State Machines in Modelica

Hilding Elmqvist[1] Fabien Gaucher[2] Sven Erik Mattsson[1] Francois Dupont[3]
[1]Dassault Systèmes AB, Ideon Science Park, SE-223 70 Lund, Sweden
[2]Dassault Systèmes, 84, Allée Galilée, 38330-Montbonnot-St-Martin, France
[3]Dassault Systèmes, 120, rue René Descartes, 29280 – Plouzané, France
Hilding.Elmqvist@3ds.com Fabien.Gaucher@3ds.com
SvenErik.Mattsson@3ds.com Francois.Dupont@3ds.com

Abstract

The scope of Modelica has been extended from a language primarily intended for physical systems modeling to modeling of complete systems by allowing the modeling of control systems including state machines.

This paper describes the state machines introduced in Modelica 3.3. Any block without continuous-time equations or algorithms can be a state of a state machine. Transitions between such blocks are modeled by a new kind of connections associated with transition conditions.

The paper gives the details for building state machines and includes several examples. In addition, the complete semantics is described using only 13 Modelica equations.

Keywords: Modelica; State Machines; Control;

1 Introduction

The scope of Modelica has been extended from a language primarily intended for physical systems modeling to modeling of complete systems by allowing the modeling of control systems including state machines and enabling automatic code generation for embedded systems.

This paper presents state machines in Modelica. A companion paper (*Elmqvist, et.al, 2012*) describes the fundamental synchronous language primitives introduced for increased correctness of control systems implementation since many more checks can be done at compile time.

The paper describes language elements to define state machines. Any block without continuous-time equations or algorithms can be a state of a state machine. Transitions between such blocks are represented by a new kind of connections associated with transition conditions.

The paper gives the details for building state machines and includes several examples. In addition, the complete semantics is described using only 13 Modelica equations.

2 States and Transitions

Modelica State Machines will be introduced gradually by means of examples.

Modelica block instances without continuous-time equations or algorithms can potentially be states of a state machine. A cluster of block instances at the same hierarchical level which are coupled by **transition** equations constitutes a state machine. All parts of a state machine must have the same clock. One and only one instance in each state machine must be marked as initial by appearing in an **initialState** equation.

2.1 A Simple State Machine

As a first example, consider the trivial state machine of Figure 1.

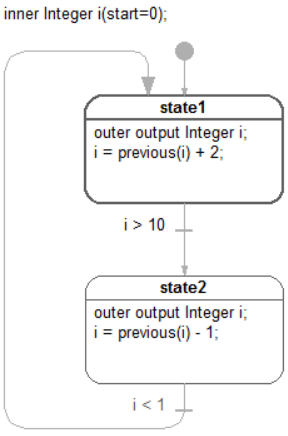

Figure 1. A simple state machine

An inner variable i is defined in the model which has two block instances state1 and state2. In the corresponding block definitions, i is declared as 'outer output' which means that i is an output from both of the blocks. In state1, i is incremented by 2 and in state2, i is decremented by 1. How such multiple definitions are handled is described below.

If state1 is active, a transition to state2 is made when i>10. If state2 is active, a transition to state1 is made when i<1.

The simulation result is shown in Figure 2.

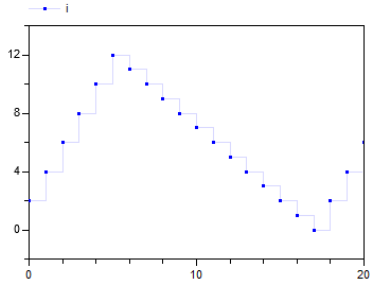

Figure 2. Plot of v of simple state machine

The Modelica code (without annotations) is:

```
model StateMachine1
  inner Integer i(start=0);

  block State1
    outer output Integer i;
  equation
    i = previous(i) + 2;
  end State1;
  State1 state1;

  block State2
    outer output Integer i;
  equation
    i = previous(i) - 1;
  end State2;
  State2 state2;

equation
  initialState(state1);
  transition(state1, state2, i > 10, immediate=false);
  transition(state2, state1, i < 1, immediate=false);
end StateMachine1;
```

2.2 Merging Variable Definitions

When a state class uses a variable in an **outer output** declaration, the equations have access to the corresponding variable declared **inner**. Special rules are then needed to maintain the single assignment rule since multiple definitions of such outer variables in different mutually exclusive states of one state machine need to be merged.

