Statecharts as a Means to Control Plant Models in LMS Imagine.Lab AMESim

Vincent Berthoux Sébastien Furic Loïc Wagner
{vincent.berthoux,sebastien.furic,loic.wagner}@lmsintl.com
LMS Imagine S.A.
7 place des Minimes
42300 Roanne, France

Abstract

This article introduces a new feature of LMS Imagine.Lab AMESim that allows users to define plant model controllers. We start by reviewing some challenging aspects of hybrid state machine handling in asynchronous Modelica-based physical simulation environments. We then describe the implementation available in AMESim, focusing on user interaction and especially static error checking and reporting.

Keywords: Statechart; Modelica; LMS Imagine.Lab AMESim

1 Background

Models of physical systems can be built out of equation-based entities (submodels) whose interaction through a connection structure yield the behavior under consideration. This way of defining physical models puts emphasis on *technological* and/or *phenomenological* aspects of modeling: one typically defines entities representing fundamental phenomena (e.g., energy storage), or entities representing technological assemblies (e.g., a cooling system), or any combination of both, as modeling 'bricks'. However, this is not the only nor always the most appropriate way of defining models. For instance, one sometimes prefers to put emphasis on *states* and *transitions* between states. This is typically the case when building controllers used to drive models. These controllers feature operating modes that can be conveniently represented as states of a certain finite state machine (consider for instance a controller having modes *start*, *run* and *stop* with transitions between these states indicating possible mode transitions).

To make the picture complete however, real-world controllers actually also feature *state variables*, leading to infinite (often uncountable) hybrid state machines. Nevertheless, the state-and-transition view is still the preferred one in most situations: this observation motivated the introduction of a new user interface feature in AMESim, allowing users to define controllers by means of a finite set of states and transitions, yet offering state variables and equations as a means to specify not only actions to be performed during state transition but also constraints to be verified in a given state.

Figure 1 shows a simple counter model expressed in the new state-and-transition view, which has been highly inspired by Harel's statechart language [1]. The corresponding user interface has been built on top of AMESim's Modelica translation chain to benefit from its automatic code generation feature.

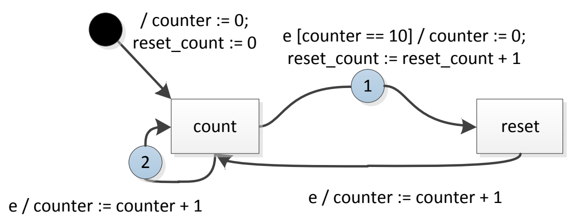

/ counter := 0;
reset_count := 0
e [counter == 10] / counter := 0;
reset_count := reset_count + 1
1
count
reset
2
e / counter := counter + 1
e / counter := counter + 1

Figure 1: A simple counter statechart

2 Why statecharts?

Variants of Harel's statecharts are today by all means one of the most popular approaches used to describe state machines in Control tools. On the other hand, LMS Imagine.Lab AMESim, which is a Physical System Modeling tool, used to favor the technological and phenomenological aspects of models—which are often the most natural ones in its application area. However, models do not necessarily classify as "pure control" or "pure physical": some of them involve a *mix* of physical and control aspects (e.g., aircraft models including aircraft missions, vehicle models including driving manoeuvers, etc.). If we want to handle such models in a simulation tool, we basically have two options: either we choose a unified representation, or we offer the ability to work with both the state-and-transition view and the technological-and-phenomenological view—and then possibly translate heterogeneous parts to a common representation under the hood. In the following we explain the reasons that have driven the choice made in AMESim, whose last release implements the second option.

2.1 Control in AMESim by means of native components

Early attempts to mix control with physics in AMESim naturally made use of the versatile native *component* concept. An AMESim component can be seen as a generic "basic brick" having one or several implementations called *submodels*, each of them specifying the *causality* attached to each of its port signals. In this paradigm, there is no fundamental difference between control signals and physical signals (i.e., signals held by *power variables* in corresponding bond graph models): it follows that native submodels can be used to implement control. However, while monolithic controllers (i.e., implemented as single submodels) can be made reasonably safe,[1] controllers built out of *smaller bricks* suffer from two weaknesses inherited from the submodel composition operation:

- the resulting flow of events is not synchronized and
- some control flow defects (resulting, for instance, in blocking models) cannot be detected at compile time.

