

State Machines in OpenModelica

Current Status and Further Development

Bernhard Thiele

PELAB

Linköping University

02. February 2015 - Open Modelica Annual Workshop

Goals of this presentation

- Introduce Modelica state machines.
- Describe the implementation approach.
- Pros and cons of the current approach and further development plans.

Motivation

- Control application often consists of:
 - 1 Data-flow parts \mapsto block diagrams
 - 2 System logic \mapsto state machines
- Previous state machine attempts were library based (e.g., *StateGraph* and *StateGraph2* library)
- However, Library based attempts not considered to be enough powerful and convenient/safe to use
- Now, Statechart like support is available as built-in language feature



Hilding Elmqvist, Fabien Gaucher, Sven Erik Mattsson, Francois Dupont State Machines in Modelica . In 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

Motivation

- Control application often consists of:
 - 1 Data-flow parts \mapsto block diagrams
 - 2 **System logic \mapsto state machines**
- Previous state machine attempts were library based (e.g., *StateGraph* and *StateGraph2* library)
- **However**, Library based attempts not considered to be enough powerful and convenient/safe to use
- Now, Statechart like support is available as **built-in language feature**



Hilding Elmqvist, Fabien Gaucher, Sven Erik Mattsson, Francois Dupont State Machines in Modelica . In 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

Motivation

- Control application often consists of:
 - 1 Data-flow parts \mapsto block diagrams
 - 2 **System logic \mapsto state machines**
- Previous state machine attempts were library based (e.g., *StateGraph* and *StateGraph2* library)
- **However**, Library based attempts not considered to be enough powerful and convenient/safe to use
- Now, Statechart like support is available as **built-in language feature**



Hilding Elmqvist, Fabien Gaucher, Sven Erik Mattsson, Francois Dupont State Machines in Modelica . In 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

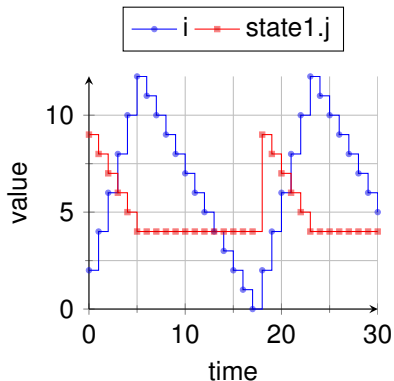
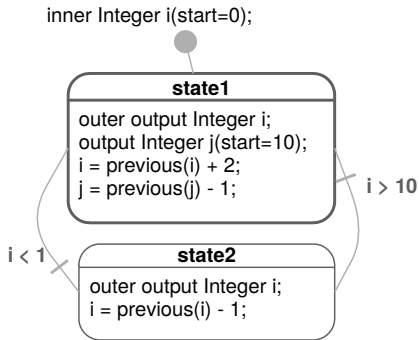
Motivation

- Control application often consists of:
 - 1 Data-flow parts \mapsto block diagrams
 - 2 **System logic \mapsto state machines**
- Previous state machine attempts were library based (e.g., *StateGraph* and *StateGraph2* library)
- **However**, Library based attempts not considered to be enough powerful and convenient/safe to use
- Now, Statechart like support is available as **built-in language feature**



Hilding Elmqvist, Fabien Gaucher, Sven Erik Mattsson, Francois Dupont State Machines in Modelica . In 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

Simple Example



- Equations are active if corresponding *clock* ticks. Defaults to a periodic clock with 1.0s sampling period.
- “i” is a shared variable, “j” is a local variable. Transitions are “*delayed*” and enter states by “*reset*”.

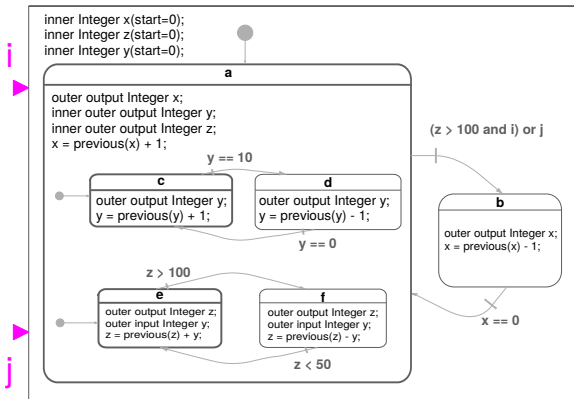
Simple Example: Modelica Code

```

model Simple_NoAnnotations "Simple state machine"
  inner Integer i(start=0);
  block State1
    outer output Integer i;
    output Integer j(start=10);
  equation
    i = previous(i) + 2;
    j = previous(j) - 1;
  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 Simple_NoAnnotations;

```


M&R-Example: Hierarchical and Parallel Composition



Semantics of Modelica state machines (and example above) inspired by



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

and by Marc Pouzet's language **Lucid Sychrone 3.0**.

