value (false).

Listing 6.23: transitionDecl

```
1  transitionDecl
2    : 'transition' '(' from=Identifier ',' to=Identifier ')' ';'
3      {
4        if(!$regionDecl::vertexMap.containsKey($from.text)) {
5          throw new RuntimeException("transition: state " + $from.text
               + " does not exist.");
6        }
7
8        if(!$regionDecl::vertexMap.containsKey($to.text)) {
9          throw new RuntimeException("transition: state " + $to.text +
               " does not exist.");
10       }
11     }
12     -> template(id={createUUID()},
13           from={$regionDecl::vertexMap.get($from.text)},
14           to={$regionDecl::vertexMap.get($to.text)}) ""
```

Line 2 Defines the `transition` keyword. This keyword has two parameters, one for where the transition start (source), and one for where the transition ends (target).

Lines 4 - 6 Checks the `vertexMap` to check if the `from` vertex has been defined, if it has not, throw a `RuntimeException` and exit.

Lines 8 - 10 Checks the `vertexMap` to check if the `to` vertex has been defined, if it has not, throw a `RuntimeException` and exit.

Lines 12 - 14 Synthesize the attributes for this rule.

Listing 6.24: transitionDecl

```
1 | 'transition' '(' from=Identifier ',' to=Identifier ')' '{' (
2       {!$transitionDecl::hasEffect}? effect=effectDecl {
            $transitionDecl::hasEffect = true;}|
3       {!$transitionDecl::hasGuard}?  guard=guardDecl {$transitionDecl
            ::hasGuard = true;}|
4       triggers+=triggerDecl
5   )* '}' ';'?
6   {
7     if(!$regionDecl::vertexMap.containsKey($from.text)) {
8       throw new RuntimeException("transition: state " + $from.text +
            " does not exist.");
9     }
10
11    if(!$regionDecl::vertexMap.containsKey($to.text)) {
12      throw new RuntimeException("transition: state " + $to.text + "
            does not exist.");
13    }
14  }
15  -> template(id={createUUID()},
16        from={$regionDecl::vertexMap.get($from.text)},
17        to={$regionDecl::vertexMap.get($to.text)},
18        effect={$effect.st},
19        guard={$guard.st},
20        triggers={$triggers}) ""
```

Line 1 Defines the `transition` keyword. This keyword has two parameters, one for where the transition start (source), and one for where the transition ends (target).

Line 2 Uses a predicate to see if the `hasEffect` flag has been set, if it has, then throw the *FailedPredicateException*. If not, match the `effectDecl` declaration and set the flag to true.

Line 3 Uses a predicate to see if the `hasGuard` flag has been set, if it has, then throw the *FailedPredicateException*. If not, match the `guardDecl` declaration and set the flag to true.

Line 4 Matches zero-or-more of `triggerDecl`, and creates a list of triggers.

Lines 7 - 9 Checks the `vertexMap` to check if the `from` vertex has been defined, if it has not, throw a *RuntimeException* and exit.

Lines 11 - 13 Checks the `vertexMap` to check if the `to` vertex has been defined, if it has not, throw a *RuntimeException* and exit.

Lines 15 - 20 Synthesize the attributes for this rule.

Listing 6.25: transitionDecl

```
1 catch [FailedPredicateException e] {
2   StringBuffer b = new StringBuffer();
3   b.append("ERR: Syntax error in line " + e.line + " at position " +
        e.charPositionInLine + ".\n");
4   b.append("ERR: Failed predicate: " + e.predicateText + ".");
5   b.append("\nTIP: Remember, effect and guard can only appear once
        per state.\n");
6   System.err.println(b.toString());
7   System.exit(-8);
8 }
```

6.4.13 Rule effectDecl

Figure 6.12: Railroad diagram for effectDecl

Listing 6.26: effectDecl

```
1 effectDecl
```

```
2   : 'effect' b=methodBody ';'?
3     -> template(id={createUUID()},
4           body={$b.text}) ""
5   ;
```

Line 2 Defines the `effect` keyword with a `methodBody` and ending in an optional semicolon.

