



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>.

# Modal Models

Thomas Huining Feng, Edward A. Lee, Xiaojun Liu,  
Stavros Tripakis, Haiyang Zheng, and Ye Zhou

## Contents

<b>8.1</b>	<b>The Structure of Modal Models</b>	<b>279</b>
	<i>Probing Further: Internal Structure of a Modal Model</i>	281
<b>8.2</b>	<b>Transitions</b>	<b>282</b>
8.2.1	Reset Transitions	282
8.2.2	Preemptive Transitions	283
8.2.3	Error Transitions	285
8.2.4	Termination Transitions	288
<b>8.3</b>	<b>Execution of Modal Models</b>	<b>292</b>
<b>8.4</b>	<b>Modal Models and Domains</b>	<b>294</b>
8.4.1	Dataflow and Modal Models	294
	<i>Probing Further: Concurrent and Hierarchical Machines</i>	295
8.4.2	Synchronous-Reactive and Modal Models	301
8.4.3	Process Networks and Rendezvous	302
<b>8.5</b>	<b>Time in Modal Models</b>	<b>302</b>
8.5.1	Time Delays in Modal Models	308
8.5.2	Local Time and Environment Time	309
8.5.3	Start Time in Mode Refinements	312
<b>8.6</b>	<b>Summary</b>	<b>313</b>
	<b>Exercises</b>	<b>314</b>

Most interesting systems have multiple modes of operation. Changes in modes may be triggered by external or internal events, such as user inputs, hardware failures, or sensor data. For example, an engine controller in a car may have different behavior when the car is in Park than when it is in Drive.

A **modal model** is an explicit representation of a finite set of behaviors (or modes) and the rules that govern transitions between them. The rules are captured by a [finite state machine](#) (FSM).

In Ptolemy II, the [ModalModel](#) actor is used to implement modal models. ModalModel is a hierarchical actor, like a [composite actor](#), but with multiple refinements instead of just one. Each refinement specifies a single mode of behavior, and 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 Chapter 6; the FSMACTOR does not support state refinements. Modal models use the same transitions and guards described in Chapter 6, plus some additional ones.

**Example 8.1:** The model shown in Figure 8.1 represents a communication channel with two modes of operation: clean and noisy. It includes a ModalModel actor (labeled “Modal Model”) with two states, *clean* and *noisy*. In the *clean* mode, the model 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 generated by a [PoissonClock](#) actor, which generates events at random times according to a Poisson process. (In a Poisson process, the time between events is given by independent and identically distributed random variables with an exponential distribution.) A sample execution of this model, in which the Signal Source actor provides an input sine wave, results in the plot shown in Figure 8.2.

This example combines three distinct models of computation (MoCs). At the top level, the timed behavior of randomly occurring events is captured using the [DE](#) domain. The next level down in the hierarchy, an FSM is used to capture mode changes. The third level uses [SDF](#) to capture untimed processing of sample data.

The process of creating a modal model is illustrated in Figure 8.3. To create a modal model in Vergil, drag in a ModalModel actor from the `Utilities` library and populate

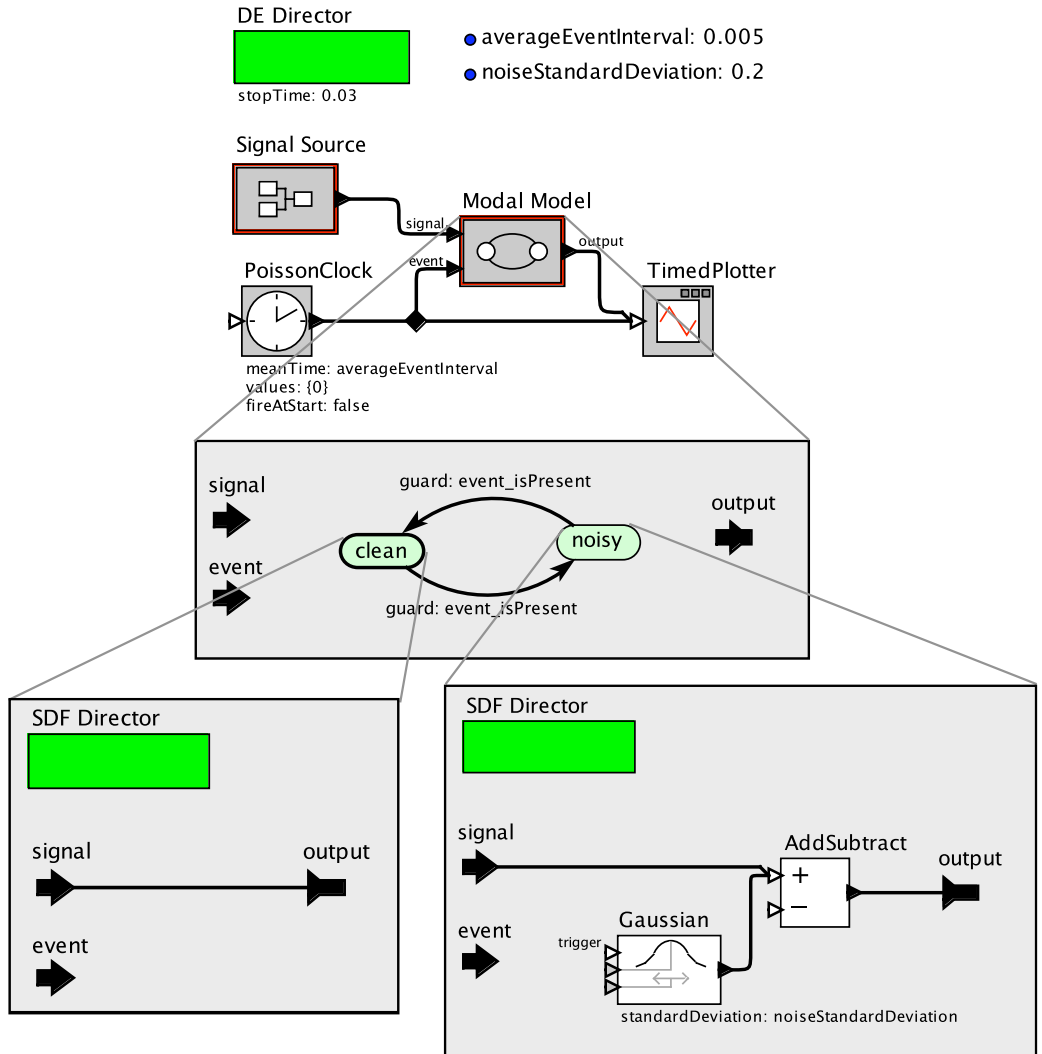


Figure 8.1: Simple modal model that has a normal (clean) operating mode, in which it passes inputs to the output unchanged, and a faulty mode, in which it adds Gaussian noise. It switches between these modes at random times determined by the PoissonClock actor. [\[online\]](#)

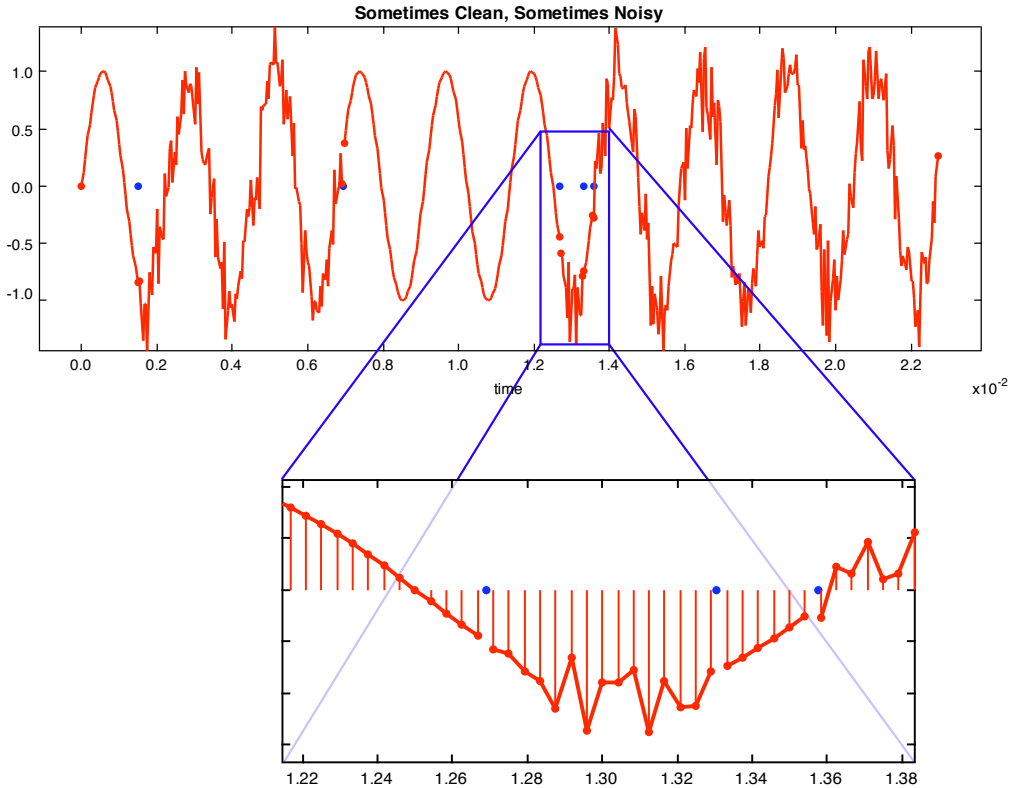


Figure 8.2: Plot generated by the model in Figure 8.1.

it with ports. Open the modal model actor and add 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 is used in the above example; it will require in each refinement a director and actors that process input data to produce outputs. The latter will enable creation of a [hierarchical FSM](#), as described in Chapter 6.