In each state, the outer output variables (v_j) are solved for ($expr_j$) and, for each such variable, a single definition is automatically formed:

v := **if** activeState($state_1$) **then** $expr_1$
elseif activeState($state_2$) **then** $expr_2$
elseif … **else** last(v)

last() is a special internal semantic operator returning its input. It is just used to mark for the sorting that the incidence of its argument should be ignored. A start value must be given to the variable if not assigned in the initial state.

Such a newly created assignment equation might be merged on higher levels in nested state machines.

2.3 Defining a state machine

The following special kinds of connect-equations are used to define transitions between states and to define the initial state:

transition(from, to, condition, immediate, reset, synchronize, priority)
Arguments "from" and "to" are block instances and "condition" is a Boolean expression. The optional arguments "immediate", "reset", and "synchronize" are of type Boolean, have parametric variability and a default of true, true, false respectively. The optional argument "priority" is of type Integer, has parametric variability and a default of 1. This operator defines a transition from instance "from" to instance "to". The "from" and "to" instances become states of a state machine. The transition fires when condition = true if immediate = true (this is called an "immediate transition") or **previous**(condition) when immediate = false (this is called a "delayed transition"). The argument "priority" defines the priority of firing when several transitions could fire. priority=1 is the highest priority. If reset = true, the states of the target state are reinitialized, i.e. state machines are restarted in initial state and state variables are reset to their start values. If synchronize=true, the transition is disabled until all state machines within the from-state have reached the final states, i.e. states without outgoing transitions.
initialState(state)
The argument "state" is the block instance that is defined to be the initial state of a state machine. At the first clock tick of the state machine, this state becomes active.

DOI 10.3384/ecp1207637

The attributes of transitions are shown graphically as illustrated in Figure 3.

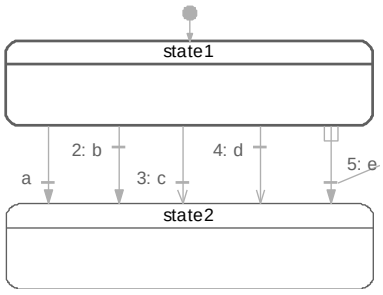

Figure 3. Graphical conventions for transitions

A transition has a perpendicular bar representing the condition which is close to the destination state for an immediate transition, else close to the source state. The arrow is filled for a reset transition otherwise non-filled. A synchronize transition has an "inverted fork" at the source state. Priority is shown preceding the condition if not equal to one. For the 5 transitions in Figure 3, the settings are as follows, from left to right:

immediate	= true, false, true, false, true;
reset	= true, true, false, false, true;
synchronize	= false, false, false, false, true;
priority	= 1, 2, 3, 4, 5.

All transitions leaving the same state must have different priorities.

It is possible to query the status of the state machine by using the following operators:

activeState(state)	Argument "state" is a block instance. The operator returns true, if this instance is a state of a state machine and this state is active at the actual clock tick. If it is not active, the operator returns false. It is an error if the instance is not a state of a state machine.
ticksInState()	Returns the number of clock ticks since a transition was made to the currently active state. This function can only be used in transition conditions of state machines not present in states of higher level state machines.
timeInState()	Returns the time duration as Real in [s] since a transition was made to the currently active state. This function can only be used in transition conditions of state machines not present in states of higher level state machines.

2.4 Immediate and Delayed Transitions

If we attempt to simulate the state machine in Figure 1 with transitions having immediate=true, we get the error message in Dymola:

```
An algebraic loop involving Integers or
Booleans has been detected.
```

The reason is that since the transition conditions involve i, the variable defined in the equations, there is a cyclic dependency or algebraic loop between the update equations for i and the update equations for state machine evolution.

2.5 Conditional Data Flows

An alternative to using outer output variables is to use conditional data flows. Since instances of blocks can be used as states of a state machine, the connection semantics of Modelica has been extended to allow several outputs to be connected to one input.

Consider the combined state machine and data flow diagram in Figure 4:

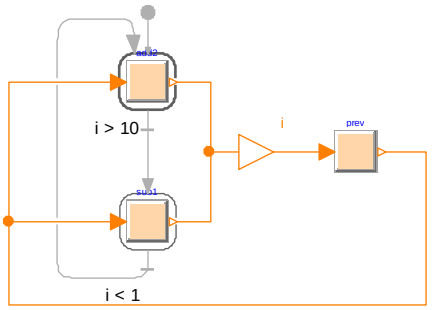

Figure 4. Combined state machine and data flow diagram

The states are instances of the block:

```
block Increment
  extends Modelica.Blocks.Interfaces.PartialIntegerSISO;
  parameter Integer increment;
equation
  y = u + increment;
end Increment;
```

with increment values 2 and -1 respectively. The outputs are connected to a protected connector called i in order to be able to use i in the transition conditions. The connector i is connected to an instance of the block:

```
block Prev
  extends Modelica.Blocks.Interfaces.PartialIntegerSISO;
equation
  y = previous(u);
```

```
end Prev;
```

