Execution of UML State Machines Using Modelica

Wladimir Schamai[1], Uwe Pohlmann[2], Peter Fritzson[3], Christiaan J.J. Paredis[4], Philipp Helle[1], Carsten Strobel[1]

[1]EADS Innovation Works, Germany
[2]University of Paderborn, Department of Computer Science, Germany
[3] Linköping University, PELAB – Programming Environment Lab, Sweden
[4]Georgia Institute of Technology, Atlanta, USA

Abstract

ModelicaML is a UML profile for the creation of executable models. ModelicaML supports the Model-Based Systems Engineering (MBSE) paradigm and combines the power of the OMG UML standardized graphical notation for systems and software modeling, and the simulation power of Modelica. This addresses the increasing need for precise integrated modeling of products containing both software and hardware. This paper focuses on the implementation of executable UML state machines in ModelicaML and demonstrates that using Modelica as an action language enables the integrated modeling and simulation of continuous-time and reactive or event-based system dynamics. More specifically, this paper highlights issues that are identified in the UML specification and that are experienced with typical executable implementations of UML state machines. The issues identified are resolved and rationales for design decisions taken are discussed.

Keywords UML, Modelica, ModelicaML, Execution Semantics, State Machine, Statechart

1 Introduction

UML [2], SysML [4] and Modelica [1] are object-oriented modeling languages. They provide means to represent a system as objects and to describe its internal structure and behavior. UML-based languages facilitate the capturing of information relevant to system requirements, design, or test data by means of graphical formalisms, crosscutting constructs and views (diagrams) on the model-data. Modelica is defined as a textual language with standardized graphical annotations for icons and diagrams, and is designed for the simulation of system-dynamic behavior.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, October, 2010, Oslo, Norway.

Copyright is held by the author/owner(s). The proceedings are published by Linköping University Electronic Press. Proceedings available at: http://www.ep.liu.se/ecp/047/

EOOLT 2010 website: http://www.eoolt.org/2010/

1.1 Motivation

By integrating UML and Modelica the strength of UML in graphical and descriptive modeling is complemented with the Modelica formal executable modeling for system dynamic simulation. Conversely, Modelica will benefit from using the selected subset of the UML-based graphical notation (visual formalisms) for editing, visualizing and maintaining Modelica models.

Graphical modeling, as promoted by the OMG [12], promises to be more effective and efficient regarding editing, human-reader perception of models, and maintaining models compared to a traditional textual representation. A unified standardized graphical notation for systems modeling and simulation will facilitate the common understanding of models for all parties involved in the development of systems (i.e., system engineers, designers, and testers; software developers, customers or other stakeholders).

From a simulation perspective, the behavior described in the UML state machine is typically translated into and thereby limited to time-discrete or event-based simulations. Modelica enables mathematical modeling of hybrid (continuous-time and discrete-time dynamic description) simulation. By integrating UML and Modelica, UML-based modeling will become applicable to the physical-system modeling domain, and UML models will become executable while covering simulation of hardware and software, with integrated continuous-time and event-based or time-discrete behavior. Furthermore, translating UML state machines into executable Modelica code enables engineers to use a common set of formalisms for behavior modeling and enables modeling of software parts (i.e., discrete or event-based behavior) to be simulated together with physical behavior (which is typically continuous-time behavior) in an integrated way.

One of the ModelicaML design goals is to provide the modeler with precise and clear execution semantics. In terms of UML state machines this implies that semantic variation points or ambiguities of the UML specification have to resolved.

The main contribution of this paper is a discussion of issues that were identified when implementing UML state machines in ModelicaML. Moreover, proposals for the resolution of these issues are presented. The issues identified or design decisions taken are not specific to the ModelicaML state machines implementation. The questions addressed in this paper will most likely be raised by anyone who intends to generate executable code from UML state machines.

1.2 Paper Structure

The rest of this paper is structured as follows: Chapter 2 provides a brief introduction to Modelica and ModelicaML and gives an overview of related research work. Chapter 3 describes how state machines are used in ModelicaML and highlights which UML state machines concepts are supported in ModelicaML so far. Chapter 4 discusses the identified issues and explains both resolution and implementation in ModelicaML. Chapter 5 provides a conclusion.

2 Background and Related Work

2.1 The Modelica Language

Modelica is an object-oriented equation-based modeling language that is primarily aimed at physical systems. The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy, signals, or other continuous-time or discrete-time interactions between system components.

2.2 ModelicaML – UML Profile for Modelica

This paper presents the further development of the Modelica Graphical Modeling Language (ModelicaML [15]), a UML profile for Modelica. The main purpose of ModelicaML is to enable an efficient and effective way to create, visualize and maintain combined UML and Modelica models. ModelicaML is defined as a graphical notation that facilitates different views (e.g., composition, inheritance, behavior) on system models. It is based on a subset of UML and reuses some concepts from SysML. ModelicaML is designed for Modelica code generation from graphical models. Since the ModelicaML profile is an extension of the UML meta-model it can be used as an extension for both UML and SysML[1]. The tools used for modeling with ModelicaML and generating Modelica code can be downloaded from [15].

