Flattening of Modelica State Machines: A Practical Symbolic Representation

Bernhard Thiele[1] Adrian Pop[1] Peter Fritzson[1]

[1]PELAB, Linköping University, Sweden, {bernhard.thiele,adrian.pop,peter.fritzson}@liu.se

Abstract

Modelica 3.3 introduced dedicated built-in language support for *state machines* that was inspired by semantics known from *Statechart* and *mode automata* formalisms. The specification describes the semantics of these constructs in terms of data-flow equations that allows it to be related to the Modelica DAE representation which is the conceptual intermediate format of Modelica code after instance creation (flattening). However, a complete transformation of state machine constructs into data-flow equations at the stage of flattening requires an early commitment to implementation details that potentially hinders model optimizations at subsequent translation phases. Also, due to the required substantial model transformation the semantic distance between the original source model and the *flattened* representation is rather large. Hence, this paper proposes a more versatile symbolic representation for flattened state machine constructs that preserves the state machine's composition structure and allows postponing optimizations to subsequent compiler phases.
Keywords: state machine, mode automata, flattening, compilation

1 Introduction

The scope of the Modelica specification is briefly stated in (Modelica Association, 2012, Section 1.2):

> The semantics of the Modelica language is specified by means of a set of rules for translating any class described in the Modelica language to a flat Modelica structure. A class must have additional properties in order that its flat Modelica structure can be further transformed into a set of differential, algebraic and discrete equations (= hybrid DAE). Such classes are called simulation models.

A typical compilation process for a Modelica language tool is structured as depicted in Figure 1. *Flat Modelica* is an intermediate representation which is further elaborated into a representation from which optimized simulation code can be generated. Conceptually, flat Modelica is closely related to a *hybrid DAE* (hybrid Differential Algebraic Equation) representation. This relationship is discussed in (Modelica Association, 2012, Appendix C). The mapping from flat Modelica to a hybrid DAE is a powerful concept, since it provides a mathematical foundation for the semantics of flat Modelica.

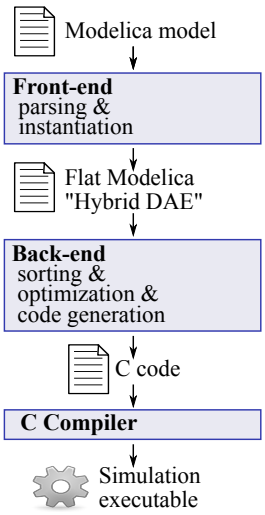


Figure 1. Outline of a typical compilation process for a Modelica language tool.

Modelica 3.3 introduced dedicated built-in language support for *clocked state machines* that was inspired by semantics known from *Statechart* (Harel, 1987) and *mode automata* formalisms (Maraninchi and Rémond, 2003), particularly the mode automata variant implemented in the Lucid Synchrone 3.0 language (Pouzet, 2006).

The Modelica specification describes the semantics of state machines by a set of rules that allows relating state machines to purely data-flow based Modelica code (Modelica Association, 2012, Chapter 17)[1]. Hence, state machine constructs are reduced to data-flow equation constructs for which the flattening process is already described in other parts of the language specification.

From this perspective it is natural to perform a complete transformation of state machine constructs to data-

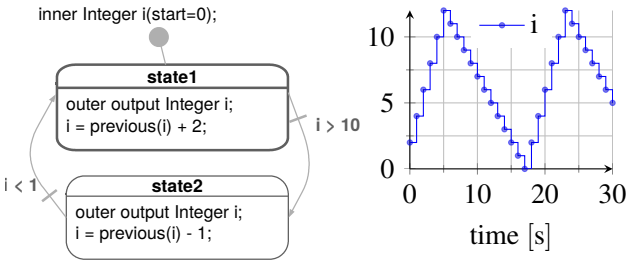
[1]A more accessible presentation of Modelica state machines with additional examples can be found in (Fritzson, 2014, Chapter 13).

flow equations during the flattening process, so that the resulting flat Modelica can be directly related to a flat hybrid DAE. However, a complete transformation of state machine constructs into data-flow equations at the stage of flattening requires an early commitment to implementation details. This commitment makes model optimizations more difficult at subsequent translation phases. Additionally, the required substantial model transformation renders the semantic distance between the source code and the flattened representation rather large which reduces the value of flat Modelica as a traceable human checkable intermediate model representation.

