Finite State Machines and Modal Models in Ptolemy II

Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-151
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html

November 1, 2009

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, Thales, and Toyota.

Finite State Machines and Modal Models in Ptolemy II [*]

Edward A. Lee
eal@eecs.berkeley.edu
Electrical Engineering & Computer Science
University of California, Berkeley

November 1, 2009

Abstract

This report describes the usage and semantics of finite-state machines (FSMs) and modal models in Ptolemy II. FSMs are actors whose behavior is described using a finite set of states and transitions between the states. The transitions between the states are enabled by guards, which are boolean-valued expressions that can reference inputs to the actor and parameters in scope. The transitions can produce outputs and can update the value of parameters in scope. Modal models extend FSMs by allowing states to have refinements, which are hierarchical Ptolemy II models. The refinements may themselves be FSMs, modal models, or any composite actor containing a director compatible with the domain in which the modal model is being used. This report describes the operational semantics, the practical usage, and the semantics of time in modal models.

1 Introduction

The behavior of actors in Ptolemy II can be defined in a number of ways. In this report, we explain how to give the behavior of an actor as a finite-state machine or a modal model. Intuitively, the **state** of a system or subsystem is its condition at a particular point in time. In general, the state affects how the (sub)system reacts to inputs. Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs.

[*]**NOTE:** If you are reading this document on screen (vs. on paper) and you have a network connection, then you can click on the figures showing Ptolemy II models to execute and experiment with those models on line. There is no need to pre-install Ptolemy II or any other software. The models that are provided online are summarized at http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/jnlp-books/doc/books/design/modal/index.htm.

For example, the `Ramp` actor, which produces a counting sequence, has state. Its reaction to a *trigger* input depends on how many times it has perviously fired. It uses a local variable to keep track of where in its counting sequence it is. This local variable is called a **state variable**, and in this case, the state variable has (typically) a numerical value.

In the case of the `Ramp` actor, the number of possible states depends on the data type of the counting sequence. If it is `int`, then there are 2^{32} possible states. If it is `double`, then there are 2^{64}. The number of possible states is very large. If the data type is `String`, then the number of possible states is infinite. Although the number of states is very large, the logic for changing from one state to the next is rather simple. So reasoning about the behavior of the actor is not difficult. The actor begins in a state given by its *init* parameter, and on each firing, increments its state by adding to it the value of the *step* parameter. (If the data type is `String`, then "adding" means concatenating.)

In contrast, it is common to have actors that have a relatively small number of possible states, but relatively complex logic for moving from one state to the next. The mechanisms described here support design, visualization, and analysis of such actors. We first explain the Ptolemy II infrastructure supporting finite state machines (FSMs), and then explain the use of FSMs to construct modal models.

The next section below explains FSMs in Ptolemy II and gives some examples of their usage. The section after that explains modal models, which extend FSMs with the ability to have hierarchical refinements of the states.

2 Finite State Machines

A **state machine** is a system whose outputs depend not only on the current inputs, but also on the current state of the system. The state of a system is a summary of everything the system needs to know about previous inputs in order to produce outputs. The state of a system may be represented by a state variable $s \in \Sigma$, where Σ is the set of all possible states that the system can be in. A **finite state machine** (**FSM**) is a state machine where Σ is a finite set.

Example 1: Consider a thermostat controlling a heater that is a state machine with states $\Sigma = \{heating, cooling\}$. If the state is $s = heating$, then the heater is on. If $s = cooling$, then the heater is off. Suppose the target temperature is 20 degrees Celsius. If the heater is on, then the thermostat allows the temperature to rise past the target to, say, 22 degrees. If the heater is off, then it allows the temperature to drop past the target to, say, 18 degrees. Thus, the behavior depends on the state, which summarizes the history by remembering whether the heater is on off. This strategy avoids **chattering**, where the heater would turn on and off rapidly when the temperature is close to the target temperature.

2.1 FSMActor

`FSMActor` is a composite actor where refinement consists of states and transitions rather than other actors. Ptolemy II provides a visual notation for these states and transitions as shown in Figure 1. In that figure, the `FSMActor` has two input and two output ports, created by the model builder. In general, and `FSMActor` can have any number of input and output ports. The actor reacts to inputs and produces outputs as specified by an FSM, shown visually at the bottom of the figure. The FSM contains a finite number of states (three in the figure). One of these states is an **initial state** (labeled `initialState` in the figure), which is the state of the actor at the beginning of execution of the model. The initial state is indicated by a bold outline. Some of the states may also be **final states**, indicated visually with a double outline (more about final states later). States are connected by **transition**s, the annotations on which determine what happens when the actor fires. Before diving

Foundations: Model of State Machines

State machines are often described in the literature as a five tuple $(\Sigma, I, O, T, \sigma)$. Σ is the set of states, and σ is the initial state. Nondeterminate state machines may have more than one initial state, in which case $\sigma \subset \Sigma$ is itself a set, although this particular capability is not supported in Ptolemy II FSMs. I is a set of possible valuations of the inputs. In Ptolemy II FSMs, I is a set of functions of the form $i \colon P_i \to D \cup \varepsilon$, where P_i is the set of input ports (or input port names), D is the set of values that may be present on the input ports at a particular firing, and ε represents "absent" inputs (i.e., $i(p) = \varepsilon$ when p`_isPresent` evaluates to false). O is similarly the set of all possible valuations for the output ports at a particular firing.

For a deterministic state machine, T is a function of the form $T : \Sigma \times I \to \Sigma \times O$, representing the transition relations in the FSM. The guards and output actions are, in fact, just encodings of this function. For a nondeterministic state machine (which is supported by Ptolemy II), the codomain of T is the powerset of $\Sigma \times O$, allowing there to be more than one destination state and output valuation.

The classical theory of state machines (Hopcroft and Ullman, 1979) makes a distinction between a **Mealy machine** and a **Moore machine**. A Mealy machine associates outputs with transitions. A Moore machine associates outputs with states. Ptolemy II supports both, using output actions for Mealy machines and state refinements in modal models for Moore machines.

Ptolemy II state machines are actually extended FSMs, which require a richer model than that given above. Extended state machines add to the model above a set V of variable valuations, which are functions of the form $v \colon N \to D$, where N is a set of variable names and D is the set of values that variables can take on. An extended state machine is a six-tuple $(\Sigma, I, O, T, \sigma, V)$ where the transition function now has the form $T : \Sigma \times I \times V \to \Sigma \times O \times V$ (for deterministic state machines). This function is encoded by the transitions, guards, output actions, and set of actions of the FSM.

into the details, however, the thermostat example may help make the notation intuitive.

Example 2: A model of a thermostat and a heater is shown in Figure 2. The reader should be able to read that figure without much help. In that figure, the `FSMActor` has a *temperature* input and a *heat* output. Its behavior is given by the FSM shown in the grey box. There are two states, $\Sigma = \{heating, cooling\}$. There are four transitions. The guard on each transition gives the conditions under which the transition is taken. The output actions on each transition give the values produced on the output ports when the transition is taken. Reading the diagram, we see that while in the *heating* state, if the *temperature* input is less than *heatOffThreshold* (22.0), then the output value is *heatingRate* (0.1). When the *temperature* input becomes greater than or equal to *heatOffThreshold*, then the FSM changes to the *cooling* state and produces output value given by *coolingRate* (-0.05). An example execution is plotted in Figure 4.

2.2 Execution Policy for an FSMActor

An FSMActor contains a set of states and transitions. One of the states is an initial state, and any number of states may be final states. Each transition has a **guard expression**, any number of **output actions**, and any number of **set actions**. At the start of execution, the state of the actor is set to the initial state. Subsequently, each firing of the actor is a sequence of steps as follows. In the `fire()` method, the actor

1. reads inputs;
2. evaluates guards on outgoing transitions of the current state;
3. chooses a transition whose guard evaluates to true; and

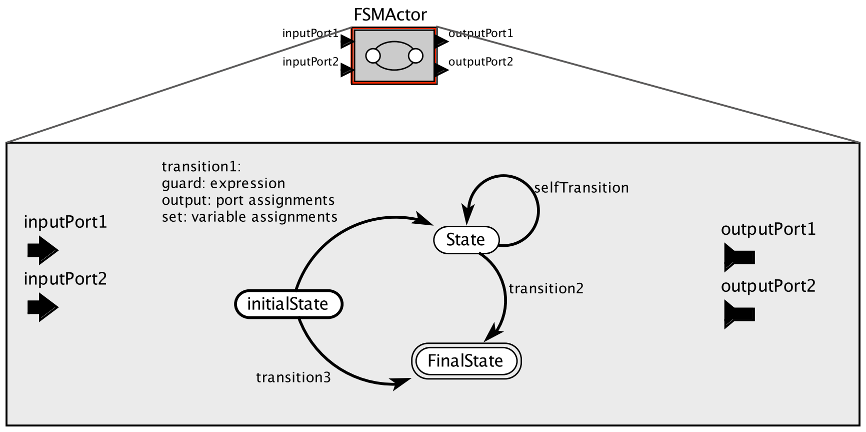


Figure 1: Visual notation for state machines in Ptolemy II.

4. executes the output actions on the chosen transition, if any.

In the postfire() method, the actor

5. executes the set actions of the chosen transition; and
6. changes the current state to the destination of the chosen transition.

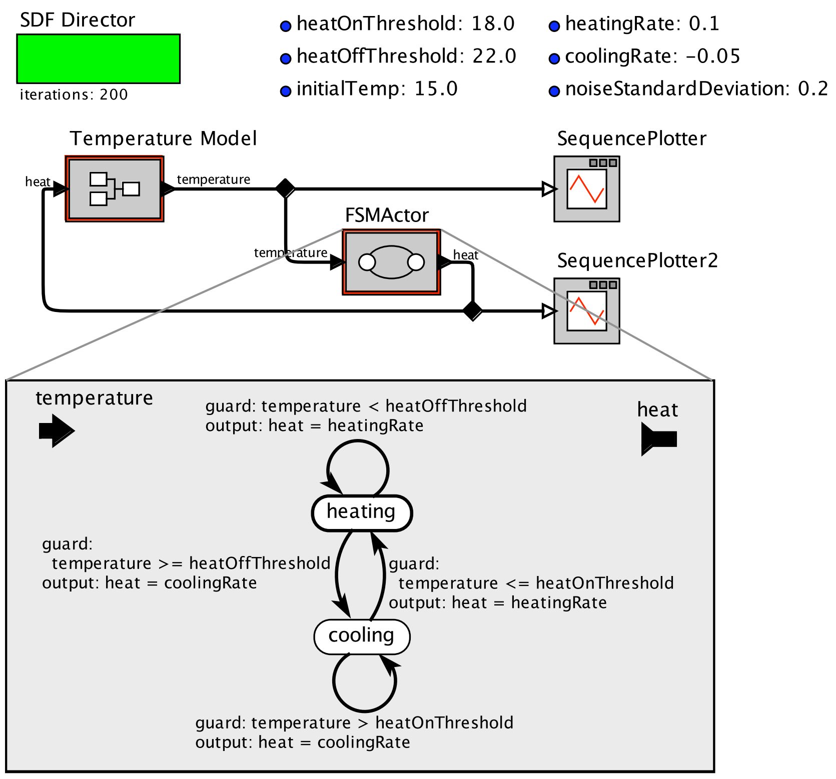

SDF Director
iterations: 200
heatOnThreshold: 18.0
heatOffThreshold: 22.0
initialTemp: 15.0
heatingRate: 0.1
coolingRate: -0.05
noiseStandardDeviation: 0.2
Temperature Model
heat
temperature
SequencePlotter
FSMActor
temperature
heat
SequencePlotter2
temperature
guard: temperature < heatOffThreshold
output: heat = heatingRate
heat
heating
guard:
temperature >= heatOffThreshold
output: heat = coolingRate
guard:
temperature <= heatOnThreshold
output: heat = heatingRate
cooling
guard: temperature > heatOnThreshold
output: heat = coolingRate

Figure 2: A model of a thermostat with hysteresis. The `Temperature Model` actor is shown in Figure 3.

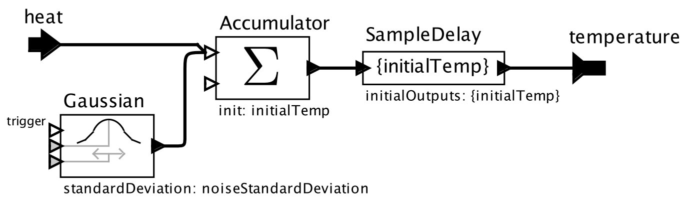


Figure 3: The `Temperature Model` composite actor of Figure 2.

These are separated into two distinct methods to support the use of this actor in domains that do a fixed-point iteration (such as SR and Continuous), as explained below in Section 2.9. In domains that do not do that (such as PN, SDF, and DDF), steps 1 through 6 can execute in sequence in each iteration, and the distinction between `fire()` and `postfire()` is not important.

