



This is a chapter from the book

## System Design, Modeling, and Simulation using Ptolemy II

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit:

<http://creativecommons.org/licenses/by-sa/3.0/>,

or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA. Permissions beyond the scope of this license may be available at:

<http://ptolemy.org/books/Systems>.

**First Edition, Version 1.0**

**Please cite this book as:**

Claudius Ptolemaeus, Editor,  
*System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, 2014.  
<http://ptolemy.org/books/Systems>.

# Finite State Machines

Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Christian Motika,  
Reinhard von Hanxleden, and Haiyang Zheng

## Contents

<b>6.1</b>	<b>Creating FSMs in Ptolemy</b>	<b>187</b>
<b>6.2</b>	<b>Structure and Execution of an FSM</b>	<b>192</b>
6.2.1	Defining Transition Guards	194
6.2.2	Output Actions	198
6.2.3	Set Actions and Extended Finite State Machines	199
	<i>Sidebar: Models of State Machines</i>	199
6.2.4	Final States	201
6.2.5	Default Transitions	204
6.2.6	Nondeterministic State Machines	205
6.2.7	Immediate Transitions	208
	<i>Probing Further: Weakly Transient States</i>	210
<b>6.3</b>	<b>Hierarchical FSMs</b>	<b>212</b>
6.3.1	State Refinements	213
6.3.2	Benefits of Hierarchical FSMs	215
6.3.3	Preemptive and History Transitions	217
6.3.4	Termination Transitions	218
6.3.5	Execution Pattern for Modal Models	220
	<i>Probing Further: Internal Structure of an FSM</i>	221
	<i>Probing Further: Hierarchical State Machines</i>	222
<b>6.4</b>	<b>Concurrent Composition of State Machines</b>	<b>223</b>
<b>6.5</b>	<b>Summary</b>	<b>226</b>
	<b>Exercises</b>	<b>228</b>

Finite state machines are used to model system behavior in many types of engineering and scientific applications. The **state** of a system is defined as its condition at a particular point in time; 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. It is represented by a state variable  $s \in \Sigma$ , where  $\Sigma$  is the set of all possible states for the system. A **finite state machine (FSM)** is a state machine where  $\Sigma$  is a finite set. In a finite state machine, a system's behavior is modeled as a set of states and the rules that govern transitions between them.

A number of Ptolemy II actors include state and behave as simple state machines. For example, the **Ramp** actor (which produces a counting sequence) has state, which is the current position in the sequence. This actor uses a local variable, called a **state variable**, to keep track of its current value. The Ramp actor's reaction to a *trigger* input depends on how many times it has previously fired, which is captured by the state variable. The number of possible states for a Ramp actor 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  $2^{64}$ . If the data type is `String`, then the number of possible states is infinite (and thus the Ramp cannot be described as a finite state machine).

Although the number of Ramp actor states is potentially very large, the logic for changing from one state to the next is simple, which makes it easy to characterize the behavior of the actor. In contrast, it is common to have actors that have a small number of possible states, but use relatively complex logic for moving from one state to the next. This chapter focuses on such actors.

This chapter discusses approaches for designing, visualizing, and analyzing finite state machines in Ptolemy II. In Chapter 8, we extend these approaches to construct **modal models**, in which the states themselves are Ptolemy II models.

## 6.1 Creating FSMs in Ptolemy

A Ptolemy II finite state machine is created in a similar manner to the previously described actor-oriented models, but it is built using states and transitions rather than actors and connections/relations. A **transition** represents the act of moving from one state to another; it can be triggered by a guard, which specifies the conditions under which the transition

is taken. It is also possible to specify output actions (actions that produce outputs when the transition is taken) and set actions (actions that set parameters when the transition is taken).

The main actor used to implement FSM models in Ptolemy II is **ModalModel**, found in the `Utilities` library.\* A ModalModel contains an FSM, which is a collection of states and transitions depicted using visual notation shown in Figure 6.1. In this figure, the ModalModel has two input and two output ports, though in general it could have any number of input and output ports. It has three states. One of these states is an **initial state** (labeled *initialState* in the figure), which is the state of the actor when the model begins execution. The initial state is indicated visually by a bold outline. Some of the states may also be **final states**, indicated visually with a double outline (more about final states later). The process for creating an FSM model in *Vergil* is shown in Figure 6.2.

To begin creating an FSM, drag the ModalModel into your model from the library. Populate the actor with input and output ports by right clicking (or control-clicking on a Mac) and selecting [Customize→Ports], clicking Add, and specifying port names and whether they are inputs or outputs. Then right click on the ModalModel and select *Open Actor*. The resulting window is shown in Figure 6.3. It is similar to other Vergil win-

---

\*You can also use **FSMActor**, found in `MoreLibraries→Automata`, which is simpler in that it does not support mode refinements, used in Section 6.3 and Chapter 8.

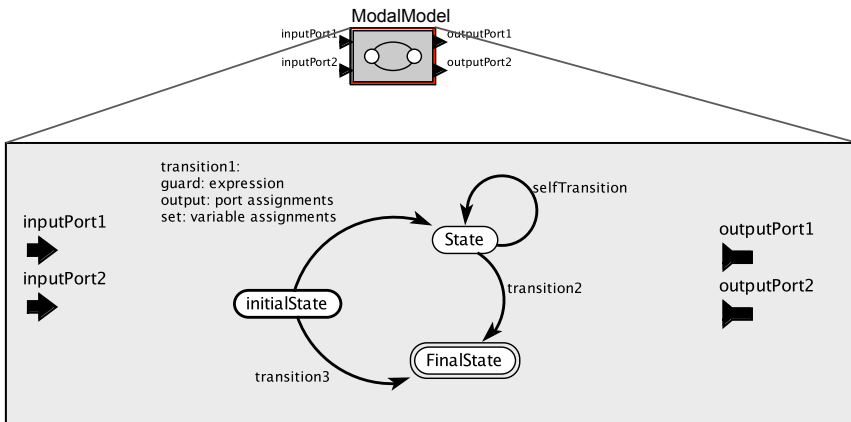


Figure 6.1: Visual notation for state machines in Ptolemy II.

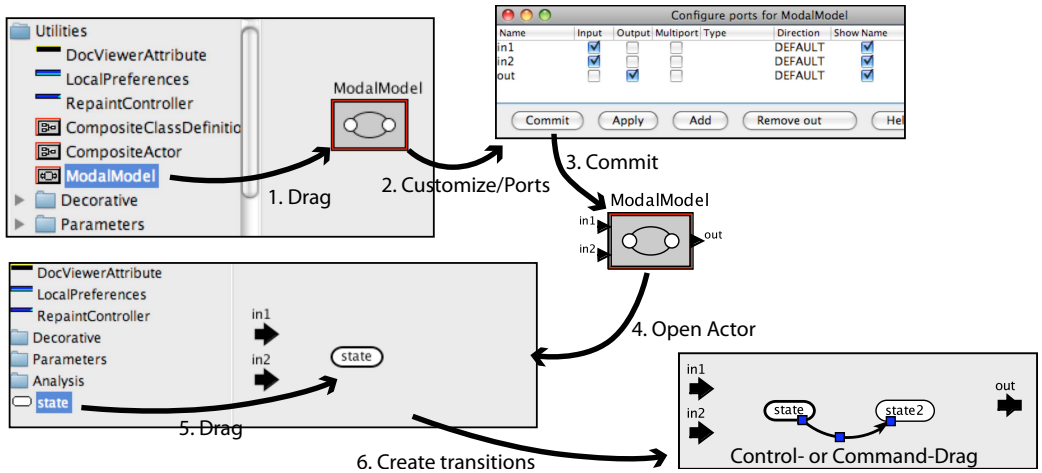


Figure 6.2: Creating FSMs in Vergil, using the `ModalModel` actor (a similar procedure applies to using the `FSMACTOR`).

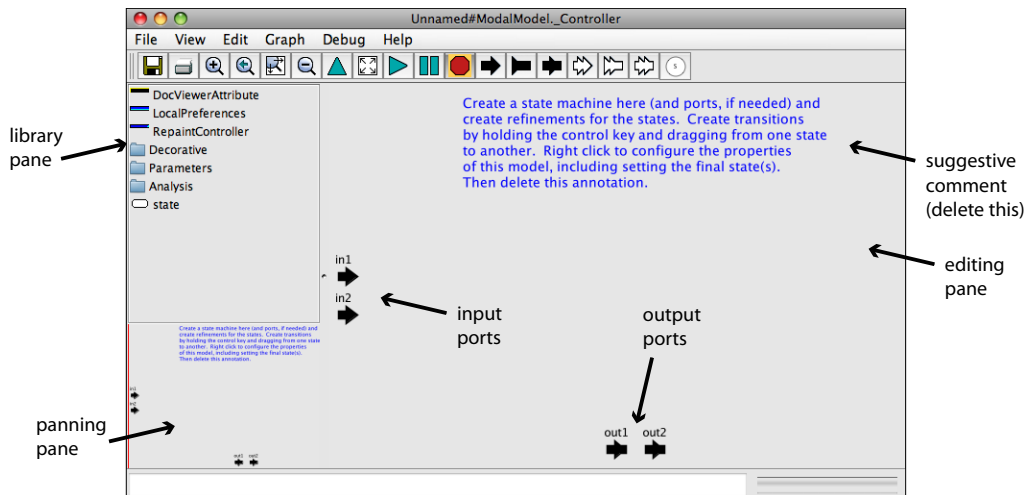


Figure 6.3: Editor for FSMs in Vergil, showing two input and two output ports, before being populated with an FSM.

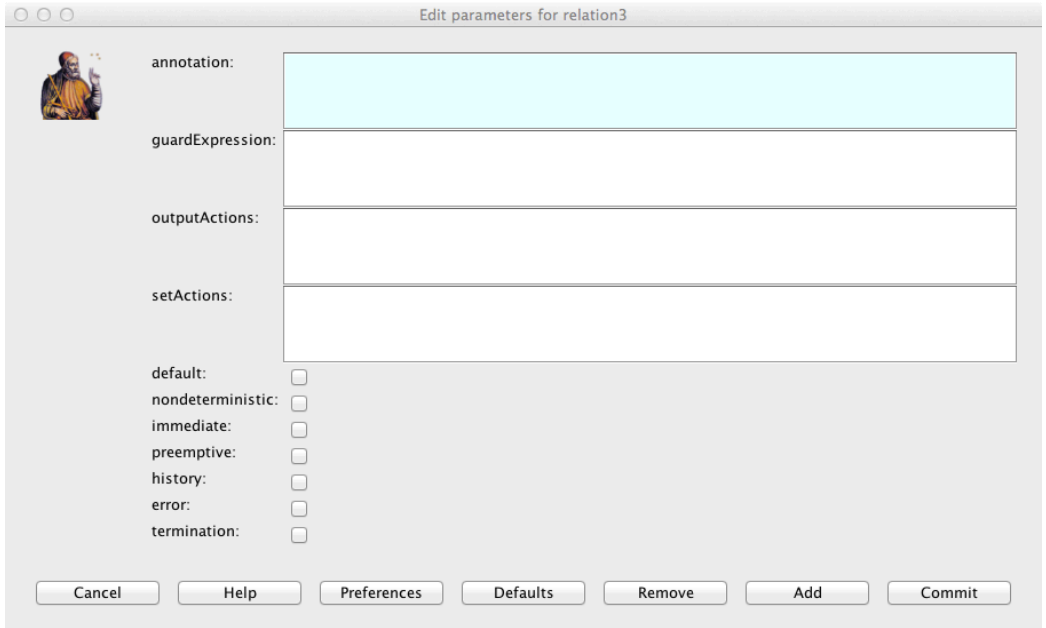


Figure 6.4: Dialog box for configuring a transition in an FSM.

dows, but has a customized library consisting primarily of a *State*, a library of parameters, and a library of decorative elements for annotating your design.