2 State Machine Flattening in Current Tools

At the time of writing, only Dymola[2] provides full support for Modelica state machines. A presentation about an early (incomplete) prototype implementation for OpenModelica[3] was given in the OpenModelica Annual Workshop (Thiele, 2015). The flat Modelica code resulting from State Machines in Dymola resembles the code generated by the above-mentioned OpenModelica prototype.

The simple state machine example presented in the original Modelica state machine paper by Elmqvist et al. (2012) is reused for illustrating the relation between the state machine Modelica code and the generated flat Modelica representation. Figure 2 shows the graphical representation of that state machine as well as a plot of its variable `i` for 30 seconds of simulation. The state machine



(a) Graphical representation. **(b)** Plot of variable `i`.

Figure 2. Simple state machine.

extension is based on Modelica's synchronous language elements extension. The discrete-time equations within a state machine are based on *clocked* variables and all variables and equations within it must be associated with the same clock. The semantics are:

- Equations are active at *clock* ticks generated by the clocks associated with the equations. If no clock is associated a default clock is used. In the example of Figure 2 the default clock used is a periodic clock with 1.0 s sampling period.
- The variable `i` is a *shared* variable between the two states `state1` and `state2`[4].
- The example uses "*delayed*" transitions[5], hence the transitions do not fire immediately if the associated condition on `i` evaluates to **`true`**. Instead they fire at the subsequent clock tick.
- Furthermore, the transitions are declared as "*reset*" transitions[6]. Reset transitions reinitialize the "states" of their target states, *i.e.*, set the values of the state variables "owned" by those states to their start values and reset nested state machines. Within the considered example `state1` and `state2` declare access to the *outer* state variable `i`. Outer variables are *not* reset if entering the state. Hence, for the considered example it makes no difference whether or not a transitions is a "reset" transition.

The Modelica model (ignoring annotations) that corresponds to the graphical representation of Figure 2 is displayed in Listing 1.

Listing 1. Modelica model corresponding to Figure 2.

```
model SimpleSM "Simple state machine"
  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
  transition(state1,state2,i > 10,
    immediate=false,reset=true,
    synchronize=false,priority=1);
  transition(state2,state1,i < 1,
    immediate=false,reset=true,
    synchronize=false,priority=1);
  initialState(state1);
end SimpleSM;
```

The flat Modelica representation generated by Dymola 2015 FD01 is reproduced in a slightly reformatted form (to save space) in Listing 2. Note that there is a discrepancy between the number of variables (three) and the

[2] `http://www.dymola.com/`
[3] `https://www.openmodelica.org/`

[4] The "outer" prefix declares that an element instance with the same name, but using prefix "inner" within the enclosing instance hierarchy is referenced.

[5] Delayed transitions are depicted by a perpendicular line close to the "from"-state. For immediate transitions this line is close to the "to"-state.

[6] Reset transitions are depicted by a filled arrow head (otherwise an open arrow head is used).

DOI 10.3384/ecp15118255

Listing 2. Flat Modelica model generated from the simple state machine model defined in Listing 1.

```
model SimpleSM
Integer i(start = 0);
Integer state1.i = i;
Integer state2.i = i;

// Equations and algorithms

// Component state1
// class SimpleSM.State1
equation
  state1.i = previous(state1.i)+2;

// Component state2
// class SimpleSM.State2
equation
  state2.i = previous(state2.i)-1;

// Component
// class SimpleSM
equation
  transition(state1,state2,i > 10,
    false,true,false,1);
  transition(state2,state1,i < 1,
    false,true,false,1);
  initialState(state1);
end SimpleSM;
```

number of equations (four). This imbalance is solved by the state machine semantics that require that outer output variables of each state are solved for and that for each such variable a single definition is formed. Hence, after substituting the alias variables in the example and merging outer variables this can be reduced to one variable and one equation, *e.g.*,

```
i := if activeState(state1) then
  previous(i)+2
    elseif activeState(state2) then
      previous(i)-1 else previous(i)
```

The last else branch can never be reached in this particular example, but it illustrates that a state variable will simply keep its current value if there is no state active in which an equation for that variable is defined.