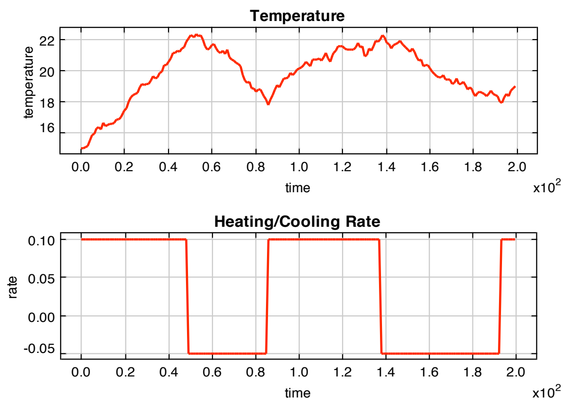


Figure 4: Two plots generated by Figure 2, showing the temperature (above) and whether the heater is on or off (below), both as a function of time.

Probing Further: Hysteresis

The thermostat in example 2 exhibits a particular form of state-dependent behavior called **hysteresis**. A system with hysteresis has the property that the absolute time scale is irrelevant. Suppose the input is a function of time, $x\colon \mathbb{R} \to \mathbb{R}$ (for the thermostat, $x(t)$ is the temperature at time t). Suppose that input x causes output $y\colon \mathbb{R} \to \mathbb{R}$, also a function of time. E.g., in Figure 4, x is upper signal and y is the lower one. For this system, if instead of x is the input is x' given by

$$x'(t) = x(\alpha \cdot t)$$

for a non-negative constant α, then the output is y' given by

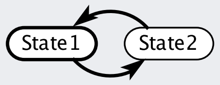
$$y'(t) = y(\alpha \cdot t) \ .$$

Scaling the time axis at the input results in scaling the time axis at the output, so the absolute time scale is irrelevant.

An alternative implementation to for the thermostat would use a single temperature threshold, but instead would require that the heater remain on or off for at least a minimum amount of time, regardless of the temperature. This design would not have the hysteresis property.

Probing Further: Internal Structure of FSMActor

`FSMActor` is a subclass of `CompositeEntity`, just like `CompositeActor`. Internally, it contains some number number of instances of `State` and `Transition`, which are subclasses of `Entity` and `Relation` respectively. The simple structure shown below:



is represented in MoML as follows:

```
<entity name="FSMActor" class="...FSMActor">
  <entity name="State1" class="...State">
    <property name="isInitialState" class="...Parameter"
      value="true"/>
  </entity>
  <entity name="State2" class="...State"/>
  <relation name="relation" class="...Transition"/>
  <relation name="relation2" class="...Transition"/>
  <link port="State1.incomingPort" relation="relation2"/>
  <link port="State1.outgoingPort" relation="relation"/>
  <link port="State2.incomingPort" relation="relation"/>
  <link port="State2.outgoingPort" relation="relation2"/>
</entity>
```

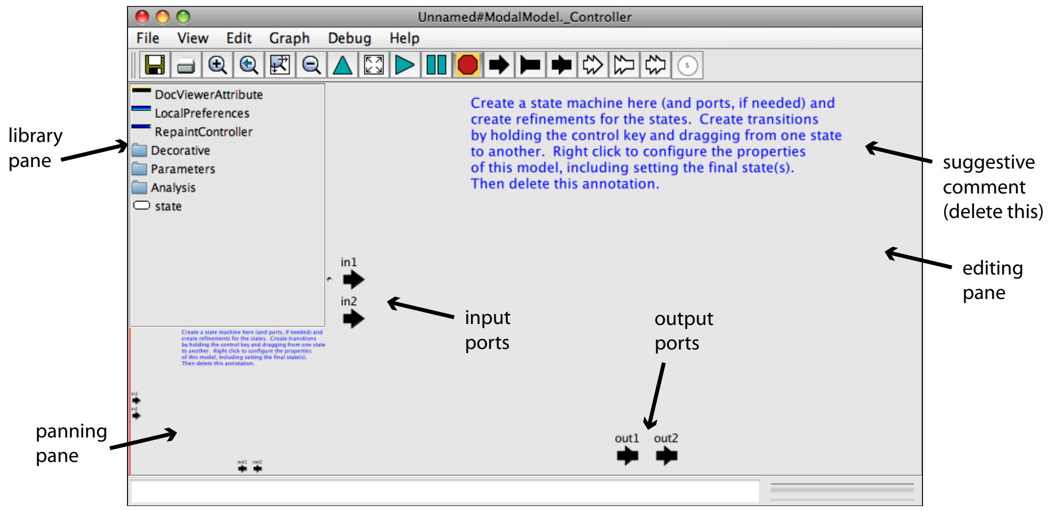
The same structure can be specified in Java as follows:

```
  import ptolemy.domains.modal.kernel.FSMActor;
  import ptolemy.domains.modal.kernel.State;
  import ptolemy.domains.modal.kernel.Transition;

  FSMActor actor = new FSMActor();
  State state1 = new State(actor, "State1");
  State state2 = new State(actor, "State2");
  Transition relation = new Transition(actor, "relation");
  Transition relation2 = new Transition(actor, "relation2");
  state1.incomingPort.link(relation2);
  state1.outgoingPort.link(relation);
  state2.incomingPort.link(relation);
  state2.outgoingPort.link(relation2);
```

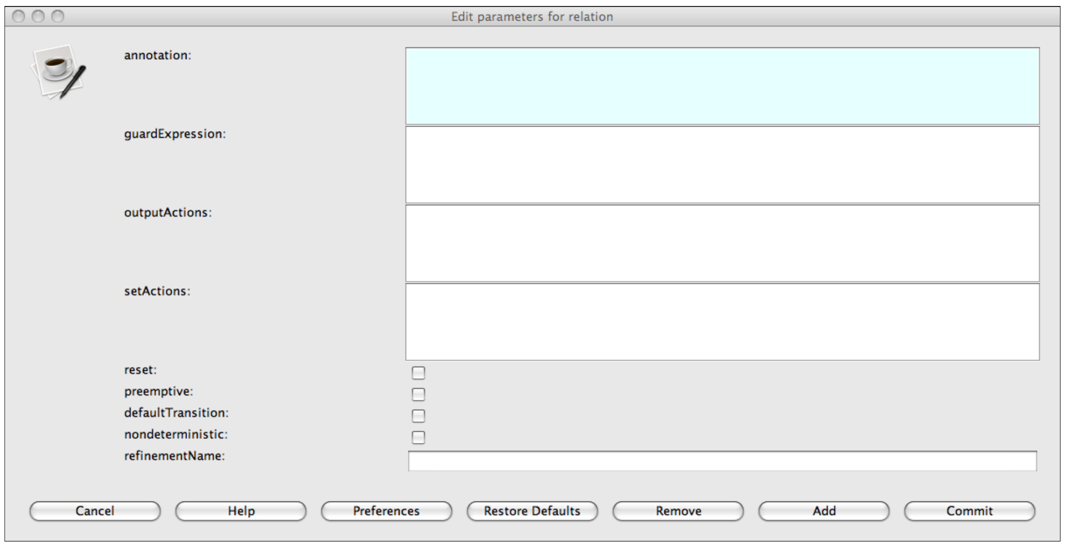
Thus, above, we see three distinct concrete syntaxes for the same structure.

After reading the inputs, this actor examines the outgoing transitions of the current state, evaluating their guard expressions. A transition is *enabled* if its guard expression evaluates to true. A blank guard expression is interpreted to be always true. The guard expression may refer to any input port and any variable in scope.



Create a state machine here (and ports, if needed) and create refinements for the states. Create transitions by holding the control key and dragging from one state to another. Right click to configure the properties of this model, including setting the final state(s). Then delete this annotation.

Figure 5: Editor for FSMs in Vergil, showing two input and two output ports, before being populated with an FSM.



annotation:
guardExpression:
outputActions:
setActions:
reset:
preemptive:
defaultTransition:
nondeterministic:
refinementName:

Figure 6: Dialog box for configuring a transition in an FSM.

2.3 Referencing Input Values of an FSMActor

If an input port name *portName* is used in a guard expression, it refers to the current input on that port on channel zero. For example, in Figure 2, in the guard expression "`temperature < heatOffThreshold`" the variable `temperature` refers to the current value in the input port *temperature* input port, and `heatOffThreshold` refers to the parameter named *heatOffThreshold*.

In many models of computation, an input may be `absent`. If a port p is `absent`, then an expression like "`p < 10`" evaluates to false. However, an expression like "`p < 10 || true`" evaluates to true. A clearer technique that leads to more readable state machine models is to use the symbol `portName_isPresent` in guard expressions. This is a boolean that is true if an input on port *portName* is present.

Recall that a multiport may have multiple channels. To refer to a channel specificatlly, a guard expression may use the symbol `portName_`*`channelIndex`*, where *`channelIndex`* is an integer between 0 and $n-1$, where n is the number of channels connected to the port. This symbol will evaluate to the value received on the port on the given channel. Similarly, a guard expression may refer to `portName_`*`channelIndex`*`_isPresent`.

Mechanics: Creating FSMs in Vergil

`FSMActor` in Vergil is in `MoreLibraries/Automata`. You can equivalently use `ModalModel` from the `Utilities` library. We recommend using `ModalModel` since it is a more general actor and can do everything `FSMActor` can do.

First, drag the actor into your model from the library. Populate the actor with input and output ports by right clicking (or control-cliking on a Mac) and selecting [`Open Actor`]. The resulting window is shown in Figure 5. It is similar to other Vergil windows, but has a customized library consisting primarily of a `State`, a library of parameters, and a library of decorative elements for annotating your design. It also includes a textual annotation with a suggestive comment that you will want to delete after reading. Drag in one or more states. To create transitions between states, **hold the control key** (or the Command key on a Mac) and click and drag from one state to the other. The grab handles on the transitions can be used to control the curvature and positioning of the transitions. Double click (or right click and select [`Configure`]) on the transition to set the guard, output actions, and set actions by entering text into the dialog box shown in Figure 6. You can also specify an annotation associated with the transition that has no effect on execution, and therefore functions like a comment.

2.4 Output Actions

Once a transition is chosen, its **output actions** are executed. The output action are specified by the *outputActions* parameter of the transition. The form of an output action is typically *portName* = *expression*, where the expression may refer to input values as above or any parameter in scope. For example, in Figure 2, the line

```
output: heat = coolingRate
```

specifies that the output port named *heat* should produce the value given by the parameter *coolingRate*. This gives the behavior of a **Mealy machine**, which is a style of state machine where outputs are produced by transitions rather than by states. **Moore machine** behavior is also achievable using state refinements that produce outputs, as explained below in Section 3.

Multiple output actions may be given by separating them with semicolons, as in *port1* = *expression1*; *port1* = *expression1*.

2.5 Set Actions and Extended Finite State Machines

The **set actions** on a transition can be used to set the values of parameters of the state machine. One practical use for this is to create an **extended FSM**, which is a finite state machine extended with a numerical state variable. It is called "extended" because the number of states now depends on the number of distinct values that the variable can take. It can even be infinite.

Example 3: A simple example of an extended FSM is shown in Figure 7, which again should be readable without much help. In this example, the `FSMActor` has parameter *count*, shown with the small blue bullet next to it. The transition from the initial state `init` to the `counting` state initializes *count* to 0 in its set action. The `counting` state has two outgoing transitions, one that is a self transition, and the other that goes to `final`. The self transition is taken as long as *count* is less than 5. That transition increments the value of *count* by one. When the value of *count* reaches 5, the transition to `final` is taken. Before taking that transition, the output is set equal to the input. Upon taking that transition, the output is henceforth constant. Therefore, the output of this model is the sequence 0, 1, 2, 3, 4, 5, 5, 5, $\cdots$.

2.6 Final States

An FSM may have **final states**, which are states that, when entered, indicate the end of execution of the state machine.

Example 4: A variant of Example 3 is shown in Figure 8. The variant has the *isFinalState* parameter of the `final` state set to `true`. This is indicated by the double outline around the state. Upon entering that state, the `FSMActor` indicates to the enclosing director that it does not wish to execute any more (it does this by returning `false` from its `postfire()` method). As a result, the output sent to the `Display` actor is the finite sequence 0, 1, 2, 3, 4, 5, 5.

In the iteration in which an `FSMActor` enters a state that is marked final, the `postfire()` method of the `FSMActor` returns false. This indicates to the enclosing director that the `FSMActor` does not wish to be fired again. Most directors will simply avoid firing that actor again, but will continue executing the model. The SDF director, however, is different. Since it assumes regular consumption and production rates for all actors, and since it constructs its schedule statically, it cannot tolerate actors that refuse to fire. Hence, the SDF director will stop execution of the model altogether if *any* actor returns `false` from `postfire()`.