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 6.4. For readability, you can also specify an annotation associated with the transition.

The `ModalModel` implementing the finite state machine can be placed within a larger model, to be executed using another director. The choice of director will depend on the application. All directors are compatible with it.

We illustrate the use of this process with a simple FSM application example, described below.

**Example 6.1:** Consider a thermostat that controls a heater. The thermostat is modeled as a state machine with states  $\Sigma = \{\text{heating}, \text{cooling}\}$ . If the state  $s = \text{heating}$ , then the heater is on. If  $s = \text{cooling}$ , then the heater is off. Suppose the target temperature is 20 degrees Celsius. It would be undesirable for the heater to cycle on and off whenever the temperature is slightly above or below the target temperature; thus, the state machine should include hysteresis around the setpoint. If the heater is on, then the thermostat allows the temperature to rise slightly above the target, to an upper limit specified as 22 degrees. If the heater is off, then it allows the temperature to drop below the target to 18 degrees. Note that the behavior of the system at temperatures between 18 and 22 degrees depends not only on the input temperature but also on the state. This strategy avoids **chattering**, where the heater would turn on and off rapidly when the temperature is close to the target temperature.

This FSM is constructed as shown in 6.5. The FSM has a *temperature* input and a *heat* output; its output specifies the rate at which the air is being heated (or cooled). This system has two states,  $\Sigma = \{\text{heating}, \text{cooling}\}$ . There are four transitions, each of which has a guard that specifies the conditions under which the transition

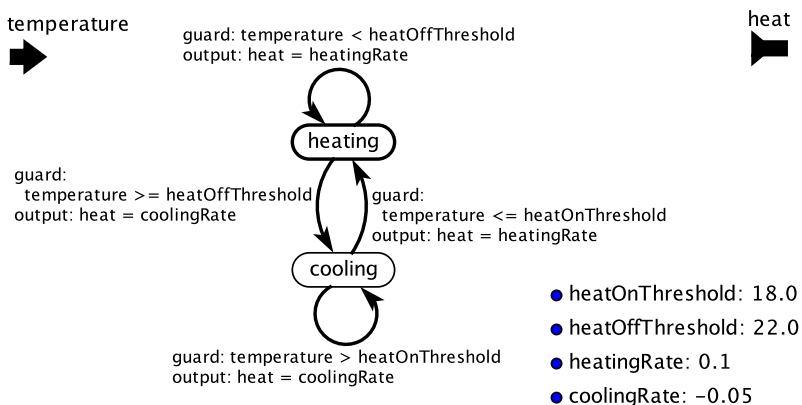


Figure 6.5: FSM model of a thermostat.

is taken. The transitions show the values produced on the output ports when the transition is taken. When the system is 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). Notice that the guards are mutually exclusive, in that in each state, it is not possible for guards on two of the outgoing transitions to evaluate to true. This makes the machine **deterministic**.

The FSM of Figure 6.5 is embedded in an SDF model as shown in Figure 6.6. The Temperature Model actor, whose definition is shown in Figure 6.7, models changes in the ambient temperature value based on the the output of the FSMActor. The system's output is plotted in Figure 6.8.

## 6.2 Structure and Execution of an FSM

As shown in the previous example, an FSM 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 executes a sequence of steps as follows. In the **fire** phase of execution, 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
4. executes the output actions on the chosen transition, if any.

In the **postfire** phase, the actor

5. executes the **set actions** of the chosen transition, which sets parameter values; and
6. changes the current state to the destination of the chosen transition.

Each of these steps is explained in more detail below. Table 6.1 summarizes the Ptolemy II notations for FSM transitions (without hierarchy), which follow those in Kieler (Fuhrmann and Hanxleden, 2010) and Klepto (Motika et al., 2010), that are explained in this chapter.









notation	description
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>An <b>ordinary transition</b>. Upon firing, if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM will choose the transition and produce the value <math>y</math> on output <math>x</math>. Upon transitioning, the actor will set the variable <math>a</math> to have value <math>b</math>.</p>
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>A <b>default transition</b>. Upon firing, if no other non-default transition is enabled and the guard <math>g</math> is <code>true</code>, then the FSM actor will choose this transition, produce outputs, and set variables in the same manner as above.</p>
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>A <b>nondeterministic transition</b>. This transition allows another nondeterministic transition to be enabled in the same iteration. One of the enabled transitions will be chosen nondeterministically.</p>
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>An <b>immediate transition</b>. If state <math>s1</math> is the <b>current state</b>, then this is like an ordinary transition. However, if state <math>s1</math> is the destination state of some transition that will be taken and the guard <math>g</math> is <code>true</code>, then the FSM will also immediately transition to <math>s2</math>. In this case, there will be two transitions in a single iteration. The output <math>x</math> will be set to value <math>y</math> upon firing, and the variable <math>a</math> will be set to <math>b</math> upon transitioning. If more than one transition in a chain of immediate transitions sets an output or variable, then the last transition will prevail.</p>
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>A <b>nondeterministic default transition</b>. A nondeterministic transition with the (lower) priority of a default transition.</p>
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>An <b>immediate default transition</b>. An immediate transition with the (lower) priority of a default transition, compared with other immediate transitions.</p>

Table 6.1: Summary of FSM transitions and their notations, which may be combined to indicate combinations of transition types. For example, a nondeterministic immediate default transition will be colored red, have the initial diamond, and be rendered with dotted lines.

### 6.2.1 Defining Transition Guards

Defining appropriate guards on state transitions is a critical part of creating a finite state machine. As we discuss below, however, the behavior of some guard expressions may cause unexpected results, depending on the director chosen to run the model. In particular, different directors handle absent input values in different ways, which can cause guard expressions to be evaluated in a manner that may seem counterintuitive.

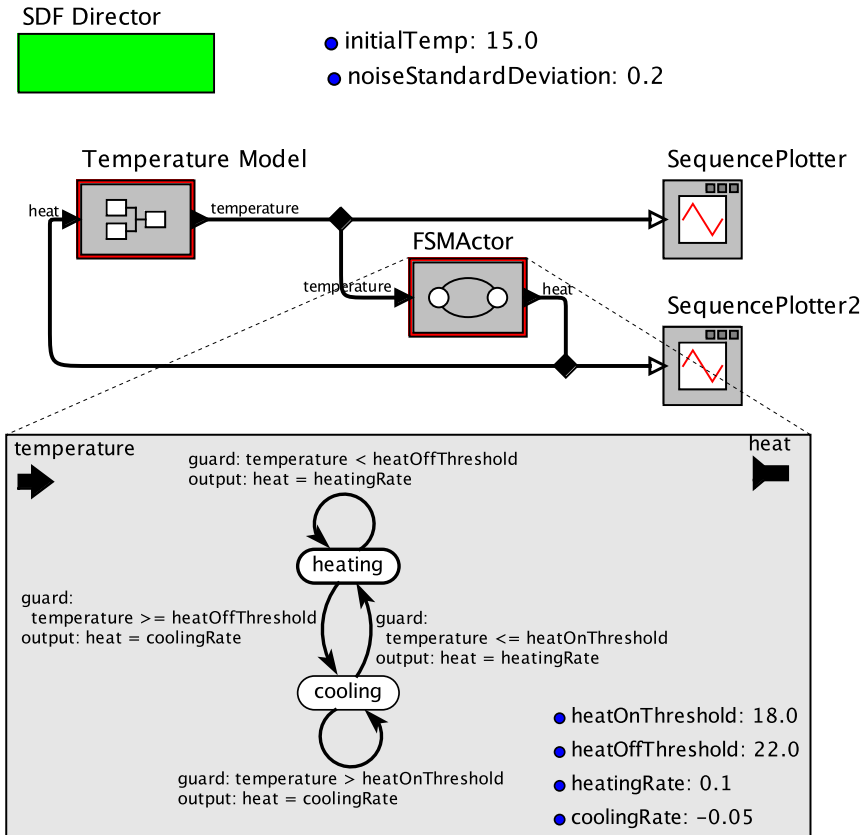


Figure 6.6: The FSM model of a thermostat of Figure 6.5 embedded in an SDF model. The Temperature Model actor is shown in Figure 6.7. [\[online\]](#)

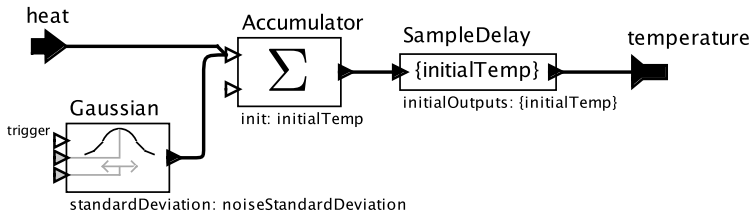


Figure 6.7: The Temperature Model composite actor of Figure 6.6.

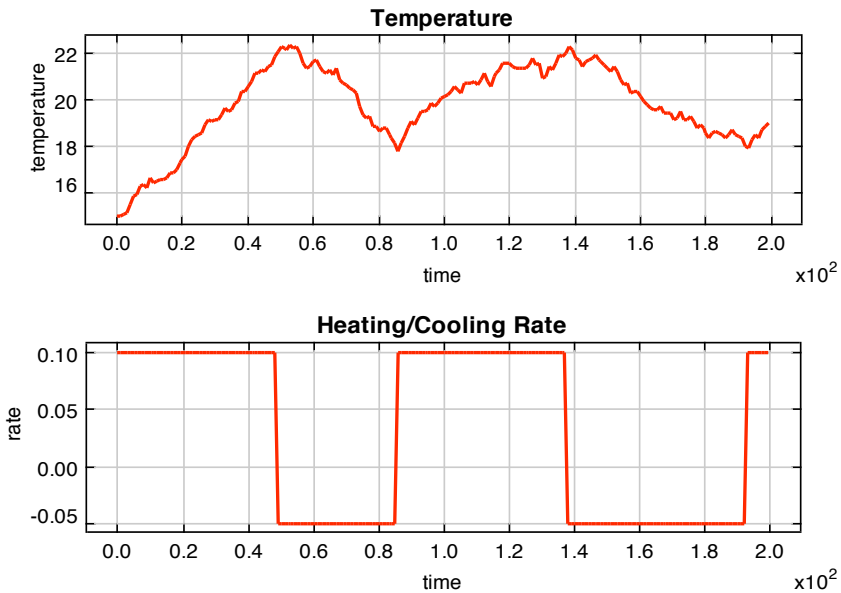


Figure 6.8: Two plots generated by Figure 6.6, showing the temperature (above) and the heating rate (below), which reflects whether the heater is on or off. Both are shown as a function of time.