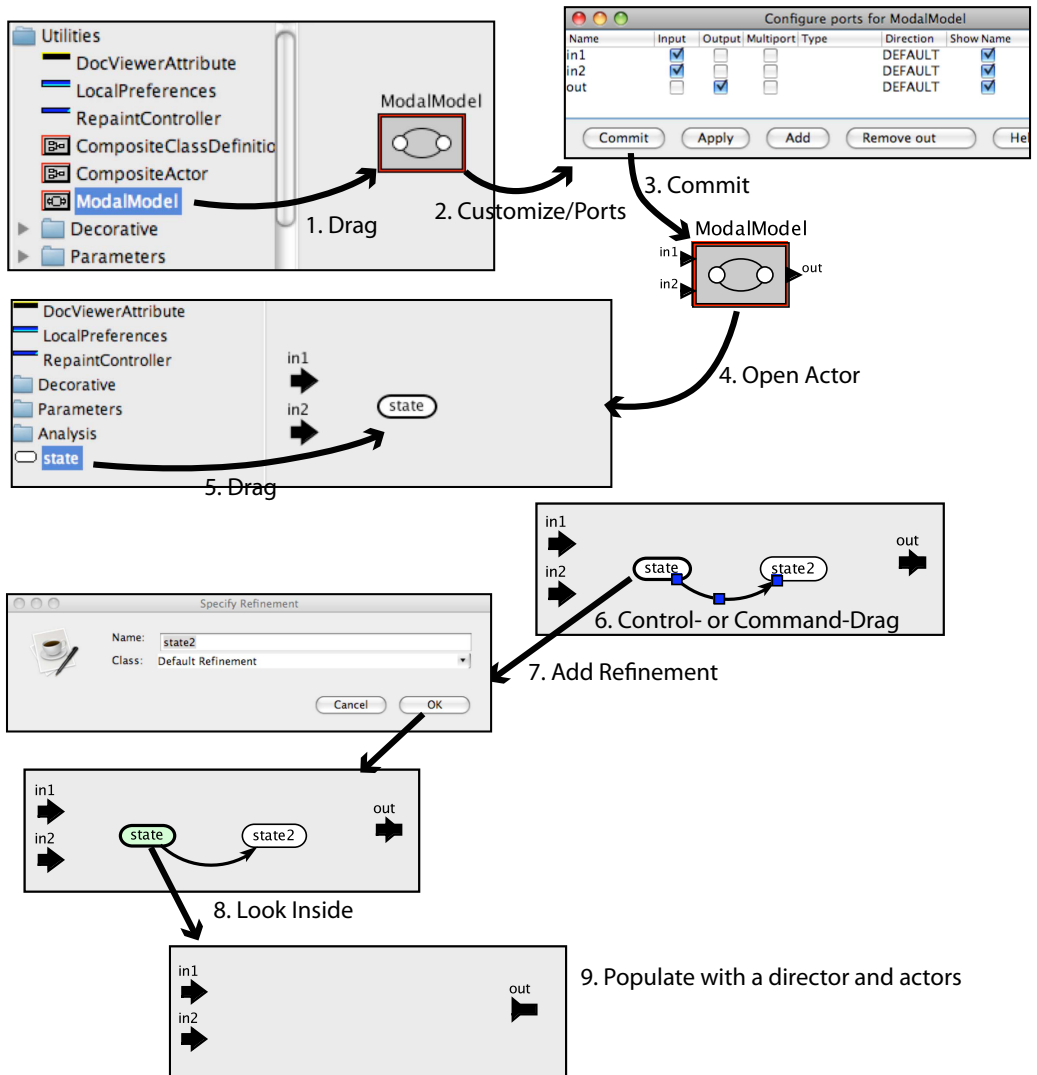


Figure 8.3: How to create modal models.

## 8.1 The Structure of Modal Models

The general structure of a modal model is shown in Figure 8.4. The behavior of a modal model is governed by a state machine, where each state is called a **mode**. In Figure 8.4, each mode is represented by a bubble (like a state in a state machine) but it is colored to indicate 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 mode's behavior. The example in Figure 8.1 shows two refinements, each of which is an **SDF** model that processes input tokens to produce output tokens.

The mode refinement must contain a director, and this director must be compatible with the director that governs the execution of the modal model actor. The example in Figure 8.1 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.

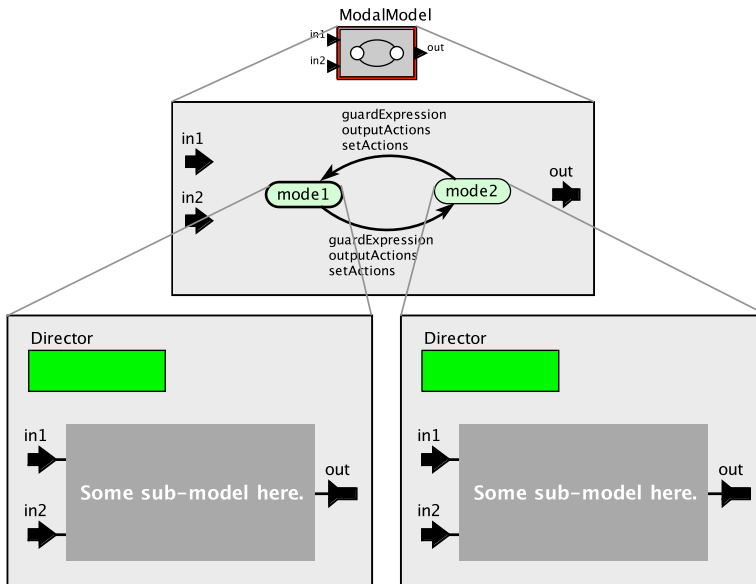


Figure 8.4: General pattern of a modal model with two modes, each with its own refinement.

Like in a finite state machine, modes are connected by arcs representing **transitions** with **guards** that specify when the transition should be taken.

**Example 8.2:** In Figure 8.1, 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 governing the time between transitions.

A variant of the structure in Figure 8.4 is shown in Figure 8.5, where two modes share the same refinement. This is useful when the behavior in different modes differs only by parameter values. For example, Exercise 2 constructs a variant of the example in Figure

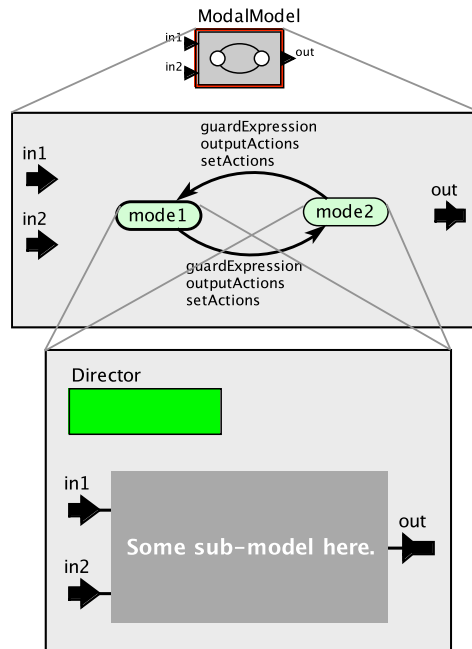


Figure 8.5: Variant of the pattern in Figure 8.4 where two modes share the same refinement.



8.1 where the *clean* refinement differs from the *noisy* refinement only by having a different parameter value for the Gaussian actor. 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 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 is when a mode has multiple refinements. This effect 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. This order can be changed by invoking `Configure` on the state (or double clicking on it) and editing the comma-separated list of refinements.

### 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. Yet in a modal model, two states can share the same refinement, which may seem to violate that general rule.

The key difference is that a `ModalModel` actor is actually a specialized composite actor that contains an instance of `FSMDirector`, an `FSMACTOR`, and any number of composite actors. Each composite actor can be a refinement for any state of the `FSMACTOR`. The `FSMACTOR` is the controller, in the sense 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. This same structure is used for the hierarchical FSMs explained in Section 6.3.

The `Vergil` user interface, however, hides this structure. When you execute an `Open Actor` command on a `ModalModel`, the user interface skips a level of the hierarchy and takes you directly to 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 architecture balances expressiveness with user convenience.

## 8.2 Transitions

All the transition types of Table 6.1 can be used with modal models. They have exactly the same meaning given in that table. The transition types shown in Table 6.3, which are explained for *hierarchical FSMs*, however, have slightly different meanings for refinements that are not FSMs. Refinements of a state in an FSM can be arbitrary *opaque composite actors* (composites that contain a director). They can even be mixed, where some refinements are FSMs and some are other kinds of models. The more general meanings for such transitions are explained in this section, and then summarized in Table 8.1.

### 8.2.1 Reset Transitions

By default, a transition is a *reset transition*, which means that the refinements of the destination state are initialized when the transition is taken. If the refinement is an FSM, as explained in Section 6.3, this simply means that the state of the FSM is set to its initial state. If that initial state itself has refinement state machines, then those too are set to their initial states. In fact, the mechanism of a reset transition is simply that the *initialize* method of the refinement is invoked. This causes all components within the refinement to be initialized.

**Example 8.3:** For the example in Figure 8.1, it does not matter whether the transitions are history transitions or not because the refinements of the two states themselves have no state. The actors in the model (Gaussian and AddSubtract) have no state, so initializing them does not change their behavior.