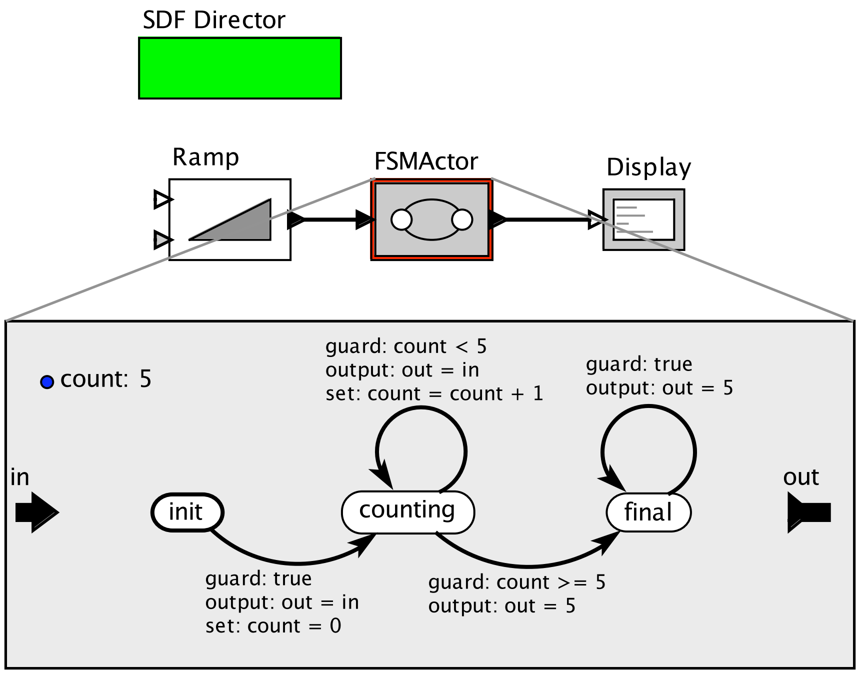


Figure 7: An extended FSM, where the *count* variable is part of the state of the system.

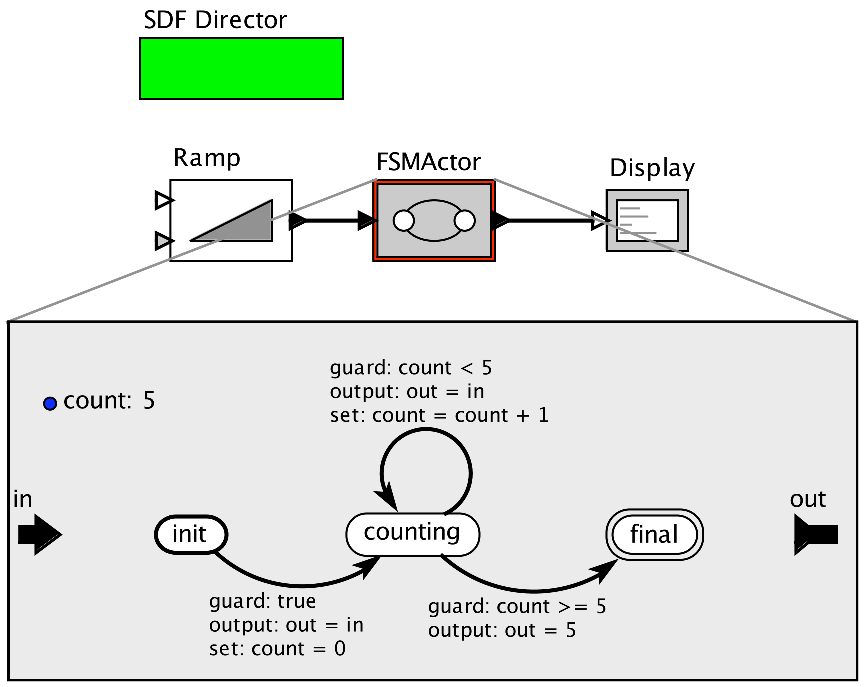


Figure 8: A state machine with a final state, which indicates the end of execution of the state machine.

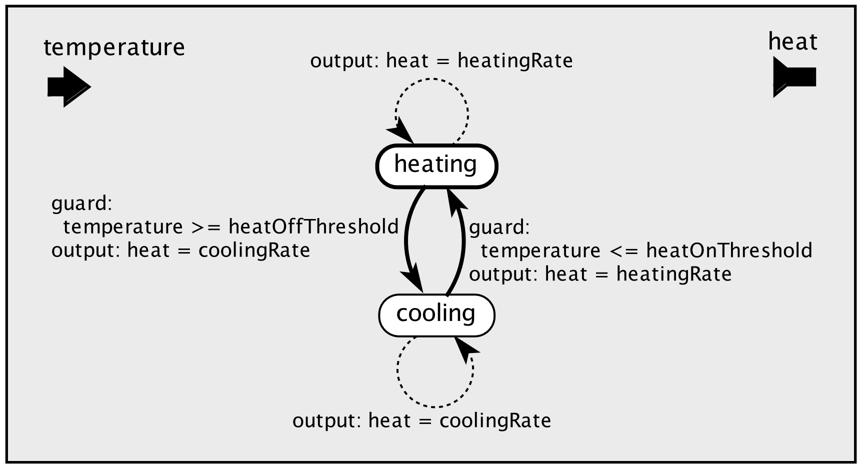

temperature
heat
output: heat = heatingRate
heating
guard:
temperature >= heatOffThreshold
output: heat = coolingRate
guard:
temperature <= heatOnThreshold
output: heat = heatingRate
cooling
output: heat = coolingRate

Figure 9: An FSM equivalent to the one in Figure 2, but using default transitions, indicated with dashed lines.

2.7 Default Transitions

An FSM may have **default transition**s, which are transitions that have the *default* parameter set to true (see Figure 6). These transitions become enabled if no other outgoing transitions of the current state are enabled. Default transitions are rendered as dashed arcs rather than solid arcs.

Example 5: The FSM of Figure 2 can be equivalently implemented using default transitions as shown in Figure 9. Here, the default transitions simply specify that the outgoing transition going to the other state is not enabled, then the FSM should remain in the same state and produce an output.

If a default transition also has a guard expression, then that transition is enabled only if the guard evaluates to true *and* there is no other non-default transition enabled. Note that using default transitions with timed models of computation can be somewhat subtle. See Section 3.7 below.

2.8 Nondeterministic State Machines

If more than one guard evaluates to true at any time, then the FSM is a **nondeterministic**. The transitions that have guards that simultaneously evaluate to true are called **nondeterministic transition**s. By default, transitions are not allowed to be nondeterministic, so if more than one guard evaluates to true, you will see an exception something like this:

```
Nondeterministic FSM error:  Multiple enabled transitions found but not
all of them are marked nondeterministic.
   in ...  name of a transition not so marked ...
```

To permit them to be nondeterministic, set the *nondeterministic* parameter to `true` on every transition that can be enabled while another another transition is enabled.

Example 6: A model of a faulty thermostat is shown in Figure 10. When the FSM is in the `heating` state, both outgoing transitions are enabled (their guards are both `true`), so either one can be taken. Both transitions are marked nondeterministic, a fact that is indicated visually by rendering the transitions in red. A plot of an execution is shown in Figure 11. Note that the heater stays on now for rather short periods of time, causing the temperature to hover around 18 degrees, the threshold at which the heater is turned on.

In a nondeterministic FSM, if more than one transition is enabled and they are all marked nondeterministic, then one is chosen at random in the `fire()` method of the `FSM` actor. If the fire() method is invoked more than once in an iteration (see Section 2.9 below), then subsequent invocations in the same iteration will always choose the same transition.

If more than one default transition leaves a state, then these transitions must also be marked nondeterministic or an exception will result. Nondeterministic default transitions are rendered as red dashed arcs.

2.9 Fixed-Point Iterations[1]

In Section 2.2 above, we explain that the execution of an `FSMActor` is divided into two segments, steps 1-4, which are performed in the `fire()` method, and steps 5-6, which are performed in the `postfire()` method. This separation is important in domains that perform a fixed-point iteration,

[1]This section may be safely skipped on a first reading unless you are particularly focusing on fixed-point domains such as SR and Continuous.

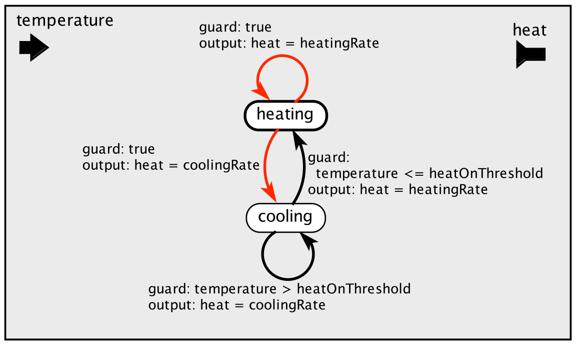

temperature
heat
guard: true
output: heat = heatingRate
heating
guard: true
output: heat = coolingRate
guard:
temperature <= heatOnThreshold
output: heat = heatingRate
cooling
guard: temperature > heatOnThreshold
output: heat = coolingRate

Figure 10: A model of a faulty thermostat that nondeterministically switches from heating to cooling.

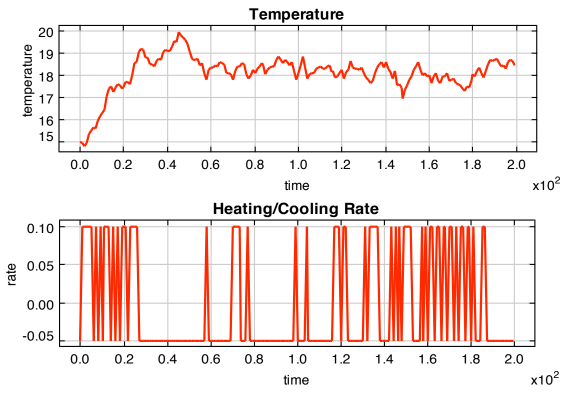


Figure 11: Plot of the thermostat FSM of Figure 10 replacing that in Figure 2.

such as SR and Continuous. In domains with a fixedpoint iteration, the `fire()` method may be invoked more than once in an iteration while the director searches for a solution. For such domains, it is important that the `fire()` method not include any persistent state changes. Steps 1-4 read inputs, evaluate guards, choose a transition, and produce outputs, but they do not commit to a state transition or change the value of any local variables.

The reason for this separation can be understood by studying the example in Figure 12. Execution of an SR model involves finding a value for each signal at each tick of a global clock. On the first tick, each of the `NonStrictDelay` actors puts the value shown in its icon on its output port (the values are 1 and 2, respectively). This defines the *in1* value for `FSMActor1` and the *in2* value for `FSMActor2`. But the other input ports remain undefined. The value of *in2* of `FSMActor1` is specified by `FSMActor2`, and the value of *in1* of `FSMActor2` is specified by `FSMActor1`. This seems to create a causality loop, but close examination of the state machines shows that there is no causality loop.

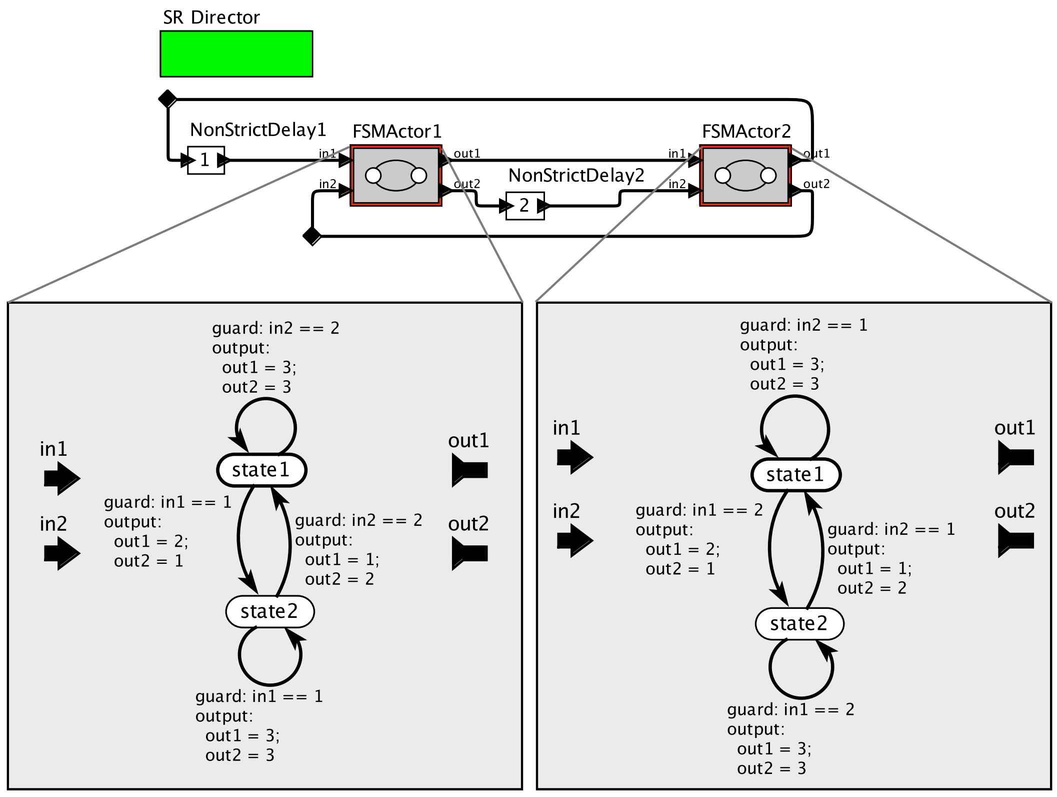


Figure 12: A model that requires separation of actions between the `fire()` method and the `postfire()` method in order to be able to converge to a fixed point.

