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.

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

- Equations are active at \textit{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 \textit{shared} variable between the two states state1 and state2\(^4\).
- The example uses “\textit{delayed}” transitions\(^5\), hence the transitions do not fire immediately if the associated condition on \(i\) evaluates to \textit{true}. Instead they fire at the subsequent clock tick.
- Furthermore, the transitions are declared as “\textit{reset}” transitions\(^6\). Reset transitions reinitialize the “\textit{states}” of their target states, \textit{i.e.}, set the values of the state variables “\textit{owned}” by those states to their start values and reset nested state machines. Within the considered example state1 and state2 declare access to the \textit{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 “\textit{reset}” transition.

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

\begin{verbatim}
model SimpleSM "Simple state machine"
  inner Integer i(start=0);
  block State1
    outer output Integer i;
    equation
      i = previous(i) + 2;
  end State1;
  block State2
    outer output Integer i;
    equation
      i = previous(i) - 1;
  end 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);
  endState1;
  initialState(state1);
end SimpleSM;
\end{verbatim}

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/
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 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;
```

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

```modelica
class SimpleSM
  Integer i(start=0);
  stateMachine smOf.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 smOf.state1;

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

The name following the “stateMachine” construct has no significant semantics and could be also omitted, e.g., “stateMachine ... end.”. Hence, it is replaced by the most inner component reference and not by a references to an intermediate “inner component” declaration.

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 ...` in the flat Modelica representation.
- Any class instance \( x \) that appears as argument in an `initialState(..)` operator results in a section `stateMachine smOf.x ... end smOf.x;` in the flat Modelica representation.
- 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.

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 ...` in the flat Modelica representation.
- Any class instance \( x \) that appears as argument in an `initialState(..)` operator results in a section `stateMachine smOf.x ... end smOf.x;` in the flat Modelica representation.
- 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.

Figure 3. Hierarchical and parallel composition example motivated from Maraninchi and Rémond (2003).
Listing 6. Preservation of state machine composition information for the hierarchical state machine from Figure 3.

class HierarchicalSM
    stateMachine smOf.a
        state a
            stateMachine smOf.a.c
                state a.c
                ...
            end a.c;
            state a.d
                ...
            end a.d;
        end smOf.a.c;
        stateMachine smOf.a.e
        ...
    end a;
    state b
    ...
end smOf.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:

    \[ x = \begin{cases} \text{activeState}(a) & \text{then } a.x \\ \text{activeState}(b) & \text{then } b.x \\ \text{previous}(x) & \end{cases} \]

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).

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

```plaintext
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


**References**

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
   // variables
   Integer smOf.a.x(start=0);
   Integer smOf.a.y;
   Integer smOf.a.z;
   stateMachine smOf.a;

   state a
      Integer $a.y;
      Integer $a.z;
      Integer $a.x;
      stateMachine smOf.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 smOf.a.c;

   stateMachine smOf.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 smOf.a.e;

      equation
         $a.z = if activeState(a.e) then $a.e.z else previous(z);
         $a.y = if activeState(a.c) then $a.c.y else previous(y);
      end smOf.a.e;

   initialState(a.e);
   transition(a.e, a.f, $a.z > 100, false, true, false, 1);
   transition(a.f, a.e, $a.z < 50, false, false, true, 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);
   initialState(a.c);
end a;

state b
   Integer $b.x;
   equation
      $b.x = -1 + previous(x);
   end b;
input Boolean i = true;
end smOf.a;

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 smOf.state1.tPriority[2] = 1
   Boolean smOf.state1.tSynchronize[2] = false
   Boolean smOf.state1.tReset[2] = true
   Boolean smOf.state1.tImmediate[2] = false
   Integer smOf.state1.tTo[2] = 1
   Integer smOf.state1.tFrom[2] = 2
   Integer smOf.state1.tPriority[1] = 1
   Boolean smOf.state1.tSynchronize[1] = false
   Boolean smOf.state1.tReset[1] = true
   Boolean smOf.state1.tImmediate[1] = false
   Integer smOf.state1.tTo[1] = 2
   Integer smOf.state1.tFrom[1] = 1
   Integer smOf.state1.nState = 2
   // variables
   Integer i(start=0);
   Boolean state2._active;
   Boolean state1._active;
   Boolean state1._init(start=true);
   Boolean state1._stateMachineInFinalState;
   Boolean state1._finalStates[1];
   Boolean state1._activeResetStates[1];
   Boolean state1._nextResetStates[1];
   Boolean state1._finalStates[2];
   Boolean state1._activeResetStates[2];
   Boolean state1._nextResetStates[2];
   Boolean state1._fire;
   Boolean state1._selectedResetState;
   Boolean state1._reset;
   Boolean state1._active;
   Boolean state1._clImmediate[2];
   Boolean state1._c[2];
Boolean smOf.state1_c[1];
equation
when { sample(1.0, 1.0), initial() } then
  state2.active = smOf.state1.active
  and smOf.state1.activeState == 2;
  state1.active = smOf.state1.active
  and smOf.state1.activeState == 1;
  smOf.state1.active = true;
  smOf.state1.reset = pre(smOf.state1.init);
  smOf.state1.init = false;
  smOf.state1.stateMachineInFinalState = smOf.state1.finalStates[
    smOf.state1.activeState];
  smOf.state1.finalStates[2] = max(
    if smOf.state1.tFrom[2] == 2 then 1 else 0,
    if smOf.state1.tFrom[1] == 2 then 1 else 0
  ) == 0;
  smOf.state1.finalStates[1] = max(
    if smOf.state1.tFrom[2] == 1 then 1 else 0,
    if smOf.state1.tFrom[1] == 1 then 1 else 0
  ) == 0;
  smOf.state1.nextResetStates[2] = if smOf.state1.active then
    if smOf.state1.selectedState == 2 then false
    else smOf.state1.activeResetStates[2]
  else pre(smOf.state1.nextResetStates[2]);
  smOf.state1.nextResetStates[1] = if smOf.state1.active then
    if smOf.state1.selectedState == 1 then false
    else smOf.state1.activeResetStates[1]
  else pre(smOf.state1.nextResetStates[1]);
  smOf.state1.activeResetStates[2] = if smOf.state1.reset then true
  else pre(smOf.state1.nextResetStates[2]);
  smOf.state1.activeResetStates[1] = if smOf.state1.reset then true
  else pre(smOf.state1.nextResetStates[1]);
  smOf.state1.nextReset = if smOf.state1.active then
    if smOf.state1.fired > 0 then
      smOf.state1.tReset[smOf.state1.fired]
    else smOf.state1.selectedReset;
  else pre(smOf.state1.nextReset);
  smOf.state1.nextState = if smOf.state1.active then
    smOf.state1.activeState
  else pre(smOf.state1.nextState);
  smOf.state1.activeReset = if smOf.state1.reset then true
  else if smOf.state1.fired > 0 then
    smOf.state1.tTo[smOf.state1.fired]
  else smOf.state1.selectedState;
  smOf.state1.fired = max(          
    if smOf.state1.tFrom[2] == 1 then 1
    else pre(smOf.state1.nextState);
    if smOf.state1.reset then 1
  else pre(smOf.state1.nextState);
  if smOf.state1.reset then 2 + pre(i)
  else pre(i);
end when;
end SimpleSM;