The connections from the state instances to i in Figure 4 are handled in a special way. It is possible to connect several outputs to inputs if all the outputs come from states of the same state machine. In such cases, we get the following constraint equations:

$u_1 = u_2 = \ldots = y_1 = y_2 = \ldots$

with u_i inputs and y_i outputs. The semantics is defined as follows. Introduce a variable v representing the signal flow and rewrite the equation above as a set of equations for u_i and a set of assignment equations for v:

v := **if** activeState($state_1$) **then** y_1 **else** last(v);
v := **if** activeState($state_2$) **then** y_2 **else** last(v);
…
u_1 = v
u_2 = v
…

The merge of the definitions of v is then made according to section 'Merging Variable Definitions'. The result of the merge is:

v = **if** activeState($state_1$) **then** y_1
elseif activeState($state_2$) **then** y_2
elseif … **else** last(v)
…

Plotting i shows the same behavior as the plot of i of the example using inner outer declarations.

3 Hierarchical State Machine Example

Consider the hierarchical state machine in Figure 5:

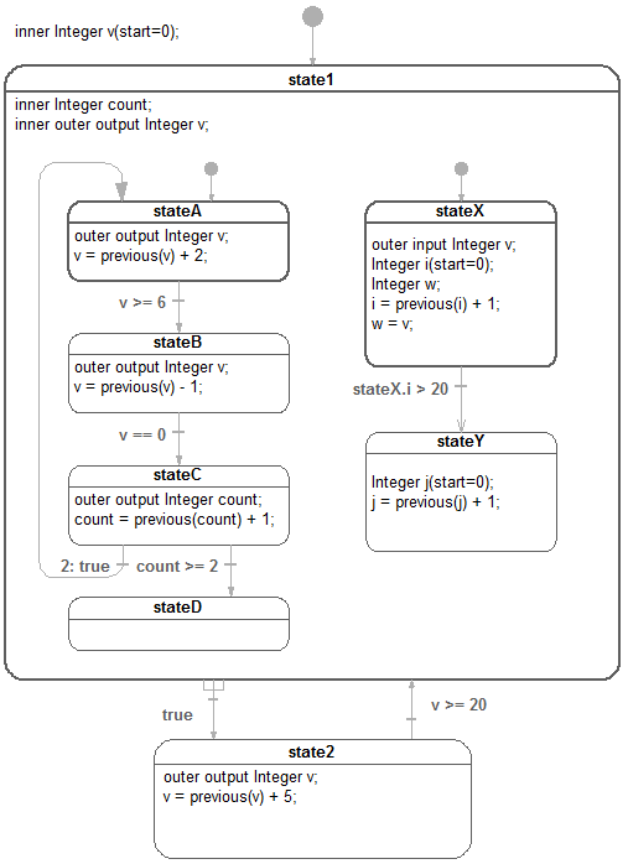

Figure 5. Hierarchical state machine

The model demonstrates the following properties:

- state1 is a meta state with two parallel state machines in it.
- stateA declares v as 'outer output'. state1 is on an intermediate level and declares v as 'inner outer output', i.e. matches lower level outer v by being inner and also matches higher level inner v by being outer. The top level declares v as inner and gives the start value.
- count is defined with a start value in state1. It is reset when a reset transition (v>=20) is made to state1.
- stateX declares the local variable w to be equal to v declared as 'inner input'.
- stateY declares a local counter j. It is reset at start and as a consequence of the reset transition (v>=20) from state2 to state1. However, the reset of j is deferred until stateY is entered by transition (stateX.i>20) although this transition is not a reset transition. This is done by marking that stateY should be reset when making the reset transition v>=20 and deferring the reset until stateY is actually entered. Synchronizing the exit from the two parallel state machines of state1 is done by using a synchronized transition.