In Figure 12, note for all states of the `FSMActor`s, each input port has a guard that depends on its value. Thus, it would seem that both inputs need to be known before any output value can be asserted, which again suggests a causality loop. However, looking closely at the left FSM, we see that the transition from `state1` to `state2` will be enabled at the first tick of the clock because *in1* has value 1, given by `NonStrictDelay1`. If the state machine is determinate, then this must be the only enabled transition. Since there are no nondeterministic transitions in the state machine, we can assume this will be the chosen transition. Once we make that assumption, we can assert both output values as shown on the transition (*out1* is 2 and *out2* is 1).

Once we assert those output values, then both inputs of `FSMActor2` become known, and it can fire. Its inputs are *in1* = 2 and *in2* = 2, so in the right state machine the transition from `state1` to `state2` is enabled. This transition asserts that *out2* of `FSMActor2` has value 1, so now both inputs to `FSMActor1` are known to have value 1. This reaffirms that `FSMActor1` has exactly one enabled transition, the one from `state1` to `state2`.

It is easy to verify that at each tick of the clock, both inputs of each state machine have the same value, so no state ever has more than one enabled outgoing transition. Determinism is preserved. Moreover, the values of these inputs alternate between 1 and 2 in subsequent ticks. For `FSMActor1`, the inputs are 1, 2, 1, $\cdots$ in ticks 1, 2, 3, $\cdots$. For `FSMActor2`, the inputs are 2, 1, 2, $\cdots$ in ticks 1, 2, 3, $\cdots$.

As explained in Section 2.2 above, in the `fire()` method, the actor

1. reads inputs;
2. evaluates guards on outgoing transitions of the current state;
3. chooses a transitions whose guard evaluates to true; and
4. executes the output actions on the chosen transition, if any.

In a fixed-point iteration, this may happen multiple times, and there are subtleties associate with each step. Specifically:

1. *reads inputs*: Some inputs may not be known. Unknown inputs cannot be read, so the actor simply doesn't read them.
2. *evaluates guards on outgoing transitions of the current state*: Some of these guards may depend on unknown inputs. These guards may or may not be able to be evaluated. For example, if the guard expression is "`true || in1`" then it can be evaluated whether the input *in1* is known or not. If a guard cannot be evaluated, then it is not evaluated.
3. *chooses a transition whose guard evaluates to true*: If exactly one transition has a guard that evaluates to true, then choose that transition. If a transition has been chosen already in a previous invocation of the `fire()` method in the same iteration, then the actor checks that the *same* transition is chosen this time. If not, it throws an exception and execution is halted. The FSM is not permitted to change its mind about which transition to take partway through an iteration. If more than one transition has a guard that evaluates to true, then the actor checks that every such transition is identified as a nondeterministic transition. If any such transition is not so marked, then the actor throws an exception. If all such transitions are marked nondeterministic, then

it chooses one of the transitions. Subsequent invocations of the `fire()` method in the same iteration will choose the same transition.

4. *executes the output actions on the chosen transition, if any*: If a transition is chosen, then the output values can all be defined. Some of these may be specified on transition itself. If they are not specified, then they are asserted to be `absent` at this tick. If all transitions are disabled (all guards evaluate to false), then all outputs are set to `absent`. If no transition is chosen but at least one transition remains whose guard cannot be evaluated, then the outputs remain unknown.

In the postfire() method, the actor

5. executes the set actions of the chosen transition; and
6. changes the current state to the destination of the chosen transition.

These actions are performed exactly once after the fixed-point iteration has determined all signal values. If any signal values remain undefined at the end of the iteration, then an exception is thrown. The model is defective.

Nondeterministic FSMs in a fixed-point domain have some subtleties. It is possible to construct a model for which there is a fixed point that has two enabled transitions but where the selection between transitions is not actually random. It could be that only one of the transitions is ever chosen. This occurs when there are multiple invocations of the `fire()` method in the fixed-point iteration, and in the first of these invocations, one of the guards cannot be evaluated because it has a dependence on an input that is not known. If the other guard can be evaluated in the first invocation of `fire()`, then the other transition will always be chosen. As a consequence, for nondeterministic state machines, the behavior may depend on the order of firings in a fixed-point iteration.

Note that default transitions may also be marked nondeterministic. However, a default transition will not be chosen unless all non-default transitions have guards that evaluate to false. In particular, it will not be chosen if any non-default transition has a guard that cannot yet be evaluated because of unknown inputs. If all non-default transitions have guards that evaluate to false and there is more than one default transition, all marked nondeterministic, then one is chosen at random.

3 Modal Models

Most interesting systems have behavior that changes over time. The changes in behavior may be triggered by user inputs, hardware failures, or sensor data, for example. A **modal model** is an explicit representation of a finite set of behaviors and the rules that govern transitions between behaviors. The switching between behaviors is governed by an FSM.

`ModalModel` is a hierarchical actor, like a composite actor, but with multiple refinements instead of just one. Each refinement gives one behavior. A state machine determines which refinement is active at any given time. The `ModalModel` actor is a more general form of the `FSMActor` described in the previous section. The `FSMActor` does not support state refinements. You can always use `ModalModel` instead of `FSMActor` and just not create state refinements. Hence, there is no real reason to use `FSMActor`.

Example 7: The model shown in Figure 13 has an actor labeled "Modal Model" that has two modes, `clean` and `noisy`. This models a communication channel with two modes of operation. In the `clean` mode, it passes inputs to the output unchanged. In the `noisy` mode, it adds a Gaussian random number to each input token. The top-level model provides an *event* signal that comes from the `PoissonClock` actor. That actor generates events at random times according to a Poisson process.[a] A sample execution of this model where the `Signal Source` actor provides a sine wave results in the plot shown in Figure 14.

[a] In a Poisson process, the time between events is given by independent and identically distributed random variables with an exponential distribution.

3.1 The Structure of Modal Models

The general pattern of a model model is shown in Figure 15. The behavior of a modal model is governed by a state machine, where each state is called a **mode**. In Figure 15, each mode is represented by a bubble, just like a state in a state machine, but filled in a light-blue color to suggest that it is a mode rather than an ordinary state. A mode, unlike an ordinary state, has a **mode refinement**, which is an opaque composite actor that defines the behavior when the mode is active. The example in Figure 13 shows two refinements, each of which is an SDF model that transforms input tokens to produce output tokens.

Note that it is essential that the refinement contain a director, and that the contained director be usable with the director that governs the execution of the modal model actor. The example in Figure 13 has an SDF director inside each of the modes and a DE director outside the modal model. SDF can generally be used inside DE, so this combination is valid.

Just like states in an ordinary state machine, modes are connected by arcs representing **transition**s. Each transition has a **guard**, which is a predicate (a boolean-valued expression) that specifies when the transition should be taken.

Example 8: In Figure 13, the transitions are guarded by the expression `event_isPresent`, which evaluates to true when the *event* input port has an event. Since that input port is connected to the `PoissonClock` actor, the transitions will be taken at random times, with an exponential random variable giving the time between transitions.

A variant of the pattern in Figure 15 is shown in Figure 16, where two modes share the same refinement. This is useful when the behavior in different modes differs only by parameter values. To construct a model where multiple modes have the same refinement, add a refinement to one of the states, giving the refinement a name (by default, the suggested name for the refinement is the same as the name of the state, but the user can choose any name for the refinement). Then, for

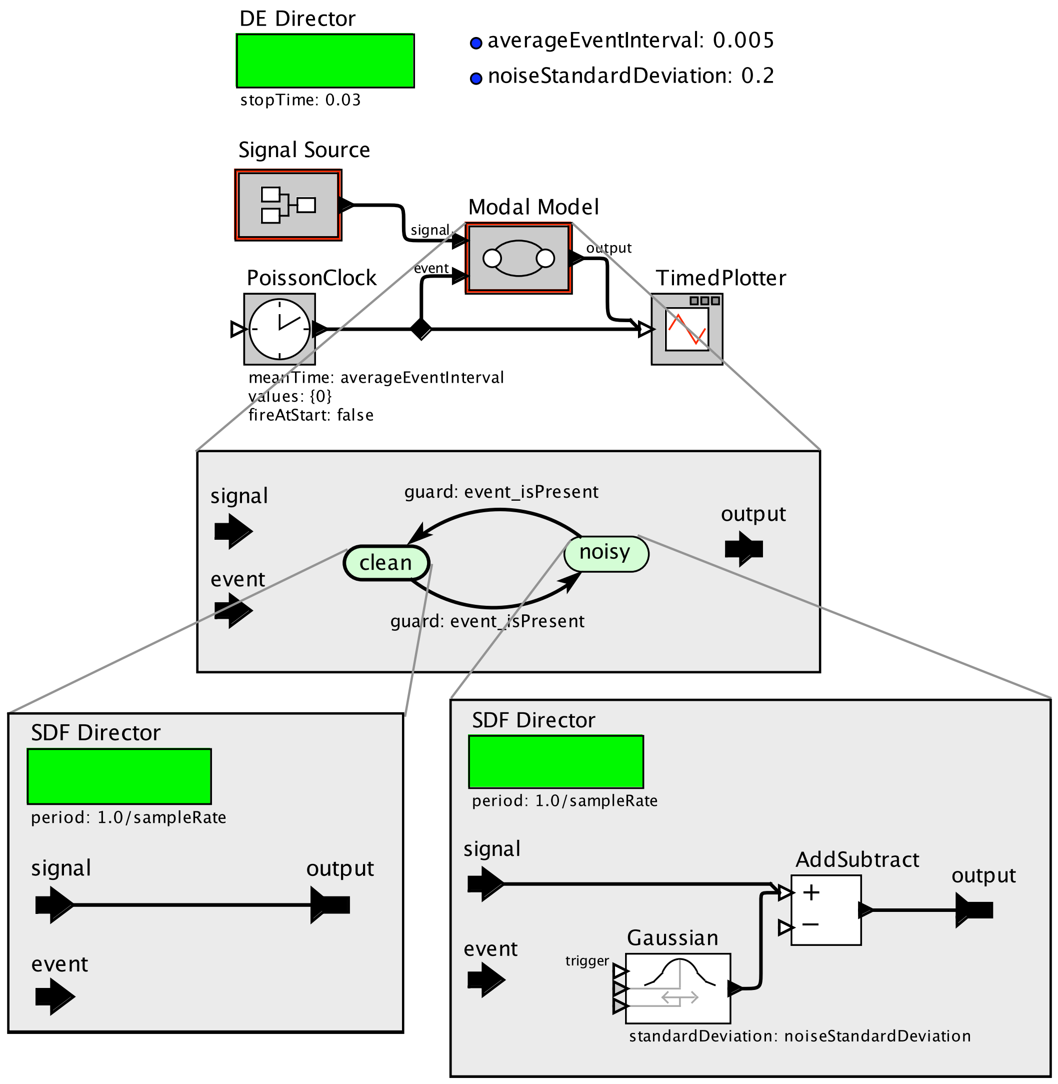


Figure 13: Simple modal model that has a normal (clean) operating mode, in which it passes inputs to the output unchanged, and a faulty mode, in which it adds Gaussian noise. It switches between these modes at random times determined by the PoissonClock actor.

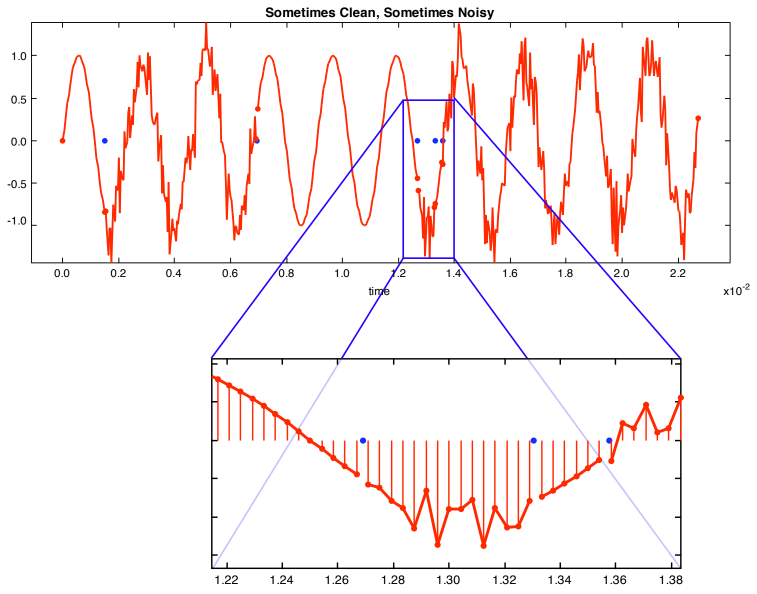


Figure 14: Plot generated by the model in figure 13.