Each transition has a **guard**, which is a predicate (a boolean-valued expression) that can depend on inputs to the state machine, parameters and variables, and outputs of **mode refinements** (which are explained in Chapter 8).

**Example 6.2:** In Figure 6.5, in the guard expression

```
temperature < heatOffThreshold,
```

the variable `temperature` refers to the current value in the port named *temperature*, and `heatOffThreshold` refers to the parameter named *heatOffThreshold*.

	guard	description
1		A blank guard always evaluates to true.
2	<code>p_isPresent</code>	True if there is a token at port <i>p</i> .
3	<code>p</code>	True if there is a token at port <i>p</i> and it has value <code>true</code> .
4	<code>!p</code>	True if there is a token at port <i>p</i> and it has value <code>false</code> .
5	<code>p &gt; 0</code>	True if there is a token at port <i>p</i> and it has value greater than zero.
6	<code>p &gt; a</code>	True if there is a token at port <i>p</i> and it has value greater than the value of the parameter <i>a</i> .
7	<code>a &gt; 0</code>	True if parameter <i>a</i> has value greater than 0.
8	<code>p &amp;&amp; q</code>	True if ports <i>p</i> and <i>q</i> both have tokens with value <code>true</code> .
9	<code>p    q</code>	True if port <i>p</i> is present and true or if <i>p</i> is present and false and <i>q</i> is present and true.
10	<code>p_0 &gt; p_1</code>	True if port <i>p</i> has a token on both channel 0 and channel 1 and the token on channel 0 is larger than the one on channel 1.
11	<code>p_1_isPresent &amp;&amp; (p_0    p_1)</code>	True if port <i>p</i> has a token on both channel 0 and channel 1 and one of the two tokens is <code>true</code> .
12	<code>timeout(t)</code>	True when time <i>t</i> has elapsed since entering the source state.

Table 6.2: Examples of guard expressions, where *p* and *q* are ports, and *a* is a parameter.

A few examples of valid guards are given in Table 6.2 for an FSM with input ports  $p$  and  $q$  and parameter  $a$ .

As shown in line 2 of the table, for any port  $p$ , the symbol `p_isPresent` may be used in guard expressions. This is a boolean that is true if an input token is present on port  $p$ . Conversely, the expression `!p_isPresent` evaluates to true when  $p$  is absent. Note that in domains where  $p$  is never absent, such as [PN](#), this expression will never evaluate to true.

If port  $p$  has no input tokens (it is *absent* on all channels), then *all* the guards in the table except number 1 are false. In particular, if  $p$  has type boolean, and it has no input tokens, then it is possible for both `p` and `!p` to be false. Similarly, it is possible for `p > 0`, `!(p > 0)`, and `p <= 0` to simultaneously evaluate to false. Of course, this can only happen if the FSM is used with a director that can fire it with absent inputs, such as [SR](#) and [DE](#).

Note that because of the way absent inputs are treated, guard 9 in the table has a particularly subtle effect. It cannot evaluate to true unless  $p$  has an input token, but it does not require that  $q$  have a token. If the intent is that both ports have a token for the transition to become enabled, then the guard should be written

```
q_isPresent && (p || q)
```

It would be clearer, though not strictly necessary, to write

```
(p_isPresent && q_isPresent) && (p || q)
```

In short, any mention of an input port  $p$  in a guard expression can cause the entire guard expression to evaluate to false if the port  $p$  is absent. But it may not evaluate to false if the subexpression involving  $p$  is not evaluated. In particular, the logical OR notated as `||` will not evaluate its right argument if the left argument is true. This is why  $q$  in guard 9 in the table is not required to be present for the guard to evaluate to true.

A consequence of this evaluation strategy is that erroneous guard expressions may not be detected. For example, if the guard expression is specified as `p.foo()`, but `foo()` is not a defined method on the data type of  $p$ , then this error will not be detected if  $p$  is known to be absent. The fact that  $p$  appears in the guard expression causes it to evaluate to false. Moreover, the expression `“true || p < 10”` always evaluates to true, whether  $p$  has a token or not.

For multiports with multiple channels, the guard expression can specify a channel using the symbol `p_i`, where  $i$  is an integer between 0 and  $n - 1$  and  $n$  is the number of channels connected to the port. For example, line 10 in Table 6.2 compares input tokens on two channels of the same input port. Similarly, a guard expression may refer to `p_i_isPresent`, as shown in line 11.

Line 12 shows a guard expression that can be used to trigger a transition after some time has elapsed. The expression `timeout(t)`, where  $t$  is a double, becomes true when the FSM has spent  $t$  time units in the source state. In domains with partial support for time, such as **SDF** and **SR**, the transition will be taken at the next firing time of the FSM greater than or equal to  $t$  (and hence, of course, will only be taken if the *period* parameter of the director is not zero); see Section 3.1.3. In domains with full support for time, such as **DE** and **Continuous**, covered in later chapters, the transition will be taken exactly  $t$  time units after entering the source state, unless some other transition becomes enabled sooner.

In all cases, the type of an input port or parameter must match the usage in an expression. For example, the expression `p || q` will trigger an exception if port  $p$  has type *int*.

## 6.2.2 Output Actions

Once a transition is chosen, its **output actions** are executed. The output action are specified by the *outputActions* parameter of the transition (see Figure 6.4). The format of an output action is typically *portName = expression*, where the expression may refer to input values (as in guard expressions) or to a parameter. For example, in Figure 6.5, the line

```
output: heat = coolingRate
```

specifies that the output port named *heat* should produce the value given by the parameter *coolingRate*.

As explained in the sidebar on page 199, the two classes of state machines are Mealy machines and Moore machines. The above-described behavior constitutes a Mealy machine; a Moore machine can be implemented using state refinements that produce outputs, as explained in Chapter 8.

Multiple output actions may be given by separating them with semicolons, as in `port1 = expression1; port2 = expression2`.

### 6.2.3 Set Actions and Extended Finite State Machines

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

#### Sidebar: Models of State Machines

State machines are often described in the literature as a five-tuple  $(\Sigma, I, O, T, \sigma)$ .  $\Sigma$  is the set of states, and  $\sigma$  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  $i: P_i \rightarrow D \cup \{absent\}$ , 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 *absent* represents “absent” inputs (i.e.,  $i(p) = absent$  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  $T: \Sigma \times I \rightarrow \Sigma \times O$ , 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 state machines](#), which require a richer model than that given above. Extended state machines add a set  $V$  of variable valuations, which are functions of the form  $v: N \rightarrow D$ , 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  $T: \Sigma \times I \times V \rightarrow \Sigma \times O \times V$  (for deterministic state machines). This function is encoded by the transitions, guards, output actions, and set of actions of the FSM.

**Example 6.3:** A simple example of an extended state machine is shown in Figure 6.9. In this example, the FSM has a parameter called *count*. 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 the state called *final*. The self transition is taken as long as *count* is less than 5. That transition increments the value of *count* by one in its set actions. In the firing after the value of *count* reaches 5, the transition to *final* is taken. At that firing, the output is set equal to 5. In subsequent firings, the output will always be 5, as specified by the self loop on the *final* state. This model, therefore, outputs the sequence 0, 1, 2, 3, 4, 5, 5,  $\dots$ .

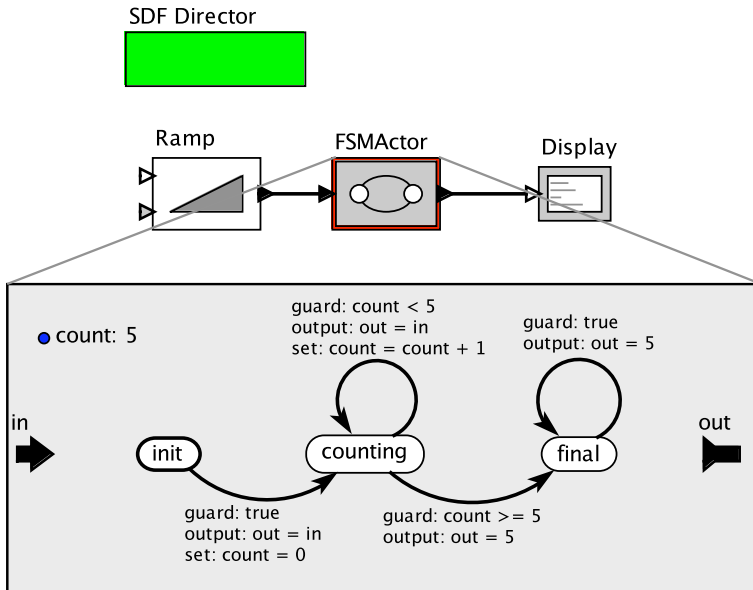


Figure 6.9: An extended state machine, where the *count* variable is part of the state of the system. [\[online\]](#)



### 6.2.4 Final States

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

**Example 6.4:** A variant of Example 6.3 is shown in Figure 6.10. This variant has the *isFinalState* parameter of the *final* state set to `true`, as indicated by the double outline around the state. Upon entering that state, the FSM 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. Notice the two 5's at the end. This underscores the fact that guards are evaluated *before* set actions are executed. Thus, at the start of the sixth firing, the input to the FSM is 5 and the value of *count* is 4. The self-loop on the *counting* state will be taken, producing output 5. At the start of the next firing, *count* is 5, so the transition to the *final* state is taken, producing another 5.

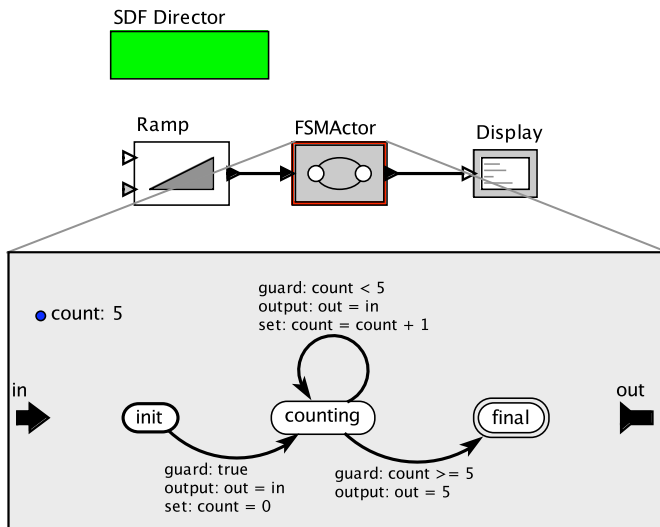


Figure 6.10: A state machine with a final state, which indicates the end of execution of the state machine. [\[online\]](#)

In the iteration in which an FSM enters a state that is marked final, the `postfire` method of the `ModalModel` or `FSMActor` returns false. This indicates to the enclosing director that the FSM does not wish to be fired again. Most directors will simply avoid firing the FSM again, but will continue executing the rest of the model. The **SDF** director, however, is different. Since it assumes the same consumption and production rates for all actors, and since it constructs its schedule statically, it cannot accommodate non-firing actors. Hence, the SDF director will stop execution of the model altogether if *any* actor returns false from `postfire`. In contrast, the **SR** director will continue executing, but all outputs of the now terminated FSM will be absent in subsequent ticks.