In the example in Figure 8.6, however, the *Ramp* actors have state. The example shows the transitions being history transitions, which produces the plot in Figure 8.7(a). In this case, the Ramp actors will resume counting from where they last left off when a state is re-entered.

If on the other hand we were to change the transitions to reset transitions, the result would be the plot in Figure 8.7(b). Each time a transition is taken, the Ramp actors are initialized (along with the rest of the refinement), so they begin again counting from zero.

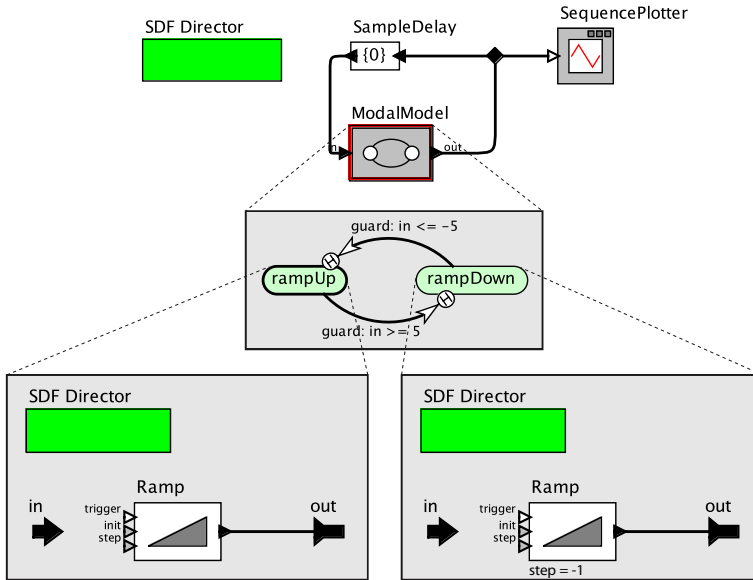


Figure 8.6: A modal model whose behavior depends on whether transitions are reset transitions or history transitions. [[online](#)]

## 8.2.2 Preemptive Transitions

For general modal models, [preemptive transitions](#) work the same way as for [hierarchical FSMs](#). If the guard is enabled, then the refinement does not execute. A consequence is that the refinement does not produce output.

**Example 8.4:** In Figure 8.8, we have modified Figure 8.6 so that the transitions are both preemptive. This means that when a guard evaluates to true, the refinement of the current state does not produce output. In this particular model, no output at all is produced in that iteration, violating the contract with [SDF](#), which expects every firing to produce a fixed, pre-determined number of tokens. An error therefore arises, as shown in the figure. This error can be corrected by producing an output on the transitions or by using a different director.

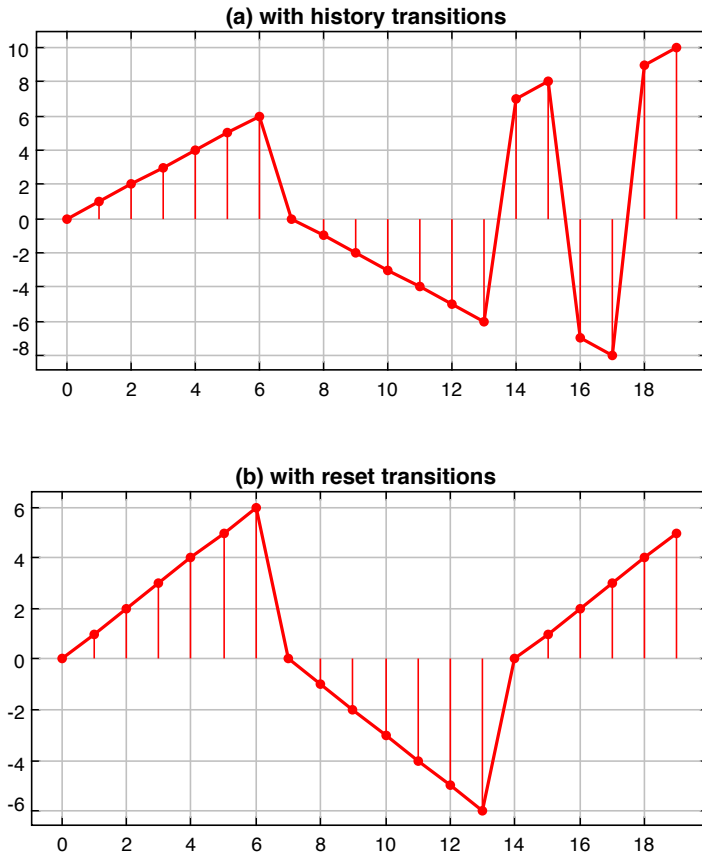


Figure 8.7: (a) The plot resulting from executing the model in Figure 8.6, which has history transitions. (b) The plot that would result from from changing the transitions to reset transitions.

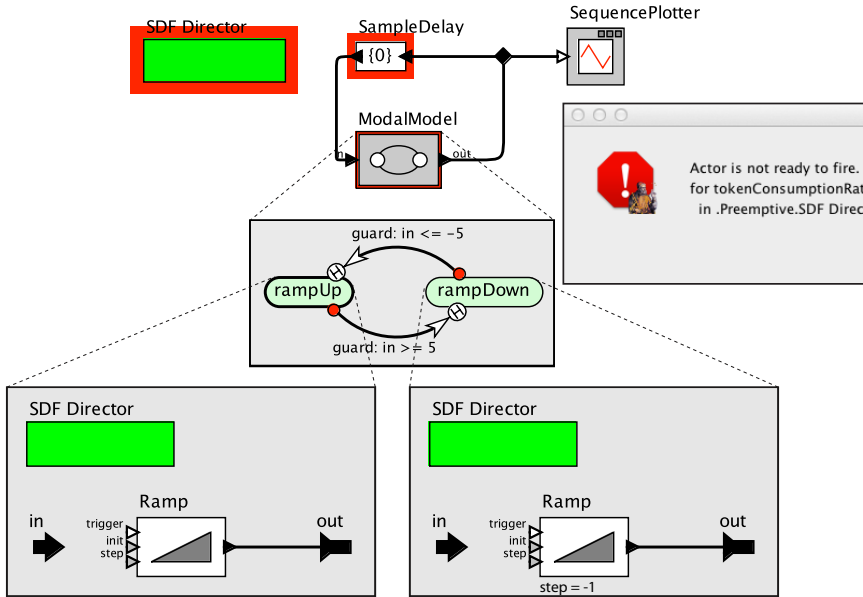
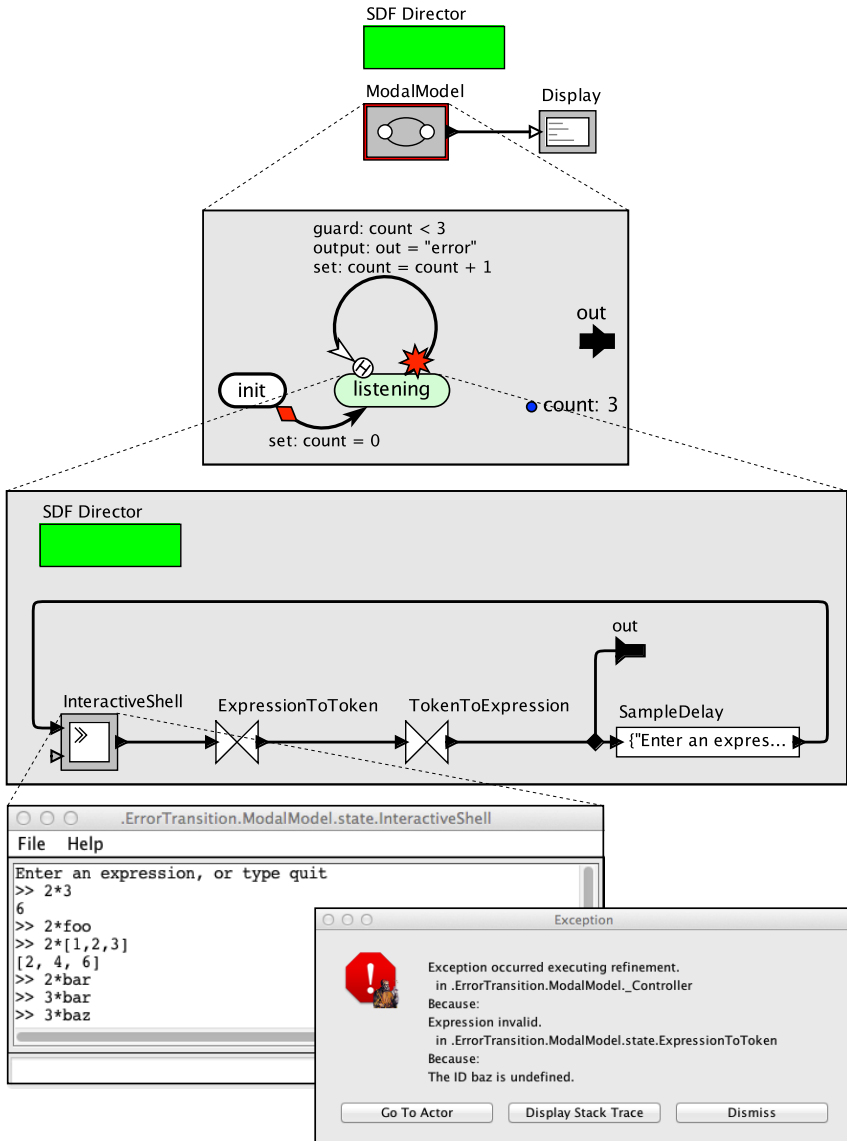


Figure 8.8: A modal model where preemptive transitions prevent the refinements from producing outputs that are expected by the SDF director. [\[online\]](#)

### 8.2.3 Error Transitions

When executing a refinement, an error may occur that causes an exception to be thrown. By default, an exception will cause the entire execution of the model to halt. This is not always desirable. It might be possible to gracefully recover from an error. To allow for this, Ptolemy II state machines include an **error transition**, which is enabled when an error occurs while executing a refinement of the current state. An error transition may also have a guard, output actions, and set actions. Some caution is necessary when using output actions, however, because if the error occurs in the [postfire](#) phase of execution of the refinement, then it may be too late to produce outputs. Most errors, however, will occur in the fire phase, so most of the time this will not be a problem.