Lines 3 - 4 Synthesize attributes for this rule.

6.4.14 Rule guardDecl

Figure 6.13: Railroad diagram for guardDecl

Listing 6.27: guardDecl

```
1 guardDecl
2   : 'guard' b=methodBody ';'?
3     -> template(id={createUUID()},
4           body={$b.text}) ""
5   ;
```

Line 2 Define the `guard` keyword with a `methodBody` and ending in an optional semicolon.

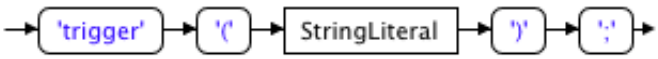
Lines 3 - 4 Synthesize attributes for this rule.

6.4.15 Rule triggerDecl

Listing 6.28: triggerDecl

```
1 triggerDecl
2   : 'trigger' '(' event=StringLiteral ')' ';'
3     -> template(id={createUUID()},
4           event={$event.text}) ""
5   ;
```

Figure 6.14: Railroad diagram for triggerDecl



Line 2 Define the `trigger` keyword with a `StringLiteral` as event, and ending in a semicolon. This will be translated into an `StringEvent` as explained before.

Lines 3 - 4 Synthesize attributes for this rule.

6.5 Examples

This section will consist of examples of basic usage of the `smjava` language.

6.5.1 Basic setup

All the examples in this section follows the same pattern, and therefore it will be explained once in this section and not included in the following examples. The examples will emulate pushing events into the machine using a simple *Queue*[6].

The basic setup is as follows:

Listing 6.29: Basic setup of smjava examples

```
1 import java.util.LinkedList;
2 import java.util.Queue;
3
4 import smlib.uml2.Event;
5 import smlib.uml2.StringEvent;
6
7 public class CLASS_NAME {
8   // INSERT EXAMPLE CODE HERE
9
10  public static void main(String[] args) throws Exception
11  {
```

[6] http://download.oracle.com/javase/6/docs/api/java/util/Queue.html

```
12      <{ CLASS_NAME }> class_instance = new <{ CLASS_NAME }>();
13      class_instance.sm.start();
14
15      Queue<Event> events = new LinkedList<Event>();
16      events.add(new StringEvent("event1"));
17      events.add(new StringEvent("event2"));
18      events.add(new StringEvent("event3"));
19
20      while(class_instance.sm.isAlive()) {
21        Event e = events.poll();
22
23        if(e != null) {
24          class_instance.sm.push(e);
25        }
26      }
27    }
28  }
```

Lines 1 - 5 Import all necessary packages.

Line 7 Define the class that has the state machine.

Line 8 This is the point where all the example code will be located.

Lines 12 - 13 Create a new instance of the class. Start the machine using the `start` method.

Lines 15 - 18 Define a set of events to be pushed into the state machine. The actual events will be different for every example.

Line 20 Defines a loop that will run until the state machine has completed, this is checked using the `isAlive` method.

Line 21
Get a new event from the event-queue.

Lines 23 - 25 If there was an event waiting, push this into the machine with the `push` method.

6.5.2 A basic state machine

Please see section 2.3 on page 19 for a description of the semantics and the constraints for this state machine, and see Figure 2.9 on page 20 for a graphical representation.

The processed version of this file is included in appendix B.

Listing 6.30: Basic.smjava

```
1 region {
2   psinitial start;
3   finalstate end;
4
5   state idle;
6
7   transition(start, idle);
8   transition(idle, end) {
9     trigger("end");
10  }
11 }
```

Line 1 Define a new region.

Lines 2 - 3 Defines a starting and a end point.

Line 5 Define a simple state. No entry or exit points.

Line 7 Defines a transition from initial to the idle state.

Lines 8 - 9 Defines a transition from idle to the final state. This transition also has a trigger associated with it, this trigger is a `StringEvent` called “end”.

6.5.3 A switch state machine

Please see section 2.4 on page 20 for a description of the semantics and the constraints for this state machine, and see Figure 2.10 on page 21 for a graphical representation.