Summary of intuitive semantics

Modelica state machines:

- Extend on the **synchronous language extension**.
- Support *hierarchic* and *parallel composition* of states, *immediate* (strong) and *delayed* (weak) transitions, entering a state with *reset* or *resume* of internal state memory (enter by history).
- *States* are instances of ordinary blocks with data-flow equations.
- Block *instances* become states if they appear as argument in `transition(..)` or `initialState(..)` operators.

Summary of intuitive semantics

Modelica state machines:

- Extend on the [synchronous language extension](#).
- Support *hierarchic* and *parallel composition* of states, *immediate* (strong) and *delayed* (weak) transitions, entering a state with *reset* or *resume* of internal state memory (enter by history).
- *States* are instances of ordinary blocks with data-flow equations.
- Block *instances* become states if they appear as argument in `transition(..)` or `initialState(..)` operators.

Summary of intuitive semantics

Modelica state machines:

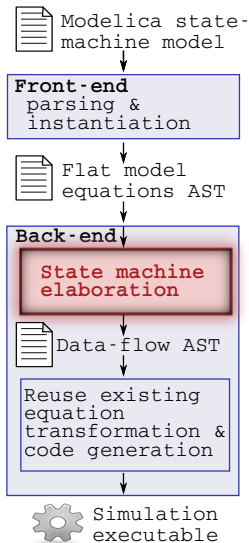
- Extend on the [synchronous language extension](#).
- Support *hierarchic* and *parallel composition* of states, *immediate* (strong) and *delayed* (weak) transitions, entering a state with *reset* or *resume* of internal state memory (enter by history).
- *States* are instances of ordinary blocks with data-flow equations.
- Block *instances* become states if they appear as argument in `transition(..)` or `initialState(..)` operators.

Summary of intuitive semantics

Modelica state machines:

- Extend on the [synchronous language extension](#).
- Support *hierarchic* and *parallel composition* of states, *immediate* (strong) and *delayed* (weak) transitions, entering a state with *reset* or *resume* of internal state memory (enter by history).
- *States* are instances of ordinary blocks with data-flow equations.
- Block *instances* become states if they appear as argument in `transition(..)` or `initialState(..)` operators.

Approach used for the OpenModelica Prototype



State machine elaboration

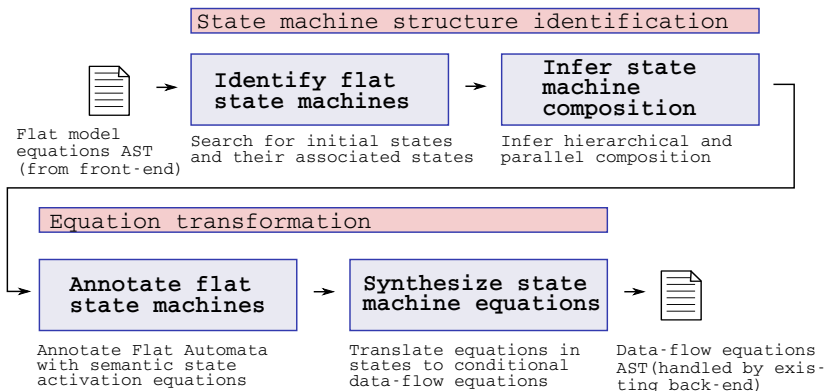
State machine control structures are translated to basic data-flow equations (*AST transformation*).

Inspired by (but simultaneously quite different to)



Jean-Louis Colaço, Bruno Pagano & Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, 2005.