As a consequence of the former, *cascades of events* are typical of models that deal with discontinuous—not necessarily piece-wise constant—signals. For instance, if a submodel's job consists in converting its real piece-wise constant input to an integer, it will trigger a fresh event each time its output changes,[2] even if in this case—the input is piece-wise constant—we know that instants corresponding to output changes form a *subset* of those corresponding to input changes.[3] This has unfortunate consequences over resulting models: for instance, it is not possible to know, when a bunch of events fire, whether these events trace back to the same cause or not. Consider our real-to-integer converter example: since each jump in the output signal slope triggers a fresh event in disregard of the reason that made the jump necessary (actually another jump, so another event) we end up having to deal with *two simultaneous* events.[4] Even if consequences over performance are generally negligible, models have to figure out someway that, given a bunch of events, some of them are "duplicates" of others to avoid treating each of them as independent events, yielding wrong results in some circumstances (see [2] for concrete examples of such wrong models). So to avoid practical synchronous issues, some form of collaboration between submodels—a *design pattern*—must be implemented by *library developers* and, moreover, *understood* and *correctly used* by end-users. Indeed, this collaboration scheme is unknown from the modeling tool which is then of no help to track down misuses of the submodels. This is arguably too much to require from both library developers and end-users, who should ideally focus on physics and control, rather than on low-level implementation details.

Also, it would be desirable to be able to statically (i.e., before execution) catch modeling errors resulting in *non-deterministic models* and *fragile models*.[5] Alas, the technological-and-phenomenological approach is again of no help here: in this paradigm, such modeling errors can only be detected—if they ever are—at runtime. As a consequence, development of models involving many discrete states can become cumbersome: it requires extensive testing to gain confidence in the correctness of the design, and whenever an error is detected, the location of the faulty submodels can be quite challenging. On the other hand, modeling with discrete states is precisely where the state-and-transition approach shines: when the state-and-transition graph of (part of) a model is explicit, it is possible to detect *at compilation time*, as we shall explain in subsequent sections, non-deterministic and

[1] We'll come back to them in subsequent sections when talking about automatic code generation for controllers.

[2] This is necessary in order to notify a change to possible listener submodels.

[3] In synchronous language terminology, we say that the *clock* corresponding to output changes is a *subclock* of the clock corresponding to input changes.

[4] This example will be further discussed in subsection 4.1.

[5] Fragile models are models whose correct execution relies on a property that escapes the automatic checker's proof capabilities. Some of those models are actually correct, but some others are not: we prefer to reject *all* suspicious models, forcing users to disambiguate correct ones (disambiguation is often easy) rather than accepting wrong models that may be hard to debug.

fragile *patterns* that may lead to runtime errors.

Clearly, the technological-and-phenomenological approach, despite its versatility, reaches its limits when complex discrete state submodels such as those involved in Control applications come into play. This observation has motivated the extension of AMESim's submodel description capabilities, which now feature a state-and-transition perspective.

2.2 Statecharts: an intuitive yet expressive graphical language

The general adoption of (variants of) statecharts in Control tools is due to their ability to concisely express complex finite state machines, making them reasonably understandable by humans.

Conciseness is mainly achieved thanks to the nice concept of *composite state*, which can be seen, at least in the original proposal by Harel [1], as a kind of "pseudo-abstraction" in the sense that this construct effectively allows some details of the equivalent, unfactored, flat machine to be abstracted away, but it is nevertheless necessary to reveal some contents to allow inner transitions. Statecharts also feature discrete state variables (updated in *actions*), which makes them suitable to describe even infinite state machines. Actually, Harel's statecharts come with many appealing features, so they constitute a very good starting point for our targeted applications. However, since they are historically strongly rooted in the discrete control world, they lack the concept of continuous state variable. So, like many others did before us, we have extended statecharts to support hybrid modeling. We have designed this extension so that it remains simple and intuitive, yet powerful enough to handle many practical applications.

We will review in the next section some theoretical aspects of timed systems that have guided integration of statechart modeling capabilities in AMESim. We will then dive in important technical achievements such as validation and automatic code generation before presenting the final result from an end-user point of view.

3 A variant of the statechart language

The graphical language implemented in AMESim is very similar to the original statechart language: a statechart is essentially a set of *states* represented with rectangular boxes (labelled `count` and `reset` in the example of Figure 1) and a set of possible *state transitions* represented with labelled arrows.