DOI 10.3384/ecp1207637

The behavior of the state machine can be seen in the plots of v and w and i of Figure 6:

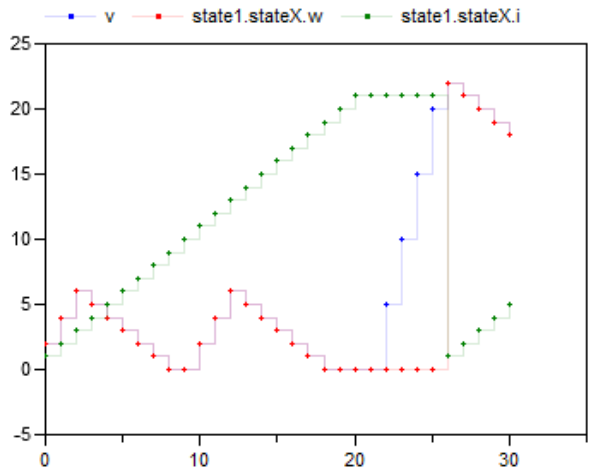

Figure 6. Behavior of hierarchical state machine

4 Adaptive Cruise Control Example

As a more useful example, we will consider a vehicle with adaptive cruise control, i.e. controller that can drive the car at a certain speed or follow the car in front at a safe distance.

The example is simplified considerably to be able to explain all the details in limited space. And the data is just designed for illustrative purposes.

The vehicle dynamics is described by the following model (without annotations):

```
model Vehicle
  parameter Real k=5000;
  parameter Real m=1000;
  parameter Real loss=5;
  Modelica.Blocks.Interfaces.RealInput ud;
  Modelica.Blocks.Interfaces.RealOutput xd;
  Modelica.Blocks.Interfaces.RealOutput vd;

  Modelica.SIunits.Distance x(start=0, fixed=true);
  Modelica.SIunits.Velocity v(start=0, fixed=true);
  Real tau;
equation
  der(x) = v;
  m*der(v) = k*tau - loss*v*abs(v);

  tau = hold(ud);
  xd = sample(x, Clock(1, 10));
  vd = sample(v, Clock(1, 10));
end Vehicle;
```

The power train is considered ideal.

A vehicle with the cruise control system is shown in Figure 7. It has an instance of the vehicle dynamics (with a car icon) with a sampled input ud on the left and two sampled outputs (period=1/10 second), xd and vd (counting from the top) to the right.

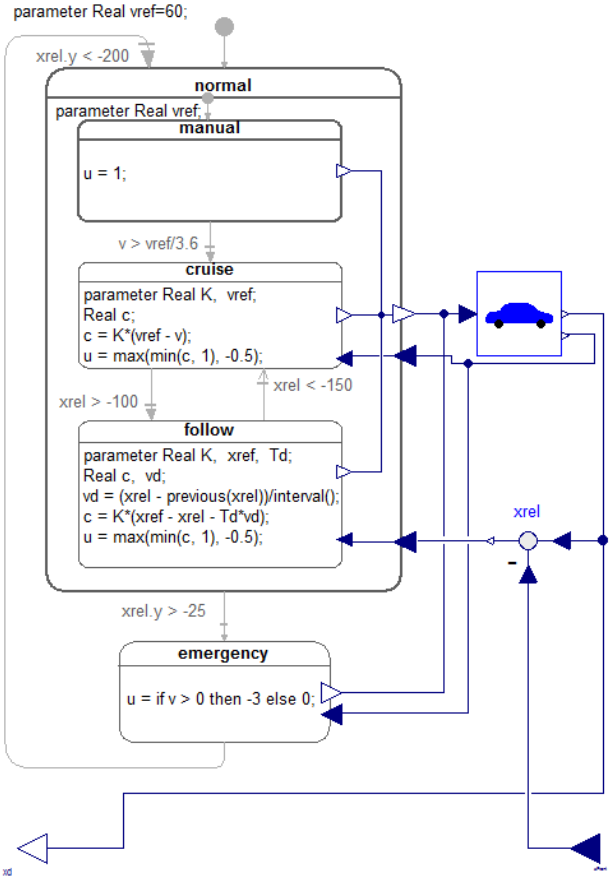

Figure 7. Vehicle with adaptive cruise controller

The top level state machine has two modes: *normal* and *emergency*. Both produces the control signal u connected to ud of the vehicle. The normal mode has vd and xrel as inputs. xrel is formed as the difference between the vehicle position and the position of the vehicle in front, available as an input.

The normal state has three states: *manual*, *cruise* and *follow*. The manual state is a simple start up state "stepping on the gas" until the desired speed has been achieved. The cruise state contains a speed controller implemented as a simple P-controller with limitation.

When the vehicle comes within 100 meters of the vehicle in front, *follow* state is entered. It contains a position controller with xref=-100. Since the vehicle is essentially a double integrator from throttle to position, a PD controller is needed. In this case a naïve implementation without filtering is shown. When the distance is larger than 150 meters, *cruise* mode is reentered.