[1] SysML itself is also a UML Profile. All ModelicaML stereotypes that extend UML meta-classes are also applicable to the corresponding SysML elements.

2.3 Related Work

In previous work, researchers have already identified the need to integrate UML/SysML and Modelica, and have partially implemented such an integration. For example, in [7] the basic mapping of the structural constructs from Modelica to SysML is identified. The authors also point out that the SysML Parametrics concept is not sufficient for modeling the equation-based behavior of a class. In contrast, [9] leverages the SysML Parametrics concept for the integration of continuous-time behavior into SysML models, whereas [8] presents a concept to use SysML to integrate models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems and provides rules for the graph-based bidirectional transformation of SysML and Modelica models.

In Modelica only one type of diagram is defined: the *connection diagram* that presents the structure of a class and shows class-components and connections between them. Modelica does not provide any graphical notation to describe the behavior of a class. In [13] an approach for using the UML-based notation of a subset of state machines and activity diagrams for modeling the behavior of a Modelica class is presented.

Regarding state machines, a list of general challenges with respect to regarding statecharts is presented in [10] and a summary on existing statechart variants is provided. In [11] the fact is stressed that different statechart variants are not compatible even though the syntax (graphical notation) is the same. It is also pointed out that the execution semantics strongly depend on the implementation decisions taken, which are not standardized in UML.

Few implementations of the translation of statecharts into Modelica exist ([6], [5][2]). However, none implements a comprehensive set of state machines concepts as defined in the UML specification.

The main focus of this paper is the resolution of issues related to the execution semantics of UML state machines. A detailed description of the execution semantics of the implementation of UML state machines in ModelicaML and additional extensions are provided in [15] and are out of the scope of this paper.

3 State Machines in ModelicaML

3.1 Simple Example

Assume one would like to simulate the behavior defined by the state machines depicted in Figure 1 using Modelica. This state machine defines part of the behavior of the

[2] StateGraph [5] uses a different graphical notation compared to the UML notation for state machines.

class SimpleStateMachine. In UML, this class is referred to as the *context* of StateMachine_0.

The UML graphical notation for state machines consists of rectangles with rounded corners representing *states*, and edges representing *transitions* between states. Transitions can only be executed if appropriate *triggers* occur and if the associated *guard* condition (e.g., **`[t>1 and x<3]`**) evaluates to true. If no triggers are defined then an empty trigger, which is always activated, is assumed. In addition, transitions can have *effects* (e.g., **`/x:=1`**).

The UML semantics define that a state machine can only be in one of the (simple) states in a region[3] at the same instance of time. The filled circle and its outgoing transition mean that, by default, (i.e. when the execution is started) the state machine is in its initial state, *State_0* (i.e., when the execution is started). The expected execution behavior is as follows:

- When the execution is started the state machine is in its initial state, *State_0*.
- As soon as the guard condition, **`[t>1 and x<3]`**, is true, *State_0* is exited, the transition effect, **`x := 1`**, is executed and *State_1* is entered. The state machine is again in a stable configuration that is referred to as an *active configuration*.
- As soon as the guard condition, **`t>1.5 and x>0`**, becomes true, *State_1* is exited, the effect, **`x:=2`**, is executed and *State_2* is entered.
- As soon as the condition **`x>1`** becomes true, *State_2* is exited, the transition effect, **`x:=3`**, is executed and *State_0* is entered.

Figure 2 shows Modelica code that performs the behavior described above. Figure 3 presents the visualized simulation results.

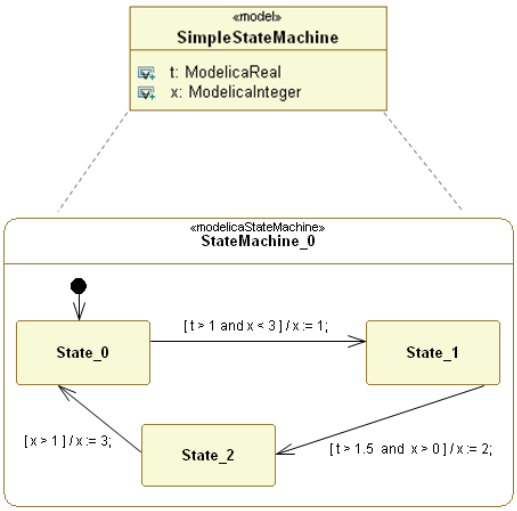

Figure 1: State machine defines part of class behavior

```
model SimpleStateMachine
  Boolean State_0 "State_0 representation";
  Boolean State_1 "State_1 representation";
  Boolean State_2 "State_2 representation";
  Integer x "discrete variable";
  Real t "continuous variable";
 equation //Code for continuous integration of t
    der(t)=time;

 algorithm //Code for StateMachine_0
  when initial() then
      State_0:=true "Activation of the initial state";
  end when;
  //Transition from State_0 to State_1
  if pre(State_0) and t > 1 and x < 3 then
      State_0:=false "Deactivation of state";
      x:=1 "Transistion effect";
      State_1:=true "Activation of state";
  //Transition from State_1 to State_2
  elseif pre(State_1) and t > 1.5 and x > 0 then
      State_1:=false "Deactivation of state";
      x:=2 "Transistion effect";
      State_2:=true "Activation of state";
  //Transition from State_2 to State_0
  elseif pre(State_2) and x > 1 then
      State_2:=false "Deactivation of state";
      x:=3 "Transistion effect";
      State_0:=true "Activation of state";
  end if;
end SimpleStateMachine;
```