Deducing the equation transformation above from the flat Modelica representation is an essential step for relating flat Modelica to a valid DAE representation. Arguably, the information about this necessary equation transformation is present in the flat Modelica in a highly implicit fashion which is not only elusive for human perception, but also difficult to reason about mechanically.

One can deduce that `state1` and `state2` are states and that `state1.i` and `state2.i` are variables declared in the respective states. However, in the flat representation it is not obvious that they are shared variables and that two of their defining equations need to be merged into a *single* definition to form a valid system of equations (otherwise there is one equation too many).

As an example of this ambiguity in the flat representation consider the *invalid* model from Listing 3 that actually has one equation too many, but still has (apart from some comments) the same flattened representation as the simple state machine model from Listing 1 (compare the respective flat Modelica representations in Listing 4 and Listing 2). Trying to simulate the model from Listing 3

Listing 3. Invalid Modelica code that has a similar flat representation as the (valid) code from Listing 1.

```
model InvalidSM "Invalid model, but
    instructive flat representation"
  inner Integer i(start = 0);
  block State1
    input Integer i; // no shared variable!
  end State1;
  State1 state1(i=i);
  block State2
    input Integer i; // no shared variable!
  end State2;
  State2 state2(i=i);
equation
  // one equation too many
  state1.i = previous(i) + 2;
  state2.i = previous(i) - 1;
  transition(state1,state2,i > 10,
    immediate=false,reset=true,
    synchronize=false,priority=1);
  transition(state2,state1, i < 1,
    immediate=false,reset=true,
    synchronize=false,priority=1);
  initialState(state1);
end InvalidSM;
```

in Dymola fails with a (correct) error message complaining about more Integer equations than Integer variables (Dymola still generates the flat Modelica representation for the model since the model can be instantiated, but it cannot be translated due to the overconstrained equation system).

The important point is that solely by inspecting the flat Modelica representation that Dymola generates it is not obvious whether it corresponds to a valid or an invalid model: the flat Modelica representation in Listing 4 is, apart from additional comments, similar to the flat Modelica representation in Listing 2.

This example should illustrate that it is quite intricate to give the correct semantics of flat Modelica state machine representations generated by current Modelica tools. The example used the flat Modelica representation generated by Dymola 2015 FD01, but similar reasoning applies to the flat Modelica generated by the first prototypical support for state machines implemented in OpenModelica. A deliberately simple example was used in order to keep the discussion comprehensible.

To give the correct semantics of state machines encoded in the considered flat Modelica representation, it is necessary to deduce structural information regarding the state machine composition, *e.g.*,

Listing 4. Flat Modelica model generated from the (invalid) model defined in Listing 3.

```
model InvalidSM
Integer i(start = 0);
Integer state1.i = i;
Integer state2.i = i;

// Equations and algorithms

// Component state1
// class InvalidSM.State1
  // extends InvalidSM
  equation
    state1.i = previous(i)+2;
  // end of extends

// Component state2
// class InvalidSM.State2
  // extends InvalidSM
  equation
    state2.i = previous(i)-1;
  // end of extends

// Component
// class InvalidSM
equation
  transition(state1,state2,i > 10,
    false,true,false,1);
  transition(state2,state1,i < 1,
    false,true,false,1);
  initialState(state1);
end InvalidSM;
```

- associate assignment equations for state variables to corresponding states,
- recover the hierarchical state machine structure,
- identify shared variables,
- identify in which equations shared variables are used in an assignment context, and finally,
- deduce which assignment equations need to be merged.

However, experience from the first prototypical implementation in OpenModelica suggest that it is hard to automatically reconstruct this information. from the flattened representation without propagating further structural information about the model from the front-end to the back-end. This crucial additional information is not visible in the flat Modelica representation. The following sections will therefore discuss a symbolic representation for flattened state machine constructs that makes such structural information explicitly available within the flat Modelica model.

3 Practical Symbolic Representation

Different approaches have been experimented with in order to find an adequate symbolic representation. One important requirement is that the representation should be flexible enough for future incorporation of continuous-time equations. Hence, it should be general enough to allow for multi-mode DAE/ODE modeling resembling the style that was advocated by Elmqvist et al. (2014) and Bouissou et al. (2014). In a first approach it was investigated whether symbolic representations developed in the context of *hybrid automata* modeling and verification (Alur et al., 1993) could be adapted and reused in a Modelica context.