**Example 6.5:** Figure 6.11 shows an **SR** model that produces a finite count, but unlike Example 6.4, the model does not stop executing when the state machine reaches its final state. The display output is shown for 10 iterations. Notice that after the FSM reaches the final state, the output of the FSM is *absent*. Notice also that the first output of the FSM is *absent*. This is because the transition from

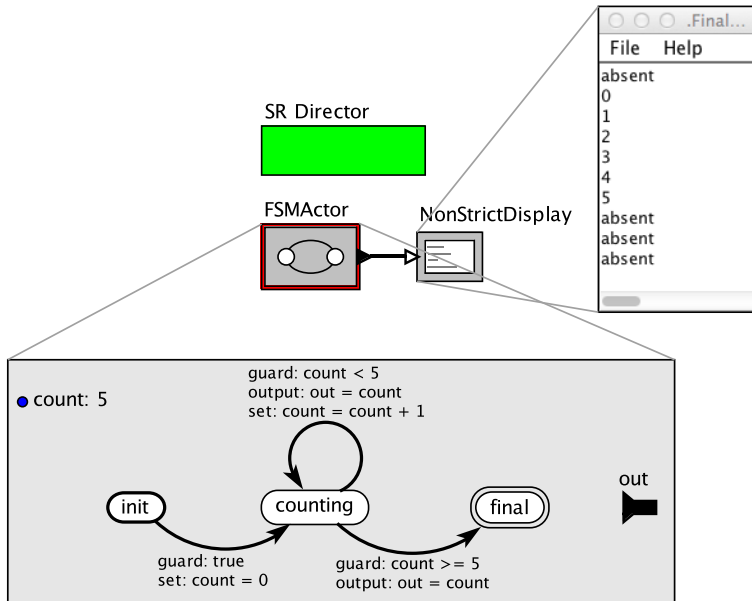


Figure 6.11: A state machine with a final state in an SR model. [[online](#)]

*init* to *counting* does not include any output action. Such a transition would not be compatible with SDF, because actors in SDF are required to produce a fixed number of outputs on each firing.

Notice the use of **NonStrictDisplay**. This actor is similar to **Display** except that it displays “absent” when the input is absent, whereas **Display** does not display anything when the input is absent.

As illustrated by the above example, SR supports a notion of absent values. Dataflow domains and **PN** have no such notion. Failure to produce outputs will starve downstream actors, preventing them from executing. An FSM with a final state in **PN** will simply stop producing outputs when it reaches the final state. This can result in termination of the entire model if it causes **starvation** (i.e., if other actors require inputs from the FSM in order to continue).

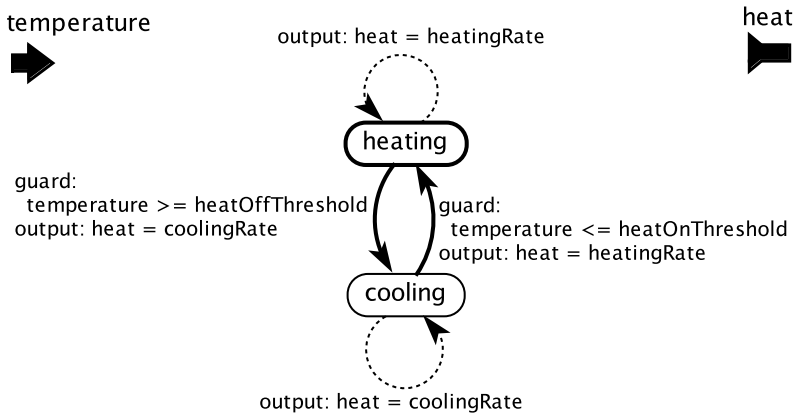


Figure 6.12: An FSM equivalent to that shown in Figure 6.5, but using default self-transitions (indicated with dashed lines). These are taken if the other outgoing transition is not enabled. [\[online\]](#)

### 6.2.5 Default Transitions

An FSM may have **default transitions**, which are transitions that have the *default* parameter set to true (see Figure 6.4). These transitions become enabled if no other outgoing (non-default) transition of the current state is enabled. Default transitions are shown as dashed arcs rather than solid arcs.

**Example 6.6:** The thermostat FSM of Figure 6.5 can be equivalently implemented using default transitions as shown in Figure 6.12. Here, the default transitions simply specify that if the outgoing transition 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 are no other non-default transitions enabled. Default transitions, therefore, provide a rudimentary form of **priority**; non-default transitions have priority over default transitions. Unlike some state-machine languages, such as *SyncCharts*, Ptolemy II FSMs offer only two levels of priority, although it is always possible to encode arbitrary priorities using guards. Note that using default transitions with timed models of computation can be somewhat tricky; see Section 8.5 in Chapter 8.

Default transitions can often be used to simplify guard expressions, as illustrated by the following example.

**Example 6.7:** Consider the counting state machine of Example 6.5, shown in Figure 6.11. We can add a *reset* input, as shown in Figure 6.13, to enable the count to be reset. If *reset* is present and true, then the state machine returns to the *init* state. However, the implementation in Figure 6.13 must then be modified; the two existing transitions out of the *counting* state must include an additional clause

```
&& (!reset_isPresent || !reset)
```

This clause ensures that the self loop on the *counting* state is only taken if the *reset* input is either absent or false. Without this clause, the state machine would have become nondeterministic, since two of the transitions out of the *counting* state could

have become simultaneously enabled. This clause, however, increases the visual complexity of the guard expression, which is functionally quite simple. Figure 6.14 shows a version where default transitions are used instead. These indicate that the machine should count only if the *reset* input is not present and *true*.

For this machine, if *reset* is present in the fourth firing, for example, then the first few outputs will be *absent*, 0, 1, 2, *absent*, 0, 1. In the iteration when *reset* is present and *true*, the output “2” is produced, and then the machine starts over.

## 6.2.6 Nondeterministic State Machines

If more than one guard evaluates to true at any time, then the FSM is a **nondeterministic FSM** (unless one of the guards is on a default transition and the other is not). The transitions that are simultaneously enabled are called **nondeterministic transitions**. By default, transitions are not allowed to be nondeterministic, so if more than one guard evaluates to true, Ptolemy will issue an exception similar to the below:

**Nondeterministic FSM error:** Multiple enabled transitions found but not all of them are marked nondeterministic.

*in ... name of a transition not so marked ...*

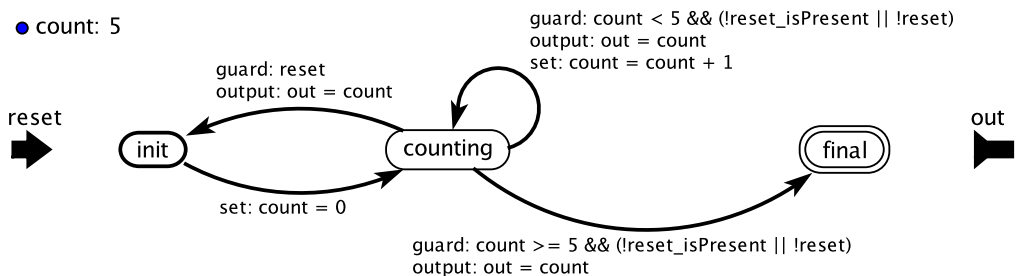


Figure 6.13: A state machine like that in Figure 6.11, but with an additional *reset* input port. [\[online\]](#)

There are cases, however, where it is desirable to allow nondeterministic transitions. In particular, nondeterministic transitions provide good models of systems that can exhibit multiple behaviors for the same inputs. They can also be useful for modeling the possibility of fault conditions where there is no information about the likelihood of a fault occurring. Nondeterministic transitions are allowed by setting the *nondeterministic* parameter to `true` on every transition that can be enabled while another transition is enabled (see Figure 6.4).

**Example 6.8:** A model of a faulty thermostat is shown in Figure 6.15. 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, indicated by the red arc color. A plot of the model's execution is shown in Figure 6.16. Note that the heater is on for relatively 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 `ModalModel` or `FSMACTOR`. If the `fire` method is invoked more than once in an iteration (see Section

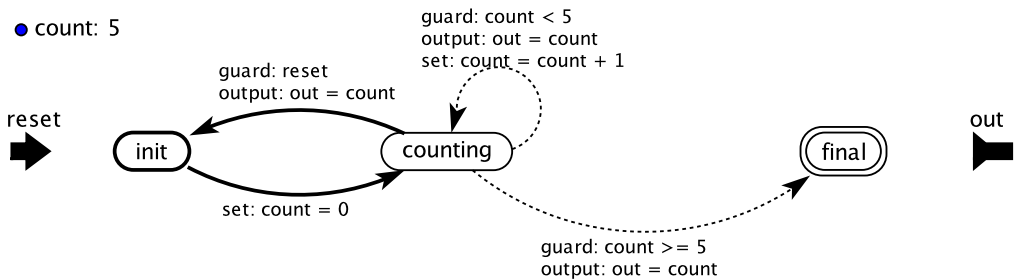


Figure 6.14: A state machine like that in Figure 6.13, but using default transitions to simplify the guard expressions. [\[online\]](#)

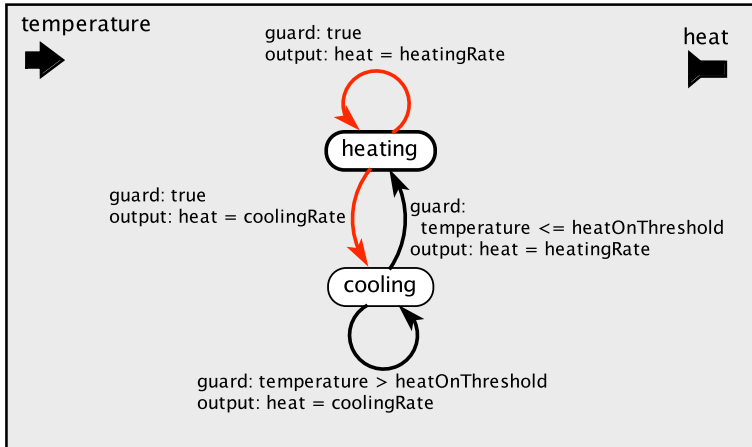


Figure 6.15: A model of a faulty thermostat that nondeterministically switches from heating to cooling. [\[online\]](#)

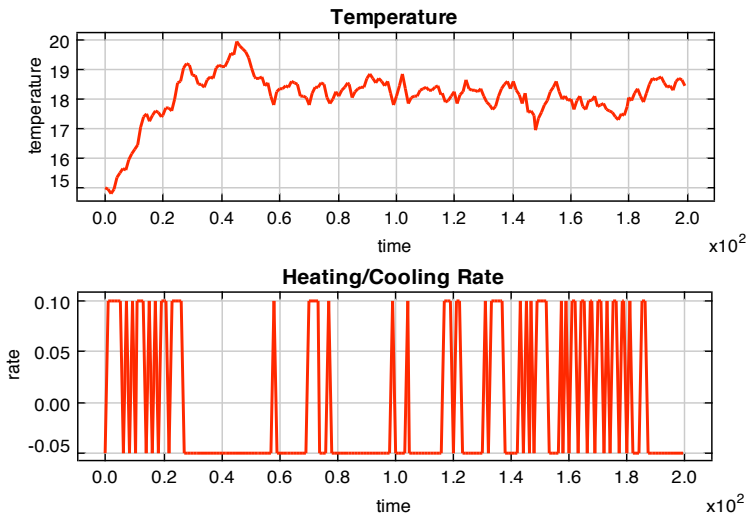


Figure 6.16: Plot of the thermostat FSM of Figure 6.15, a variant of Figure 6.5.