The emergency state is entered when the distance to the car in front is less than 25 meters independently in which substate normal is in. Maximum braking power (-3) is then applied until the car has stopped. When the distance is again 200 meters, the normal

state is entered with a reset transition, i.e. the substate of manual of state normal is activated.

The architecture with two entirely different controllers for speed and position was chosen to illustrate the possibility in particular regarding how the data flow connections can be used. (Adaptive cruise control can also be achieved using a cascade controller with an inner speed loop.)

A model of a platoon of 5 CruisingVehicles was built. The desired speed vref is set as {100, 60, 65, 50, 25} km/h. The initial speeds are the same except for the last car (cruisingVehicle) which is standing still. The distances between the cars are 200 meters.

The results of simulation are shown in Figure 8: position on top and velocity below. All cars slow down to follow the first car (cruisingVehicle4) at 25 km/h at a distance of 100 meter.

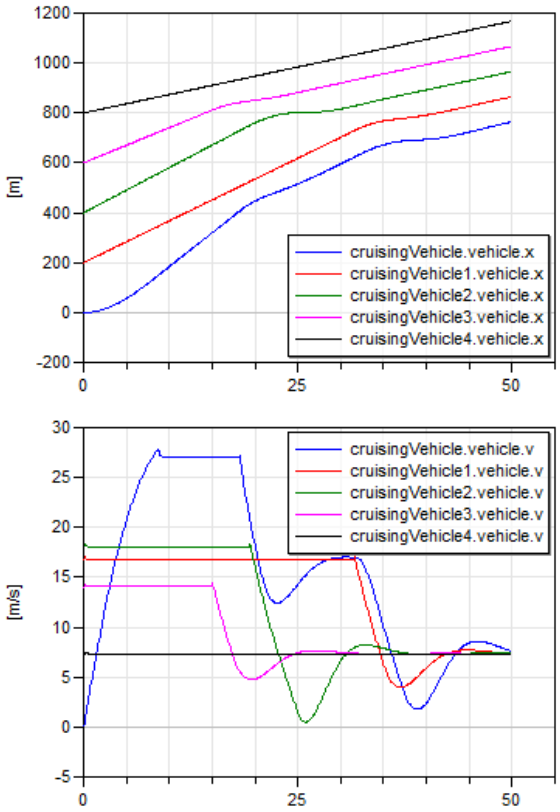

Figure 8. Positions and velocities of vehicles in a platoon

The control signals are shown in Figure 9.

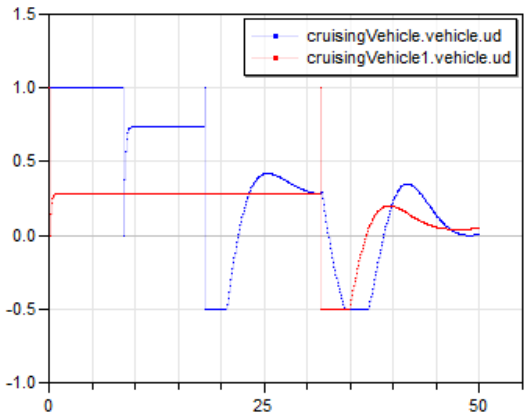

Figure 9: Control signals

The implementation of the cruise state shown in Figure 7 is a bit simplified using a parameter vref for the velocity set point. Usually, the triggering of going from manual to cruise mode is done by a button. The cruise mode is then picking up the current speed and uses that as a set point. Such an implementation can be made as follows:

```
model Cruise
  parameter Real K = 1;
  Real c, vref;
  Boolean reinit(start=true) = false;
  Modelica.Blocks.Interfaces.RealOutput u;
  Modelica.Blocks.Interfaces.RealInput v;
equation
  vref = if previous(reinit) then v else previous(vref);
  c = K*(vref-v);
  u = max(min(c, 1),-0.5);
end Cruise;
```

This is a general modeling idiom for special treatment when a state is entered. The equation for reinit is reinit = false. However, the start value is true, so previous(reinit) gives a pulse at the first cycle if a reset transition is made to the state.

So the desired behavior is achieved by a reset transition from manual to cruise, but a non-reset transition from follow to cruise, since in the last case, the stored vref should be used.

A platoon of 100 vehicles can easily be constructed using an array of CruisingVehicles:

```
model Platoon
  parameter Integer n=100;
  CruisingVehicle cruisingVehicle[n](vref=linspace(100, 50.5, n));
  Modelica.Blocks.Sources.Constant const(k=10000);
equation
  connect(const.y, cruisingVehicle[n].xFront);
  for i in 1:n-1 loop
    connect(cruisingVehicle[i+1].xd,
            cruisingVehicle[i].xFront);
  end for;
end Platoon;
```


DOI 10.3384/ecp1207637

This is a good example of how well the state machine concept is integrated in Modelica allowing to use data flows between states, using modifiers for parameterization, using redeclare of classes and components and using arrays of a mixture of state machines and continuous dynamical models.