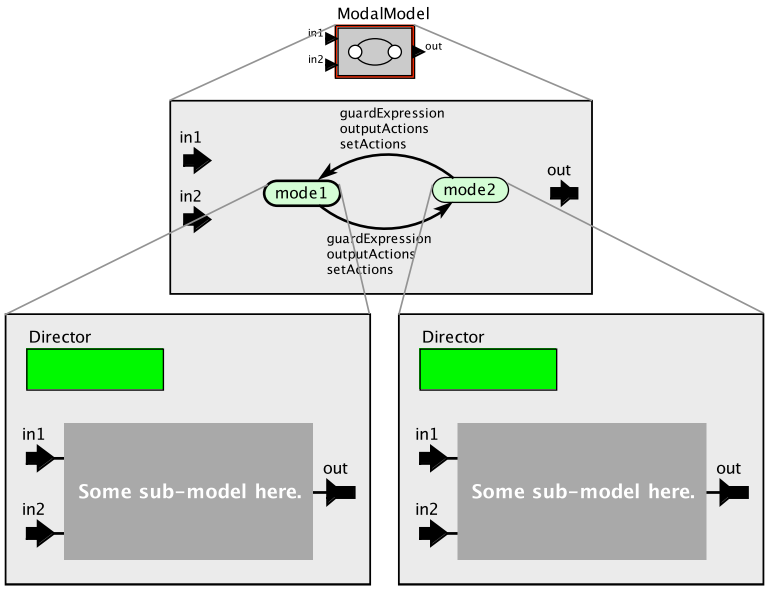


Figure 15: General pattern of a modal model with two modes, each with its own refinement.

another state, instead of choosing [Add Refinement], choose [Configure] (or simply double click on the state) and specify the refinement name as the value for the *refinementName* parameter. Both modes will now have the same refinement.

Another variant of the pattern is where a mode has multiple refinements. This can be accomplished by executing [Add Refinement] multiple times or by specifying a comma-separated list of refinement names for the *refinementName* parameter. These refinements will execute in the order that they are added.

3.2 Hierarchical FSMs

A particularly useful form of modal model is a **hierarchical FSM**. This is a modal model where the state refinements are themselves state machines.

Example 9: A hierarchical FSM that combines the normal and faulty thermostats of Examples 2 and 6 is shown in Figure 17. In this model, a `Bernoulli` actor is used to generate a *fault* signal (which will be `true` with probability 0.01). When the *fault* signal is `true`, the modal model will transition to the faulty state and remain there for 10 iterations before returning the the `normal` mode. The state refinements are the same as those in Figures 9 and 10, giving normal and faulty behavior of the thermostat.

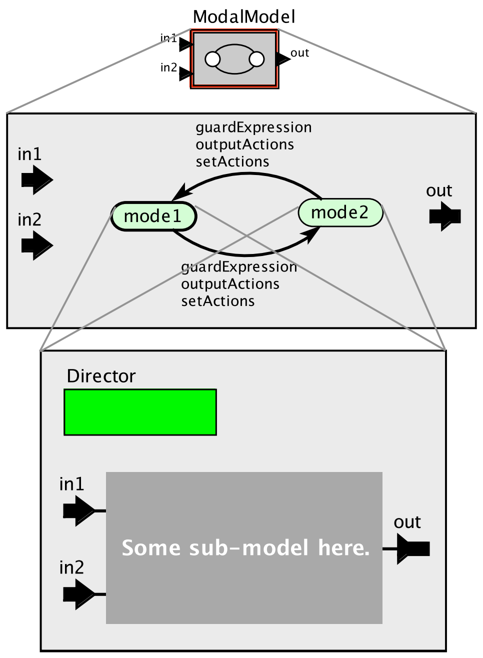


Figure 16: Variant of the pattern in Figure 15 where two modes share the same refinement.

To create a hierarchical FSM, when you select [Add Refinement] in the context menu of a state in your state machine, instead of choosing [Default Refinement], you should choose [State Machine Refinement]. The inside state machine can reference the input ports and write to the output ports, and its states can themselves have refinements (either Default Refinements or State Machine Refinements).

Notice that the model in Figure 17 combines a **stochastic state machine** with a nondeterministic FSM. The stochastic state machine has random behavior, but an explicit probability model is provided in the form of the Bernoulli actor. The nondeterministic FSM also has random behavior,

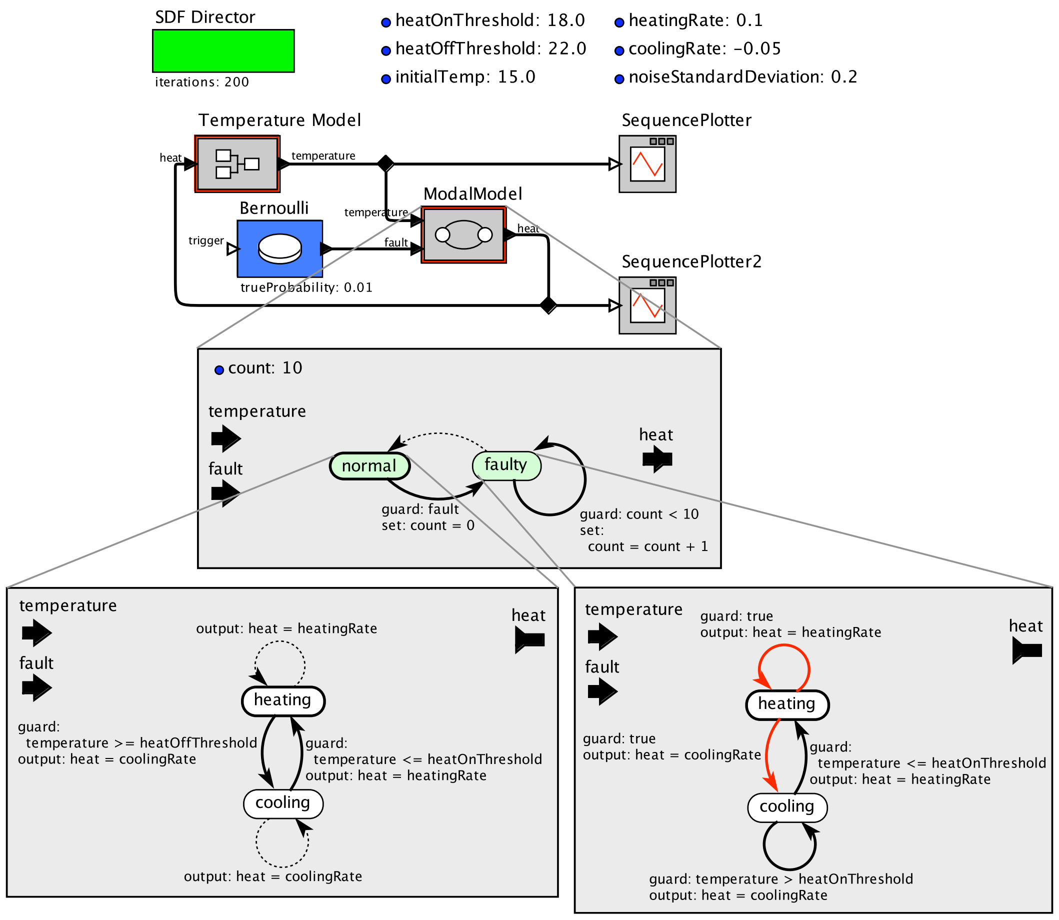


Figure 17: A hierarchical FSM that combines the normal and faulty thermostats of Examples 2 and 6.

but no probability model is provided.

3.3 Preemptive Transitions

A **preemptive transition** is a transition that may prevent the execution of the current state refinement. A transition is preemptive if its *preemptive* parameter evaluates to true. The syntax in Vergil

Further Reading: Concurrent Composition of State Machines

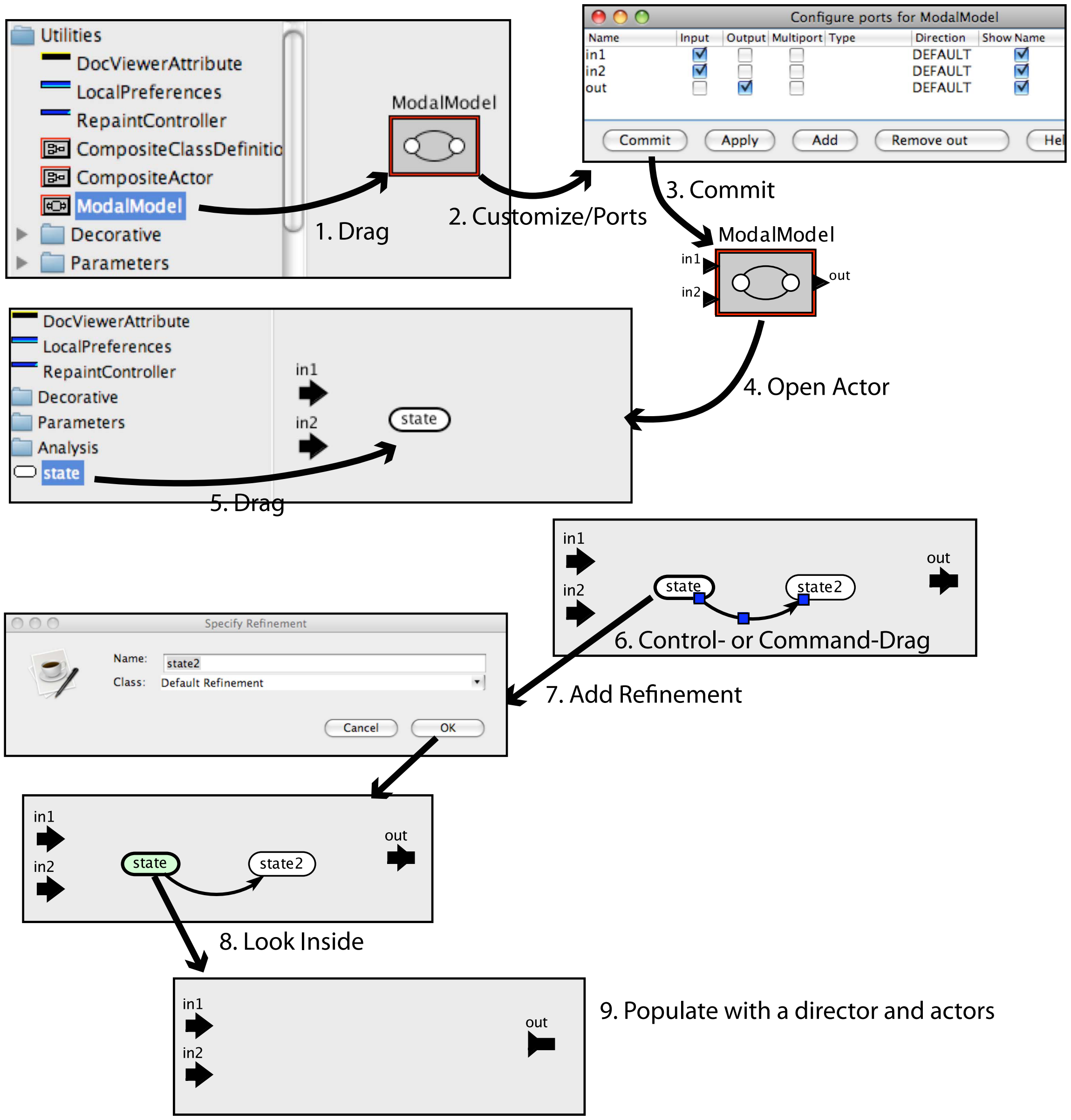
State machines have a long and distinguished history in the theory of computation (Hopcroft and Ullman, 1979). Concurrent composition of state machines is a more recent area of study, and continues to experience considerable flux. An early model for such concurrent composition is **Statecharts**, due to David Harel (Harel, 1987). With Statecharts, Harel introduced the notion of **and states**, where a state machine can be in both states *A* and *B* at the same time. On careful examination, the Statecharts model is concurrent composition of hierarchical FSMs under an SR model of computation. Statecharts are therefore (roughly) equivalent to modal models combining hierarchical FSMs and the SR director in Ptolemy II (Eker et al., 2003). Statecharts were realized in a software tool called **Statemate** (Harel et al., 1990).

Harel's work triggered a flurry of activity, resulting in many variants of the model (Beeck, 1994). Harel's visual syntax was subsequently adopted to become part of **UML**, the **unified modeling language** (Booch et al., 1998). A particularly elegant version is **SynchCharts** (André, 1996), which provides a visual syntax to the Esterel synchronous language (Berry and Gonthier, 1992).

One of the key properties of synchronous composition of state machines is that it becomes possible to model a composition of components as itself a state machine. A straightforward mechanism for doing this results in a state machine whose state space is the cross product of the individual ones. More sophisticated mechanisms have been developed, such as interface automata (de Alfaro and Henzinger, 2001).

Hybrid systems can also be viewed as modal models, where the concurrency model is a continuous time model (Maler et al., 1992; Henzinger, 2000; Lynch et al., 1996). In the usual formulation, hybrid systems couple FSMs with ordinary differential equations (ODEs), where each state of the FSMs is associated with a particular configuration of ODEs. A variety of software tools have been developed for specifying, simulating, and analyzing hybrid systems (Carloni et al., 2006).