6.4 below), then subsequent invocations in the same iteration will always choose the same transition.

### 6.2.7 Immediate Transitions

Thus far we have only considered the case where each firing of an FSM results in a single transition. It is possible, however, to take more than one transition in a single firing, by using an **immediate transition**. If a state *A* has an immediate transition to another state *B*, then that transition will be taken in the same firing as a transition into state *A* if the guard on the immediate transition is true. The transition into and out of *A* will occur in the same firing. In this case, *A* is called a **transient state**.

**Example 6.9:** In Example 6.7, the output of the thermostat is absent in the first iteration and in the iteration immediately following a *reset*. These absent outputs can be avoided by marking the transition from *init* to *counting* immediate, as shown in Figure 6.17. This change has two effects. First, when the model is initialized, the transition from *init* to *counting* is taken immediately (during initialization), which sets the `count` variable to 0. Thus, in the first iteration of the state machine, it will be in state *counting*. This prevents the initial *absent* from appearing at the output. Instead, the output in the first iteration will be 0.

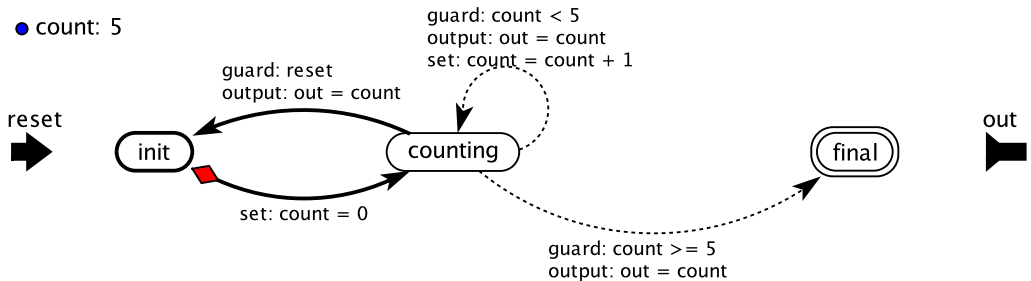


Figure 6.17: A state machine like that in Figure 6.14, but using an immediate transition to prevent absent outputs before counting. The immediate transition is indicated by a red diamond. [online]



The second effect is that in the *counting* state, if the *reset* input is present and true, then the machine will transition from *counting* to *init*, and back to *counting*, in the same iteration, resetting the `count` variable to 0.

For this machine, if *reset* is present in the fourth firing (for example), then the first few outputs displayed will be as follows: 0, 1, 2, 3, 0, 1. In the iteration when *reset* is present and true, the output “3” is produced by the transition back to *init*, and then the machine starts over.

Note that a transient state is not quite the same thing as a state in which the state machine spends zero time; because of [superdense time](#) in Ptolemy II, a state machine may spend zero time in a state, but the transition into the state and out of the state occur in different firings, at different [microstep](#) indexes (see sidebar on page 210).

When a state machine reacts, the state at the start of the reaction is called the **current state**. The current state may have immediate transitions coming out of it. For this to be the case, it is necessary that in the previous reaction the guards on these transitions evaluated to false; otherwise, the state would have been transient and would not have become the current state. When the current state has both immediate and non-immediate transitions out of it, those two classes of transitions are treated identically. There is no distinction between them, and no priority order between them. If an immediate and non-immediate transition out of the current state both have guards that evaluate to true, then either one of them needs to a [default transition](#), or both of them need to be marked nondeterministic.

If there are immediate transitions out of the [initial state](#), then their guards are evaluated when the FSM is initialized, and if the guard is true, then the transition is taken before the FSM starts executing. Notice that in some domains, such as [SR](#), outputs produced prior to the start of execution will never be observed by the destination, so in those domains, an immediate transition out of an initial state should not produce outputs.

Immediate, default, and nondeterministic transitions can be used in combination to sometimes dramatically simplify a state machine diagram, as illustrated in the following example.

**Example 6.10:** An **ABRO** state machine is a class of FSM that waits for a signal *A* and a signal *B* to arrive. Once both have arrived, it produces an output *O*, unless a reset signal *R* arrives, in which case it starts all over, waiting for *A* and *B*.

This pattern is used to model a variety of applications. For example,  $A$  may represent a buyer for a widget,  $B$  a seller, and  $O$  the occurrence of a transaction.  $R$  may represent the widget becoming unavailable.

Specifically, the system has boolean-valued inputs  $A$ ,  $B$ , and  $R$ , and a boolean-valued output  $O$ . Output  $O$  will be present and true as soon as both inputs  $A$  and  $B$  have been present and true. In any iteration where  $R$  is present and true, the behavior is restarted from the beginning.

An implementation of this state machine is shown in Figure 6.18. The initial state,  $nAnB$  (short for “not  $A$  and not  $B$ ”) represents the situation where neither  $A$  nor  $B$  has arrived. The state  $nAB$  represents the situation where  $A$  has not arrived but  $B$  has – and so on.

### Probing Further: Weakly Transient States

A state machine may spend zero time in a state without the use of immediate transitions. Such a state is called a **weakly transient state**. It is not quite like the [transient states](#) of Section 6.2.7, which have [immediate transitions](#) that move out of the state *within a single reaction*. A weakly transient state is the final state of one reaction, and the current state of the next reaction, but no [model time](#) elapses between reactions. 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.

When a transition is taken in an FSM, the [FSMACTOR](#) or [ModalModel](#) calls the `fireAtCurrentTime` method of its enclosing director. This method requests a new firing in the next [microstep](#) regardless of whether any additional inputs become available. If the director honors this request (as timed directors typically 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 before model time has advanced. Note also that in a [modal model](#) (see Section 6.3 and Chapter 8), 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 (see Chapter 9), since the transition may represent a discontinuity in otherwise continuous signals. The discontinuity translates into two distinct events with the same time stamp.

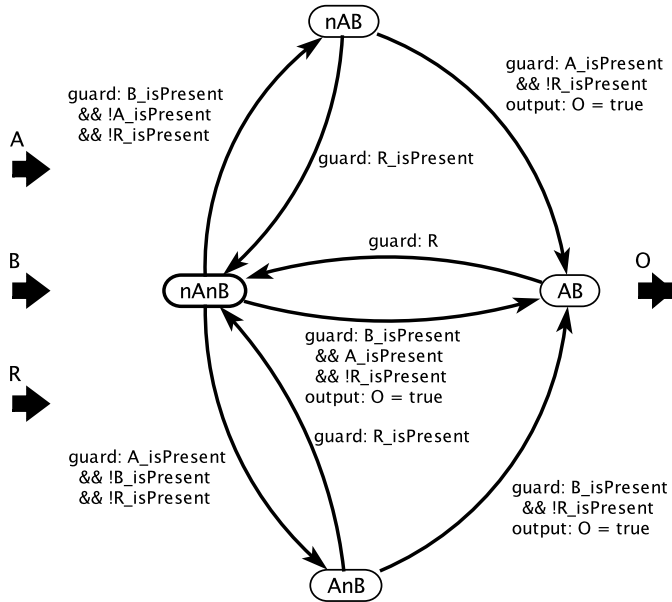


Figure 6.18: A brute-force implementation of the classic ABRO state machine. [\[online\]](#)

The guard expressions in Figure 6.18 can be difficult to read (although it can get much worse — see Exercise 4). An alternative implementation that is easier to read (once you are familiar with the transition notation, which are summarized in Table 6.1) is shown in Figure 6.19. This example uses nondeterminate, default, and immediate transitions to simplify guard expressions.

Immediate transitions may write to the same output ports that are written to by previous transitions taken in the same iteration. FSMs are **imperative**, with a well-defined sequence of execution, so the output of the FSM will be the *last* value written to an out-

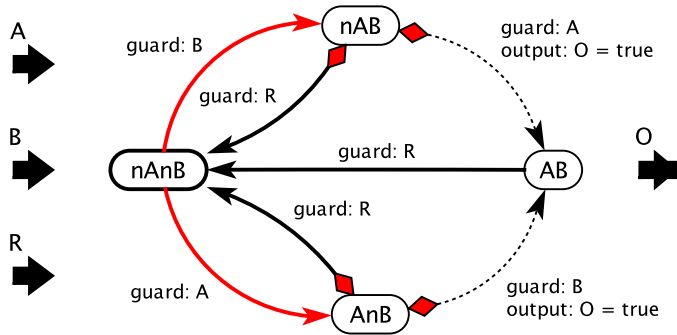


Figure 6.19: An implementation of the ABRO state machine that leverages default transitions, immediate transitions, and nondeterministic transitions to simplify the guard expressions. [\[online\]](#)

put port in a chain of transitions. Similarly, immediate transitions may write to the same parameter in their [set actions](#), overwriting values written in previously taken transitions.

## 6.3 Hierarchical FSMs

It is always possible (and encouraged) to construct an FSM by using the [ModalModel](#) actor rather than the [FSMActor](#). The [ModalModel](#) actor allows states to be defined with one or more **refinements**, or submodels. In Chapter 8 we discuss the general form of this approach, called [modal models](#), where the submodel can be an arbitrary Ptolemy II model. Here, we consider only the special case where the submodel is itself an FSM. The approach yields a **hierarchical FSM** or **hierarchical state machine**.

To create a hierarchical FSM, select [Add Refinement](#) in the context menu for a state, and choose [State Machine Refinement](#), as shown in Figure 6.20. This creates a state machine refinement (submodel) that can reference the higher-level state machine’s input ports and write to the output ports. The refinement’s states can themselves have refinements (either [Default Refinements](#) or [State Machine Refinements](#)).

In addition to the transition types of Table 6.1, hierarchical state machines offer additional transition types, summarized in Table 6.3. These will be explained below.

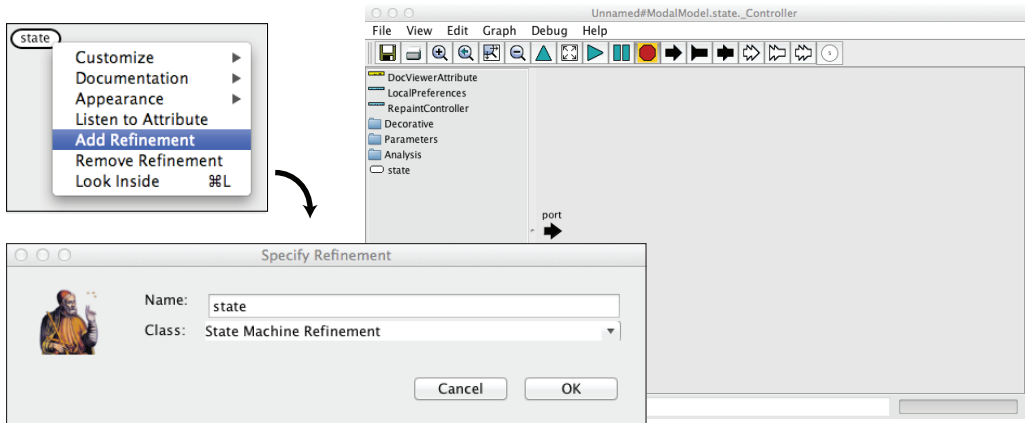


Figure 6.20: How to add a refinement to the state of a ModalModel.