The basic idea in this first approach was to generate flat state machine representations and use *interconnection relations* to describe parallel and hierarchical compositions. This idea was motivated by a versatile notion of composition described by Tabuada (2009) in the context of hybrid system modeling. However, the representations became large (depending on the example about twofold the size compared to the representation proposed below), appeared rather artificial in the context of Modelica, and required many decisions during the flattening process that seem to be better postponed to the back-end.

Therefore, a more lightweight approach is proposed. It is based on the following basic ideas:

- Preserve the state machine hierarchy by introducing the notions of **`stateMachine`** and **`state`**.

  stateMachine Consists of a set of mutually exclusive states that are related by transitions (flat state machine).

  state Consists of variable declarations and equations associated to that state. May have nested state machines.

- Generate the equations necessary for merging shared variables from mutual exclusive states.

The resulting representation has the property that the number of equations and variables must be equal for a valid system.

Instead of the terms “stateMachine” and “state” one might prefer to use “automaton” and “mode” which capture that a system operates in a certain *mode* and that the automaton logic allows to change the active mode. However, the proposed terms correspond to the terms that are used in the state machine chapter of the Modelica 3.3 specification.

3.1 Simple State Machine Example

With the proposed extension, the flat representation of the simple state machine example from Listing 1 translates to the flat Modelica displayed in Listing 5. Note that two auxiliary variables `$state1.i`, `$state2.i` have been

DOI 10.3384/ecp15118255

Listing 5. Extended flat Modelica model proposed to be generated for the simple state machine model defined in Listing 1.

```
class SimpleSM
Integer i(start=0);
stateMachine sm0f.state1
  state state1
    Integer $state1.i;
  equation
    $state1.i = previous(i) + 2;
  end state1;

  state state2
    Integer $state2.i;
  equation
    $state2.i = previous(i) - 1;
  end state2;
end sm0f.state1;

equation
  i = if activeState(state1) then $state1.i
    elseif activeState(state2) then $state2.i
    else previous(i);
  transition(state1,state2,i > 10,
    false,true,false,1);
  transition(state2,state1,i < 1,
    false,true,false,1);
  initialState(state1);
end SimpleSM;
```

introduced. They are substitutes for `state1.i`, `state2.i` in the respective states and are used in the generated variable merging equation. However, in case `state1.i` and `state2.i` appear as argument in a **`previous`**`(..)` operator the variable `i` is substituted instead.

The "`$`" prefix shall denote an auxiliary variable that is related to the subsequent variable name, but not strictly identical (*e.g.*, `i` = `state1.i` = `state2.i`, but `i` ≠ `$state1.i` ≠ `$state2.i`).

Compared to the flat Modelica representation of Listing 2, the semantics of the simple state machine example is more explicitly represented in the flat Modelica of Listing 5 which re-enables the possibility to interpret the flat Modelica representation in a meaningful manner. Note that Listing 5 has the desirable property that the number of equations equals the number of unknowns since the variable merging equation (*i.e.*, the equation for `i`) is made explicitly visible.

3.2 Hierarchical State Machine Example

The hierarchical state machine example shown in Figure 3 is motivated by the example given by Maraninchi and Rémond (2003). The state machine receives a stream of input values `i` and `j` and computes the variables `x`, `y`, and `z`. For this example the input values have been set to the constant values `i`=**`true`** and `j`=**`false`**.

Listing 6 shows how information about the structural composition for the hierarchical state machine from Figure 3 is preserved in the proposed flat Modelica representation. The complete listing for the flat Modelica representation is given in Appendix A.