Girault et al. (1999) showed that, in fact, FSMs can be combined hierarchically with a rich variety of concurrent models of computation. They called such compositions ***charts** or **starCharts**, where the star represents a wildcard. Several active research projects continue to explore explore expressive variants of concurrent state machines. **BIP** (Basu et al., 2006), for example, composes state machines using rendezvous interactions. Alur et al. (1999) give a very nice study of semantic questions around concurrent FSMs, including various complexity questions. Prochnow and von Hanxleden (2007) describe sophisticated techniques for visual editing of concurrent state machines.



Name	Input	Output	Multiport	Type	Direction	Show Name
in1	✓	☐	☐		DEFAULT	✓
in2	✓	☐	☐		DEFAULT	✓
out	☐	✓	☐		DEFAULT	✓

Name: state2
Class: Default Refinement

Figure 18: How to create modal models.

Mechanics: Creating Modal Models

The creation of a modal model is illustrated in Figure 18. The process starts by dragging in a `ModalModel` from the `Utilities` library and populating it with ports. Then, open the modal model and populate it with one or more states and transitions. To create the transitions, hold the Control key (or the Command key on a Mac) and click and drag from one state to the other. To add a refinement, right click on a state and select [`Add Refinement`]. You can choose a [`Default Refinement`] or a [`State Machine Refinement`]. The former will require a director and actors that process input data to produce outputs. The latter will enable creation of a hierarchical FSM.

Probing Further: Internal Structure of a Modal Model

In Ptolemy II, every object (actor, state, transition, port, parameter, etc.) can have at most one container. In a modal model, two states can share the same refinement. This seems to violate the principle of "at most one container."

The implementation in Ptolemy II, however, does not violate this principle. A `ModalModel` actor is actually a specialized composite actor that contains an instance `FSMDirector`, an `FSMActor`, and any number of composite actors. Each composite actor is a candidate to be a refinement for any state of the `FSMActor`. The `FSMActor` is the controller, in that it determines which mode is active at any time. The `FSMDirector` ensures that input data is delivered to the `FSMActor` and all active modes.

The Vergil user interface, however, hides this structure. When you execute an [`Open Actor`] command on a `ModalModel`, the user interface actually skips a level of the hierarchy and takes you directly the `FSMActor` controller. It does not show the layer of the hierarchy that contains the `FSMActor`, the `FSMDirector`, and the refinements. Moreover, when you [`Look Inside`] a state, the user interface goes up one level of the hierarchy and opens *all* refinements of the selected state. This somewhat intricate architecture balances expressiveness with user convenience.

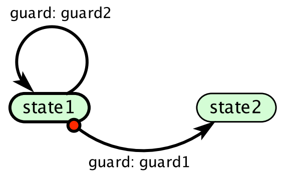


Figure 19: A preemptive transition, if chosen, results in the refinement not being executed. It is indicated by the red circle at the start of the transition.

is as shown in Figure 19, which shows a red circle at the origin of the transition. In that figure, if the current state is `state1` and the expression *guard1* evaluates to `true`, then the refinement of `state1` will not be executed. Moreover, if *guard1* evaluates to true, then *guard2* will not be evaluated at all.

3.4 Reset Transitions

A reset transition, if chosen, results in the refinement of the *destination* state being reset to its initial condition. It is indicated by the open arrowhead at the end of the transition, as shown in Figure 20. Specifically, after the transition has been taken, the `initialize()` method of the refinements of the destination state will be called. If, for example, the destination state has an FSM as its refinement, then the state of that refinement FSM will be set to the initial state.

A reset transition may be used to restart an FSM after it has reached a final state, as illustrated in the following example.

Example 10: We can use a final state to temporarily stop execution of a submodel, and a reset transition to restart it. In Figure 21, the `ModalModel` has only a single state and a reset transition that is taken whenever the input is a multiple of 10. When this transition is taken, the refinement will be initialized. The refinement is an extended FSM that counts 5 inputs, copying them unchanged to the output each time. When 5 inputs have arrived, the refinement transitions to the final state and stops executing. Subsequent iterations will produce no output (the output of the `ModalModel` will be absent). When the refinement is re-initialized, then it will begin again, copying the next five inputs to the output, and then stopping again. For this model, assuming the `Ramp` actor produces the sequence 1, 2, 3, $\cdots$, then the output will be 1, 2, 3, 4, 5, followed by five absents, followed by 11, 12, 13, 14, 15, followed by another five absents, etc. The SR director is used here to make the iterations and absent values explicit.

This use of final states is sometimes referred to in the literature as **normal termination**. The submodel stops executing when it enters a final state and can be restarted by a reset.

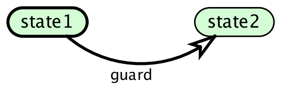


Figure 20: A reset transition, if chosen, results in the refinement of the destination state being reset to its initial condition. It is indicated by the open arrowhead at the end of the transition.

3.5 Transition Refinements

A transition may also have refinements. To create a **transition refinement**, right click on the transition and select [`Add Refinement`]. The syntax is shown in Figure 22. Specifically, a transition with a refinement is shown with a bolder line than a transition without a refinement. The transition refinement fires when the transition is chosen and postfires when the transition is committed. When it fires, it can produce outputs.

Interestingly, a transition refinement can also take as inputs the outputs of the state refinement from which the transition originated. The way this works is that an output port (named, say *out*, as in the figure) has a sister port with "_in" appended to the name (*out_in* in the figure). That sister port provides whatever data the state refinements produced on that output port prior to the transition being chosen.

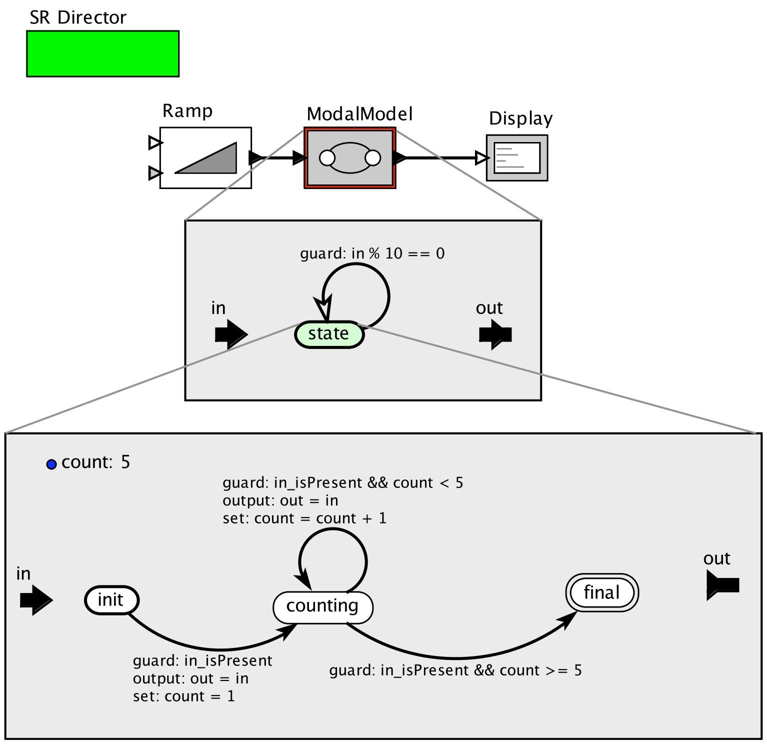


Figure 21: A reset transition may be used to restart an FSM after it has reached a final state.

3.6 Execution Policy for Modal Models

Execution of a `ModalModel` is similar to the execution of an `FSMActor` as described in Section 2.2. In outline, in the `fire()` method, the `ModalModel` actor

1. reads inputs;
2. evaluates the guards of preemptive transitions out of the current state;
3. if no preemptive transition is enabled, the actor
   1. fires the refinements of the current state (if any); and
   2. evaluates guards on non-preemptive transitions out of the current state;
3. chooses a transition whose guard evaluates to true, giving preference to preemptive transitions;
4. executes the output actions of the chosen transition; and
5. fires the transition refinements of the chosen transition.

In `postfire()`, the `ModalModel` actor

1. postfires the refinements of the current state if they were fired;
2. executes the set actions of the chosen transition;

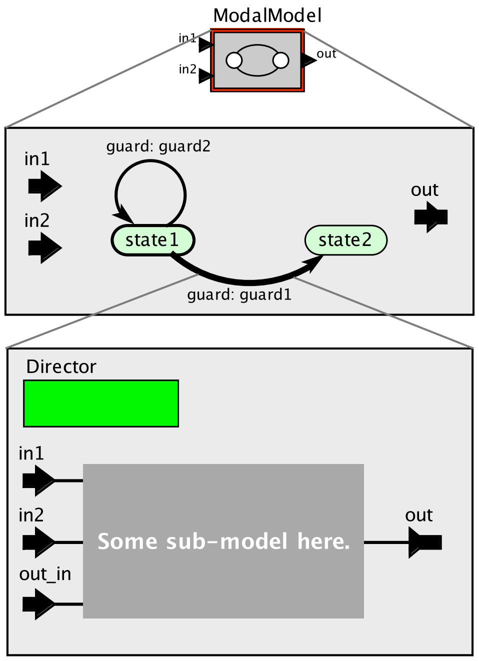


Figure 22: A transition refinement is a sub-model that fires when the transition is chosen and postfires when the transition is actually committed in the `postfire()` method of the container.

3. postfires the transition refinements of the chosen transition;
4. changes the current state to the destination of the chosen transition; and
5. initializes the refinements of the destination state if the transition is a reset transition.

The `ModalModel` actor makes no persistent state changes in its `fire()` method, so as long as the same is true of the refinement directors and actors, a modal model may be used in any domain. How it behaves in each domain, however, can be somewhat subtle, particularly with domains that have fixed-point semantics and when nondeterministic transitions are used. The behavior is exactly as described above for `FSMActor`, with the only exception being that state and transition refinements are fired and postfired at appropriate times in the execution. One subtlety is that if a preemptive transition is enabled, then the guards of non-preemptive transitions will not be evaluated. Thus, nondeterminism never results from guards of a preemptive and a non-preemptive transition both becoming true in an iteration.

Note that state refinements are fired before any non-preemptive guard is evaluated. One consequence of this is that the outputs resulting from firing the refinement can be referenced in the guards of non-preemptive transitions! Thus, whether a transition is taken can depend on how the current refinement reacts to the inputs. In fact, the astute reader may have already noticed in the figures here that the when the controller `FSMActor` of a `ModalModel` is shown, that the icon for output ports does not look like a normal output port (see for example the *heat* port in the middle diagram of Figure 17). This icon, in fact, is the icon for a port that is both an input and an output. In fact, it serves both of these roles for the modal model controller, since it can influence the evaluation of guards and it can provide outputs to the environment.

In a firing, it is possible that the current state refinement produces an output, and a transition that is taken also produces an output on the same port. In this case, only the second of these outputs will appear on the output of the `ModalModel`. However, the first of these output values, the one produced by the refinement, may affect whether the transition is taken. That is, it can affect the guard. If in addition a transition refinement writes to the output, then that value will be produced, overwriting the value produced either by the state refinement or the output action on the transition.

3.7 Time and Modal Models

Many Ptolemy II directors implement a timed model of computation. The `ModalModel` and `FSMActor` are themselves untimed, but they have certain features to support their use in timed domains.

The FSMs we have described so far are **reactive**, meaning that they only produce outputs in reaction to inputs. In a timed domain, the inputs have time stamps. For a reactive FSM, the time stamps of the outputs are the same as the time stamps of the inputs. Thus, the FSM appears to be reacting in zero time. It is instantaneous, from the perspective of the timed domain.

However, in a timed domain, it is also possible to define spontaneous FSMs. A **spontaneous FSM** or **spontaneous modal model** is one that produces outputs even when inputs are absent.

Example 11: The model in Figure 23 switches between two modes every 2.5 time units. In the `regular` mode it generates a regularly-spaced clock signal with period 1.0 (and with value 1, the default output value for `DiscreteClock`). In the `irregular` mode, it generates randomly spaced events using a `PoissonClock` actor with a mean time between events set to 1.0 and value set to 2. The result of a typical run is plotted in Figure 24, with a shaded background showing the times over which it is in the two modes. The output events from the `ModalModel` are spontaneous in that they are not necessarily produced in reaction to input events.