### 6.3.1 State Refinements

The execution of a modal model follows a simple pattern. When the modal model fires, first, its refinements fire. Then its guards are evaluated and a transition may be chosen. A refinement may produce outputs, and so may a transition, but since the refinement is fired first, if both produce output values on the same port, the transition will overwrite the value produced by the refinement. Execution is strictly sequential.

Postfire is similar. When the modal model postfires, it first postfires its refinements, and then commits the transition, executing any set actions on the transition. Again, if the refinement and the transition write to the same variable, the transition prevails.

Note that a state can have more than one refinement. To create a second refinement, invoke `Add Refinement` again. Refinements are executed in order, so if two refinements of the same state produce a value on the same output or update the same variable, then the second one will prevail.<sup>†</sup> The last output value produced becomes the output of the modal model for the firing. It overwrites the actions of the first, as with chains of [immediate transitions](#). To change the order in which refinements execute, simply double click on the state and edit the *refinementName* parameter, which is a comma-separated list of refinements.

<sup>†</sup>If you wish to have refinements that execute concurrently, see Chapter 8.

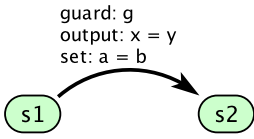
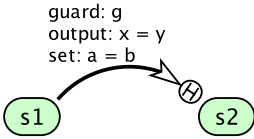
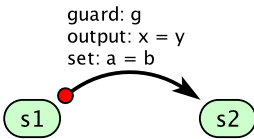
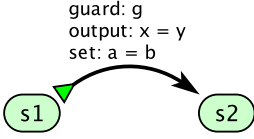
notation	description
	<p>An <b>ordinary transition</b>. Upon firing, the refinement of the source state is fired first, and then if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM will choose the transition. It will produce the value <math>y</math> on output port <math>x</math>, overwriting any value that the source state refinement might have produced on the same port. Upon transitioning (in postfire), the actor will set the variable <math>a</math> to have value <math>b</math>, again overwriting any value that the refinement may have assigned to <math>a</math>. Finally, the refinements of state <math>s2</math> are reset to their initial states. For this reason, these transitions are sometimes called <b>reset transitions</b>.</p>
	<p>A <b>history transition</b>. This is similar to an ordinary transition, except that when entering state <math>s2</math>, the refinements of that state are <i>not</i> reset to their initial states, but rather resume from whatever state they were in when the refinement was last active. On first entry to <math>s2</math>, of course, the refinements will start from their initial states.</p>
	<p>A <b>preemptive transition</b>. If the current state is <math>s1</math> and the guard is true, then the state refinement (the FSM sub-model) for <math>s1</math> will not be invoked prior to the transition.</p>
	<p>A <b>termination transition</b>. If all refinements of state <math>s1</math> reach a final state and the guard is true, then the transition is taken.</p>

Table 6.3: Summary of FSM transitions and their notations for hierarchical state machines. Here, we assume all refinements are themselves FSMs, although in Chapter 8 we will see that refinements can be arbitrary Ptolemy II models.

It is also possible for a refinement to be the refinement of more than one state. To add a refinement to a state that is already the refinement of another state, double click on the state and insert the name of the refinement into the *refinementName* parameter.

### 6.3.2 Benefits of Hierarchical FSMs

Hierarchical FSMs can be easier to understand and more modular than flat FSMs, as illustrated in the following example.

**Example 6.11:** A hierarchical FSM that combines the normal and faulty thermostats of Examples 6.1 and 6.8 is shown in Figure 6.21.

In this model, a *Bernoulli* actor is used to generate a *fault* signal (which will be *true* with some fixed probability, shown as 0.01 in the figure). When the *fault* signal is *true*, the modal model will transition to the faulty state and remain there for ten iterations before returning to the *normal* mode. The state refinements are the same as those in Figures 6.12 and 6.15, modeling the normal and faulty behavior of the thermostat.

The transitions from *normal* to *faulty* and back in top-level FSM are *preemptive transitions*, indicated by the red circles on their stem, which means that when the guards on those transitions become true, the refinement of the current state is not executed, and the refinement of the destination state is reset to its initial state. In contrast, the self-loop transition from *faulty* back to itself is a *history transition*, which, as we will explain below, means that when the transition is taken, the destination state refinement is not initialized. It resumes where it left off.

An equivalent flat FSM is shown in Figure 6.22. Arguably, the hierarchical diagram is easier to read and more clearly expresses the separate normal and faulty mechanisms and how transitions between these occur. See Exercise 7 of this chapter for a more dramatic illustration of the potential benefits of using a hierarchical approach.

Notice that the model in Figure 6.21 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 non-deterministic FSM also has random behavior, but no probability model is provided.

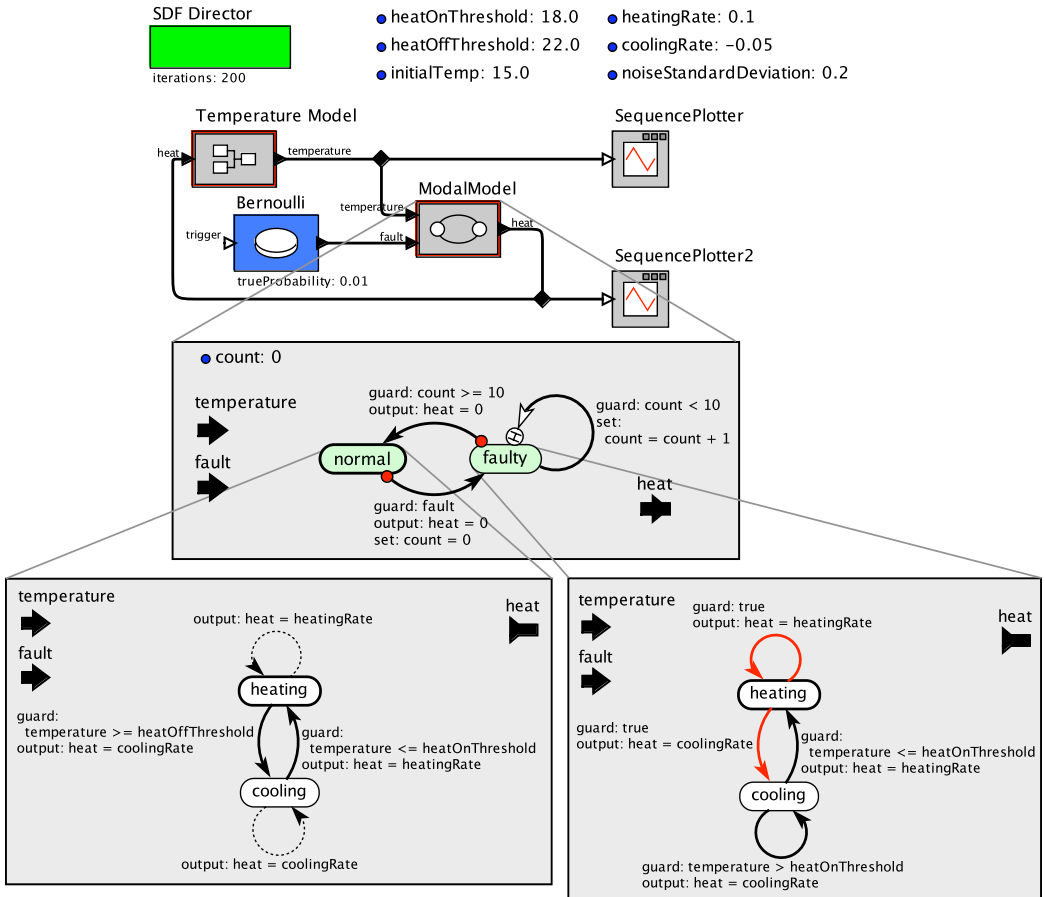


Figure 6.21: A hierarchical FSM that combines the normal and faulty thermostats of Examples 6.1 and 6.8. [\[online\]](#)



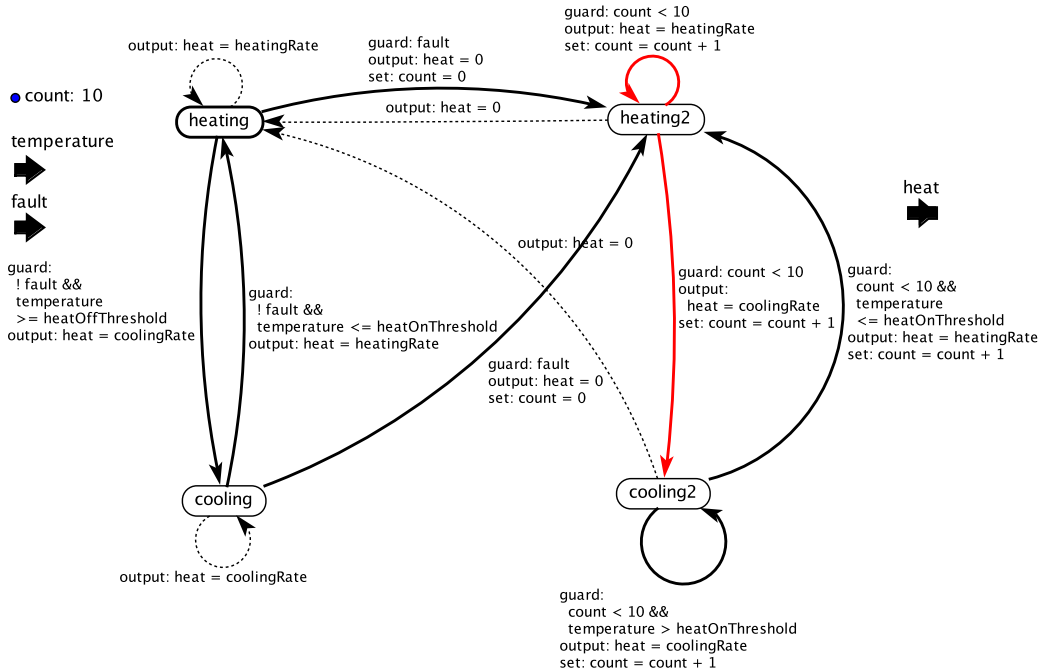


Figure 6.22: A flat FSM version of the hierarchical FSM of Figure 6.21. [online]

### 6.3.3 Preemptive and History Transitions

A state that has a refinement is shaded in light green in Vergil, as shown in Figure 6.21. The top-level FSM in that figure also uses two new specialized transitions, which we now explain (see Table 6.3).

The first is a **preemptive transition**, indicated by red circle at the start of the transition. In a firing where the current state has a preemptive transition leading to another state, the refinement does not fire if the guard on the transition is true. It is preempted by the transition.<sup>‡</sup> If the transition out of the *normal* state was not preemptive in this example, then in an iteration where the *fault* input is true and present, the refinement FSM of the *normal* state would nonetheless produce a normal output. The preemptive transition prevents this

<sup>‡</sup>In the literature, this is sometimes called **strong preemption**, where **weak preemption** refers to a normal transition out of a state that allows the refinement to execute.

from occurring. In iterations where a fault occurs, the preemptive transition generates outputs that are not the normal outputs produced by the *normal* or *faulty* submodels. The model shown in the figure assigns the outputs the value 0 in the iteration when either a transition occurs from *normal* to *faulty*, or vice versa.