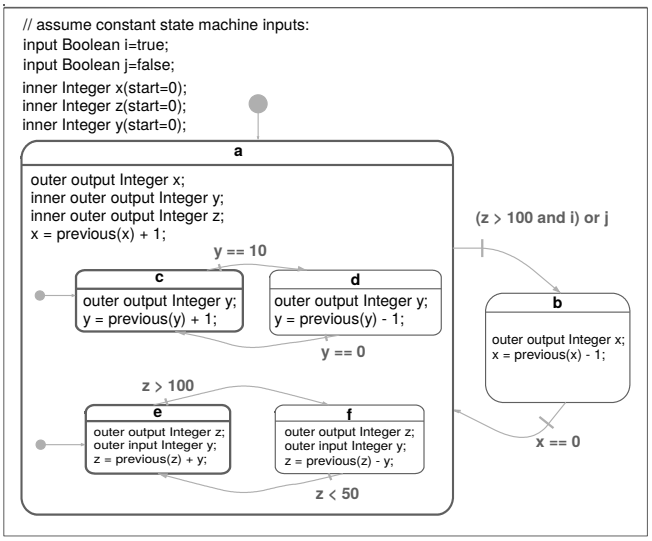


Figure 3. Hierarchical and parallel composition example motivated from Maraninchi and Rémond (2003).

3.3 Summary of Rules

The rules for mapping from state machines specified in Modelica to the proposed flat representation can be summarized as follows:

- Any class instance `x` that appears as argument in an **`initialState`**`(..)` or **`transition`**`(..)` operator results in a section **`state`** `x ...` **`end`** `x;` in the flat Modelica representation.
- Any class instance `x` that appears as argument in an **`initialState`**`(..)` operator results in a section **`stateMachine`** `sm0f.x ...` **`end`** `sm0f.x;` in the flat Modelica representation[7].
- States that are connected by transition relations are collected in a **`stateMachine`** section. The identifier of the **`stateMachine`** section encodes the component reference of the initial state of that state machine. Hence, a **`stateMachine`** section collects states that belong to the same *flat* state machine.
- For any **`outer output`** or **`inner outer output`** variable declaration `x`, an auxiliary variable `$x` is introduced and all occurrences of `x` are replaced by `$x` unless `x` appears as argument in a **`previous`**`(..)` operator in which case `x` is replaced by its corresponding most **`inner`** component reference[8].

[7] The name following the "stateMachine" construct has no significant semantics and could be also omitted, *e.g.*, "stateMachine ... end;".

[8] Hence, it is replaced by the most inner component reference and not by a references to an intermediate "inner outer output" declaration.

Listing 6. Preservation of state machine composition information for the hierarchical state machine from Figure 3.

```
class HierarchicalSM
  stateMachine sm0f.a
    state a
      stateMachine sm0f.a.c
        state a.c
          ...
        end a.c;
        state a.d
          ...
        end a.d;
      end sm0f.a.c;
      stateMachine sm0f.a.e
        state a.e
          ...
        end a.e;
        state a.f
          ...
        end a.f;
      end sm0f.a.e;
      ...
    end a;
    state b
      ...
    end b;
  end sm0f.a;
  ...
end HierarchicalSM;
```

- **outer** variables that are not declared as **output** are replaced by their corresponding **inner** component references.

- Equations for merging shared variables are introduced according to the following rules:

  - For any **inner** (or **inner outer**) variable that is referenced by an **outer** (or **inner outer**) variable declaration in one or more states (within the same hierarchy) of a **stateMachine** section, a merging equation is formed at the instance level in which the **stateMachine** is defined.

  - The merging equation assigns the inner variable the value of the corresponding auxiliary variable of the currently active state in the following form:

    ```
    x = if activeState(a) then $a.x
      elseif activeState(b) then $b.x
      else previous(x);
    ```

Further on, it is possible to improve the comprehensibility of the state machine representation by collecting all transitions and merging equations associated to a **stateMachine** section within that associated section (note that this is a pretty-print consideration and not a requirement for giving an unambiguous semantics).

4 Implementation

The proposed flattening for state machines has been implemented in the OpenModelica compiler. The process is depicted in Figure 4. State machines are first flattened in the compiler front-end according to the approach that was outlined in Section 3. After that, the state machine structures are further elaborated in the back-end where they are translated to basic data-flow equations (by transforming the abstract syntax tree (AST)). The translation to data-flow equations is inspired by the state machine compilation approach described by Colaço et al. (2005) and is a fairly direct encoding of the equations provided in the *Semantics Summary* of (Modelica Association, 2012, Section 17.3.4).

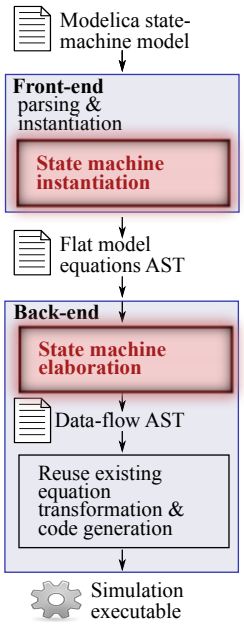


Figure 4. Outline of the state machine compilation process.