A transition can be associated with a *trigger*, a *guard* and *actions*. A transition is taken when an event corresponding to its trigger occurs if its guard evaluates to true. In that case, the actions—which are *state variable* assignments—are executed. For instance, the transition from `count` to `reset` in Figure 1 labelled

```
e [counter == 10] /
  counter := 0;
  reset_count := reset_count + 1
```

means that when an event associated with `e` occurs while `count` is active and the state variable `counter` is equal `10`, a transition from `count` to `reset` is taken, `counter` is set to `0` and `reset_count` is incremented.

Triggers, or event generators, are defined at statechart creation time by boolean expressions that create events on their rising edges.

In addition to these general features, a statechart can be augmented with inputs from and outputs to AMESim. An output can either refer to a state variable, i.e. a discrete variable, or to a continuous signal controlled by state activations.

4 Theoretical aspects of statecharts integration in AMESim

When Modelica-based tools *simulate* the dynamic behavior of a system, they actually try to find a reasonable approximation of the solution of a system of equations that is supposed to capture the behavior of interest. This system of equations generalizes the so-called state space representation: it roughly consists in inputs, outputs and internal variables constrained by sets of equations whose (possibly dynamic) activation determine the trajectories of the model. This form constitutes the actual *denotation* of the corresponding *desugared program*[6] from which tools need to deduce the desired approximation. Now, given a desugared program, how is this approximation actually obtained? Of course, at some point, it depends on design choices made in the simulation tool.[7] But, given a common model description language like Modelica, any implementation is supposed to follow the same *operational semantics* which state *how to compute the solution—not an approximation—of any well-behaved program*:

[6]"Flat program" in Modelica.

[7]A tool may favor one or several classes of problems (e.g., marginally stable problems, discrete problems, etc.). This contributes to the tool's added value.

this is somewhat the "reference implementation"—although a "virtual" one[8]—of a correct interpreter of the modeling language. Unfortunately, in practice, defining sound operational semantics for a physical system modeling language is such a huge work that, for most languages available today (including Modelica), only *informal semantics* (i.e., given in written human language) are available. Consequently, many inconsistencies simply cannot be spotted, because, contrary to formal descriptions, informal descriptions do not easily allow *reasoning* about the semantic model itself. This partly explains why, as pointed out in, e.g., [3], physical simulation tools experience difficulties in correctly handling some hybrid problems.

In AMESim, we had to take this fact into consideration when designing the new state-and-transition mode (which interacts with continuous behavior) so that we practically avoid most issues encountered in an unchecked implementation.

In the following subsections, the issues involved with coupling statecharts with AMESim models and the design choices made to circumvent them are discussed.

4.1 Synchrony vs. Simultaneity issues

Statecharts describe systems that *react* to the environment, i.e., they respond to external *triggers* by executing state transitions according to their internal state and their inputs. In other words, a statechart merely describes a function that computes a new internal state from a previous state and a set of inputs while the environment is responsible for providing the inputs and for deciding *when* this function should be used to compute a new state.

A first source of non-determinism originates from the fact that the execution of a statechart can sometime be triggered by several different event sources *at the same time*. The issue is then to tell if the corresponding events are actually one and the same, i.e. they are dependent, or *synchronous*, events, or if they are unrelated, i.e., *asynchronous*, and actually occur in sequence but just seem to be simultaneous due to numerical approximations.

As an example, let us consider the simple statechart of Figure 2.

It simply says that

- if the event `e1` occurs while in state `s0`, the statechart transitions from `s0` to `s1`,

[8]This guarantees implementation independence.

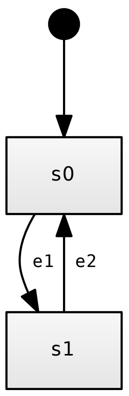


Figure 2: A simple statechart

- if the event `e2` occurs while in state `s1`, the statechart transitions from `s1` to `s0` and
- the statechart starts with `s0` being active.

If the statechart is executed because only one of `e1` or `e2` has occurred, the computation of the new state is quite straightforward. But what should happen if, for example, `s0` is active and both `e1` and `e2` are sensed simultaneously? Different interpretations are possible.

A first interpretation might be to assume that events that occur simultaneously are *exactly* the same, i.e. that they originate from the same *primary* external source and should consequently only be taken into account once. In the example this means that the new active state should be `s1`.