This example illustrates a number of subtle points about time. Examining the plot in Figure 24, we see that an event with value 1 and another with value 2 is produced at time 0. Why? The initial state is `regular`, and the execution policy described in section 3.6 explains that the refinement of that initial state is fired before guards are evaluated. That firing produces the first output of the

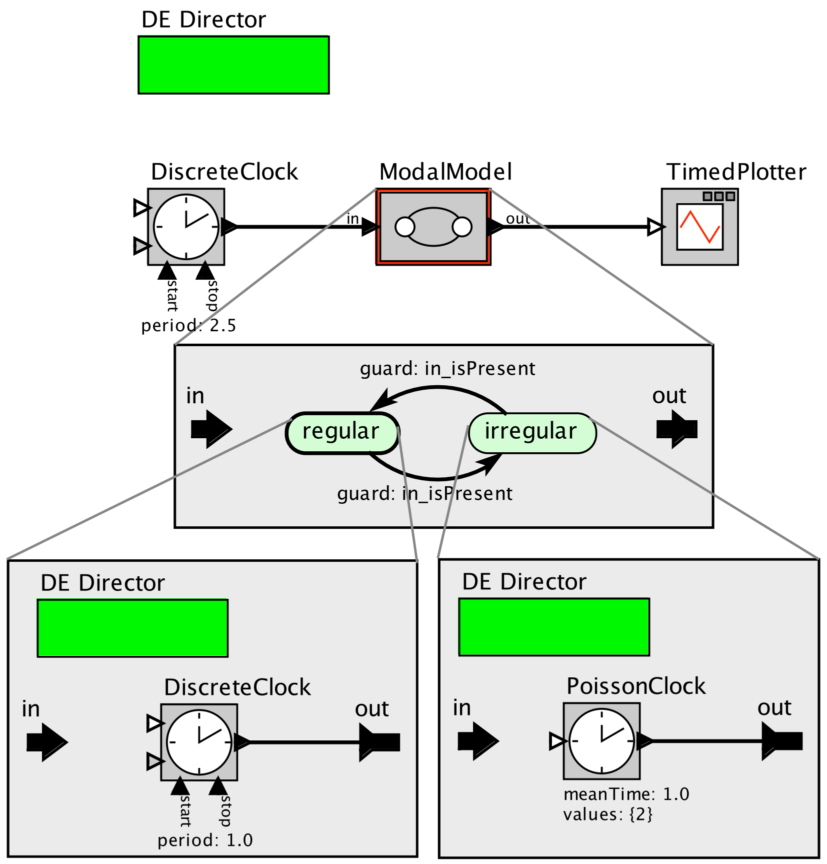


Figure 23: A spontaneous modal model, which produces output events that are not triggered by input events.

DiscreteClock. If we had instead used a preemptive transition, as shown in Figure 25, then that first output event would not appear.

The second event in Figure 24 (with value 2) at time zero is produced because the PoissonClock, by default, produces an event at the time the execution starts, time zero. This event is produced in the second iteration of the ModalModel, after entering the irregular state. Although the event has the same time stamp as the first event (both occur at time zero), they have a well-defined ordering. The event with value 1 appears before the event with value 2. In Ptolemy II, the value of

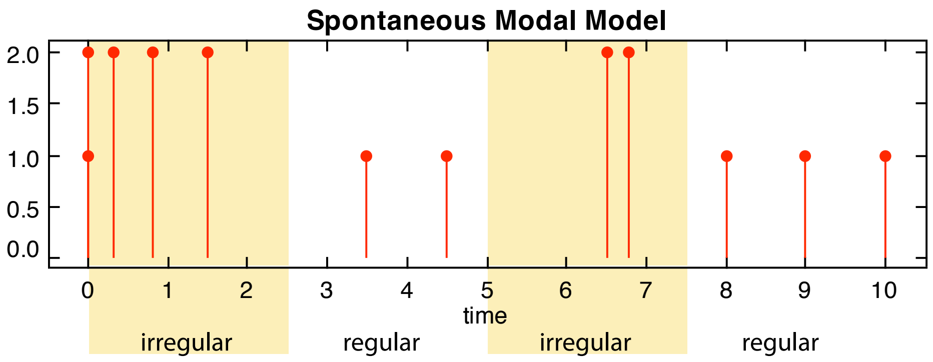


Figure 24: A plot of the output from one run of the model in Figure 23.

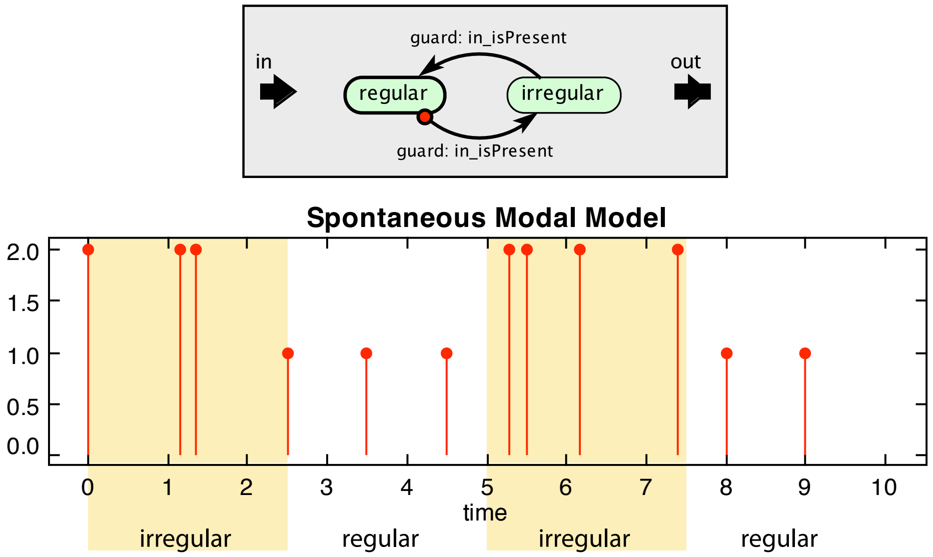


Figure 25: A variant of Figure 23 where a preemptive transition prevents the initial firing of the DiscreteClock.

time is actually represented by a pair of numbers called a tag, $(t,n) \in \mathbb{R} \times \mathbb{N}$, rather than a single number. The first of these numbers, t, is called the time stamp. It approximates a real number (it is a quantized real number with a specified precision). We interpret the time stamp t to represent the number of seconds (or any other time unit) since the start of execution of the model. The second of these numbers, n, is called the time index or microstep, and it represents a sequence number for events that occur at the same time stamp. In our example, the first event (with value 1) has tag $(0,0)$, and the second event (with value 2) has tag $(0,1)$. If we had set the *fireAtStart* parameter of the `PoissonClock` actor to `false`, then the second event would not occur.

Notice further that the `DiscreteClock` actor in the `regular` mode refinement has period 1.0, but produces events at times 0.0, 3.5, and 4.5, 8.0, 9.0, etc.. These are not multiples of 1.0 from the start time of the execution. Why?

The modal begins in the `regular` mode, but spends zero time there. It immediately transitions to the `irregular` mode. Hence, at time 0.0, the `regular` mode becomes inactive. While it is inactive, its local notion of time does not advance. It becomes active again at global time 2.5, but its local notion of time is still 0.0. Therefore, it has to wait one more time unit, until time 3.5, to produce the next output.

This notion of local time is important to understanding timed modal models. Very simply, local time stands still while a mode is inactive. Actors that refer to time, such as `TimedPlotter` and `CurrentTime`, have a parameter *useLocalTime*, which defaults to `false`. Thus, by default, a plotter will always plot events on the global time line. If no actor accesses global time, however, then a mode refinement will be completely unaware that it was ever suspended. It does not appear as if time has elapsed.

Another interesting property of the output of this model is that no event is produced at time 5.0, when the `irregular` mode becomes active again. This follows from the same principle. The `irregular` mode became inactive at time 2.5, and hence, from time 2.5 to 5.0, its local notion of time has not advanced. When it becomes active again at time 5.0, it resumes waiting for the right time (local time) to produce the next output from the `PoissonClock` actor.[2]

If an event is desired at time 5.0, then a reset transition can be used, as shown in Figure 26. The `initialize()` method of the `PoissonClock` causes an output event to be produced *at the time of the initialization*.

3.8 Time Delays in Modal Models

Time delays interact with modal models in interesting ways, as illustrated by the next example.

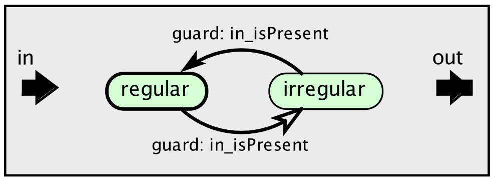
[2]Interestingly, because of the memoryless property of a Poisson process, the time to the next event after becoming active is statistically identical to the time between events of the Poisson process. But this fact has little to do with the semantics of modal models.

Example 12: Figure 27 shows a model that produces a counting sequence of events spaced one time unit apart, and then alternately delays the events by one time unit and does not delay them. In the `delay` mode, a `TimeDelay` actor imposes a time delay of one time unit. In the `noDelay` mode, the input is sent directly to the output without delay. The result of executing this model is shown in Figure 28. Notice that the value 0 emerges at time 2. Why?

The model begins in the `delay` mode, so it is that mode that receives the first input, which has value 0. However, the modal model transitions immediately out of that mode to the `noDelay` mode. So no time elapses. The `delay` mode becomes active again at time 1, but at that time, its local time has not advanced. Local time is still 0. Therefore, it still has to delay the input with value 0 by one time unit, It produces that output therefore at time 2, just before transitioning out again to the `noDelay` mode.

3.9 Time in Transition Refinements

A transition refinement can also make reference to current time, but it makes little sense to have transition refinements that are not reactive. Local time in a transition refinement will always be the same as local time in the environment in which the modal model containing the transition is executing. Hence, any access to local time via, for example, the `CurrentTime` actor will yield the time at which the transition is taken.



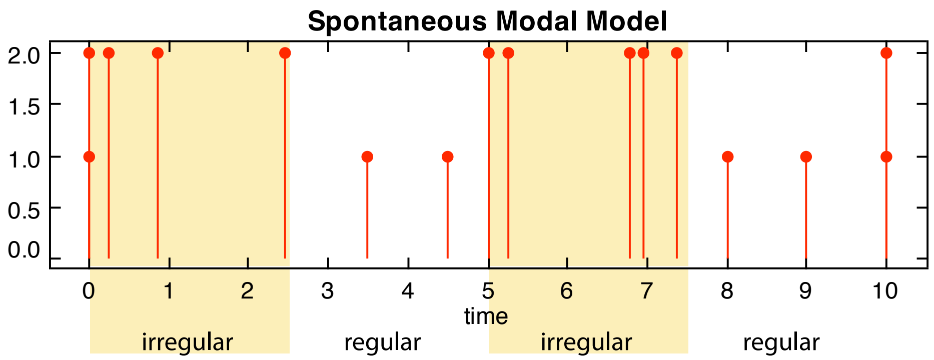


Figure 26: A variant of Figure 23 where a reset transition causes the PoissonClock to produce events when the `irregular` mode is reactivated.

Conceptually, a transition refinement is always active for exactly zero time. It takes no time to take a transition. As a consequence, it would not make much sense to include in a transition refinement spontaneous event generators like `DiscreteClock` or `PoissonClock`. Neither does it make much sense to include time delay actors.

3.10 Time and FSMs

It is possible also to create a simple FSM (not a modal model) that is spontaneous in a sense, though it will not be able to produce outputs at arbitrary times. It can produce outputs at initialization time and can produce outputs with the same time stamp but higher indexes than input events.

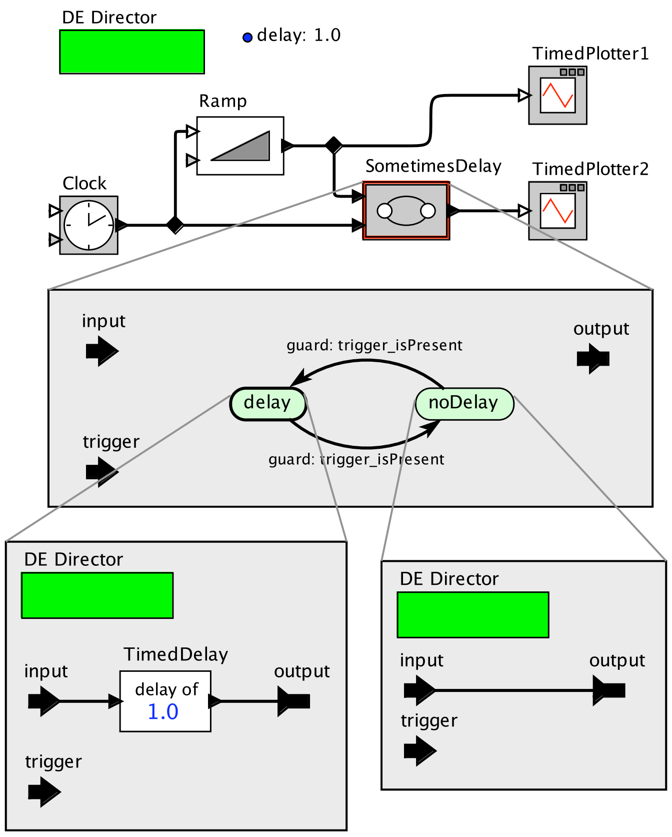


Figure 27: A modal model that switches between delaying the input by one time unit and not delaying it.

Example 13: The model shown in Figure 29 includes an `FSMActor` that is both reactive and spontaneous. It produces an output at time zero (with value 0) despite the fact that it receives no input at time zero (the *fireAtStart* parameter of the `PoissonClock` actor is `false`). This output is produced because the transition out of the `init` state has guard `true`, and hence is taken immediately on initialization of the state machine. That transition has an output action, `out = 0`.

After initialization, when the first input event arrives (near time 0.7, with value 1), the input enables the transition from the `wait` to the `duplicate` state. That transition copies the value of the input to the output and also records the value of the input in a local variable *recordedInput*. Once the state machine has entered the `duplicate` state, the transition back to `wait` is immediately enabled, so it is taken in the next microstep. Thus, the state machine produces a second output at the same time (around 0.7), as shown in the plot. The second output has twice the value of the first, making both outputs visible in the plot.