**Example 8.5:** A model with an error transition is shown in Figure 8.9. Like Example 8.6, this model includes an [InteractiveShell](#) actor, which allows the user to

Figure 8.9: A modal model with an error transition. [\[online\]](#)

type in arbitrary text. In this model, what the user types is then sent to an [ExpressionToToken](#) actor, which parses what the user types, interpreting the text as an expression in the Ptolemy II expression language (see Chapter 13). Of course, the user may type an invalid expression, which will cause ExpressionToToken to throw an exception.

In the FSM, the *listening* state has an error transition self loop. The error transition is indicated by the red star at its stem. It is enabled when the refinement of the *listening* state has thrown an exception and its guard (if there is a guard) is true. In this case, the guard ensures that this transition is taken no more than three times. After it has been taken three times, it will no longer be enabled.

An example of an execution of this model is shown at the bottom of the figure. Here, the user first types in a valid expression, “2\*3,” which produces the result 6. Then the user types an invalid expression, “2\*f○○.” This is invalid because there is no variable named “f○○” in scope. This triggers an exception, which will be caught by the error transition.

In this simple example, the error transition simply returns to the same state. In fact, this transition is also a [history transition](#), so the refinement is not reinitialized. This could be dangerous with error transitions because an exception may leave the refinement in some inconsistent state. But in this case, it is OK. Were this a reset transition, then the [InteractiveShell](#) would be initialized after the error is caught. This would cause the shell window to be cleared, erasing the history of the interaction with the user.

On the fourth invalid expression, “3\*baz,” the error transition guard is no longer true, so the exception is not caught. This causes the model to stop executing and an exception window to appear, as shown at the bottom of the figure.

Error transitions provide quite a powerful mechanism for recovering from errors in a model. When an error transition is taken, two variables are set that may be used in the guard or the output or set actions of this transition:

- *errorMessage*: The error message (a string).
- *errorClass*: The name of the class of the exception thrown (also a string).

In addition, for some exceptions, a third variable is set:

- *errorCause*: The Ptolemy object that caused the exception.

For the above example, the *errorCause* variable will be a reference to the ExpressionTo-Token actor. This is an ObjectToken on which you can invoke methods such as `getName` in the guard or output or set actions of this transition (see Chapter 14).

### 8.2.4 Termination Transitions\*

A [termination transition](#) behaves rather differently when the state refinements are general Ptolemy models rather than [hierarchical FSMs](#). Such a transition is enabled when all refinements of the current state have terminated, but for general Ptolemy models, it is not possible to know whether the model has terminated prior to the [postfire](#) phase of execution. As a consequence, if at least one of the refinements of the current state is a default refinement (vs. a state machine refinement), then:

- the termination transition is not permitted to produce outputs, and
- the termination transition has lower priority than any other transition, including default transitions.

The reason for these constraints is a bit subtle. Specifically, in many domains ([SR](#) and [Continuous](#), for example), the postfire phase is simply too late to be producing outputs. The outputs will not be seen by downstream actors. Second, the guards on all other transitions (non-termination transitions) will be evaluated in [fire](#) phase of execution, and a transition may be chosen before it is even known whether the termination transition will become enabled.

As a consequence of these constraints, termination transitions are not as useful for general refinements as they are for hierarchical FSMs. Nevertheless, they do occasionally prove useful.

**Example 8.6:** Figure 8.10 shows a model that uses a termination transition. The key actor here is the [InteractiveShell](#), which opens a dialog window into which the user can type, as shown in Figure 8.11. The InteractiveShell asks the user to type something, or to type “quit” to stop. When the user types “quit,” the postfire

---

\*Termination transitions are rather specialized. The reader may want to skip this subsection on a first reading.



method of the InteractiveShell returns false, which causes the bottom SDFDirector to terminate the model (SDF terminates a model when any actor terminates because the SDF contract to produce a fixed number of tokens can no longer be honored).

In the FSM, the transition from *listening* to *check* is a termination transition, so it triggers when the user types quit. This transition has a **set action** of the form:

```
response = yesNoQuestion("Do you want to continue?")
```

which invokes the yesNoQuestion function to pop up a dialog asking the user a question, as shown in Figure 8.11 (see Table 13.16 in Chapter 13 for information

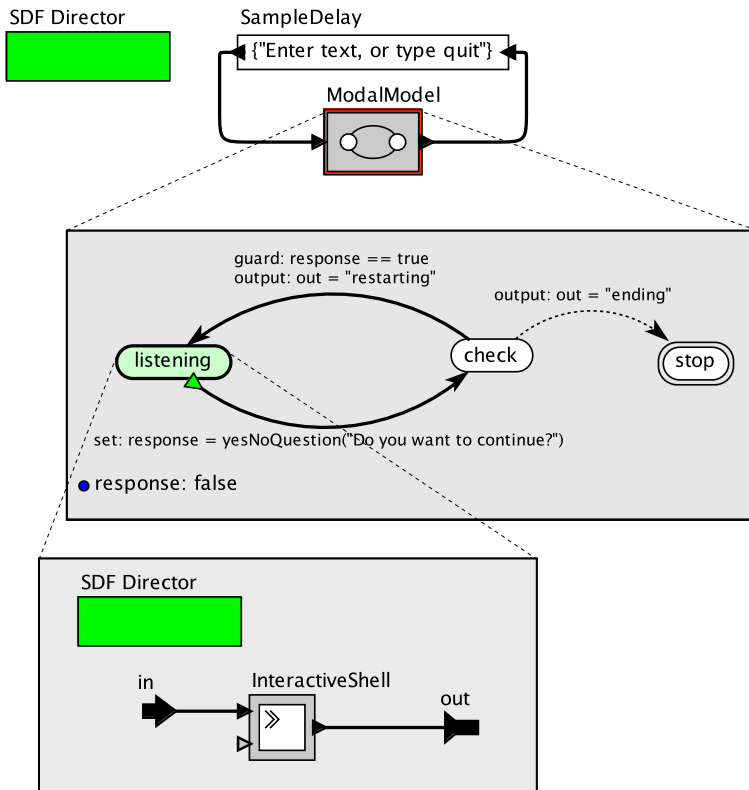


Figure 8.10: A modal model with a termination transition. [[online](#)]

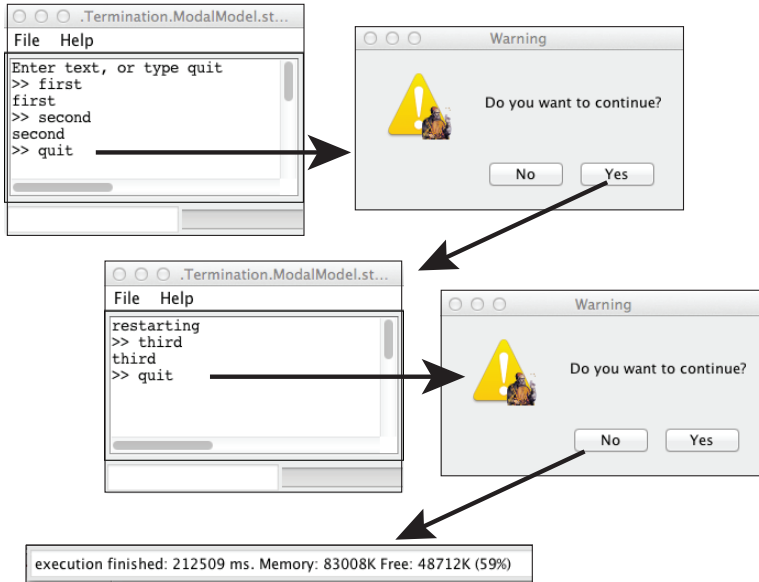


Figure 8.11: An execution of the model in Figure 8.10.

about the `yesNoQuestion` function). If the user responds “yes” to the question, then the transition sets the *response* parameter to true, and otherwise it sets it to false. Hence, in the next iteration, the FSM will either take a reset transition back to the initial listening state, opening another dialog, or it will transition to *stop*, a [final state](#). Transitioning to a final state will cause the top-level SDF director to terminate.