Another interpretation could be that `e1` and `e2` are *independent* and that the fact that they appear simultaneous is just a numerical artefact. In that case, the statechart would need to be executed twice according to the sequence in which `e1` and `e2` have occurred. If `e1` happens first, `s1` would become the active state after the first execution and `s0` would again be after the second one. It should be noted that in this interpretation it is not enough to know that the events are independent: the order in which events occur needs to be determined as well.

Both interpretations can be perfectly valid depending on the external context. Let us reuse examples similar to those presented in [2] to illustrate this fact.

Let us assume that the events `e1` and `e2` occur when some external signals `i1` and `i2` respectively become larger or equal to zero.

Figure 3 shows an AMESim model where the same signal source is connected to both `i1` and `i2`. The source defines a piece-wise constant signal that is -1 when $t < 1$ and 1 when $t >= 1$. As a result, events `e1` and `e2` happen simultaneously when $t = 1$. In that case, it would make sense to consider that `e1` and `e2` refer to the same events and to adopt the corresponding interpretation in the statechart.

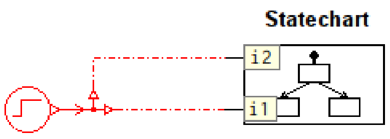


Figure 3: Dependent event sources

Let's now consider the model shown in Figure 4. The same statechart model is fed the outputs of position sensors attached to two identical mass with viscous friction models. If the velocities and positions of the masses are initialized to the same values, both positions will become non-negative simultaneously (if they ever do). However, it does not make sense to consider that these events are related as the models that generate them have nothing to do with each other. It seems much more natural to take them into account one after the other as if they had been sensed in sequence. The order of this sequence is not that important here as there is absolutely no reason to favor one model over the other. Actually, in real life, two seemingly identical systems submitted to the same inputs will always behave slightly differently at some scale because of uncontrolled parameters.

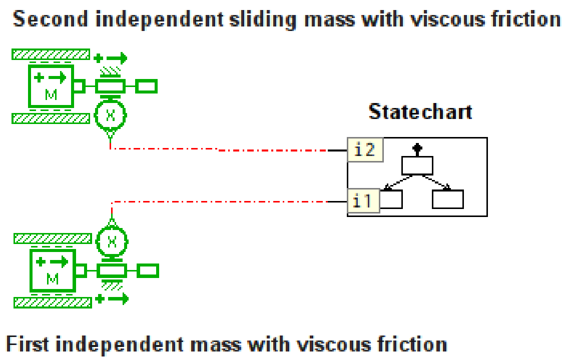

Second independent sliding mass with viscous friction
Statechart
i2
i1
M
M
First independent mass with viscous friction

Figure 4: Independent event sources

A third way of using the statechart of Figure 2 is presented in Figure 5. This model illustrates the idea expressed in Subsection 2.1 about cascading events: a piece-wise constant signal is fed to `i1` while its integer part is fed to `i2`. Both inputs cross zero at the same time, generating simultaneous events. One could consider that they are the same events, as in the model in Figure 3, but one could also consider that `e2` is a *consequence* of `e1` and should then be handled *after* `e1`.

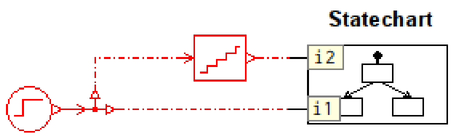


Figure 5: Independent event sources where order matters

The examples presented above show that very different meanings can be given to simultaneous events. Unfortunately, a continuous-time modeling environment is unable to give any insight about which is the expected one, the information being simply unavailable. This shortcoming is particularly critical when executing a statechart as making wrong decisions in a discrete model can radically alter the course of a simulation compared to continuous-time detailed physical models where energy conservation principles make models more robust with regard to non-determinism.

4.2 Observability of state transitions

In AMESim, just like in Modelica, discrete states can only be assigned *once* when a continuous-time event occurs.[9] This makes up another obstacle in the way of coupling a statechart with a continuous-time model. Indeed, a statechart may need to execute several transitions *without increasing the elapsed time in the model*, thus updating its state multiple times as a response to a unique continuous-time event.

Let us consider for example the statechart of Figure 6 featuring an input `i`, a discrete real output `o` and an event generator `e`. The statechart starts in state `s0`. When `e` generates an event, state `s1` is entered and `o` is set to `1.0`. Besides, if the guard `i > 0.0` is satisfied at that time, `s2` directly becomes the new state and `o` is set to `2.0`, *without awaiting a new event*.