A **current state** may have preemptive, default preemptive, non-preemptive, and default non-preemptive transitions. The guards on these transitions are checked in that order, giving four priority levels. Similarly, immediate transitions may also be preemptive and/or default transitions, so they again have four possible priority levels (see Exercise 9).

The second of the two specialized transitions is a **history transition**, indicated by an out-lined arrowhead and a circle with an “H.” When such a transition is taken, the refinement of the destination state is *not* initialized, in contrast to an ordinary transition. Instead, it resumes from the state it was last in when the refinement was previously active. In Figure 6.21, the self transition from *faulty* back to itself is a history transition because its purpose is to just count iterations, not to interfere with the execution of the refinement.

Transitions that are not history transitions are often called **reset transitions**, because they reset the destination refinements.

#### 6.3.4 Termination Transitions

A **termination transition** is a transition that is enabled only when the refinements of the current state reach a final state. The following example uses such a transition to significantly simplify the **ABRO** example.

**Example 6.12:** A hierarchical version of the ABRO model of Figure 6.19 is shown in Figure 6.23. At the top level is a single state and a preemptive reset transition that is triggered by an input *R*. Below that is a two-state machine that waits in *waitAB* until the two refinements of *waitAB* transition reach a final state. Its transition is a termination transition, indicated by the green diamond at its stem. When that the termination transition is taken, it will transition to the final state called *done* and produce the output *O*. Each refinement of *waitAB* waits for one of *A* or *B*, and once it receives it, transitions to a final state.

In each firing of the modal model, while in *waitAB*, both of the lowest level refinements execute. In this case, it does not matter in which order they execute.

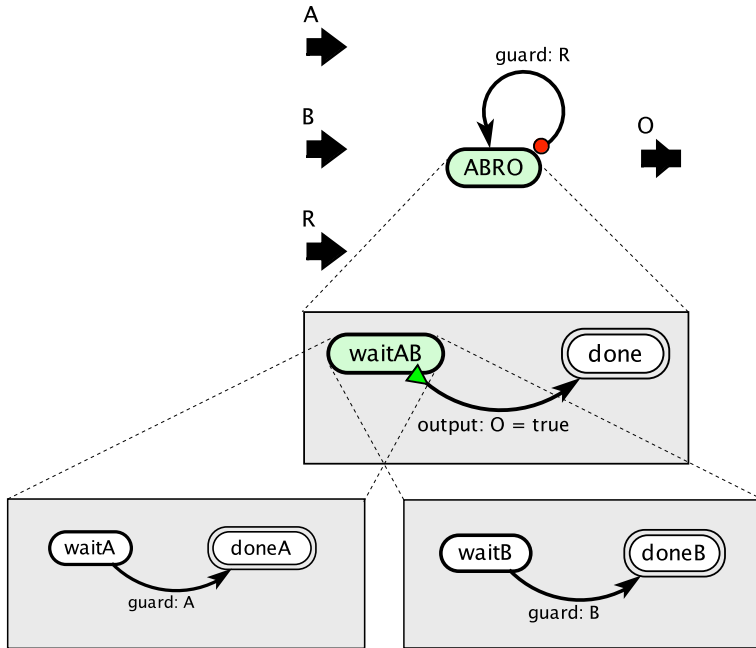


Figure 6.23: A hierarchical version of the ABRO model of Figure 6.19. [online]

Hierarchical machines can be much more compact than their flat counterparts. Exercise 5 (at the end of the chapter), for example, illustrates that if you increase the number of signals that ABRO waits for (making, for example **ABCRO**, with three inputs), then the flat machine gets very large very quickly, whereas the hierarchical machine scales linearly.

This use of transitions triggered by entering final states in the refinements is sometimes referred to as **normal termination**. The submodel stops executing when it enters a final state and can be restarted by a reset. André (1996) points out that specialized termination transitions are not really necessary, as local signals can be used instead (see Exercise 6). But they can be convenient for making diagrams simpler.

### 6.3.5 Execution Pattern for Modal Models

Execution of a [ModalModel](#) proceeds in two phases, fire and postfire. In `fire`, it:

1. reads inputs, makes inputs available to current state refinements, if any;
2. evaluates the guards of preemptive transitions out of the current state;
3. if a preemptive transition is enabled, the actor choses that transition and executes its output actions.
4. if no preemptive transition is enabled, then it:
  - a. fires the refinements of the current state (if any), evaluating guards on transitions of the lower-level FSM and producing any required outputs;
  - b. evaluates guards on the non-preemptive transitions of the upper-level FSM (which may refer to outputs produced by the refinement); and
  - c. executes the output actions of the chosen transition of the upper-level FSM.

In `postfire`, the `ModalModel` actor

1. postfires the refinements of the current state if they were fired, which includes executing set actions on any chosen transitions in the lower-level FSM and committing its state change;
2. executes the set actions of the chosen transition of the upper-level FSM;
3. changes the current state to the destination of the chosen transition; and
4. initializes the refinements of the destination state if the transition is a reset transition.

The transitions out of a state are checked in the following order:

1. preemptive transitions,
2. preemptive default transitions,
3. non-preemptive transitions, and
4. non-preemptive default transitions.

For transitions emerging from the [current state](#) (the state at the start of a reaction), no distinction is made between immediate and non-immediate transitions. The distinction only matters upon entering a state, when [immediate transitions](#) are also checked in the same order as above (preemptive, preemptive default, non-preemptive, and non-preemptive default immediate transitions).

## Probing Further: Internal Structure of an FSM

**FSMActor** is a subclass of **CompositeEntity**, just like **CompositeActor**. Internally, it contains some 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:

```

1 <entity name="FSMActor" class="...FSMActor">
2   <entity name="State1" class="...State">
3     <property name="isInitialState" class="...Parameter"
4       value="true"/>
5   </entity>
6   <entity name="State2" class="...State"/>
7   <relation name="relation" class="...Transition"/>
8   <relation name="relation2" class="...Transition"/>
9   <link port="State1.incomingPort" relation="relation2"/>
10  <link port="State1.outgoingPort" relation="relation"/>
11  <link port="State2.incomingPort" relation="relation"/>
12  <link port="State2.outgoingPort" relation="relation2"/>
13 </entity>

```

The same structure can be specified in Java as follows:

```

1 import ptolemy.domains.modal.kernel.FSMActor;
2 import ptolemy.domains.modal.kernel.State;
3 import ptolemy.domains.modal.kernel.Transition;
4 FSMActor actor = new FSMActor();
5 State statel = new State(actor, "State1");
6 State state2 = new State(actor, "State2");
7 Transition relation = new Transition(actor, "relation");
8 Transition relation2 = new Transition(actor, "relation2");
9 statel.incomingPort.link(relation2);
10 statel.outgoingPort.link(relation);
11 state2.incomingPort.link(relation);
12 state2.outgoingPort.link(relation2);

```

Thus, above, we see three distinct concrete syntaxes for the same structure. A **ModalModel** contains an **FSMActor**, the controller, plus each of the refinements.

### Probing Further: Hierarchical State Machines

State machines have a long history in the theory of computation (Hopcroft and Ullman, 1979). An early model for hierarchical FSMs is **Statecharts**, developed by Harel (1987). As with Ptolemy II FSMs, states in Statecharts can have multiple refinements, but unlike ours, in Statecharts the refinements are not executed sequentially. Instead, they execute concurrently, roughly under the *synchronous-reactive* model of computation. We achieve the same effect with *modal models*, as shown in Chapter 8. Another feature of Statecharts, not provided in Ptolemy II, is the ability for a transition to cross levels of the hierarchy.

The **Esterel** synchronous language also has the semantics of hierarchical state machines, although it is given a textual syntax rather than a graphical one (Berry and Gonthier, 1992). Esterel has a rigorous SR semantics for concurrent composition of state machines (Berry, 1999). **SyncCharts**, which came later, provides a visual syntax (André, 1996).

**PRET-C** (Andalam et al., 2010), **Reactive C (RC)** (Boussinot, 1991), and **Synchronous C (SC)** (von Hanxleden, 2009) are C-based languages inspired by Esterel that support hierarchical state machines. In both RC and PRET-C, state refinements (which are called “threads”) are executed sequentially in a fixed, static order. The PRET-C model is more restricted than ours, however, in that distinct states cannot share the same refinements. A consequence is that refinements will always be executed in the same order. The model in Ptolemy II, hence, is closer to that of SC, which uses “priorities” that may be dynamically varied to determine the order of execution of the refinements.

Both RC and PRET-C, like our model, allow repeated writing to outputs, where the last write prevails. In RC, however, if such an overwrite occurs after a read has occurred, a runtime exception is thrown. Our model is closer to that of PRET-C, where during an iteration, outputs function like variables in an ordinary imperative language. Like RC and PRET-C, only the last value written in an iteration is visible to the environment on the output port of the FSM. In contrast, Esterel provides **combine operators**, which merge multiple writes into a single value (for example by adding numerical values).

## 6.4 Concurrent Composition of State Machines<sup>§</sup>

Since FSMs can be used in any Ptolemy II domain, and most domains have a concurrent semantics, a Ptolemy user has many ways to construct concurrent state machines. In most domains, an FSM behaves just like any other actor. In some domains, however, there are some subtleties. In this section we particularly focus on issues that arise when constructing feedback loops in domains that perform [fixed-point iteration](#), such as the [SR](#) and [Continuous](#) domains.

As described earlier, when an FSM executes, it performs a sequence of steps in the `fire` method, and additional steps in the `postfire` method. This separation is important in constructing a fixed point, because the `fire` method may be invoked more than once per iteration while the director searches for a solution, and it cannot include any persistent state changes. Steps 1-4 in the `fire` method of the FSM 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.

**Example 6.13:** Consider the example in Figure 6.24, which requires that the `fire` method be invoked multiple times. As explained in Chapter 5, execution of an SR model requires the director to find a value for each signal at each tick of a global clock. On the first tick, each of the [NonStrictDelay](#) actors places 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 may appear to create a causality loop, but as discussed below, it does not.

In Figure 6.24, note that for all states of the FSMActors, each input port has a guard that depends on the port's value. Thus, it would seem that both inputs need to be known before any output value can be asserted, which 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

<sup>§</sup>This section may be safely skipped on a first reading unless you are particularly focusing on fixed-point domains such as SR and Continuous.

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*.