Listing 6.31: OnOff.smjava

```
1 region {
2   psinitial start;
3   finalstate end;
4
5   state on;
6   state off;
7
8   transition(start, off);
```

```
9
10   transition(off, on) {
11     trigger("on");
12   }
13
14   transition(off, end) {
15     trigger("end");
16   }
17
18   transition(on, off) {
19     trigger("off");
20   }
21 }
```

Line 1 Define a new region.

Lines 2 - 3 Defines a starting and a end point.

Lines 5 - 6 Defines two simple states, no entry or exit points.

Line 8 Define a transition from the starting point to the default state.

Lines 10 - 12 Define a transition from `off` to `on` with the `StringEvent` trigger `on`.

Lines 14 - 16 Defines a transition from the `off` state to the `finalstate`. This ends the state machine.

Lines 18 - 20 Define a transition from `on` to `off` with the `StringEvent` trigger `off`.

6.5.4 A deep history based state machine

Please see section 2.7 on page 24 for a description of the semantics and the constraints for this state machine, and see Figure 2.13 on page 25 for a graphical representation.

Listing 6.32: DeepHistory.smjava

```
1 region {
2   psinitial start;
3   finalstate end;
4
```

```
5   psdeephistory history;
6   state pause;
7
8   state composite {
9     region {
10      psinitial start;
11      finalstate end;
12
13      state idle1;
14      state idle2;
15
16      transition(start, idle1);
17      transition(idle1, idle2) {
18        trigger("continue1");
19      }
20
21      transition(idle2, end) {
22        trigger("continue2");
23      }
24    }
25  }
26
27  transition(start, composite);
28
29  transition(composite, end) {
30    trigger("finish");
31  }
32
33  transition(composite, pause) {
34    trigger("suspend");
35  }
36
37  transition(pause, history) {
38    trigger("resume");
39  }
40
41  transition(history, start);
42 }
```

Line 1 Define a new region.

Lines 2 - 3 Defines a starting and a end point.

Line 5 Creates a deep history state. Will be used to resume the state machine when paused.

Line 6 The state in which the state machine is paused.

Lines 8 - 25 Definition of the composite state. The inner block resembles the Basic state machine of the first example. The only difference is that the transition from `idle1` to `idle2` trigger on the `StringEvent continue1` and the transitions from `idle2` to `end` trigger on the `StringEvent continue2`.

Line 27 Defines a transition from `start` to `composite`.

Lines 29 - 31 Defines a transition from `composite` to `end` with the `StringEvent finish`.

Lines 33 - 35 Defines a transition from `composite` to `pause` with the `StringEvent suspend`.

Lines 37 - 39 Defines a transition from `pause` to `history` with the `StringEvent resume`.

Line 41 Defines a transition from `history` to `start`.

6.6 Conclusion

Extending a well-established language with new keywords is not something that can be done without some level of controversy. Keywords in a language should be just that, words that are key to the operation of the language, and for everything else there are frameworks.

As with the framework that was developed, an extension of the Java language with the UML specification in mind has little room for exploring different solutions to the same problem. And as such, the keywords in the `smjava` language quite naturally exposed themselves.

The implementation clearly shows that it is possible to extend the language with features from UML state machines, although not all features were implemented, it doesn't take much imagination to implement the rest of the features.

There are however some points that should be explored further:

Language clutter

In the current version of the `smjava` language there has been added

12 new keywords. This is not even the full set of features from the UML specification, and with the 50 keywords [7] already defined in Java, adding these would increase the amount of keywords with 24%.

It's quite clear that a JSR[8] describing these keywords would never be approved, since it messes too much with the language, and although there are plenty of use-cases for state machines, this will never be something every application needs.

An approach taken by several frameworks that extend the language today, is using an extra pre-processing step that takes away all the non-java parts and replacing them with their Java equivalents (often done at the bytecode level). This was how AspectJ[9] was able to implement support for aspect oriented programming in Java. This is somewhat related to the source-to-source translation that was done in this chapter, just using source-to-bytecode translation instead.