This clearly contradicts the assumptions stated above. In Modelica, for instance, something like the partial Modelica code in Listing 1 would need to be written, which is invalid as the second `when` clause introduces an algebraic loop involving `s1` and because `s1` and `o` are potentially constrained by two equations at once when `e` occurs.

Listing 1: Invalid Modelica code for the statechart of

[9]The values of certain states can actually be changed by a Modelica simulator as part of a solving process, but these intermediate values should never be relied upon in a model.

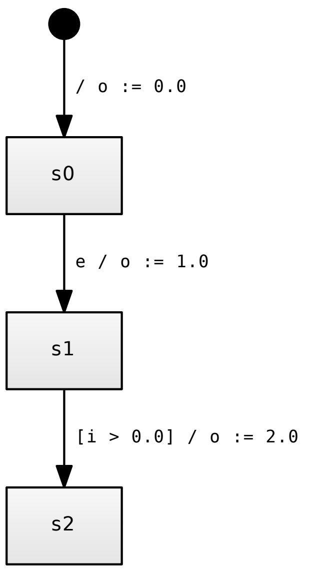


Figure 6: A statechart involving a potential double assignment

Figure 6

```
when initial() then
  s0 = true;
  s1 = false;
  s2 = false;
  o = 0.0;
elsewhen e then
  if pre(s0) then
    s0 = false;
    s1 = true;
    o = 1.0;
  end if;
end when;

// when state s1 is entered...
when s1 then
  if i > 0.0 then
    s1 = false;
    s2 = true;
    o = 2.0;
  end if;
end when;
```

One could, of course, think of working around these obstacles by “inlining” the intermediate transition, i.e. by rewriting the statechart with a direct transition from 0 to s2 as done in Listing 2.

Listing 2: Modelica code for the statechart of Figure 6 using “inlined” transitions

```
when initial() then
  s0 = true;
  s1 = false;
  s2 = false;
  o = 0.0;
elsewhen e then
  if pre(s0) then
    if i > 0.0 then
      s0 = false;
      s2 = true;
      o = 2.0;
    else
      s0 = false;
      s1 = true;
      o = 1.0;
    end if;
  end if;
end when;
```

This is valid Modelica, but the behavior is not exactly the expected one. Indeed, from an external point of view, o jumps from 0.0 to 2.0 directly without ever taking the 1.0 value. This may seem harmless for one used to physical continuous-time modeling, but what if this output was fed to another discrete part, e.g. another statechart, that relies on it to significantly alter its behavior? Figure 7 shows an AMESim model that depends on the intermediate value being taken. Event e is fired when the input becomes positive which at the same time satisfies the guard i > 0.0. The output is connected to a discrete subsystem that increments a counter when it gets exactly equal to 1.0. This means of course that the counter will only be incremented if the intermediate value is properly observed.

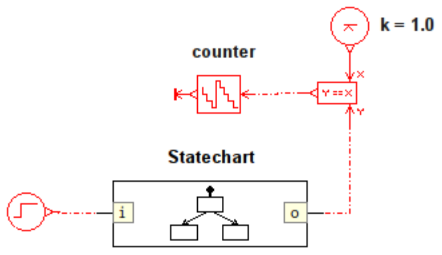


Figure 7: A model exposing observability issues

It should be noted that using Modelica algorithms to rewrite the model as shown in Listing 3 exposes the same issue as o is only equal to 1.0 during an intermediate step inside the algorithm.

Listing 3: Modelica code for the statechart of Figure 6 using an algorithm

```
algorithm
  when initial() then
    s0 := true;
    s1 := false;
    s2 := false;
    o := 0.0;
  elsewhen e then
    if pre(s0) then
      s0 := false;
      s1 := true;
      o := 1.0;
    end if;
  end when;

  // when state s1 is entered...
```

```
    when s1 then
      if i > 0.0 then
        s1 := false;
        s2 := true;
        o := 2.0;
      end if;
    end when;
```

4.3 Design choices

When designing the statechart extension to AMESim, the main focus was placed on the robustness, reliability and usability of the solution. That is why special care was taken to avoid as much as possible the issues presented above. This necessarily lead to additional rules in our variant of the statechart language that will be justified in this section.