Figure 2: Corresponding Modelica code

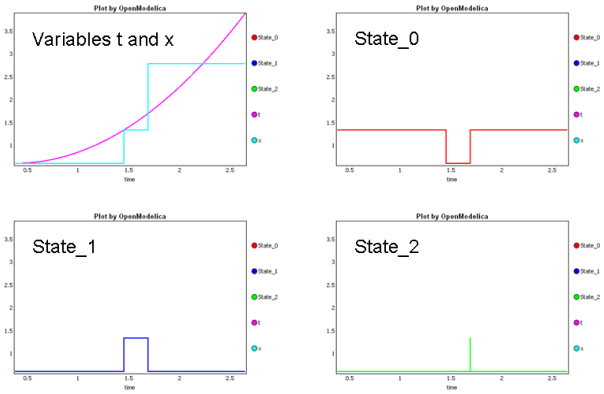

Figure 3: Simulation results

[3] If a composite state or multiple regions are defined for a state machine, then it means that the state machine is in multiple (simple) states at the same time.

3.2 State Machines in ModelicaML

UML2 defines two types of state machines: behavior state machines and protocol state machines. Behavior state machines are used to model parts of class behavior. ModelicaML state machines are derived from UML behavior state machines. Compared to behavior state machines, protocol state machines are limited in terms of expressiveness and are tailored to the need to express protocols or to define the lifecycle of objects. Since this is not the main intended area of application for ModelicaML, protocol state machines are not taken into account. Consequently, none of the chapters of the UML specification that address the protocol state machines are considered.

State machines are used in ModelicaML to model parts of class behavior. A behavioral class (i.e., Modelica class, model or block) can have 0..* state machines as well as 0..* other behaviors (e.g. equation or algorithm sections). This is different from a typical UML application where usually only one state machine is used to represent the *classifierBehavior*. In ModelicaML it is possible to define multiple state machines for one class which are executed in parallel. This possibility allows the modeler to structure the behavior by separating it into individual state machines. When multiple state machines are defined for one class they are translated into separate algorithm sections in the generated Modelica code. This implies that they cannot set the same class variables (e.g., in entry/do/exit or transition effects) because it would result in an overdetermined system.

UML state machines are typically used to model the reactive (event-based) behavior of objects. Usually an event queue is implemented that collects all previously generated events which are then dispatched one after the other (the order is not fully specified by UML) to the state machine and may cause state machine reactions. Strictly speaking, a UML state machine reacts (is evaluated) only when events are taken from the event queue and dispatched to the state machine.

ModelicaML uses Modelica as the execution (action) language. In contrast to the typical implementations of UML state machines, the Modelica code for a ModelicaML state machine is evaluated continuously, namely, after each continuous-time integration step and, if there are event iterations, at each event iteration. Event iterations concept is defined by Modelica ([1], p.25) as follows: "*A new event is triggered if at least for one variable v "pre(v) <> v" after the active model equations are evaluated at an event instant. In this case, the model is at once re-evaluated. This evaluation sequence is called "event iteration". The integration is restarted, if for all v used in pre-operators the following condition holds: "pre(v) == v"*.". The definition of pre(v) is the following: "*Returns the "left limit" $y(t^{pre})$ of variable y(t) at a time instant t. At an event instant, $y(t^{pre})$ is the value of y after the last event iteration at time instant t ...*" (see [1], p.24).

Furthermore, the following definition is essential to understand the execution of Modelica code (see Modelica specification [1], p.84): "*Modelica is based on the synchronous data flow principle and the single assignment rule, which are defined in the following way:*

- *All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.*
- *At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).*
- *Computation and communication at an event instant does not take time. [If computation or communication time has to be simulated, this property has to be explicitly modeled].*
- *The total number of equations is identical to the total number of unknown variables (= single assignment rule)."*

3.2.1 Transformation of State Machines to Modelica Code

Two main questions need to be considered when translating a ModelicaML state machine into Modelica:

- The first question is whether a library should be used or whether a code generator should be implemented. One advantage of using a library is that the execution semantics can be understood simply by inspecting the library classes. In order to understand the execution semantics of generated code one also needs to understand the code generation rules.
- The behavior of a ModelicaML state machine can be expressed by using Modelica algorithmic code or by using equations. Note that the statements inside an algorithm section in Modelica are executed exactly in the sequence they are defined. This is different from the equation sections which are declarative so that the order of equations is independent from the order of their evaluation. The Modelica StateGraph library [5] uses equations to express behavior that is similar to the UML state machine behavior.