State-Machine Elaboration



M&R-Example: Information Available in Flat Model AST

```

class MRExample "Flattened example from slide 6"
  input Boolean i = true, j = false; // assume constant SM inputs
  inner Integer x(start = 0), z(start = 0), y(start = 0);
  inner outer output Integer a.y = y, a.z = z;
  outer output Integer a.x = x, a.c.y = a.y, a.d.y = a.y;
  outer output Integer a.e.z = a.z, a.f.z = a.z, b.x = x;
equation
  initialState(a);
  initialState(a.e);
  initialState(a.c);
  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);
  transition(a, b, z > 100 and i or j, false, true, false, 1);
  transition(b, a, x == 0, false, false, false, 1);
  a.c.y = 1 + previous(a.c.y);
  a.d.y = -1 + previous(a.d.y);
  a.e.z = previous(a.e.z) + a.y;
  a.f.z = previous(a.f.z) - a.y;
  a.x = 1 + previous(a.x);
  b.x = -1 + previous(b.x);
end MRExample;

```


M&R-Example: State machine structure identification

- 1 Identify flat state machines by computing transitive closure over transition relations:

Flat SM	States
sm0f.a	a, b
sm0f.a.c	a.c, a.d
sm0f.a.e	a.e, a.f

- 2 Infer state machine composition (*state refinements*) from the list of flat state machines:

$$R_{\text{sm0f.a}}(a \mapsto \{R_{\text{sm0f.a.c}}(a.c, a.d) \parallel R_{\text{sm0f.a.e}}(a.e, a.f)\}, b)$$

M&R-Example: Annotate flat state machines

Abridged and simplified activation equations for smOf.a:

```

constant Integer smOf.a.tTo[2] = {2,1}; // transition "to"
constant Integer smOf.a.tFrom[2] = {1,2}; // transition "from"
Boolean smOf.a.init(start=true) = false; // false except start value
// Ensure SM reset at first clock tick
Boolean smOf.a.reset := previous(smOf.a.init);
// delayed transitions
Boolean smOf.a.c[2] :=
  {previous(z > 100 and i or j), previous(x == 0)};
// State update starts from previous active state
Integer smOf.a.selectedState :=
  if smOf.a.reset then 1 else previous(activeState);
// If several can fire, the highest priority is chosen:
Integer smOf.a.fired := max{
  if (smOf.a.tFrom[2] == smOf.a.selectedState
    then smOf.a.c[2] else false) then 2 else 0,
  if (if smOf.a.tFrom[1] == smOf.a.selectedState
    then smOf.a.c[1] else false) then 1 else 0)}
// A reset forces the activeState to be the initial state
Integer smOf.a.activeState = if smOf.a.reset then 1
  else (if smOf.a.fired > 0 then smOf.a.tTo[smOf.a.fired]
    else smOf.a.selectedState);

```

M&R-Example: Synthesize state machine equations

- 1 Merge activation equation annotations of flat state machines and add them to the flat equation AST.
- 2 Translate equations in states to conditional data-flow equations, e.g.,:

```
x := if smOf.a.activeState == 1
     then previous(x) + 1 else previous(x) - 1;
```

Note: [Current prototype](#) uses a workaround due to clocked synchronous features not being implemented yet and wraps all SM related equations in:

```
when sample(0, 1.0) then
  x := if smOf.a.activeState == 1
       then pre(x) + 1 else pre(x) - 1;
end when;
```

Current implementation pros and cons

Pros:

- Implementation can be added to OMC in a modular manner (as pre-optimization module in the back-end).
- Remaining modules for equation sorting, optimization, and code generation can be used without modification.

Cons:

- State machine structure identification from the flat model AST requires costly elaboration.
- Activation equations lead to many new equations and variables (costly in terms of performance and memory efficiency).
- Extensive symbolic transformation complicates traceability and leads to error messages that are not helpful.

Outlook

- First state machine prototype (partial implementation of complete semantics, about 3000 LoC) will be merged into the main development branch in the coming weeks.
- In parallel, ongoing implementation of clocked synchronous constructs in OpenModelica ongoing at FH Bielefeld. Currently, equation partitioning achieved.
- Future challenges: Complete implementation, *efficiency* of translation process and generated code, *adequacy* of translation approach, *traceability*, *error messages*, *debugging*, *hybrid state machines* . . .

Specialities of the state machine semantics

Modelica state machine:

- All variables in the state machine are on the same clock — this is in contrast to to the Mode-Automata design paradigms where modes (\equiv states) should behave like clocks.
- Consequently, variables of inactive states are accessible/readable whenever the clock of the state machine ticks.
- “Shared” variables are realized by instance hierarchy name lookup of “**inner**” declarations with merging of variable definitions that correspond to “**outer output**” declarations of (mutual exclusive) states. They are kept constant if no defining state is active.
- Non-normative specification text suggests the use of “**inner outer output**” for intermediate instance levels of “shared” variables.
- “Immediate” and “delayed” transitions are significant different to Lucid Synchrone: All transitions are “immediate”, “delayed” transitions are “immediate” transitions wrapped in a **previous(..)**.
- A “reset” will enforce the initial state to be active even if a transition from the initial state could fire immediately.