The current back-end implementation does not lead to very efficient code, *e.g.*, translation of the simple state machine example from Listing 1 to data-flow equations leads to 24 (mostly **Boolean**) data-flow variables (and equations) in the back-end (see Appendix B). However, this can be optimized to produce fewer variables and equations in future versions of the compiler without having to change the underlying symbolic representation produced by the front-end. The advantage of the current back-end implementation is the possibility to reuse most of the existing equation transformation and code generation facilities without further modification.

The current prototype needs a workaround to compensate for the not yet implemented support for Modelica's clocked synchronous language extension (Modelica Association, 2012, Chapter 16). The "hack" in the back-end is to wrap all state machine related equations in a **when**-equation with a sampling period of one second and replace

DOI 10.3384/ecp15118255

all **previous**(..) operators by **pre**(..) operators, similarly to following code snippet:

```
when sample(0.0, 1.0) then
  i = if sm0f.state1.activeState == 2
        then -1 + pre(i)
          else 2 + pre(i);
end when;
```

This restriction can be lifted easily as soon as the clocked synchronous language elements are fully supported by our compiler. Meanwhile the workaround allows to experiment with state machine implementations in parallel and independently to ongoing work related to synchronous languages elements support.

5 Conclusion

This paper proposed a dedicated symbolic representation for flattened Modelica state machines. The representation explicitly preserves crucial structural and relational information in the human readable flat Modelica representation. Hence, the proposed representation avoids ambiguities and can be interpreted straightforwardly by human inspection. This is in contrast to the ambiguous and hard to interpret flat representations of state machine models which are generated by existing tools. Furthermore, this representation is well suited for further computational processing in the back-end, because it becomes unnecessary to elaborately re-construct important structural information solely from the basic data-flow equations that are typically available at that later compiler phase.

At the same time the proposed representation strives to avoid an early commitment to implementation details for the specified state machine logic, *i.e*, it refrains from performing a full translation of the state machine constructs to basic clocked synchronous data-flow equations in the flattened representation. In that way it allows one to postpone implementation decisions, that would potentially hinder code optimization techniques, to later translation stages.

The approach has been implemented in the OpenModelica compiler and successfully tested on a number of state machine models.

Acknowledgements

This work has been supported by Vinnova in the ITEA2 MODRIO project, by EU in the INTO-CPS project, and by the Swedish Government in the ELLIIT project. The Open Source Modelica Consortium supports the OpenModelica project.

References

R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid systems*, 736:209–229, 1993.

Marc Bouissou, Hilding Elmqvist, Martin Otter, and Albert Benveniste. Efficient Monte Carlo simulation of stochastic hybrid systems. In Hubertus Tummescheit and Karl-Erik Årzén, editors, 10th *Int. Modelica Conference*, Lund, Sweden, March 2014. doi:10.3384/ecp14096715.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 173–182, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. doi:10.1145/1086228.1086261.

Hilding Elmqvist, Fabien Gaucher, Sven Erik Mattsson, and Francois Dupont. State Machines in Modelica. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp1207637.

Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica extensions for Multi-Mode DAE Systems. In Hubertus Tummescheit and Karl-Erik Årzén, editors, 10th *Int. Modelica Conference*, Lund, Sweden, May 2014. doi:10.3384/ECP14096183.

Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley IEEE Press, 2014. ISBN 9781-118-859124.

David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. doi:10.1016/0167-6423(87)90035-9.

Florence Maraninchi and Yann Rémond. Mode-Automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46:219–254, 2003.

Modelica Association. Modelica—A Unified Object-Oriented Language for Systems Modeling v3.3. Standard Specification, May 2012. Available at http://www.modelica.org/.

Marc Pouzet. *Lucid Synchrone Tutorial and Reference Manual*, 2006.

Paulo Tabuada. *Verification and Control of Hybrid Systems A Symbolic Approach*. Springer US, 2009. doi:10.1007/978-1-4419-0224-5.