In ModelicaML, the behavior of one state machine is translated into algorithmic code that is generated into one algorithm section of the containing class[4]. The rationale

[4] A Modelica class can have 0..* algorithm sections.

for the decision to implement a specific code generator instead of implementing a library and to use algorithm statements instead of equations is the following:

- The behavior expressed by a state machine is always causal. There is no need to use the acausal modeling capability of Modelica. By using algorithm (with predefined causality) no sorting of equations is required.
- Furthermore, for the implementation of inter-level transitions, i.e. transitions which cross states hierarchy borders, the deactivation and activation of states and the execution sequence of associated actions (exit/entry action of states or state transitions effects) has to be performed in an explicitly defined order. This is hard to achieve when using equations that are sorted based on their data dependencies.

3.2.2 Combining Continuous-Time and Discrete-Time Behavior

The fact that a ModelicaML state machine is translated into algorithmic Modelica code implies that all actions (transition effects, or entry/do/exit actions of states) can only be defined using algorithmic code. Hence, it is not possible to insert equations into transition effects or entry/do/exit actions of states. However, it is possible to relate the activation of particular equations based on the activation or deactivation of state machine states. This is supported by the dedicated `IsInState()`-macro in ModelicaML. Vice versa, state machines can react on the status of any continuous-time variable.

3.2.3 Event Processing (Run-To-Completion Semantics Applicability)

UML, [2] p. 565, defines the *run-to-completion* semantics for processing events. When an event is dispatched to a state machine, the state machine must process all actions associated with the reaction to this event before reacting to further events (which might possibly be generated by the transitions taken). This definition implies that, even if events occur simultaneously, they are still processed sequentially. In practice, this requires an implementation of an event queue that ultimately prevents events from being processed in parallel. This can lead to ambiguous execution semantics as is pointed out in section 4.2.

The problem with event queues, as discussed in section 4.2, does not exist in ModelicaML.If events occur simultaneously (at the same simulated time instant or event iteration) in ModelicaML state machines they are processed (i.e. consumed) in parallel in the next evaluation of the state machine.

However, the definition of the *run-to-completion* semantics is still applicable to Modelica and, thus, to ModelicaML state machines in the sense that when an event has occurred a state machine first finishes its reactions to this event before processing events that are generated during these reactions.

4 State Machines Execution Semantics Issues Discussion

4.1 Issues with Instantaneous States: Deadlocks (Infinite Looping)

In Modelica the global variable *time* represents the simulated real time. Computations at event iterations do not consume simulated time. Hence, states can be entered and exited at the same simulated point in time. For instance, Figure 3 shows State_2 being entered and exited at the same simulated point in time. However, in ModelicaML, a state cannot be entered and exited during the same event iteration cycle, i.e., variables that represent states cannot be set and unset in the same event iteration cycle. This is ensured by using the *pre(state activation status)* function in the conditions of state transition code. This enforces that the entire behavior first reacts to an event before reacting to the events that are generated during event iterations (i.e. re-evaluation of the equation system).

When instantaneous states are allowed it is possible to model deadlocks that lead to an infinite loop at the same simulated point in time. Consider Figure 4 (left): The behavior will loop infinitely without advancing the simulated time and a simulation tool will stop the simulation and report an issue.

If such a behavior is intended and the state machine should loop continuously and execute actions, the modeler can break the infinite looping by adding a time delay[5] to one of the involved transitions (see the right state machine in Figure 4). In doing so the simulation time is advanced and the tool will continue simulating.

In large models deadlocks can exist which are not as obvious as in the simple example depicted in Figure 4. Infinite looping is often not modeled on purpose and is hard to prevent or to detect. This issue is subject to future research in ModelicaML.

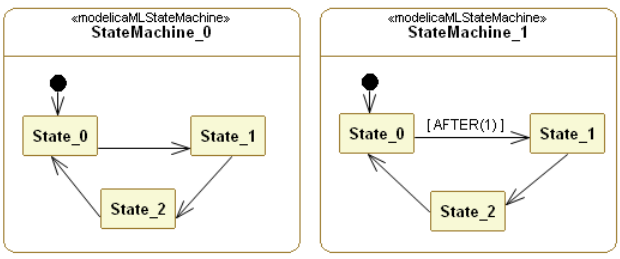

Figure 4: Deadlocks (infinite looping) example

[5] AFTER(expression) is a ModelicaML macro. It is expanded to the guard condition `state_local_timer > expression`.

4.2 Issues With Concurrency When Using Event Queues

Consider the state machine in Figure 5 modeled in IBM Rational Rhapsody [14]. The events *ev1* and *ev2* are generated simultaneously at the same time instant when entering the initial states of both regions. However, it is not obvious to the modeler in which order the generated events are dispatched to the state machine.

When the simulation is started Rhapsody shows that the events are generated and put into the event queue in the following order: *ev1, ev2*. When restarting the simulation the order of events in the queue will always be the same. Obviously, there is a mechanism that determines the order of region executions based on the model data.