DSL for UML state machines

A possible solution to the keyword problem is creating a textual DSL for state machines instead. This DSL could act as a intermediary step between the model and the code implementation. In this DSL one would be free to add all the required keywords without cluttering the Java language. A code-generation step could then be added to make sure that the DSL is translated into actual code. Since this DSL would be small and compact, it could also open up the possibility for a two-way conversation between the model and the DSL.

***smjava* translated to framework code**

Since the `smjava` language was implemented using a translator, it is quite obvious that the addition of UML state machine to the language would simply be syntactic sugaring, and not really something that demands the implementation of new keywords. A framework could easily been used instead of extending the language.

[7] `http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`

[8] `http://jcp.org/en/jsr/overview`

[9] `http://www.eclipse.org/aspectj/`

Even if the new language might not be a perfect candidate for inclusion in the Java language, for our uses it served its purpose well. The development of a Java extension for state machines gives a much clearer syntax for developing state machines in code, and would be a great help for developing at the code level.

Chapter 7

Conclusion and Future Work

7.1 Overview

This section will give a short overview of the conclusions made in this thesis.

State Pattern In chapter 4 it was shown that state patterns can be easily extended to gain some of the more common properties of state machines (guards, behaviors). But when it comes to the more advanced features of state machines (orthogonality, sub-states, etc.) the pattern quickly becomes too cluttered, and loses its most important feature; simplicity.

Java Framework In chapter 5 it was shown that a fully capable framework for creating and running state machines can be created in Java. The framework follows the UML meta-model and this adds a level of complexity that might hinder the adoption of this method. There was also developed an importer for XMI files, which opens up the possibility for using e.g. the Eclipse UML2 Tools project for modeling state machines.

Extended Java In chapter 6 it was shown that extending the Java language with features from the UML state machine specification [14] is not a difficult task, the nature of the state machine meta-model makes it easy to directly map keywords from UML into Java.

While this is true, the resulting language is in the best case grown

by 24% and makes the language unnecessary cluttered with state machine specific keywords.

For full featured UML state machine support, it has been shown that using a framework is the only approach that gives us full support without compromise, like limiting us to a pattern, or by using pre-processors for extending the language.

7.2 Future work

This section will further look into possible improvements of the approaches that was chosen. Special care will we taken with the framework approach since that was concluded as the best approach.

7.2.1 Extended Java / State Pattern

It has already been shown [18, 9] that extending languages with features from state patterns is a task that can be done without interfering too much with the language, and I believe this is something that can also be implemented in the Java language. As it has been shown in this thesis, extending a language with the full feature set of UML state machines adds too much clutter to a language, and a state pattern approach might allow for a simpler implementation of the notion of *reactive* systems in Java.

7.2.2 Java Framework

For full UML state machine support in Java, it has been shown that the currently best approach is using a framework based on the UML metamodel.

There are quite a few improvements that can be made to the current framework:

Thread support As it was stated in the beginning of the framework chapter, the approach that was chosen when it comes to concurrency in the runtime system is a simple multiplexing system which means that in reality only one action can happen at any time.

What this actually means, is that if we have two concurrent regions, all stepping will be finished in region 0, before region 1 is stepped at all.

The framework should be restructured so that every orthogonal state puts its regions into their own thread. Special care has to be made to make sure that the system remains thread-safe, and this will also give some burden to the implementor. If e.g. we have two behaviors in two different regions both accessing the same variable from the context class at the same time, it will have to be protected by the `synchronized` flag.

Fluent Builder Since most of the classes in the meta-model needs a number of properties before they can be constructed, a number of constructors exists for every class. E.g. the `Transition` class has 11 constructors, covering every possible combination of properties for a transition.

This could be fixed by using a common design pattern called a fluent builder, this is a variant of the normal builder [5] and it makes the code almost self-describing (if done correctly).

The fluent builder works by having a separate class for defining the instance you want. E.g. for the `Transition` class, there would be a `TransitionBuilder` class, with a static method called `transition` which creates a builder, and then a separate method called `build` or `create` that actually makes the configured object.