Bernhard Thiele. State Machines in OpenModelica - Current Status and Further Development. In *OpenModelica Annual Workshop*, Linköping, Sweden, 2. February 2015. Open Source Modelica Consortium (OSMC) and Linköping University (LiU). URL `http://www.modprod.liu.se/openmodelica-2015?l=en`.

A Flat Modelica for the Hierarchical State Machine Example

The complete listing of the flat Modelica representation of the hierarchical state machine example from Section 3.2.

```
class HierarchicalSM
  Integer x(start = 0);
  Integer z(start = 0);
  Integer y(start = 0);
  input Boolean i = true;
  input Boolean j = false;
  stateMachine sm0f.a
    state a
      Integer $a.y;
      Integer $a.z;
      Integer $a.x;
      stateMachine sm0f.a.c
        state a.c
          Integer $a.c.y;
        equation
          $a.c.y = 1 + previous(y);
        end a.c;

        state a.d
          Integer $a.d.y;
        equation
          $a.d.y = -1 + previous(y);
        end d;
      end sm0f.a.c;

      stateMachine sm0f.a.e
        state a.e
          Integer $a.e.z;
        equation
          $a.e.z = previous(z) + y;
        end a.e;

        state a.f
          Integer $a.f.z;
        equation
          $a.f.z = previous(z) - y;
        end a.f;
      end sm0f.a.e;

    equation
      $a.z = if activeState(a.e) then $a.e.z
        elseif activeState(a.f) then $a.f.z
        else previous(z);
      $a.y = if activeState(a.c) then $a.c.y
        elseif activeState(a.d) then $a.d.y
        else previous(y);
      initialState(a.e);
      transition(a.e, a.f, $a.z > 100,
        false, true, false, 1);
      transition(a.f, a.e, $a.z < 50,
        false, true, false, 1);
      transition(a.c, a.d, $a.y == 10,
        false, true, false, 1);
      transition(a.d, a.c, $a.y == 0,
        false, true, false, 1);
      $a.x = 1 + previous(x);
      initialState(a.c);
    end a;

    state b
      Integer $b.x;
    equation
      $b.x = -1 + previous(x);
    end b;
  end sm0f.a;
equation
  z = if activeState(a) then $a.z
    else previous(z);
  x = if activeState(a) then $a.x
    elseif activeState(b) then $b.x
    else previous(x);
  transition(a, b, z > 100,
    false, true, false, 1);
  transition(b, a, x == 0,
    false, true, false, 1);
  initialState(a);
end HierarchicalSM;
```

B Back-End Equations

The state machine elaboration in the back-end translates the state machine representation from the front-end to basic data-flow equations (see Figure 4). The intermediate system of equations can be retrieved from the back-end by using debugging functions. The listing below shows the equation system which is generated for the simple state machine example from Listing 1. For better readability the debug output has been reformatted to resemble the typical flat Modelica style. Obviously, the behaviour of the simple state machine can already be described by a fraction of the actually generated equations. However, such optimizations are not performed in the current prototype implementation.

```
model SimpleSM
// parameters
Integer sm0f.state1.tPriority[2]  = 1
Boolean sm0f.state1.tSynchronize[2]  = false
Boolean sm0f.state1.tReset[2]  = true
Boolean sm0f.state1.tImmediate[2]  = false
Integer sm0f.state1.tTo[2]  = 1
Integer sm0f.state1.tFrom[2]  = 2
Integer sm0f.state1.tPriority[1]  = 1
Boolean sm0f.state1.tSynchronize[1]  = false
Boolean sm0f.state1.tReset[1]  = true
Boolean sm0f.state1.tImmediate[1]  = false
Integer sm0f.state1.tTo[1]  = 2
Integer sm0f.state1.tFrom[1]  = 1
Integer sm0f.state1.nState  = 2
// variables
Integer i(start=0);
Boolean state2._active;
Boolean state1._active;
Boolean sm0f._state1._init(start=true);
Boolean sm0f._state1._stateMachineInFinalState;
Boolean sm0f._state1._finalStates[2];
Boolean sm0f._state1._finalStates[1];
Boolean sm0f._state1._nextResetStates[2];
Boolean sm0f._state1._nextResetStates[1];
Boolean sm0f._state1._activeResetStates[2];
Boolean sm0f._state1._activeResetStates[1];
Boolean sm0f._state1._nextReset;
Integer sm0f._state1._nextState;
Boolean sm0f._state1._activeReset;
Integer sm0f._state1._activeState;
Integer sm0f._state1._fired;
Boolean sm0f._state1._selectedReset;
Integer sm0f._state1._selectedState;
Boolean sm0f._state1._reset;
Boolean sm0f._state1._active;
Boolean sm0f._state1._cImmediate[2];
Boolean sm0f._state1._c[2];
Boolean sm0f._state1._cImmediate[1];
```