Next, these events are dispatched to the state machine one after the other. First the event *ev1* is dispatched and the transition to *state_1* is executed, then the event *ev2* is dispatched and the transition from *state_1* to *state_2* is executed, as shown in Figure 6.

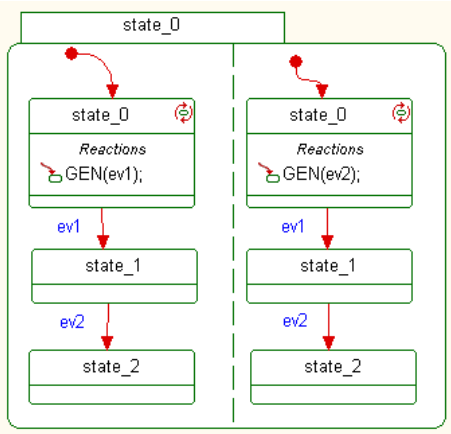

Figure 5: Events queue issue 1

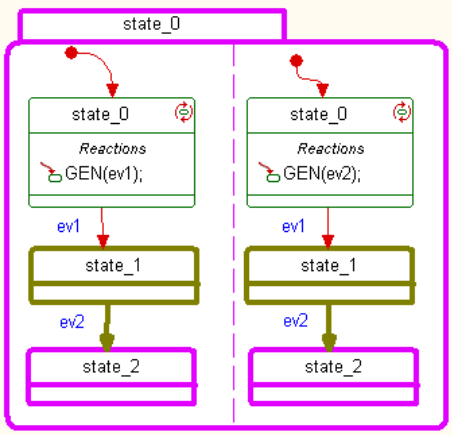

Figure 6: Events queue issue 1 simulation. The state machine ends up in state_2 in both regions.

According to this behavior the occurrence of *ev2* is delayed. However, with the state machine in *state_1* no *ev2* occurs. The event *ev2* occurs when the state machine is in *state_0*. This behavior seems to be similar to the concept of *deferred events* described in the UML specification ([2], p.554): "*An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event pool while another non-deferred event is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.*".

Consider Figure 7. It shows a slightly modified state machine. The event e*v2* is generated inside the left region and the *ev1* is generated inside the right region. The order of events in the queue is now reversed: *ev2, ev1*.

Figure 8 shows the simulation result. In contrast to the assumption above, the event *ev2* is not deferred. It is dispatched to the state machine and discarded after the transition to *state_1* is taken. The state machine finally stays in *state_1*, which is a different behavior than in Figure 6.

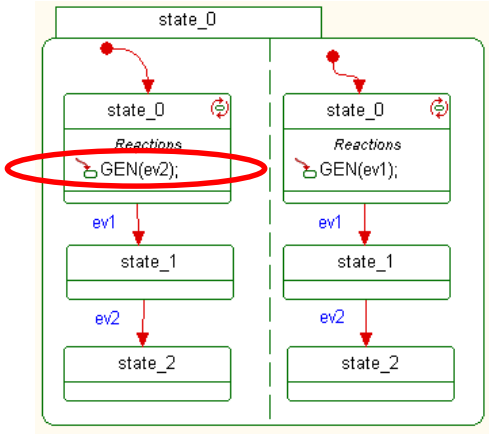

Figure 7: Events queue issue 2

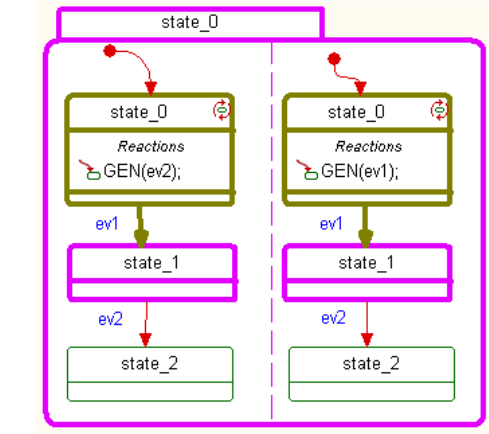

Figure 8: Events queue issue 2 simulation. The state machine ends up in state_1 in both regions.

Along with the fact that the modeler cannot control the execution order of the parallel regions (this issue is ad-

dressed in section 4.3) and, thus, the order in which events are generated, the main issue here is that it leads to behavior that is unpredictable and cannot be expected from the modeler's perspective.

Figure 9 shows the same model in ModelicaML. In contrast to the examples above, regardless of whether the event *ev2* is generated in the right region or in the left region, the execution behavior is the same – the state machine always ends up in *state_1* in both regions.

This is because the events that occur simultaneously are processed (i.e., dispatched and consumed) in parallel during the next state machine evaluation cycle. When the transitions from the states *state_0* to *state_1* in both regions are executed, event *ev2* is also consumed and the state machine stays in both states *state_1* because with the state machines in states *state_1* in both regions no event *ev2* is generated.