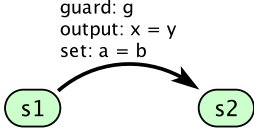
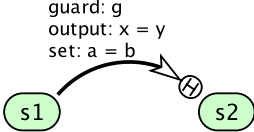
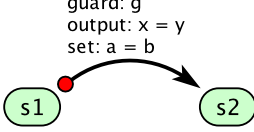
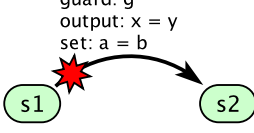
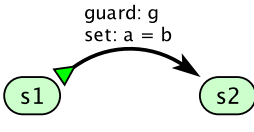
notation	description
	<p>An <b>ordinary transition</b>. Upon firing, the refinement of the source state is fired first, and then if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM will choose the transition. It will produce the value <math>y</math> on output 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 initialized. 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> initialized. On first entry to <math>s2</math>, of course, the refinements will have been initialized.</p>
	<p>A <b>preemptive transition</b>. If the current state is <math>s1</math> and the guard is true, then the state refinement for <math>s1</math> will not be iterated prior to the transition.</p>
	<p>An <b>error transition</b>. If any refinement of state <math>s1</math> throws an exception or a model error, and the guard is true, then this transition will be taken. The output and set actions of the transition can refer to special variables <i>errorMessage</i>, <i>errorClass</i>, and <i>errorCause</i>, as explained in Section 8.2.3.</p>
	<p>A <b>termination transition</b>. If all refinements of state <math>s1</math> have returned false on postfire, and the guard is true, then the transition is taken. Notice that since it cannot be known until the postfire phase that this transition will be taken, the transition cannot produce outputs. For most domains, postfire is too late to produce outputs. Moreover, this transition has lower priority than all other transitions, including <a href="#">default transitions</a>, because it cannot become enabled until postfire.</p>

Table 8.1: Summary of modal model transitions and their notations. We assume the state refinements are arbitrary Ptolemy II models, each with a director.

## 8.3 Execution of Modal Models

Execution of a `ModalModel` is similar to the execution of an `FSMACTOR`. 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; and
4. executes the output actions 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;
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 `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. Its behavior in each domain may have subtle differences, however, particularly in domains that use fixed-point iteration or when nondeterministic transitions are used. In the next section (Section 8.4), we discuss the use of modal models in various domains.

Note that state refinements are fired before guards on non-preemptive transitions are evaluated. One consequence of this ordering is that the guards can refer to the outputs of the refinements. Thus, whether a transition is taken can depend on how the current refinement reacts to the inputs. The astute reader may have already noticed in the figures here that output ports shown in the FSM do not look like normal output ports (notice the *output* ports in Figures 8.1 and 8.4). In the FSM, these output ports are actually both an output and an input. It serves both of these roles in the FSM. An output of the current state refinement is also an input to the FSM, and guards can refer to this input.

**Example 8.7:** Figure 8.12 shows a variant of the model in Figure 8.10 that includes a guard that references an output from a refinement. This guard customizes the response when the user types “hello,” as shown at the bottom of the figure.

The above example shows that the current state refinement and a transition’s output action can both produce outputs on the same output port. Since execution of FSMs is strictly sequential, there is no ambiguity about the result produced on the output of the ModalModel. It is always the last of the values written to the output in the firing.

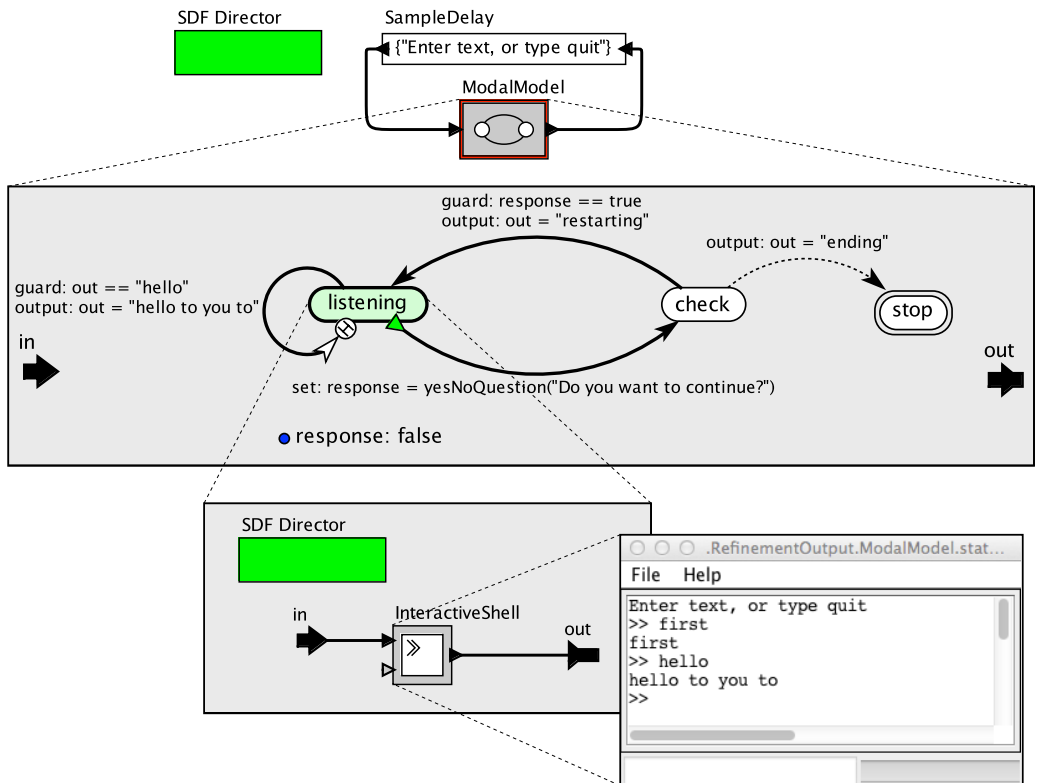


Figure 8.12: A variant of the model in Figure 8.10 that includes a guard that references an output from a refinement. [\[online\]](#)

a chain of [immediate transitions](#), each passing through states that have refinements that write to the same output port, and each transition also writing to the same output port. These writes always occur in a well-defined order, and only the last of these writes will be visible outside the modal model.

## 8.4 Modal Models and Domains

Our modal model examples so far have mostly used the [SDF](#) and [DE](#) domains in simple ways. For the DE examples, such as that in [Figure 8.1](#), the modal model fires when there is an event on at least one of the input ports. Some inputs may be absent, and transitions may be triggered by the presence (or absence) of an input. The modal model may or may not produce an output on each output port; if it does not, then the output will be absent. The only real subtlety with using modal models in DE concerns that passage of time, which will be considered below in [Section 8.5](#).

However, the role of modal models in some other domains is not so simple. In this section, we discuss some of the subtleties.

### 8.4.1 Dataflow and Modal Models

Our SDF examples so far have all been [homogeneous SDF](#), where every actor consumes and produces a single token. When the modal model in these examples fires, all inputs to the modal are present and contain exactly one token. And the firing of the modal model results in one token produced on each output port, with the exception on of [Figure 8.9](#), where an error prevents production of the output token.

With some care, modal models can be used with multirate SDF models, as illustrated by the following example.

**Example 8.8:** In the example shown in [Figure 8.13](#), the refinements of each of the states require 10 samples in order to fire, because of the [SequenceToArray](#) actor. This model alternates between averaging 10 input samples and computing the maximum of 10 input samples. Each firing of the ModalModel executes the current refinement for one [iteration](#), which in this case processes 10 samples. As you can see from the resulting plot, when the input is a sine wave, averaging sequences of 10 samples yields another sine wave, whereas taking the maximum does not.

## Probing Further: Concurrent and Hierarchical Machines

An early model for concurrent and hierarchical FSMs is [Statecharts](#), developed by [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 a 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. Specifically, use of the SR director in a mode refinement to govern concurrent actors, each of which is a state machine, provides a variant of Statecharts. 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 ([von der Beeck, 1994](#)). One variant was adopted to become part of [UML](#) ([Booch et al., 1998](#)). A particularly elegant version is [SyncCharts](#) ([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 a state machine. A straightforward mechanism for doing this results in a state machine whose state space is the cross product of the individual state spaces. More sophisticated mechanisms have been developed, such as interface automata ([de Alfaro and Henzinger, 2001](#)).

[Hybrid systems](#) (Chapter 9) 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 ([Carlioni et al., 2006](#)).

[Girault et al. \(1999\)](#) first showed that FSMs can be combined hierarchically with a 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 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.

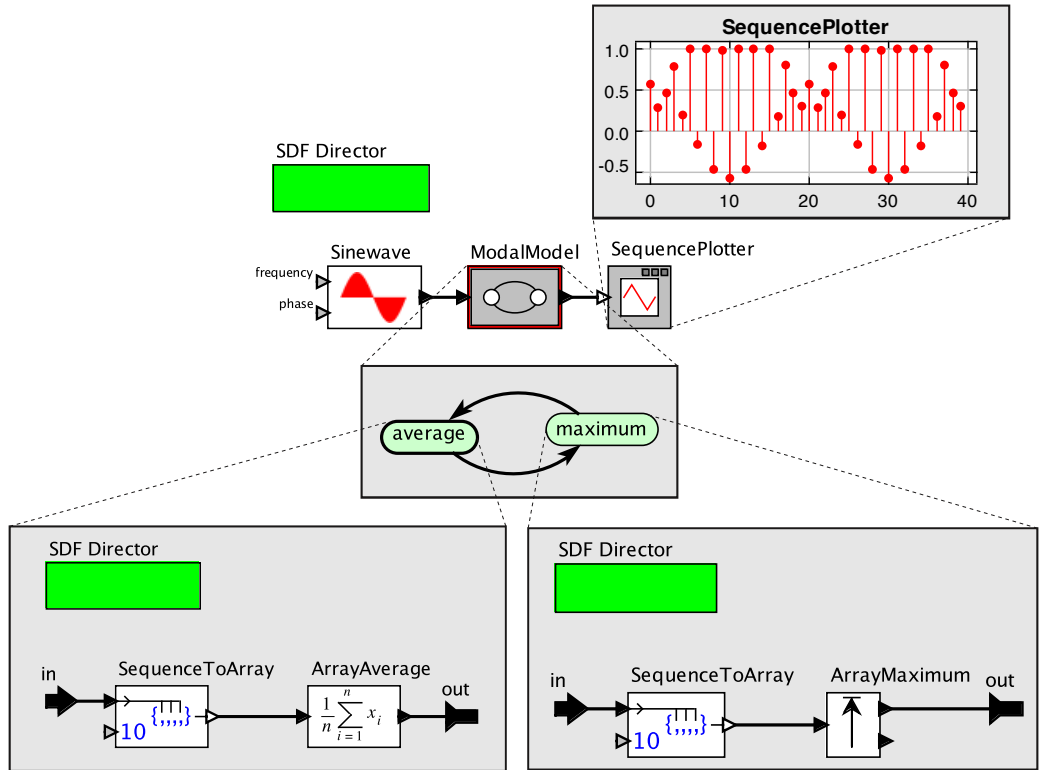


Figure 8.13: An SDF model where the **ModalModel** requires more than one token on its input in order to fire. [\[online\]](#)