5 State Machine Semantics

This section is not intended for normal users of Modelica state machines. It is included since the precise semantics can be described using only 13 Modelica equations and is thus a convenient reference for advanced users and tool developers.

For the purpose of defining the semantics of state machines, assume that the data of all transitions are stored in an array of records, t:

```
record Transition
  Integer from;
  Integer to;
  Boolean immediate = true;
  Boolean reset = true;
  Boolean synchronize = false;
  Integer priority = 1;
end Transition;
```

The transitions are sorted with lowest priority number last in the array. The states are enumerated from 1 and up. The transition conditions are stored in a separate array c[:] since they are time varying.

The semantics model is a discrete-time system with inputs {c[:], active, reset}, outputs {activeState, activeReset, activeResetStates[:]} and states {nextState, nextReset, nextResetStates[:]}. For a top level state machine, active is always true. For sub-state machines, active is true only when the parent state is active. For a top level state machine, reset is true at the first activation only. For sub-state machine, reset is propagated from the state machines higher up.

5.1 State Activation

The state update starts from nextState, i.e.,what has been determined to be the next state at the previous time. selectedState takes into account if a reset of the state machine is to be done.

```
output Integer selectedState =
  if reset then 1 else previous(nextState);
```

The integer fired is calculated as the index of the transition to be fired by checking that selectedState is the from-state and the condition is true for an immediate transition or previous(condition) is true for a delayed transition. The max function returns the index of the transition with highest priority or 0.

```
Integer fired =
  max(if (if t[i].from == selectedState then (if t[i].immediate
  then c[i] else previous(c[i])) else false) then i else 0
  for i in 1:size(t,1));
```

The start value of c is false. This definition would require that the previous value is recorded for all transitions conditions. Below is described an equivalent semantics which just requires to record the value of one integer variable delayed. The integer immediate is calculated as the index of the immediate transition to potentially be fired by checking that selectedState is the from-state and the condition is true. The max function returns the index of the transition with true condition and highest priority or 0.

```
Integer immediate =
  max(if (if  t[i].immediate and t[i].from == selectedState then
   c[i] else false) then i else 0 for i in 1:size(t,1));
```

In a similar way, the Integer delayed is calculated as the index for a potentially delayed transition, i.e. a transition taking place at the next clock tick. In this case the from-state needs to be equal to nextState:

```
Integer delayed =
  max(if (if not t[i].immediate and t[i].from == nextState  then
   c[i] else false) then I else 0 for i in 1:size(t,1));
```

The transition to be fired is determined as follows, taking into account that a delayed transition might have higher priority than an immediate:

```
Integer fired = max(previous(delayed), immediate);
```

nextState is set to the found transitions to-state:

```
Integer nextState = if active then (if fired > 0 then t[fired].to
  else selectedState) else previous(nextState);
```

In order to define *synchronize* transitions, each state machine must determine which are the final states, i.e. states without from-transitions and to determine if the state machine is in a final state currently:

```
Boolean finalStates[nStates] =
  {max(if t[j].from == i then 1 else 0 for j in 1:size(t,1)) == 0
   for i in 1:nStates};
Boolean stateMachineInFinalState = finalStates[activeState];
```

To enable a *synchronize* transition, all the stateMachineInFinalState conditions of all state machines within the meta state must be true.

5.2 Reset Handling

A state can be reset for two reasons:

- The whole state machine has been reset from its context. In this case, all states must be reset, and the initial state becomes active.
- A reset transition has been fired.
  Then, its target state (and its sub-state machines) are reset, but not other states.

The first reset mechanism is handled by the activeResetStates and nextResetStates vectors. The state machine reset flag is propagated and maintained to each state individually:

```
output Boolean activeResetStates[nStates] =
  {if reset then true else previous(nextResetStates[i])
  for i in 1:nStates};
```

until a state is eventually executed, then its corresponding reset condition is set to false:

```
Boolean nextResetStates[nStates] = if active then
  {if activeState == i then false else activeResetStates[i]
  for i in 1:nStates}
```

The second reset mechanism is implemented with the selectedReset and nextReset variables. If no reset transition is fired, the nextReset is set to false for the next cycle.