Consider the `Transition` class again, a normal usage of this would be:

```
new Transition("t1", on, off, r);
```

Using a fluent builder wouldn't necessarily make the code shorter, but would make it more clear:

```
transition("t1").from(on).to(off).onRegion(r).build();
```

The actual build process at the end should throw an exception for classes that violate semantic rules and constraints.

Sub-machine support One feature that is clearly lacking from the framework, is sub-machines. Sub-machines allow state machines to be composed by several state machines, and helps in both in reducing the complexity of the state machine, and also support reuse of state machine (although this has more limited scope).

Tool support Another feature that is missing is good support from tools. A custom tool could be created that would help in bridging the gap between the programming language (the framework) and the diagram itself.

Some notable features of this tool would be:

- Ability to create UML state machine diagrams based on the meta-model.
- The tool should be able to save the diagrams in a simple format that would be imported directly by the framework, thereby skipping over the entire code-generation step. It would be ideal if the tool could also generate the code, if the project was supposed to continue living without using a graphical notation.
- Using a tool like EMFText[1] to create on-the-fly EMF representations of the classes in the project, like the context class, the events, and such.

  The diagram tool could then use this information to directly bind events / behaviors (from the context) onto states and transitions.

Constraint language A simple side-effect free Java-like language should be created to enable simpler implementation of the constraints in the system. The language should be able to access the context objects properties and operations.

State machines as behaviors Today, all behaviors must be implemented as the command pattern (using the *Behavior* interface). This should be extended to more closely follow the meta-model, this would

[1] `http://www.emftext.org`

mean that the `StateMachine` class would implement the *Behavior* interface, and as a result would be allowed to be used for all the behaviors in the system.

Bibliography

[1] Franck Barbier. Support the uml state machine diagrams at runtime. *Model Driven Architecture - Foundations and Applications, Lecture notes in Computer Science*, 5095/2008:338–348, 2008.

[2] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer, 1st edition, October 25, 1982.

[3] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 3rd edition, September 25, 2003.

[4] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 1st edition, October 25, 2004.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, November 10, 1994.

[6] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.

[7] David Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, May 1988.

[8] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Ole Lehrmann Madsen. Towards integration of state machines and object-oriented languages. *Technology of Object-Oriented Lan-*

guages and Systems, 1999. Proceedings of, pages 261–274, July 1999.

[10] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[11] George H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September, 1955.

[12] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, 1st edition, April, 2006.

[13] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34:129–153, April 1, 1956.

[14] OMG. *OMG Unified Modeling Language (tm) (OMG UML), Superstructure*. OMG, 2.2 edition, February 2, 2009.

[15] Dan Pilone. *UML 2.0 Pocket Reference*. O'Reilly Media, 1st edition, March, 2006.

[16] Dan Pilone and Neil Pitman. *UML 2.0 In A Nutshell*. O'Reilly Media, 1st edition, July, 2005.

[17] James Rumbaugh, Ivar Jacobsen, and Grady Booch. *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2nd edition, May 29, 2005.

[18] A Taivalsaari. Object-oriented programming with modes. *Journal of object-oriented programming*, 6:25–32, 1993.

Appendices

Appendix A

Instructions for the included source code

This appendix will show a brief summary of the included source code, and how to run it.

The source code can be found on:

For download:
`http://folk.uio.no/~morteoh/source.tar.gz`

For browsing:
`http://folk.uio.no/~morteoh/source`

A.1 The source directory

This section will give a overview of where files are located in the source directory.

Directory "extlib" This directory contains the ANTLR 3 jar file (used for processing smjava files)

Directory "misc" This directory contains the statepattern examples from chapter 4.

Directory "smexamples" This directory contains the different examples used in the tutorial in chapter 2, and should be openend in Eclipse Galileo (3.5)[1].

[1] http://www.eclipse.org/galileo/

This directory can be imported into Eclipse as an existing project.

Directory "smjava" This directory contains all the files for the *smjava* pre-processor.