In order for multirate modal models to work, it is necessary to propagate the production and consumption information from the refinements to the top-level SDF director. To do this, you must change the *directorClass* parameter of the **ModalModel** actor, as shown in Figure 8.14. The default director for a **ModalModel** makes no assertion about tokens produced or consumed, because it is designed to work with any Ptolemy II director, not specifically to work with SDF. The **MultirateFSMDirector**, by contrast, is designed to cooperate with SDF and convey production and consumption information across levels of the hierarchy.

In certain circumstances, it is even allowed for the consumption and production profiles of the refinements to differ in different modes. This has to be done very carefully, however. If



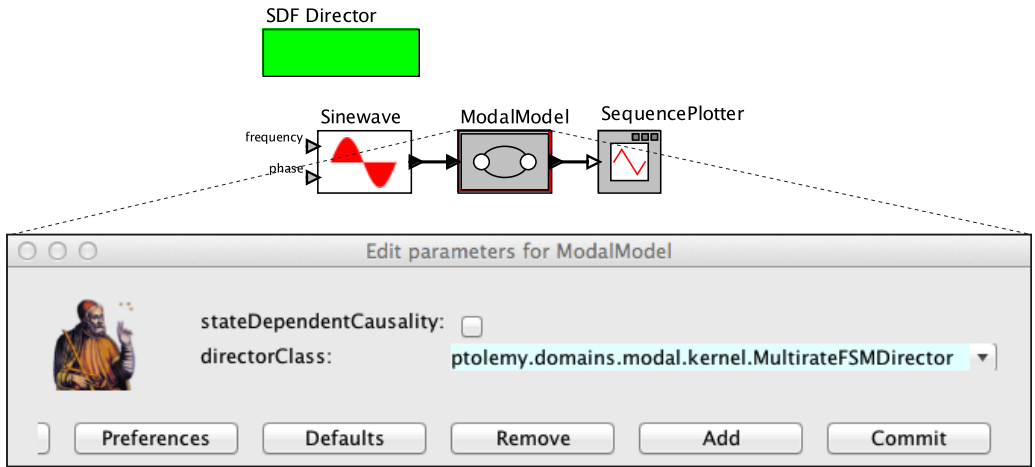


Figure 8.14: In order for the ModalModel to become an SDF actor with non-unit consumption and production on its inputs and outputs, it has to use the specialized MultirateFSMDirector.

different modes have different production and consumption profiles, then the ModalModel actor is actually not an SDF actor. Nevertheless, the SDF director will sometimes tolerate it.

**Example 8.9:** In Figure 8.13, for example, you can get away with changing the parameters of the [SequenceToArray](#) actor so that they differ in the two refinements. Behind the scenes, each time a transition is taken, the SDF director at the top level notices the change in the production consumption profile and compute a new schedule.

This is a major subtlety, however, with relying on the SDF director to recompute the schedule when an actor's production and consumption profile changes. Specifically, the SDF director will only recompute the schedule after a [complete iteration](#) has executed (see Section 3.1.1). If the production and consumption profile changes *in the middle of a complete iteration*, then the SDF director may not be able to finish the complete iteration.

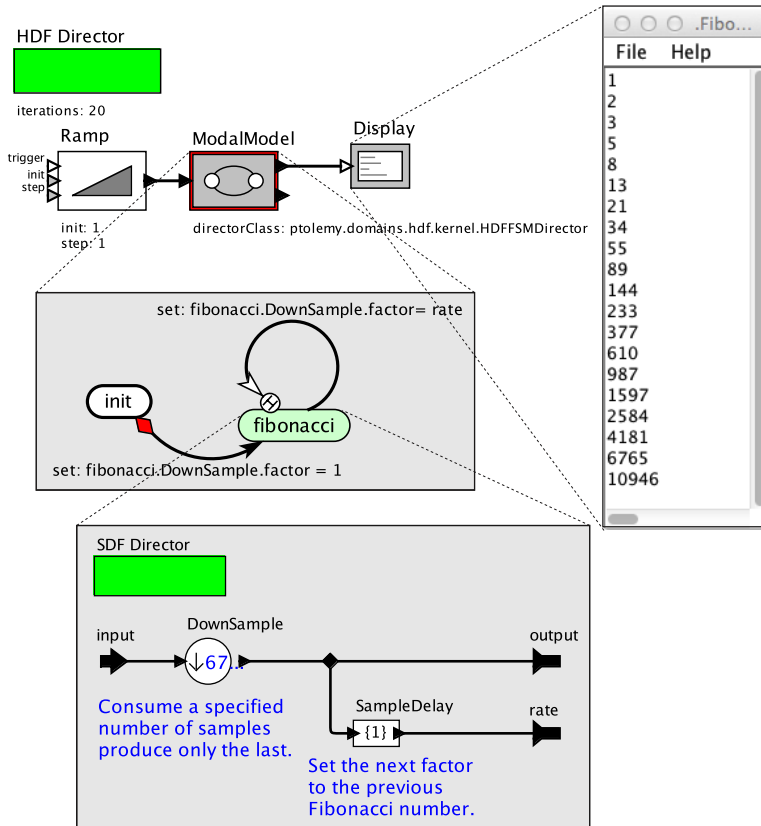


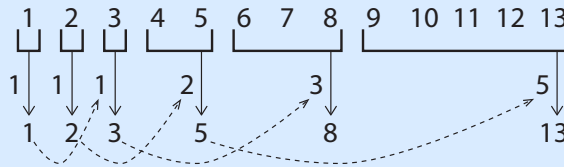
Figure 8.15: A model that calculates the Fibonacci sequence using a heterochronous dataflow model. This model is due to Ye Zhou. [\[online\]](#)

You may see errors about actors being unable to fire due to insufficient input tokens or errors about buffer sizes being inadequate.

The [heterochronous dataflow \(HDF\)](#) director provides a proper way to use multirate modal models with SDF. With this director, it is necessary to select the **HDFFSMDirector** for the *directorClass* of the ModalModel. These two directors cooperate to ensure that transitions of the FSM are taken *only after each complete iteration*. This combination is very expressive, as illustrated by the following examples.

**Example 8.10:** The model in Figure 8.15 uses HDF to calculate the **Fibonacci** sequence. In the Fibonacci sequence, each number is the sum of the previous two numbers. One way to generate such a sequence is to extract the Fibonacci numbers from a counting sequence (the natural numbers) by sampling each number that is a Fibonacci number. This can be done by a **DownSample** actor where the  $n$ -th Fibonacci number is generated by downsampling with a factor given by the  $(n-2)$ th Fibonacci number.

The calculation is illustrated in the following figure:



The top row shows the counting sequence from which we select the Fibonacci numbers. The downward arrows show the amount of downsampling required at each stage to get the next Fibonacci number. A downsampling operation simply consumes a fixed number of tokens and outputs only the last one. The first two downsampling factors are fixed at 1, but after that, the downsampling factor is itself a previously selected Fibonacci number.

In the model, the FSM changes the *factor* parameter of a **DownSample** actor each time it fires. The HDF director calculates a new schedule each time the downsampling rate is changed, and the new schedule outputs the next Fibonacci number.

**Example 8.11:** Another interesting example is shown in Figure 8.16. In this example, two increasing sequences of numbers are merged into one increasing sequence. In the initial state, the ModalModel consumes one token from each input and outputs the smaller of the two on its upper output port, and the larger of the two on its lower output port. The smaller, of course, is the first token of the merged sequence. The larger of the two is fed back to the input port named *previous* of the ModalModel.