A state in which the state machine spends zero time is called a **transient state**. In the previous example, both the `init` and the `duplicate` states are transient. Note that any state that has default transitions (without guards or with guards that evaluate to true immediately) is a transient state, since exiting the state is always immediately enabled after entering the state.

3.11 Modal Model Principles

Modal models are clearly subtle and expressive, particularly for timed models. It is useful to step back and ponder the principles that govern the design choices in the Ptolemy II implementation

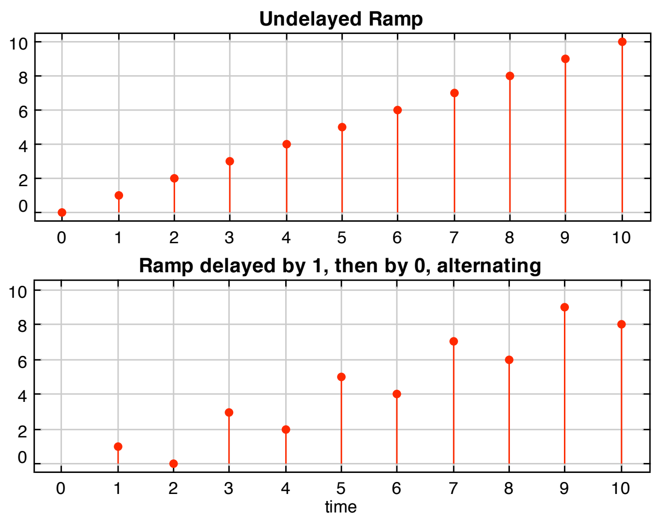


Figure 28: The result of executing the model in Figure 27.

of modal models. The key idea behind a mode is that it specifies a portion of the system that is active only part of the time. When it is inactive, does it cease to exist? Does time pass? Can its state evolve? These are not easy questions because the desired behavior depends very much on the application.

In Ptolemy II, the guiding principle is that when a mode is inactive, local time stands still, but global time passes. An inactive mode is therefore in a state of suspended animation. Local time within a mode will lag the time in its environment. The amount of lag begins at zero, and increases each time the mode becomes inactive. When an event crosses a hierarchical boundary into or out of the mode, its time stamp is adjusted by the amount of the lag. Thus, within the mode, time seems uninterrupted.

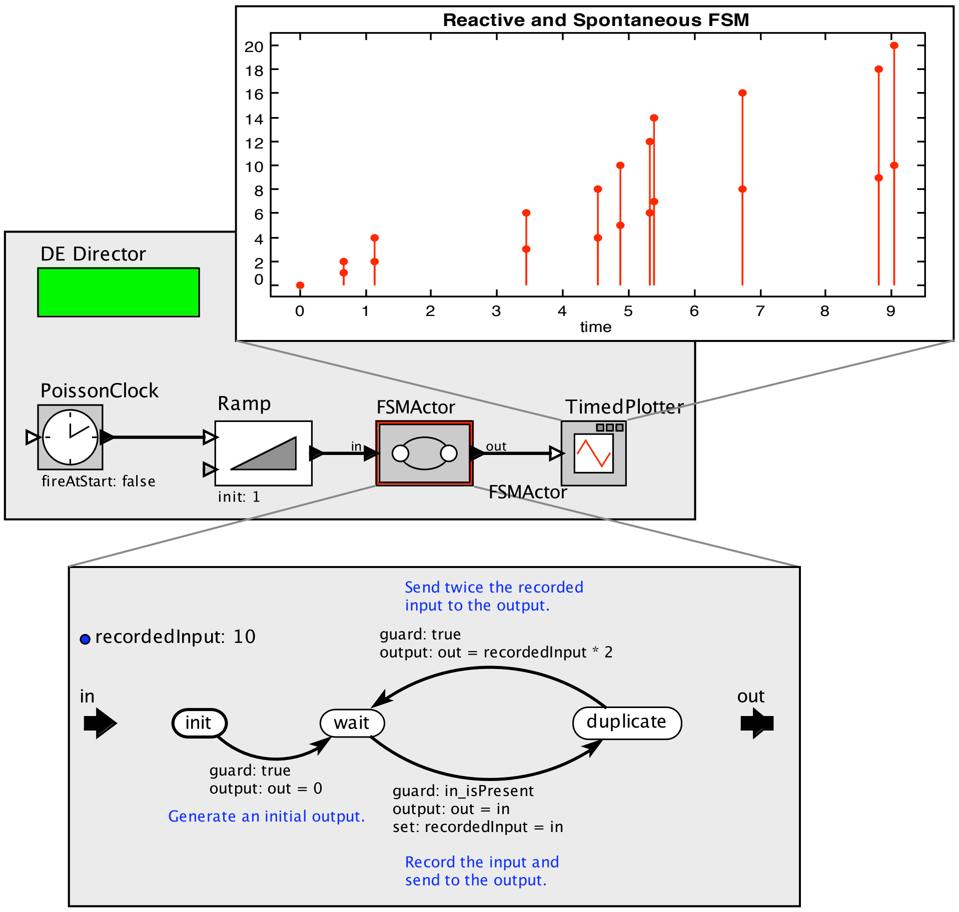


Figure 29: An FSM that spontaneously produces events at the start time and at the times of input events.

A key point is that being inactive is not necessarily the same as not getting inputs and not being observed. This point is illustrated in the model of Figure 30, which shows two instances of `DiscreteClock`, labeled `DiscreteClock1` and `DiscreteClock2`, which have the same parameter values. `DiscreteClock2` is inside a modal modal model labeled `ModalClock`, and `DiscreteClock1` is outside of any modal model. The output of `DiscreteClock1` is filtered by a modal model labeled `ModalFilter` that selectively passes the input to the output. The two modal models are controlled by the same `ControlClock`, which determines when they switch between the `active` and `inactive` states. Three plots are shown. The top plot is simply the output of `DiscreteClock1` unmodified in any way. The middle plot is the result of switching between observing and not observing the output of `DiscreteClock1`. The bottom plot is the result of activating and deactivating `DiscreteClock2`, which is otherwise identical to `DiscreteClock1`.

The `DiscreteClock` actors in this example are set to produce a sequence of values, 1, 2, 3, 4, cyclically. Consequently, in addition to being timed, these actors have state, since they need to recall which was the last output value in order to produce the next output value. When `DiscreteClock2` is inactive, its state does not change, and time does not advance. Thus, when it becomes active again, it simply resumes where it left off.

4 Conclusion

FSMs and modal models in Ptolemy II provide a very expressive way to build up complex model behaviors. As a consequence of this expressiveness, it takes some practice to learn to use them well. This report is intended to provide a reasonable starting point. Readers who wish to probe further are encouraged to examine the documentation for the Java classes that implement these mechanisms. Many of these are accessible when running Vergil by right clicking and selecting [`Documentation`]. Please send comments to eal@eecs.berkeley.edu.

Probing Further: Implementation of Transient States

When a transition is taken in an FSM, the `FSMActor` or `ModalModel` calls `fireAtCurrentTime()` on its enclosing director. This method requests a new firing in the next microstep regardless of whether any additional inputs become available. If the director respects this request (normally timed directors do), then the actor will be fired again at the current time, one microstep later. This ensures that if the destination state has a transition that is immediately enabled (in the next microstep), then that transition will be taken immediately. Note also that in a modal model, if the destination state has a refinement, then that refinement will be fired at the current time in the next microstep. This is particularly useful for continuous-time models, since the transition may represent a discontinuity in otherwise continuous signals. The discontinuity translates into two distinct events with the same time stamp.

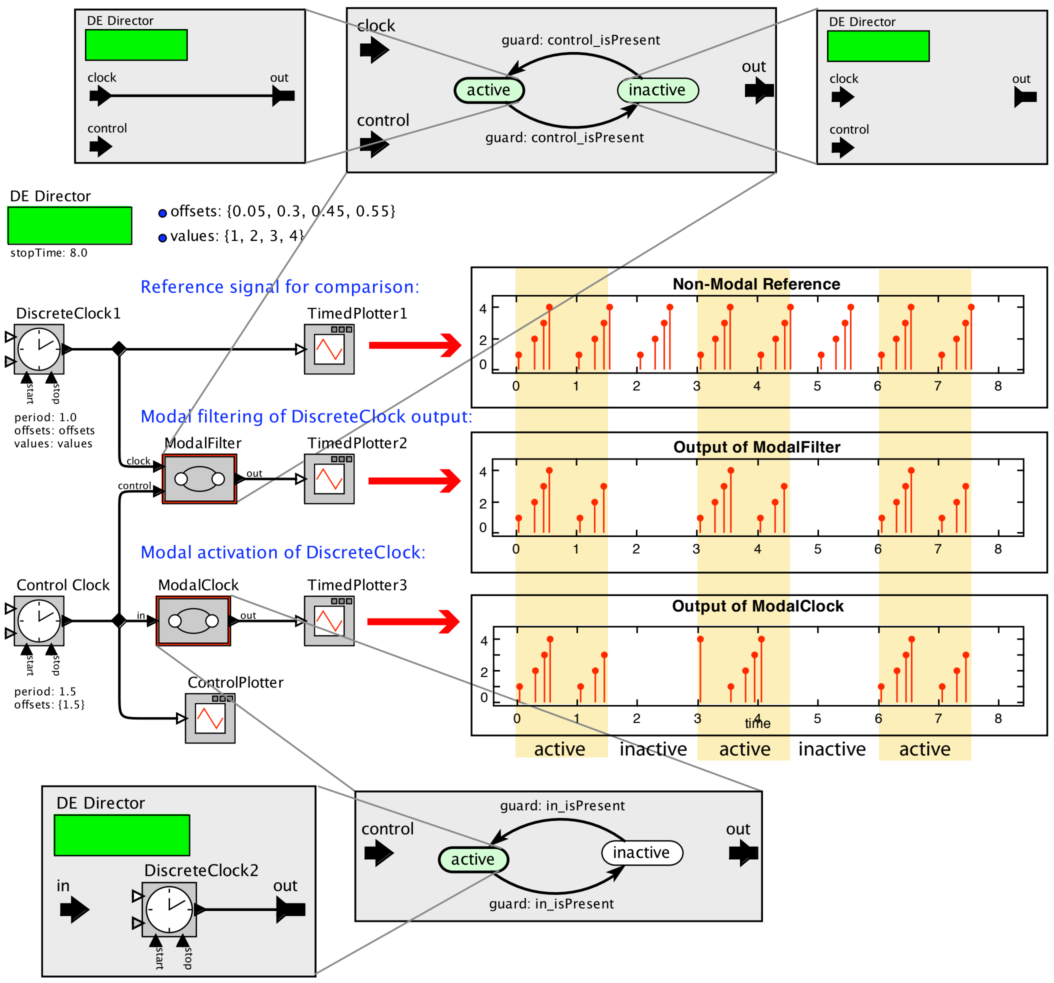


Figure 30: A model that illustrates that putting a stateful timed actor such as DiscreteClock inside a modal model is not the same switching between observing and not observing its output.

Acknowlegements

Several people in addition to the author contributed to the FSM and modal-model infrastructure in Ptolemy II. The modal domain was created primarily by Thomas Huining Feng, Xiaojun Liu, and Haiyang Zheng. The graphical editor in Vergil for state machines was created by Stephen Neuendorffer and Hideo John Reekie. Joern Janneck and Stavros Tripakis contributed to the semantics for timed state machines. Christopher Brooks created the online version of the models accessible by hyperlink from this document, and has also contributed enormously the Ptolemy II software infrastructure. Other major contributors include David Hermann, Jie Liu, and Ye Zhou.

References

Alur, R., S. Kannan, and M. Yannakakis, 1999: Communicating hierarchical state machines. In *26th International Colloquium on Automata, Languages, and Programming*, Springer, vol. LNCS 1644, pp. 169–178.

André, C., 1996: SyncCharts: A visual representation of reactive behaviors. Tech. Rep. RR 95–52, rev. RR (96–56), I3S.

Basu, A., M. Bozga, and J. Sifakis, 2006: Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, pp. 3–12.

Beeck, M. v. d., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytopil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152.

Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.

Carloni, L. P., R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, 2006: Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, **1(1/2)**.

de Alfaro, L. and T. Henzinger, 2001: Interface automata. In *ESEC/FSE 01: the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, **91(2)**, 127–144.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**.

Henzinger, T. A., 2000: The theory of hybrid automata. In Inan, M. and R. Kurshan, eds., *Verification of Digital and Hybrid Systems*, Springer-Verlag, vol. 170 of *NATO ASI Series F: Computer and Systems Sciences*, pp. 265–292.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Lynch, N., R. Segala, F. Vaandrager, and H. Weinberg, 1996: Hybrid I/O automata. In Alur, R., T. Henzinger, and E. Sontag, eds., *Hybrid Systems III*, Springer-Verlag, vol. LNCS 1066, pp. 496–510.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.

Prochnow, S. and R. von Hanxleden, 2007: Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ACM/IEEE, Nashville, TN, USA.