In the base of this directory, the "smjava.stg" StringTemplate file can be found. The directory *examples* contains all the examples from the smjava chapter. In the directory *src/main/antlr3/* the grammar file for ANTLR 3 can be found. In the directory *src/main/-java/net/mortenoh/smjava/* the source for the pre-processor can be found.

This directory can be imported into Eclipse as an existing project.

Directory "smlib" This directory contains all the project files for the state machine framework.

The directory *src/main/java/net/mortenoh/smlib/examples/* contains the smlib version of the tutorial examples. The directory *src/main/java/net/mortenoh/smlib/runtime/* contains all the runtime files. The directory *src/main/java/net/mortenoh/smlib/uml2/* contains all files for the meta-model.

This directory can be imported into Eclipse as an existing project.

A.2 Running the examples

This section will explain how to compile and run the examples for the smlib and smjava chapters. Running the source from the statepattern chapter will not be explained since these are just plain Java files. Commands needed to input events to the examples are not given, since example of this have already been given.

You will need a working Java installation and Maven 2 for this to work and this has only been tested on OS X and Linux. For running on Windows, there will be a need to manually replace the `smjavac` shell script (should not be too difficult).

A.2.1 Compiling and running smlib examples

The source for the framework can be compiled with "mvn package" command, and this will result in a file called *smlib-1.0-SNAPSHOT.jar* in the

"target" directory.

The examples can be run with:

```
1 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      basic.BasicSM
2 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      choice.ChoiceSM
3 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      history.DeepHistorySM
4 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      onoff.OnOffSM
5 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      onoff.OnOffXMI
6 java -cp target/smlib-1.0-SNAPSHOT.jar net.mortenoh.smlib.examples.
      orthogonal.ForkOrthogonalSM
```

A.2.2 Compiling and running smjava examples

These files are already compiled for convenience. The output from the ANTLR tool is placed in the src-gen directory.

Before trying to compile and run these examples, please make sure that the framework is packaged first.

Listing A.1: Processing the examples

```
1 ./smjavac examples/Basic.smjava
2 ./smjavac examples/OnOff.smjava
3 ./smjavac examples/DeepHistory.smjava
```

Listing A.2: Compiling the processed examples

```
1 javac -classpath ../smlib/target/smlib-1.0-SNAPSHOT.jar examples/
      Basic.java
2 javac -classpath ../smlib/target/smlib-1.0-SNAPSHOT.jar examples/
      OnOff.java
3 javac -classpath ../smlib/target/smlib-1.0-SNAPSHOT.jar examples/
      DeepHistory.java
```

Listing A.3: Running the processed examples

```
1 java -cp ../smlib/target/smlib-1.0-SNAPSHOT.jar:examples Basic
2 java -cp ../smlib/target/smlib-1.0-SNAPSHOT.jar:examples OnOff
3 java -cp ../smlib/target/smlib-1.0-SNAPSHOT.jar:examples DeepHistory
```

Appendix B

Processed Basic.smjava example

This appendix shows how the “Basic.smjava” file looks processed, the rest of the processed files are located in the “source/smjava/examples” directory of the included source.

Listing B.1: Unprocessed Basic.smjava

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3
4  import net.mortenoh.smlib.uml2.Event;
5  import net.mortenoh.smlib.uml2.StringEvent;
6
7  public class Basic {
8    region {
9      psinitial start;
10     finalstate end;
11
12     state idle;
13
14     transition(start, idle);
15     transition(idle, end) {
16       trigger("end");
17     }
18   }
19
20   public static void main(String[] args) {
21     Basic basic = new Basic();
```

```
22    basic.sm.start();
23
24    Queue<Event> events = new LinkedList<Event>();
25    events.add(new StringEvent("end"));
26
27    while(basic.sm.isAlive()) {
28      Event e = events.poll();
29
30      if(e != null) {
31        basic.sm.push(e);
32      }
33    }
34  }
35 }
```

Listing B.2: Processed Basic.smjava (Basic.java)