5.3 Activation handling

The execution of a sub-state machine has to be suspended when its enclosing state is not active. This activation flag is given as a Boolean input `active`. When this flag is true, the sub-state machine maintains its previous state, by guarding the equations of the state variables `nextState`, `nextReset` and `nextResetStates`.

5.4 Semantics Summary

The entire semantics model is given below:

```
model StateMachineSemantics "Semantics of state machines"
  parameter Integer nStates;
  parameter Transition t[:]
    "Array of transition data sorted in priority";
  input Boolean c[size(t,1)]
    "Transition conditions sorted in priority";
  Boolean active "true if the state machine is active";
  Boolean reset "true when the state machine should be reset";

  Integer selectedState = if reset then 1 else previous(nextState);
  Boolean selectedReset = if reset then true
     else previous(nextReset);

// For strong (immediate) and weak (delayed) transitions
  Integer immediate = max(if (if t[i].immediate and t[i].from ==
       selectedState then c[i] else false) then i else 0
     for i in 1:size(t,1));

  Integer delayed = max(if (if not t[i].immediate and t[i].from ==
       nextState then c[i] else false) then i else 0 for i in 1:size(t,1));

  Integer fired = max(previous(delayed), immediate);
  output Integer activeState = if reset then 1
     elseif fired > 0 then t[fired].to else selectedState;
  output Boolean activeReset = if reset then true
     elseif fired > 0 then t[fired].reset else selectedReset;

// Update states
  Integer nextState = if active then activeState
     else previous(nextState);
  Boolean nextReset = if active then false
     else previous(nextReset);

// Delayed resetting of individual states
  output Boolean activeResetStates[nStates] = {if reset then true
       else previous(nextResetStates[i]) for i in 1:nStates};
  Boolean nextResetStates[nStates] = if active then
     {if selectedState == i then false else activeResetStates[i]
     for i in 1:nStates}
     else previous(nextResetStates);

  Boolean finalStates[nStates] = {max(if t[j].from == i then 1 else 0
     for j in 1:size(t,1)) == 0 for i in 1:nStates};
  Boolean stateMachineInFinalState = finalStates[activeState];
end StateMachineSemantics;
```

6 Comparison to Other State Machine Formalisms

State machines needed to be introduced in Modelica to enable modeling of complete systems. Several attempts have been made: (*Mosterman et. al. 1998)*, defines state machines in an object-oriented way with Boolean equations. A more powerful state machine formalism was introduced in StateGraph (*Otter et. al. 2005*). A prototype mode automata formalism was implemented (*Malmheden et. al. 2008*) using a built-in concept of modes. Certain problems of potentially unsafe models in StateGraph were removed in the StateGraph2 library (*Otter et. al. 2009*). These efforts showed that state machine support must be natively supported in the language.

The presented state machines of Modelica 3.3 have a similar modeling power as Statecharts (*Harel, 1987*) and State Machine Diagrams of SysML (*Friedenthal 2008*).

The semantics of the state machines defined in this paper is inspired by mode automata (*Maraninchi 2002*) and basically the same as Lucid Synchrone 3.0 (*Pouzet 2006*), or its clone LCM (Logical Control Module) (*Gaucher et.al. 2009*). Some minor properties are different compared to Lucid Synchrone 3.0,

DOI 10.3384/ecp1207637

in particular regarding transition conditions. Lucid Synchrone has two kinds of transitions: namely "strong" and "weak". Strong transitions are executed before the actions of a state are evaluated while weak transitions are executed after. This can lead to surprising behavior, because the actions of a state are skipped if it is activated by a weak transition and exited by a true strong transition. For this reason, the state machines in Modelica use "immediate" (= the same as "strong") and "delayed" transitions. Delayed transitions are "immediate" transitions where the condition is automatically delayed with an implicit **previous**(...).

Note that safety critical control software in aircrafts is often defined with such kind of state machines, such as using the Scade 6 Tool Suite from Esterel Technologies (*Dormoy 2008*) that provides a similar formalism as Lucid Synchrone, with minor differences such as the ability to associate actions to transitions in addition to states. Scade also provides synchronize semantics by means of synchronization transitions between several parallel sub-state machines being in states which have been declared final.

Stateflow (*Mathworks 2012*), while being very expressive, suffers from "numerous, complex and often overlapping features lacking any formal definition", as reported by (*Hamon, et.al, 2004)*.

The presented Modelica approach has the important feature that at one clock tick, there is only one assignment to every variable (for example, it is an error if state machines are executed in parallel and they assign to the same variable at the same clock tick; such errors are detected at compile-time).