First, it was decided to avoid the simultaneity issues described in Subsection 4.1 altogether by making sure that the execution of a statechart is always independent of the interpretation given to simultaneous events. This means that if any two events occur at the same time, assuming that they are synchronous or asynchronous should not change the upcoming computation. In the language described in this paper, events triggering a statechart simultaneously are simply forbidden.[10]

This rule is partially enforced by the statechart environment of AMESim by statically checking that a transition cannot generate an event that would conflict with the one that triggered the transition in the first place. What is more, simultaneous external events are detected at runtime and result in aborting the simulation. This guarantees the deterministic execution of a statechart provided that no dependent event sources are created by directly connecting—without inserting a continuous or discrete state variable as a buffer—an output to an input.[11]

One may argue that another solution might have been to stick to one interpretation anytime events occur simultaneously. But what if the only interpretation that makes sense is precisely the other one like in the model of Figure 4, where assuming that the events are dependent is clearly not expected? That is why signaling ambiguous situations was favored over making arbitrary choices behind the scene. As a side note, the "independent events" interpretation brings its lot of additional questions. How can the ordering of events be determined? How can several events be processed without increasing the continuous time and without encountering the issues of Subsection 4.2?[12]

[10]This rule may be made less restrictive in the future if it turns out that it significantly enlarges the range of valid models.

[11]Avoiding this is the responsibility of the environment and cannot be enforced locally in a statechart.

Similarly, to avoid the observability issues presented in Subsection 4.2, the decision was made to enforce that the *effect* of every taken transition can always be observed from outside a statechart. This means that an *output* cannot be set more than once during one execution of a statechart. This property can be checked statically assuming that the first rule about simultaneous events is enforced.

5 Practical validation of statecharts

To ensure the safe execution of a statechart definition, the fulfilment of the aforementioned constraints has to be statically checked before generating code. Beside trivial checks such as making sure that the state machine contains a unique initial state or that at least one event generator is present (to ensure time progression), a few non-trivial checks have also been implemented. These checks are presented hereafter.

5.1 Transition expression checking

Writing correct transition expressions is error-prone: as in any other textual language, one can easily make syntax and semantic errors such as referring to a non-existent variable or event generator, or combine incompatible expressions.

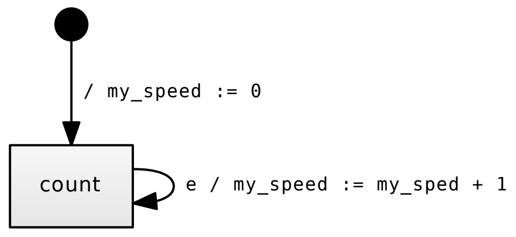


Figure 8: A statechart with a typo in an identifier

In order to avoid runtime errors due to unknown identifiers, each input, output and event must be declared before use. All local variables have to be initialized, and thus implicitly declared, on the initial transition of a statechart. These rules imply that all variables are known at compile time and allow rejecting the statechart of Figure 8 where `my_sped` is not a valid variable name.

[12]For instance, a state transition may change an output that in turn invalidates an event that has already been placed in the processing queue.

Avoiding element misuse requires a more complex solution: a static type system. This type system was designed to avoid the need for user provided type annotations, which would make writing transition expressions cumbersome. The Hindley/Damas/Milner type inference algorithm [4, 5] is used to type check all transitions without type annotations.

The basic idea of type inference is to traverse all expressions of a program to gather various typing constraints and correlations and then to resolve all those constraints in a second pass, thus attributing their final types to variables.

Let us consider for example the following expression

```
e / v := 2 + v; x := sin(v) + x
```

and the primitive type definitions below.

`sin`	$\text{Real} \to \text{Real}$
`2`	$\forall \alpha \in \{\text{Integer}, \text{Real}\}, \alpha$
`(+)`	$\forall \alpha \in \{\text{Integer}, \text{Real}\}, \alpha \to \alpha \to \alpha$

The first step to perform type inference on the example expression is to attribute free type variables to the variables in use, i.e. v will get type β, x type δ and e type γ.

As e is used in the trigger section of the expression, the typer can deduce the constraint $\gamma = \text{Event}$, which gives the final type for e: Event.

In the first action, the type of the (+) operator implies that both its operands must have the same type, which can be either Integer or Real. This means that the type of variable v must satisfy the following constraint:

$$\beta \in \{\text{Integer}, \text{Real}\}.$$

On the other hand, the type constraint for 2 matches exactly the one for the operands of (+) and can then be omitted in the next equivalences as it brings no additional information.