DOI 10.3384/ecp15118255

```
Boolean sm0f._state1._c[1];
equation
when {sample(1.0, 1.0), initial()} then
  state2.active = sm0f.state1.active
    and sm0f.state1.activeState == 2;
  state1.active = sm0f.state1.active
    and sm0f.state1.activeState == 1;
  sm0f.state1.active = true;
  sm0f.state1.reset = pre(sm0f.state1.init);
  sm0f.state1.init = false;
  sm0f.state1.stateMachineInFinalState =
    sm0f.state1.finalStates[
      sm0f.state1.activeState];
  sm0f.state1.finalStates[2] = max(
    if sm0f.state1.tFrom[2] == 2 then 1 else 0,
    if sm0f.state1.tFrom[1] == 2 then 1 else 0
    ) == 0;
  sm0f.state1.finalStates[1] = max(
    if sm0f.state1.tFrom[2] == 1 then 1 else 0,
    if sm0f.state1.tFrom[1] == 1 then 1 else 0
    ) == 0;
  sm0f.state1.nextResetStates[2] =
    if sm0f.state1.active then
      if sm0f.state1.selectedState == 2 then false
        else sm0f.state1.activeResetStates[2]
    else pre(sm0f.state1.nextResetStates[2]);
  sm0f.state1.nextResetStates[1] =
    if sm0f.state1.active then
      if sm0f.state1.selectedState == 1 then false
        else sm0f.state1.activeResetStates[1]
    else pre(sm0f.state1.nextResetStates[1]);
  sm0f.state1.activeResetStates[2] =
    if sm0f.state1.reset then true
    else pre(sm0f.state1.nextResetStates[2]);
  sm0f.state1.activeResetStates[1] =
    if sm0f.state1.reset then true
    else pre(sm0f.state1.nextResetStates[1]);
  sm0f.state1.nextReset = if sm0f.state1.active
    then false
    else pre(sm0f.state1.nextReset);
  sm0f.state1.nextState =
    if sm0f.state1.active
      then sm0f.state1.activeState
    else pre(sm0f.state1.nextState);
  sm0f.state1.activeReset =
    if sm0f.state1.reset then true
    else
      if sm0f.state1.fired > 0 then
        sm0f.state1.tReset[sm0f.state1.fired]
      else sm0f.state1.selectedReset;
  sm0f.state1.activeState =
    if sm0f.state1.reset then 1
    else
      if sm0f.state1.fired > 0 then
        sm0f.state1.tTo[sm0f.state1.fired]
      else sm0f.state1.selectedState;
  sm0f.state1.fired = max(
    if
      if sm0f.state1.tFrom[2] ==
        sm0f.state1.selectedState then
          sm0f.state1.c[2]
      else false
        then 2 else 0,
    if
      if sm0f.state1.tFrom[1] ==
        sm0f.state1.selectedState then
          sm0f.state1.c[1]
      else false
        then 1 else 0);
  sm0f.state1.selectedReset =
    if sm0f.state1.reset then true
    else pre(sm0f.state1.nextReset);
  sm0f.state1.selectedState =
    if sm0f.state1.reset then 1
    else pre(sm0f.state1.nextState);
  sm0f.state1.c[2] =
    pre(sm0f.state1.cImmediate[2]);
  sm0f.state1.cImmediate[2] = i < 1;
  sm0f.state1.c[1] =
    pre(sm0f.state1.cImmediate[1]);
  sm0f.state1.cImmediate[1] = i > 10;
  i = if sm0f.state1.activeState == 2
      and sm0f.state1.active then -1 + pre(i)
    else if sm0f.state1.activeState == 1
      and sm0f.state1.active then 2 + pre(i)
        else pre(i);
end when;
end SimpleSM;
```