Modelica, Lucid Synchrone, LCM and Scade 6 all have the property that data flow and state machines can be mutually hierarchically structured, i.e. that, for example a state of a state machine can contain a block diagram in which the blocks might contain state machines.

7 Conclusions

We have described how state machines can be modeled in Modelica 3.3. Instances of blocks connected by transitions with one such block marked as an initial state constitute a state machine. Hierarchical state machines can be defined with reset or resume semantics, when re-entering a previously executed state. Parallel sub-state machines can be synchronized when they reached their final states. Special merge semantics have been defined for multiple outer output definitions in mutually exclusive states as well as conditional data flows.

8 Acknowledgements

The authors are very thankful to Albert Benveniste, Marc Pouzet, Martin Otter, Martin Malmheden, Daniel Weil, Torsten Blochwitz, Peter Fritzson, Carl-Fredrik Abelson, Hans Olsson and other Modelica Association members for stimulating discussions and feedback during evolutions of the Modelica 3.3 specification.

The authors appreciate the partial funding of this work by the Swedish funding organization VINNOVA (funding number: 2008-02291) within the ITEA2 MODELISAR project (http://www.itea2.org/project/result/download/result/5533).

References

Dormoy F.X. (2008): **SCADE 6 A Model Based Solution For Safety Critical Software Development**, ERTS EMBEDDED REAL TIME SOFTWARE 2008, TOULOUSE, FRANCE, http://www.esterel-technologies.com/EN-50128/files/ERTS2008-SCADE-6-A-Model-Based-Solution-For-Safety-Critical-Software.pdf

Elmqvist H., Otter M., and Mattsson S.E. (2012): **Fundamentals of Synchronous Control in Modelica**. Proceedings of 9th International Modelica Conference, Munich, Germany, September 3-5.

Friedenthal S., Moore A., and Steiner R. (2008): **A Practical Guide to SysML –The Systems Modeling Language**, Elsevier Inc.

Gaucher F., Closse E., Weil D. (2009): **The LCM Language Primer**, Dassault Systèmes Internal Report, Grenoble, France, 2009

Hamon G., and Rushby J. (2004). **An operational semantics for Stateflow**. In Fundamental Approaches to Software Engineering (FASE)'04, volume 2984 of LNCS, pages 229–243, Barcelona, Spain, 2004. Springer. http://fm.csl.sri.com/~rushby/papers/sttt07.pdf

Harel, D. (1987): **Statecharts: A Visual Formalism for Complex Systems**. Science of Computer Programming 8, 231-274. Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel. www.inf.ed.ac.uk/teaching/courses/seoc1/-2005_2006/resources/statecharts.pdf

Malmheden M., Elmqvist H., Mattsson S.E., Henriksson D., and Otter M. (2008): **ModeGraph - A Modelica Library for Embedded Control**

Based on Mode-Automata. B. Bachmann (editor), in Proc. of Modelica'2008 conference, Bielefeld, Germany. www.modelica.org/events/modelica2008/Proceedings/sessions/session3a3.pdf

Maraninchi, F. and Rémond, Y. (2002): **Mode-Automata: a New Domain-Specific Construct for the Development of Safe Critical Systems.** http://wwwverimag.imag.fr/~maraninx/SCP2002.html

MathWorks (2012): **R2012a Documentation - Stateflow** http://www.mathworks.com/help/toolbox/stateflow/

Modelica Association (2012): **Modelica Language Specification Version 3.3**. https://www.modelica.org/documents/ModelicaSpec33.pdf.

Mosterman P., M. Otter, and H. Elmqvist. (1998): **Modeling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica**. Proceedings of SCSC'98, Reno, Nevada, USA, Society for Computer Simulation International, pp. 314–319.

Otter M., K.-E. Årzén, and I. Dressler (2005): **StateGraph – A Modelica Library for Hierarchical State Machines**. Proceedings of the 4th International Modelica Conference, Hamburg, Germany, ed. G. Schmitz, pp. 569-578. http://www.modelica.org/events/Conference2005/online_proceedings/Session7/Session7b2.pdf

Otter M., Malmheden M., Elmqvist H., S.E. Mattsson, and C. Johnsson (2009): **A New Formalism for Modeling of Reactive and Hybrid Systems**. Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009. http://www.ep.liu.se/ecp/043/041/ecp09430108.pdf

Pouzet M. (2006): **Lucid Synchrone, Version 3.0, Tutorial and Reference Manual**. http://www.di.ens.fr/~pouzet/lucid-synchrone/

 DOI 10.3384/ecp1207637