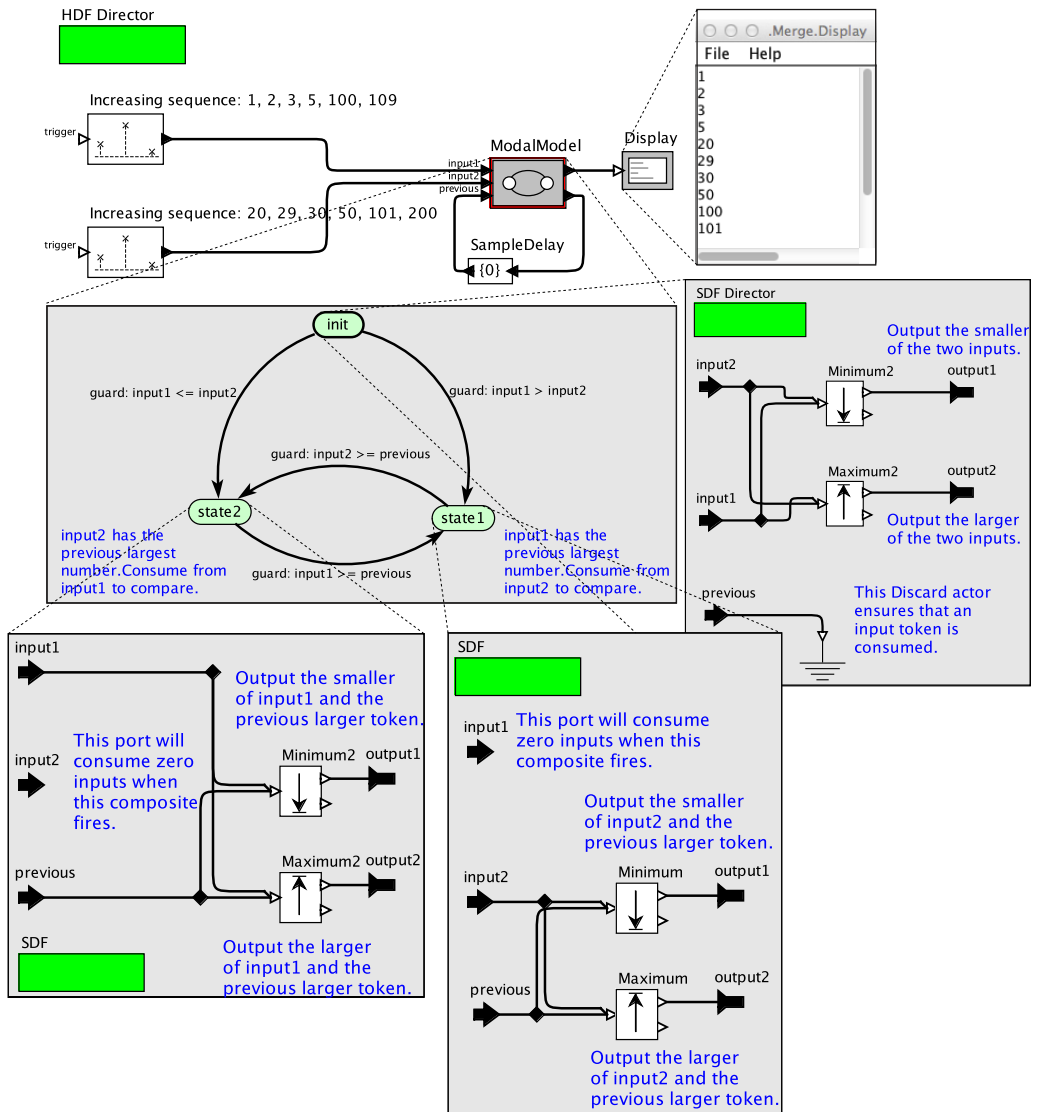


Figure 8.16: A heterochronous dataflow model that merges two numerically increasing streams into one numerically increasing stream (due to Ye Zhou and Brian Vogel). [\[online\]](#)

If the larger input came from *input1*, then the FSM transitions to *state1*. The refinement of this state does not read a token at all from *input1* (its consumption parameter will be zero). Instead, it reads one from *input2* and compares it against the value that was fed back.

If instead the initial larger input came from *input2*, then the FSM transitions to *state2*, which reads from *input1* and compares that input against the value that was fed back.

These examples demonstrate that HDF allows consumption and production rates to vary dynamically. In each case, the production and consumption profiles of the modal models are determined by the model inside the current state refinement.

The HDF model of computation was introduced by [Girault et al. \(1999\)](#), and the primary author of the HDF director is Ye Zhou. It has many interesting properties. Like SDF, HDF models can be statically analyzed for [deadlock](#) and [bounded buffers](#). But the MoC is much more flexible than SDF because data-dependent production and consumption rates are allowed. In order to use it, however, the model builder has to fully understand the notion of a [complete iteration](#), because this notion will constrain when transitions are taken in the FSM.

## 8.4.2 Synchronous-Reactive and Modal Models

The [SR](#) domain, explained in Chapter 5, can use modal models in very interesting ways. The key subtlety, compared with DE or dataflow, is that SR models may have feedback loops that require iterative convergence to a [fixed point](#). An example of such a feedback loop using FSMs is given in Section 6.4.

The key issue, then, is that when a ModalModel actor fires in the SR domain, some of its inputs may be unknown. This not the same as being absent. When an input is unknown, we don't know whether it is absent or present.

In order for modal models to be useful in feedback loops, it is important that the modal model be able to assert outputs even if some inputs are unknown. Asserting an output means specifying that it is either absent or present, and if it is present, optionally giving it a value. But the modal model has to be very careful to make sure that it does not make

assertions about outputs that become incorrect when the inputs become known. This constraint ensures that the actor is [monotonic](#).

If a ModalModel actor fires with some inputs unknown, then it must make a distinction between a transition that is known to not be enabled and one where it is not known whether it is enabled. If the guard refers to unknown inputs, then it cannot be known whether a transition is enabled. This makes it challenging, in particular, to assert that outputs are absent. It is not enough, for example, that no transition be enabled in the current state. Instead, the modal model has to determine that every transition that could potentially make the output present is *known to be not enabled*.

This constraint becomes subtle with chains of [immediate transitions](#), because all chains emanating from the current state have to be considered. If in any transition in such a chain has a guard that is not known to be true or false, then the possible outputs of all subsequent transitions have to be considered. If there are state refinements in chains of immediate transitions, then it becomes extremely difficult to assert that an output is absent when not all inputs are known.

Because of these subtleties, we recommend avoiding using modal models in feedback loops that rely on the modal model being able to assert outputs without knowing inputs. The resulting models can be extremely difficult to understand, so that even recognizing correct behavior becomes challenging.

### 8.4.3 Process Networks and Rendezvous

Modal models can be used with [PN](#) and [Rendezvous](#), but only in a rather simple way. When a ModalModel actor fires, it will read from each input port (in top-to-bottom order), which in each of these domains will cause the actor to block until an input is available. Thus, in both cases, a modal model always consumes exactly one token from each input. Whether it produces a token on the output, however, will depend on the FSM.

## 8.5 Time in Modal Models

Many Ptolemy II directors implement a timed [model of computation](#). The [ModalModel](#) actor and [FSMActor](#) are themselves untimed, but they include 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. The FSM appears to be reacting instantaneously.

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

**Example 8.12:** The model shown in Figure 8.17 uses the `timeout` function, described in Section 6.2.1, in the guard expression to trigger a transition every 1.0 time units. This is a spontaneous FSM with no input ports at all.

**Example 8.13:** The model in Figure 8.18 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

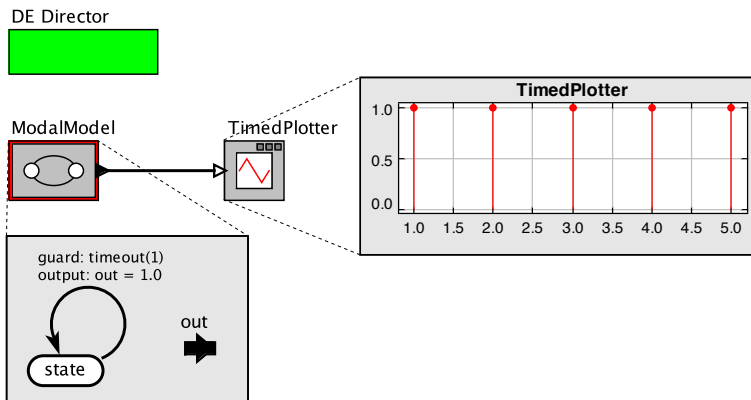


Figure 8.17: A spontaneous FSM, which produces output events that are not triggered by input events. [[online](#)]

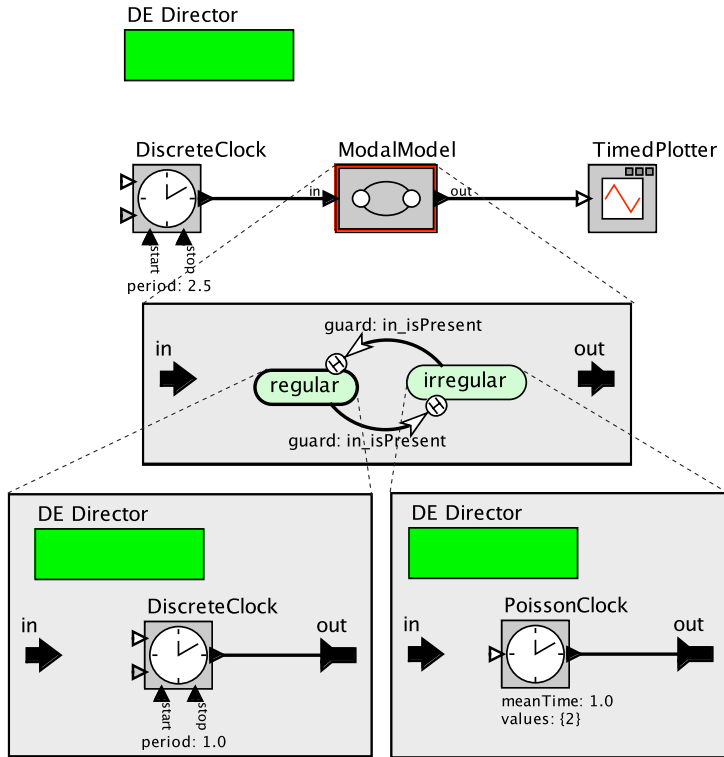


Figure 8.18: Another spontaneous modal model, which produces output events that are not triggered by input events. [\[online\]](#)

*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 8.19, with a shaded background showing the times during which it is in the two modes. The output events from the *ModalModel* are spontaneous; they are not necessarily produced in reaction to input events.