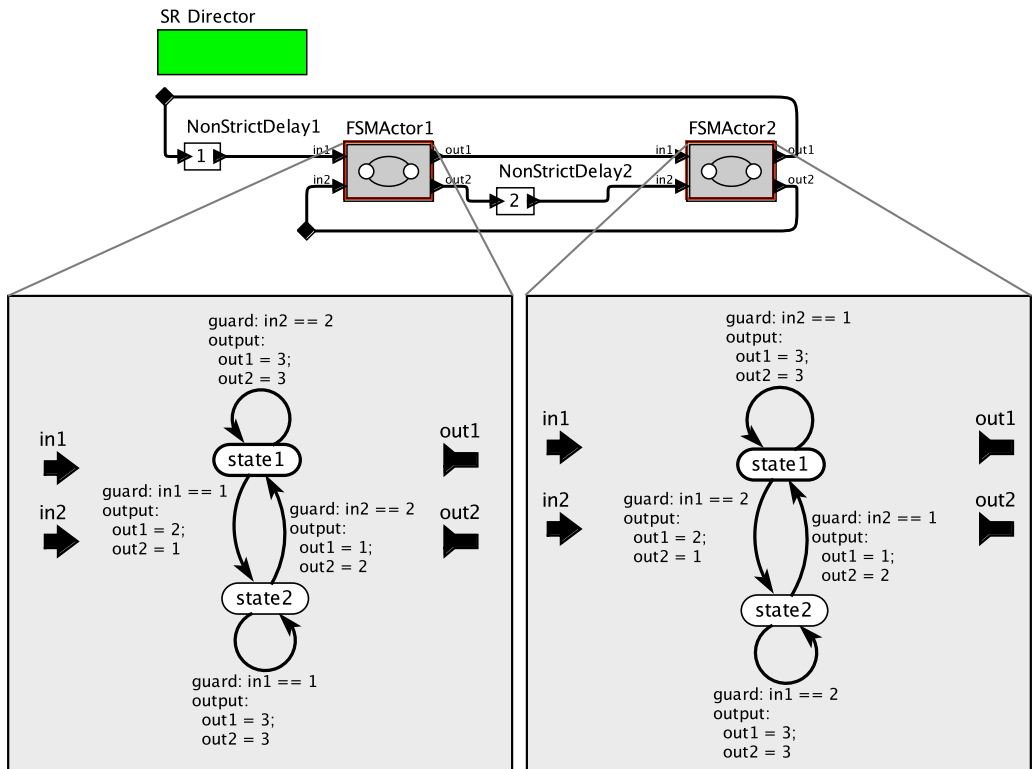


Figure 6.24: A model that requires separation of actions between the `fire` method and the `postfire` method in order to be able to converge to a fixed point. [\[online\]](#)



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,  $\dots$  in ticks 1, 2, 3,  $\dots$ . For `FSMActor2`, the inputs are 2, 1, 2,  $\dots$  in ticks 1, 2, 3,  $\dots$ .

To understand a fixed-point iteration, it is helpful to examine more closely the four steps of execution of the `fire` method explained in Section 6.2 above.

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 that transition is chosen. If a transition has already been chosen 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 issues 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 marked as nondeterministic, then the actor issues 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 the 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 all of the above, choosing a transition may actually amount to choosing a chain of transitions, if there are [immediate transitions](#) enabled.

As described earlier, in the `postfire()` method, the actor executes the set actions of the chosen transition and 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, the model is considered defective, and an error message will be issued.

Nondeterministic FSMs that are executed in a domain that performs fixed-point iteration involve additional 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 are multiple nondeterministic default transitions, then one is chosen at random.

## 6.5 Summary

This chapter has introduced the use of finite-state machines in Ptolemy II to define actor behavior. Finite-state machines can be constructed using the [FSMACTOR](#) or [ModalModel](#) actors, where the latter supports hierarchical refinement of states in the FSM and the former does not. A number of syntactic devices are provided to make FSM descriptions more compact. These include the ability to manipulate variables ([extended state machines](#)), default transitions, immediate transitions, preemptive transitions, and hierarchical state machines, to name a few. Transitions have [output actions](#), which are executed in the `fire` method when a transition is chosen, and [set actions](#), which are executed in the `postfire` method and are used to change the value of variables. This chapter also briefly introduces concurrent composition of state machines, but that subject is studied in much more depth

in Chapter 8, which shows how state refinements can themselves be concurrent Ptolemy II models in another domain.

## Exercises

1. Consider a variant of the thermostat of example 6.1. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. If the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 30 seconds. In both cases, once the 30 seconds have elapsed, it returns to the initial state.
  - (a) Create a Ptolemy II model of this thermostat. You may use an SDF director and assume that it runs at a rate of one iteration per second.
  - (b) How many possible states does your thermostat have? (**Careful!** The number of states should include the number of possible valuations of any local variables.)
  - (c) The thermostat in example 6.1 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,  $x: \mathbb{R} \rightarrow \mathbb{R}$  (for the thermostat,  $x(t)$  is the temperature at time  $t$ ). Suppose that input  $x$  causes output  $y: \mathbb{R} \rightarrow \mathbb{R}$ , also a function of time. E.g., in Figure 6.8,  $x$  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  $\alpha$ , 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. Does your new thermostat model have this property?

2. Exercise 1 of Chapter 5 asks for a model that recognizes the difference between single and double mouse clicks. Specifically, the actor should have an input port named *click*, and two output ports, *singleClick* and *doubleClick*. When a `true` input at *click* is followed by  $N$  *absents*, the actor should produce output `true` on *singleClick*, where  $N$  is a parameter of the actor. If instead a second `true` input occurs within  $N$  ticks of the first, then the actor should output a `true` on *doubleClick*.

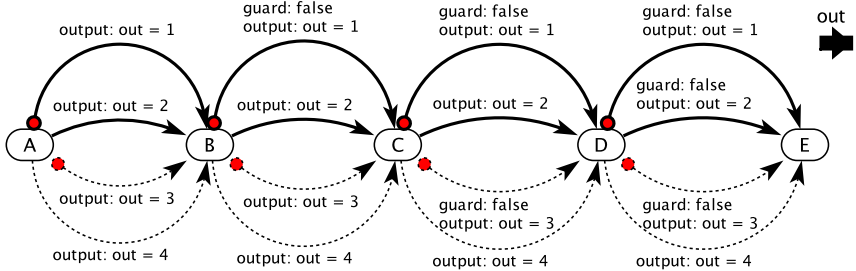
- (a) Create an implementation of this actor using an [extended state machine](#).
  - (b) How does your model behave if given three values *true* within  $N$  ticks on input port *click*, followed by at least  $N$  *absent* ticks.
  - (c) Discuss the feasibility and attractiveness of implementing this as a simple FSM, with no use the arithmetic variables of extended state machines.
3. A common scenario in embedded systems is where a component  $A$  in the system monitors the health of another component  $B$  and raises an alarm. Assume  $B$  provides sensor data as timed events. Component  $A$  will use a local clock to provide a regular stream of local timed events. If component  $B$  fails to send sensor data to component  $A$  at least once in each clock interval, then something may be wrong.
- (a) Design an FSM called MissDetector with two input ports, *sensor* and *clock*, and two output ports *missed* and *ok*. Your FSM should produce an event on *missed* when two *clock* events arrive without an intervening *sensor* event. It should produce an *ok* event when the first *sensor* event after (or at the same time that) a *clock* event arrives.
  - (b) Design a second FSM called StatusClassifier that takes inputs from your first FSM and decides whether component  $B$  is operating normally. Specifically, it should enter a *warning* state if it receives *warningThreshold* *missed* events without an intervening *ok* event, where *warningThreshold* is a parameter. Moreover, once it enters a *warning* state, it should remain in that state until at least *normalThreshold* *ok* events arrive without another intervening *ok*, where *normalThreshold* is another parameter.
  - (c) Comment about the precision and clarity of the English-language specification of the behavior in this problem, compared to your state machine implementation. In particular, find at least one ambiguity in the above specification and explain how your model interprets it.
4. Figures 6.18, 6.19, and 6.23 show the [ABRO](#) example implemented as a finite state machine, discussed in Example 6.10. In these realizations, in an iteration where the reset input  $R$  arrives, the output  $O$  will not be produced, even if in the same iteration  $A$  and  $B$  arrive.

Make a variant of each of these that performs [weak preemption](#) upon arrival of  $R$ . That is,  $R$  prevents the output  $O$  from occurring only if it arrives strictly before both  $A$  and  $B$  have arrived. Specifically:

- (a) Create a weak preemption ABRO that like Figure 6.18, uses only ordinary transitions and has no hierarchy.
  - (b) Create a weak preemption ABRO that like Figure 6.19, uses any type of transition, but has no hierarchy.
  - (c) Create a weak preemption ABRO that like Figure 6.23, uses any type of transition and hierarchy.
5. Figures 6.19 and 6.23 show the ABRO example implemented as a flat and a hierarchical state machine, respectively. Construct corresponding flat and hierarchical ABCRO models, which wait for three inputs, *A*, *B*, and *C*. If you had to wait for, say, 10 inputs, would you prefer to construct the flat or the hierarchical model? Why?
6. André (1996) points out that termination transitions are not necessary, as local signals can be used instead. Construct a hierarchical version of ABRO like that in Example 6.12 but without termination transitions.
7. The hierarchical FSM of Example 6.11 uses reset transitions, which initialize each destination state refinement when it is entered. It also uses preemptive transitions, which prevent firing of the refinement when taken. If these transitions were not reset or preemptive transitions, then the flattened equivalent machine of Figure 6.22 would be much more complex.
- (a) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *faulty* and back are not preemptive.
  - (b) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *faulty* and back are preemptive, as in Figure 6.21, but are also history transitions instead of reset transitions.
  - (c) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *error* and back are nonpreemptive history transitions.
8. Consider the compact implementation of the ABRO state in Figure 6.19.
- (a) Is it possible to do a similarly compact model that does not use nondeterminism?
  - (b) Can a similarly compact variant of ABCRO be achieved without nondeterminism?

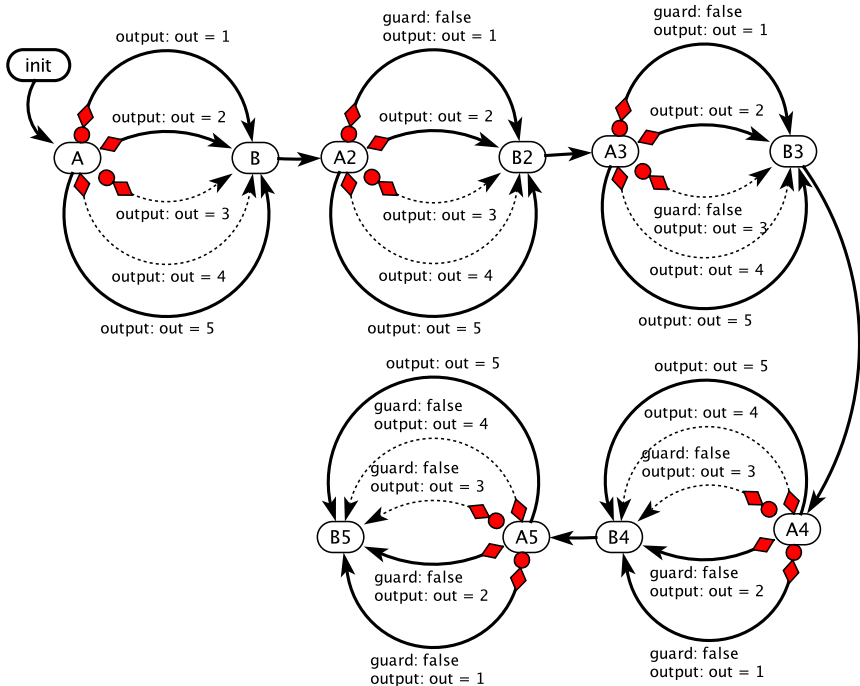
9. This exercise studies relative priorities of transitions.

(a) Consider the following state machine:



Determine the output from the first six reactions.

(b) Consider the following state machine:



Determine the output from the first six reactions.