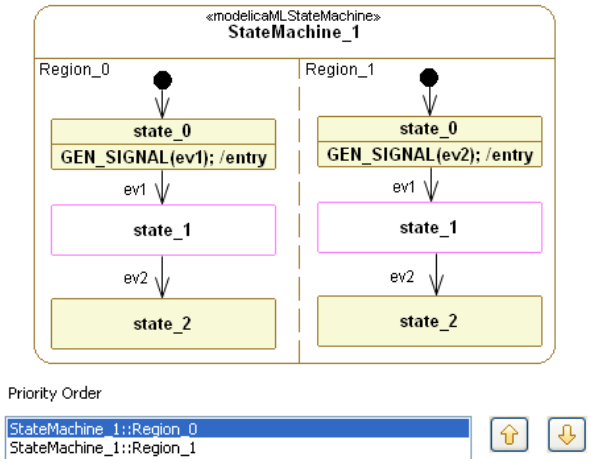

Figure 9: Same model in ModelicaML. The state machine ends up in state_1 in both regions.

4.3 Issue with Concurrent Execution in Regions

Regions are executed in parallel, i.e. at the same simulated time instant. However, the corresponding Modelica code in the algorithm section is still sequential (procedural). To ensure determinism and make the semantics explicit to the modeler, in ModelicaML each region is automatically given a priority relative to its neighboring regions (a lower priority number implies higher execution order priority). This may be necessary when there are actions that set the same variables in multiple parallel regions[6], as illustrated in Figure 10.

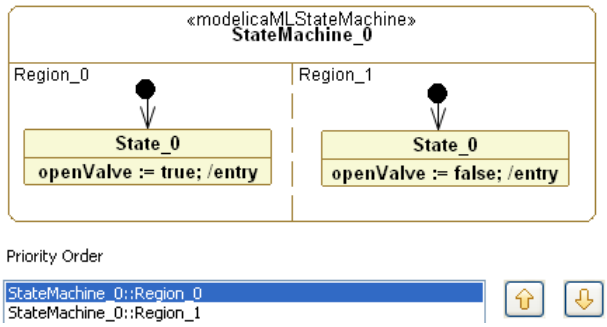

Figure 10: Definition of priority for parallel regions

Since *Region_0* is given a higher execution priority, it will be executed prior to *Region_1*, which has lower priority. The result is that `openValve` is set to `false`. Note that if *State_0* in *Region_0* was a composite state, then its internal behavior would also be executed before the behavior of *State_0* in *Region_1*.

Priorities are set by default by the ModelicaML modeling tool. The modeler can change priorities and, in doing so, define the execution sequence explicitly to ensure the intended behavior.

The regions priority definition is also used for exiting and entering composite states as well as inter-level transitions as discussed in sections 4.5 and 4.6.

4.4 Issues with Conflicting Transitions

When a state has multiple outgoing transitions and trigger and guard conditions overlap, i.e. they can be true at the same time, then the transitions are said to be in conflict ([2], p.566): "*Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously.*" When triggers are defined for a transition, then there is no issue with overlapping guard conditions because simultaneous events are not processed in parallel in UML. This is different in ModelicaML because events are processed in parallel and can overlap. However, it is not clear from the UML specification what should happen if conflicting transitions do not have any triggers but only have guard conditions defined, that can evaluate to true at the same time. This issue is addressed in section 4.4.1

Furthermore, in case the conflicting transitions are at different hierarchy levels, UML defines the following: "*In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an implicit priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy.*" This issue is described in section 4.4.2.

[6] This example is artificial and is meant for illustration purposes only. Normally, modeling such behaviour will probably be avoided.

4.4.1 Priorities for State-Outgoing Transitions

Consider the state machine in Figure 11. If *x* and *y* are greater than *2* at the same time both guard conditions evaluate to true. In ModelicaML, which transition will be taken then is determined by the transition execution priority defined by the modeler.

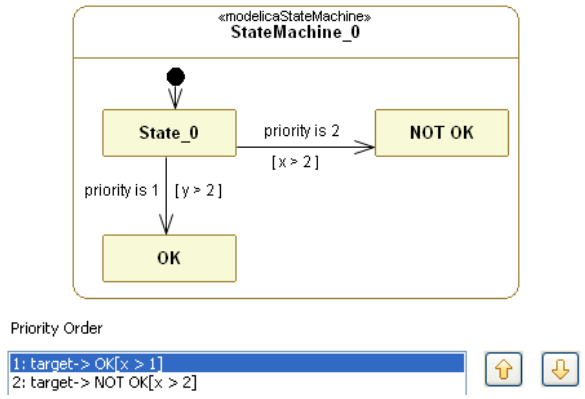

Figure 11: Priorities definition for state-outgoing transitions

As for regions (discussed in section 4.3), conflicting transitions coming out of a state are prioritized in ModelicaML. Priorities for transitions are set by default by the modeling tool. The modeler can change priorities and thereby ensure deterministic behavior.

4.4.2 Priority Schema for Conflicting Transitions at Different State Hierarchy Levels

UML defines the priority schema for conflicting transitions that are at different levels as follows (see p.567): "… *By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states."* No rationale is documented for this decision as pointed out in [11].