Similarly, the use of the (+) operator in the second action yields the following constraint:

$$\delta \in \{\text{Integer}, \text{Real}\}.$$

The presence of the `sin` function generates tighter constraints, as it bounds the types of the input and output variables, yielding:

$$\beta = \text{Real},$$
$$\delta = \text{Real}.$$

Gathering all the inequalities together gives the following final typing equation system:

$$\beta = \text{Real}, \beta \in \{\text{Integer}, \text{Real}\},$$
$$\delta = \text{Real}, \delta \in \{\text{Integer}, \text{Real}\}.$$

It can then be simplified to deduce that v has type Real and that x also has type Real. An impossibility to simplify type equations, like obtaining Integer = Real would have meant a type error which should be reported to the user.

5.2 Activation chain analysis

Transitions that have no trigger section do not necessarily stop the execution of a statechart; they are taken as far as their guards allow, without waiting for another event.

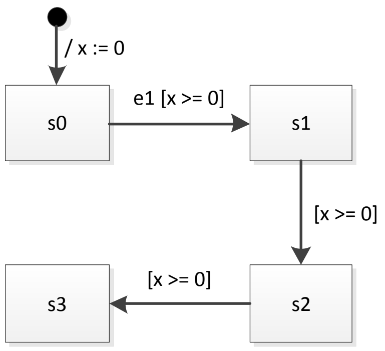


Figure 9: An activation chain

The execution of the statechart of Figure 9 exhibits such a behavior: when the event e1 is raised while in s0, the state machine will take the transition to s1 as x is equal to 0, then take the transition to s2, as the transition has no trigger, and finally to s3. To an external observer, the visible state activation will go from s0 to s3 directly. A path made of transitions that can be taken globally during an execution of a statechart is called an activation chain.

This behavior can results in infinite looping if not handled carefully, like in Figure 10. The semantics of our statechart language imply that the execution will continue indefinitely between s0 and s1 (highlighted in red), without ever resuming continuous-time simulation.

To ensure that a statechart will never stall a simulation, activation chain cycling is forbidden, hence forcing users to break cycles with triggers. For example, fixing the statechart of Figure 10 requires the addition of a trigger on the transition between states s1 and s0.

The possibility to take several transitions in a single execution leads to potential duplicated assignments of variables, as in Figure 6 where variable o is assigned

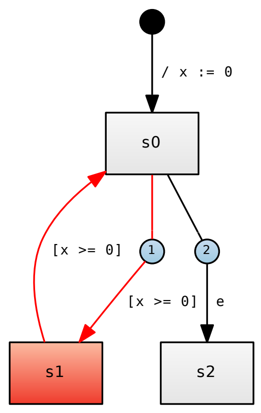


Figure 10: A cyclic activation chain

twice, once on the transition from `s0` to `s1` and once on the transition from `s1` to `s2`.

As explained in Subsection 4.2 about transition observability issues, this behavior is undesired and the implementation prevents it by analyzing the assignments along every activation chain and rejecting the activation chains that assign a variable more than once.

5.2.1 Practical chains analysis

The algorithm used to check the invariants mentioned above is mainly a depth-first search coupled with memoization. Each state is visited to compute its activation chains and forward assigned variables while the set of visited states in the current activation chain and the set of already assigned variables are maintained. Any intersection with the visited values and the ones already stored results in an error.

Another possibility would be to use a data-flow framework and express activation and assignment as liveness information to be propagated by the framework.

6 Code generation strategy

In this section, an overview of a code generation strategy leveraging the existing AMESim Modelica tool chain is presented.

6.1 Describing statecharts in Modelica

Modelica fits well to our purpose as a statechart that passes the validation stages discussed in the previous section can be described by a Modelica model that is:

- *valid*, i.e., is guaranteed to compile without error, thanks to the syntax and type check phases, and
- *sound* with respect to mixed discrete/continuous semantics as all ambiguous models are filtered out by the restrictions regarding simultaneous events.

Listing 4 shows how Modelica code can be generated to describe the statechart of Figure 1.