```
 1 import java.util.LinkedList;
 2 import java.util.Queue;
 3
 4 import net.mortenoh.smlib.uml2.Event;
 5 import net.mortenoh.smlib.uml2.StringEvent;
 6
 7 public class Basic {
 8
 9   public static interface statemachine
10   {
11     public void start();
12     public boolean isAlive();
13     public void push(net.mortenoh.smlib.uml2.Event e);
14   }
15
16   public statemachine sm = new _impl_statemachine();
17
18   @SuppressWarnings("unused")
19   private final class _impl_statemachine implements statemachine
20   {
21     private net.mortenoh.smlib.runtime.v2.RT _runtime;
22     private net.mortenoh.smlib.uml2.StateMachine _stateMachine;
23
24     private java.util.concurrent.BlockingQueue<net.mortenoh.smlib.
           uml2.Event> _runtime_queue;
25     private java.lang.Thread _runtime_thread;
26
27     public _impl_statemachine() {
```

```
28        _create_statemachine();
29        _statemachine_validate();
30        _create_runtime();
31      }
32
33      private void _create_statemachine()
34      {
35        _stateMachine = new net.mortenoh.smlib.uml2.StateMachine();
36        net.mortenoh.smlib.uml2.Region id5148537001872048123 =
              _stateMachine.getRegions().get(0);
37        net.mortenoh.smlib.uml2.PseudoState id7390234900874313665 = new
               net.mortenoh.smlib.uml2.PseudoState(id5148537001872048123)
              ;
38        net.mortenoh.smlib.uml2.FinalState id8045455508827321486 = new
              net.mortenoh.smlib.uml2.FinalState(id5148537001872048123);
39        net.mortenoh.smlib.uml2.State id9032382805906406051 = new net.
              mortenoh.smlib.uml2.State("id9032382805906406051",
              id5148537001872048123);
40
41
42        net.mortenoh.smlib.uml2.Transition id2424066601685896859 = new
              net.mortenoh.smlib.uml2.Transition("id2424066601685896859",
               id7390234900874313665, id9032382805906406051,
              id5148537001872048123);
43
44
45        net.mortenoh.smlib.uml2.Transition id3674477414337365959 = new
              net.mortenoh.smlib.uml2.Transition("id3674477414337365959",
               id9032382805906406051, id8045455508827321486,
              id5148537001872048123);
46
47        net.mortenoh.smlib.uml2.Trigger id7110958751908446737 = new net
              .mortenoh.smlib.uml2.Trigger(id3674477414337365959, new net
              .mortenoh.smlib.uml2.StringEvent("end"));
48      }
49
50      private void _create_runtime()
51      {
52        _runtime = new net.mortenoh.smlib.runtime.v2.RT(_stateMachine);
53        _runtime.prepare();
54
55        _runtime_thread = new java.lang.Thread(_runtime);
56        _runtime_queue = _runtime.getEventQueue();
57      }
```

```
58
59    private void _statemachine_validate()
60    {
61      try {
62        _stateMachine.validate();
63      } catch (net.mortenoh.smlib.uml2.SemanticException e) {
64        e.printStackTrace();
65      }
66    }
67
68    private void _runtime_push_event(net.mortenoh.smlib.uml2.Event e)
69    {
70      _runtime_queue.add(e);
71    }
72
73    private void _runtime_start()
74    {
75      _runtime_thread.start();
76    }
77
78    public void start()
79    {
80      _runtime_start();
81    }
82
83    public boolean isAlive()
84    {
85      return _runtime_thread.isAlive();
86    }
87
88    public void push(net.mortenoh.smlib.uml2.Event e)
89    {
90      _runtime_push_event(e);
91    }
92  }
93
94  public static void main(String[] args) {
95    Basic basic = new Basic();
96    basic.sm.start();
97
98    Queue<Event> events = new LinkedList<Event>();
99    events.add(new StringEvent("end"));
100
101   while(basic.sm.isAlive()) {
```

```
102       Event e = events.poll();
103
104       if(e != null) {
105         basic.sm.push(e);
106       }
107     }
108   }
109 }
```