Consider the state machine in Figure 12. What should happen when *x* and *y* are greater than *2* at the same time? This case is not addressed in the UML specification because no triggers are defined for these transitions. One possible answer could be: Transition to state *NOT OK* is taken. Another answer could be: Transition to state *NOT OK* is taken and then transition to state *OK* is taken. Yet another answer could be: Transition to state *OK* is taken and since *State_0* is deactivated no further reaction inside *State_0* is to be expected. The latter is implemented in ModelicaML.

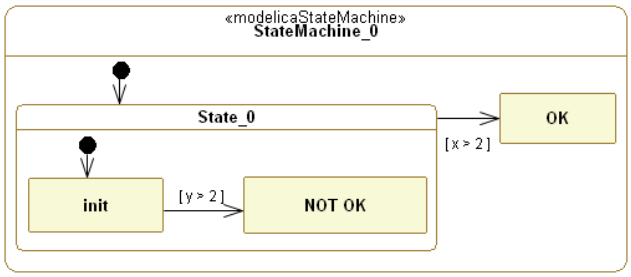

Figure 12: Transition at higher levels have higher priority

In ModelicaML the priority scheme is different from UML. In ModelicaML, outgoing transitions of a composite state have higher execution priority than transitions inside the composite state. The rationale for this decision is as follows:

- This semantics is more intuitive and clear. For example, if states are described using sub-state machines (presented in their own diagrams), the modeler cannot know if a particular transition is taken unless he or she has inspected all lower-level sub-state machines diagrams. The UML priority scheme can also lead to behavior where the composite state is never exited because events are consumed at some level further down the hierarchy of the composite state.
- The ModelicaML priority scheme reduces the complexity of the code generator. For example, as already mentioned above, event queues are not used in Modelica. To avoid that transitions of a composite state consume events that are already consumed inside the composite states, events need to be marked or removed from the queue. Such an implementation would drastically increase the size and complexity of the code generator as well as of the generated Modelica code.

4.5 Issues with Inter-Level Transitions

This section discusses issues concerning the execution order of actions as a result of transition executions. Actions can be *entry/do/exit* actions of states or *effect* actions of transitions. The order in which actions are executed is important when actions are dependable, i.e. if different actions set or read the same variables.

Consider the state machine in Figure 13. Assume that each state has entry and exit actions, and that *Region_0* is always given the highest priority and *Region_x* the lowest.

When the state machine is in state *a* and *cond1* is true, the question is in which order the states are activated and the respective entry actions are executed. This case is not addressed in the UML specification.

From the general UML state machine semantics definition we can deduce that sub-states cannot be activated as long as their containing (i.e. composite) state is not activated. For example, it is clear that the state *b* has to be

activated before the states *c* and *i* can become active. Furthermore, we can argue that since the modeler explicitly created an inter-level transition the state *e2* should be activated first, i.e. before states in neighboring regions at the same (i.e. *f*) or at a higher state hierarchy level (i.e. *g, h, or i*). Hence, when the inter-level transition from state *a* to state *e2* is taken the partial states activation sequence and its resulting order of entry-actions execution should be: *b, c, d, e2*. However, in which sequence shall the states *f, g, h* and *i* be activated? Possible answers are: *i, h, g, f,* or *i, g, h, f* , or *f, g, h, i*, or *f, h, g, i*.

In ModelicaML, this issue is resolved as follows: First all states containing the target state and the target state itself are activated. Next, based on the region execution priority, the initial states from neighboring regions are activated. Since the priority in this example is defined for regions from left (highest) to right (lowest) for each composite state, the activation order would be *b, c, d, e2, f, g, h, i*. Vice versa, if the region priority would be defined the other way around (from right to left) the activation order would be *b, c, d, e2, i, h, g, f*.

A similar issue exists regarding the transition to state *a* when the state machine is in state *e2* and when *cond2* is true. Here the states deactivation and exit actions execution order are involved. The resulting deactivation sequence in ModelicaML would be: *e2, f, d, g* and *h* (based on the region priority definition), c, *i* and *b*.

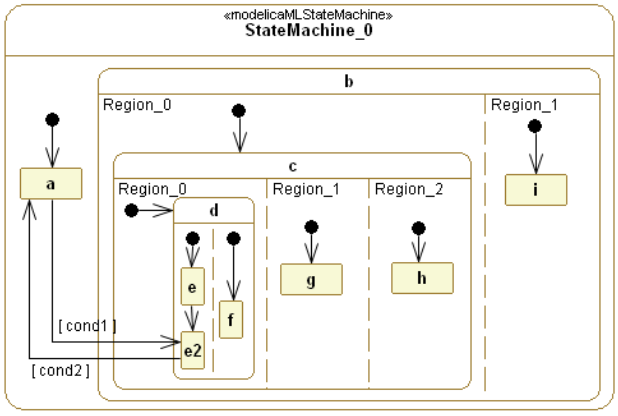

Figure 13: Inter-level transition example

4.6 Issues with Fork and Join

Like section 4.5, this section addresses issues regarding the activation and deactivation of states. However, in this case, UML fork and join constructs are regarded. Consider the state machine on Figure 14. With the state machine in state *a* the questions are:

- In which sequence are states *b, c, d, e,* and *f* activated when the transitions (fork construct) from state *a* is executed?
- In which sequence are states *b, c, d, e,* and *f* deactivated when the transitions (join construct) to state *g* are executed?