Listing 4: Code generated for the statechart of Figure 1

```
  e = pulse > 0.0;
  when initial() then
    reset_count = 0;
    counter = 0;
    st2 = true;
    st1 = false;
  elsewhen e then
    if pre(st2) then
      if pre(counter) == 10 then
        counter = 0;
        reset_count = pre(reset_count) + 1;
        st2 = false;
        st1 = true;
      else
        counter = pre(counter) + 1;
        reset_count = pre(reset_count);
        st2 = true;
        st1 = false;
      end if;
    elseif pre(st1) then
      counter = pre(counter) + 1;
      reset_count = pre(reset_count);
      st2 = true;
      st1 = false;
    else
      counter = pre(counter);
      reset_count = pre(reset_count);
      st2 = pre(st2);
      st1 = pre(st1);
    end if;
  end when;
```

Each event generator is associated with an `elsewhen` clause with the logic to handle this specific event as the body of the clause. This logic is encoded as cascading `if` statements representing the various transitions, the transition priorities being used to order the conditions. The equations describe the accumulated content of the action part of every transition and of the state activation updates.

Activation chains are generated recursively, for each level of the chain, the final body standing for the concatenation of all the actions. For instance, adding a transition from `reset` to `count` with the expression `[reset_count > 2]` and a priority below the existing one in the example counter statechart would create an activation chain resulting in the code shown in Listing 5.

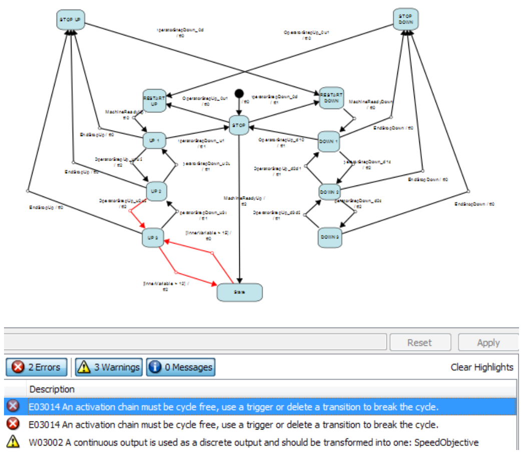
Listing 5: Code generated for the statechart of Figure 1 augmented with an activation chain

```
  ...
```

```
if pre(st2) then
  if pre(counter) == 10 then
    if pre(reset_count) > 2 then
      counter = 0;
      reset_count = pre(reset_count) + 1;
      st2 = true;
      st1 = false;
    else
      counter = 0;
      reset_count = pre(reset_count) + 1;
      st2 = false;
      st1 = true;
    end if;
  else
    counter = pre(counter) + 1;
    reset_count = pre(reset_count);
    st2 = true;
    st1 = false;
  end if;
elseif pre(st1) then
...
```

7 Graphical user interface aspects

AMESim provides an editor to let users easily create statecharts and statically validate them, highlighting erroneous parts in red as seen in Figure 11.



E03014 An activation chain must be cycle free, use a trigger or delete a transition to break the cycle.
E03014 An activation chain must be cycle free, use a trigger or delete a transition to break the cycle.
W03002 A continuous output is used as a discrete output and should be transformed into one: SpeedObjective

Figure 11: A statechart with a structural error highlighted in red

The editor also serves as a debugger as it is able to replay the behavior of a statechart during simulation, highlighting the active states and showing the values taken by all variables. Additional post-processing features are available, such as the possibility to easily jump between state changes. Timing diagrams representing state activations can be obtained using the regular AMESim data plotting facilities by displaying the activation variable associated with every state.

8 Conclusion and perspectives

In this paper, a few challenging issues associated with modeling and simulating hybrid models in an asynchronous environment are discussed.

A practical solution implemented in AMESim to avoid those issues and to offer a reliable and usable user interface is presented. This solution is built on top of the AMESim Modelica tool chain and demonstrates how a specific language can be implemented in terms of a more general language. The condition is, of course, that any model expressed in the specific language can somehow also be expressed without loss of meaning in the base language, which demands special care. However, the advantages of this approach are appealing as it is then possible to combine the convenience associated with a user friendly dedicated language with the power of a more general underlying language allowing to connect models expressed using different paradigms together. A core language can thus be extended to new applicative domains without being altered, retaining its generality.

Future work involves extending the supported subset of the statechart language, for example to allow parallel states. Developing practical and scalable means of building statecharts by composing smaller ones is another interesting perspective.

References

[1] David Harel. Statecharts: A visual formalism for complex systems, 1987.

[2] Sébastien Furic. Enforcing reliability of discrete-time models in modelica. In *Proceedings of the 8th International Modelica Conference*, 2011.

[3] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877 – 910, 2012.

[4] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.