This case is also not addressed in the UML specification. In ModelicaML, first the parent states of the transition target-states are activated. Then the target states themselves are activated based on the fork-outgoing transition priority. Next the initial states in the neighboring regions are activated based on the region priority definition.

Again assume that each state has entry and exit actions, and that *Region_0* is always given the highest priority and *Region_x* the lowest. The resulting states activation sequence (and respective execution of entry actions) for the fork construct would be: *b, d* and *e* (based on the fork-outgoing transitions priority), *c* and *f* (based on their region priority). The resulting deactivation for the join construct would be: *d* and *e* (based on the join-incoming transitions priority), *c* and *f* (based on their region priority definition), *b*. In any case, the modeler can define the execution order explicitly.

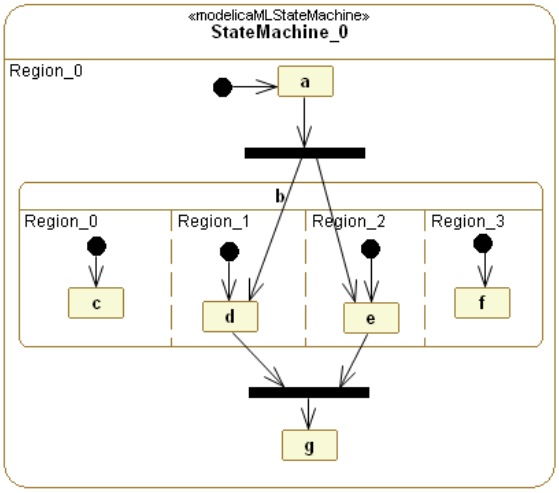

Figure 14: Fork and join example

5 Conclusion

This paper presents a proof of concept for the translation of UML state machines into executable Modelica code. It presents how the executable semantics of UML state machines are defined in ModelicaML and how state machines are used to model parts of class behavior in ModelicaML. The ModelicaML prototypes can be downloaded from [15].

Furthermore, this paper highlights issues that should be addressed in the UML specification and makes proposals on how to resolve them. Section 4.2 questions the use of an event queue that prevents simultaneous events from being processed in parallel. When procedural code is used for the implementation of state machines execution, sec-

tion 4.3 makes a proposal to include priority for regions. A regions priority also supports the definition of states activation or deactivation order in case of inter-level state transitions (section 4.5) or fork/join constructs (section 4.6). Section 4.4.1 introduces execution priority for conflicting state-outgoing transitions in order to allow the modeler to control the execution and to ensure that the state machine behaves as intended.

References

[1] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. www.modelica.org

[2] OMG. OMG Unified Modeling Language ™ (OMG UML). Superstructure Version 2.2, February 2009.

[3] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, 2004.

[4] OMG. OMG Systems Modeling Language (OMG SysML™), Version 1.1, November 2008.

[5] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattsson, Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. Proceedings of the 7th International Modelica Conference, Como, Italy. September 20-22, 2009.

[6] Ferreira J. A. and Estima de Oliveira J. P., Modelling Hybrid Systems Using Statecharts And Modelica. Department of Mechanical Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL), Department of Electronic Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL)

[7] Adrian Pop, David Akhvlediani, Peter Fritzson. Towards Unified Systems Modeling with the ModelicaML UML Profile. International Workshop on Equation-Based Object-Oriented Languages and Tools. Berlin, Germany, Linköping University Electronic Press, www.ep.liu.se, 2007

[8] Thomas Johnson, Christian Paredis, Roger Burkhart. Integrating Models and Simulations of Continuous Dynamics into SysML. www.omgsysml.org

[9] Johnson, T. A. Integrating Models and Simulations of Continuous Dynamic System Behavior into SysML. M.S. Thesis, G.W. Woodruff School of Mechanical Engineering, Georgia Institute of Technology. Atlanta, GA. 2008

[10] M. von der Beeck. A Comparison of Statecharts Variants. In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 863, pages 128-148. Springer, 1994.

[11] Michelle L. Crane and Juergen Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal School of Computing, Queen's University Kingston, Ontario, Canada

[12] Object Management Group (OMG). www.omg.org

[13] Wladimir Schamai, Peter Fritzson, Chris Paredis, Adrian Pop. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. Proceedings of the 7th International Modelica Conference, Como, Italy. September 20-22, 2009

[14] IBM® Rational® Rhapsody® Designer for Systems Engineers, http://www-01.ibm.com/software/rational/products/rhapsody/designer/

[15] ModelicaML - A UML Profile for Modelica. www.openmodelica.org/index.php/developer/tools/134