This example illustrates a number of subtle points about the use of time in modal models. In Figure 8.19, we see that two events are produced at time zero: one with a value of 1, and one with a value of 2. Why? The initial state is *regular*, and the execution policy



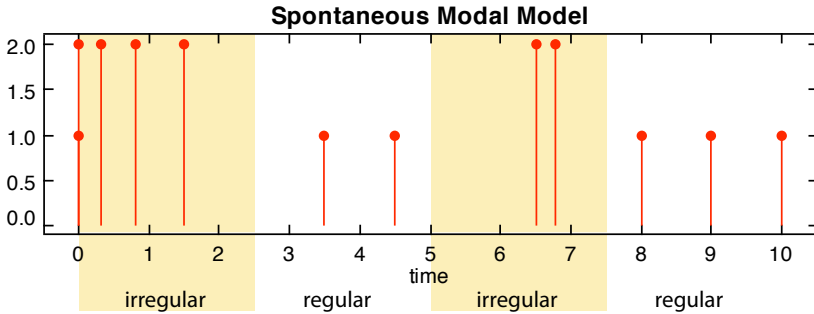


Figure 8.19: A plot of the output from one run of the model in Figure 8.18.

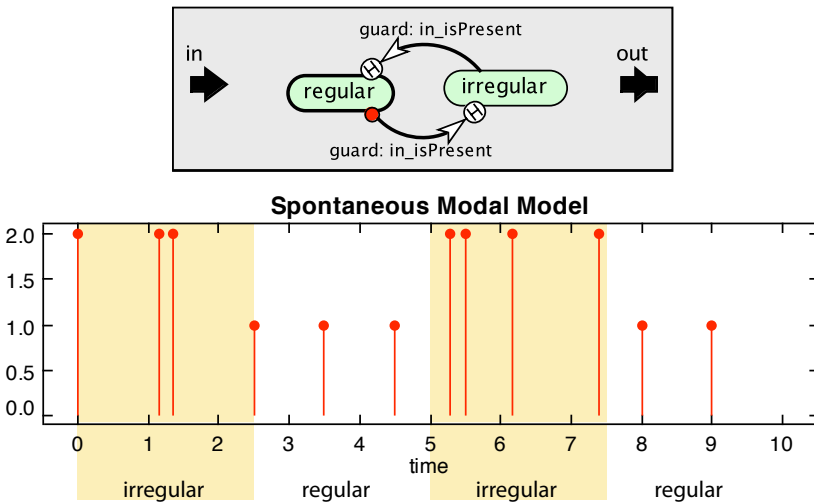


Figure 8.20: A variant of Figure 8.18 where a preemptive transition prevents the initial firing of the DiscreteClock.

described in section 8.3 explains that the refinement of that initial state is fired before guards are evaluated. That firing produces the first output of the `DiscreteClock`, with value 1. If we had instead used a preemptive transition, as shown in Figure 8.20, then that first output event would not appear.

The second event in Figure 8.19 (with value 2) at time zero is produced because the `PoissonClock`, by default, produces an event at the time when execution starts. 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.

As previously described, in Ptolemy II, the value of time is represented by a pair of numbers,  $(t, n) \in \mathbb{R} \times \mathbb{N}$ , rather than a single number (see Section 1.7). 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) that have elapsed since the start of execution of the model. The second of these numbers,  $n$ , is called the **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 time does not advance. It becomes active again at global time 2.5, but its local 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 `Timed-Plotter` and `CurrentTime`, can base their responses on either local time or global time, as specified in the parameter *useLocalTime* (which defaults to `false`). 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.

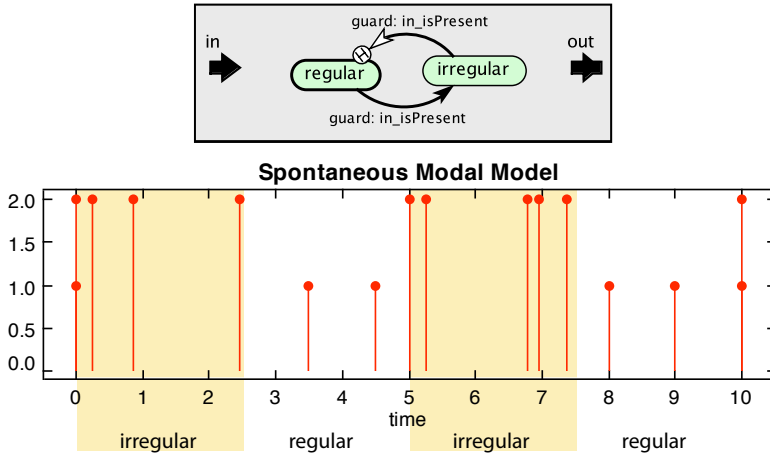


Figure 8.21: A variant of Figure 8.18 in which a reset transition causes the PoissonClock to produce events when the *irregular* mode is reactivated.

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 behavior follows from the same principle described above. 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.<sup>†</sup>

If an event is desired at time 5.0 (when the *irregular* mode becomes active) then a [reset transition](#) can be used, as shown in Figure 8.21. The `initialize` method of the PoissonClock causes an output event to be produced *at the time of the initialization*. A reset transition causes local time to match the environment time (where environment time is the time of the model in which the modal model resides; these distinct time values are discussed further in the next section). The time lag between local time and environment time goes to zero.

<sup>†</sup>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.

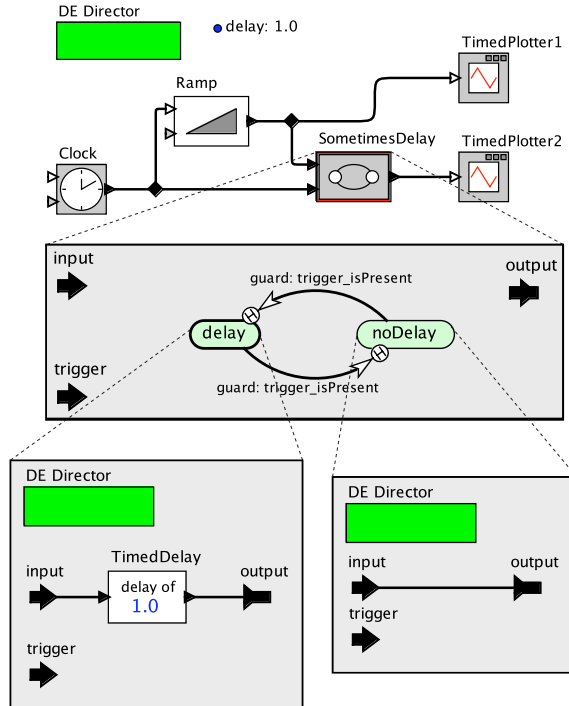


Figure 8.22: A modal model that switches between delaying the input by one time unit and not delaying it. [\[online\]](#)

### 8.5.1 Time Delays in Modal Models

The use of time delays in a modal model can produce several interesting effects, as shown in the example below.

**Example 8.14:** Figure 8.22 shows a model that produces a counting sequence of events spaced one time unit apart. The model uses two modes, *delay* and *noDelay*, to delay every other event by one time unit. In the *delay* mode, a **TimeDelay** actor imposes a 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 8.23. Notice that the value 0 is produced at time 2. Why?

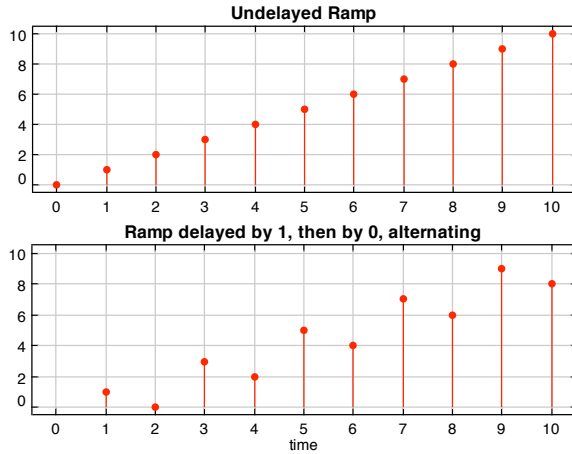


Figure 8.23: The result of executing the model in Figure 8.22.

The model begins in the *delay* mode, which receives the first input. This input has value 0. However, the modal model transitions immediately out of that mode to the *noDelay* mode, with zero elapsed time. The *delay* mode becomes active again at time 1, but its local time is still time 0. Therefore, it must delay the input with value 0 by one time unit, to time 2. Its output is produced at time 2, just before transitioning to the *noDelay* mode again.

## 8.5.2 Local Time and Environment Time

As shown in the previous examples, modal models may have complex behaviors, particularly when used in timed domains. It is useful to step back and ponder the principles that govern the design choices in the Ptolemy II implementation 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 to answer because the desired behavior depends on the application.

In a modal model, there are potentially four distinct times that can affect the behavior of the model: local time, environment time, global time, and real time. **Local time** is the time within the mode (or other local actor). **Environment time** is the time within the model that contains the modal model. **Global time** is the model time at the top level of a hierarchical model. **Real time** is the wall-clock time outside the computer executing the model.

In Ptolemy II, the guiding principle is that when a mode is inactive, local time stands still, while environment time (and 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 by an **accumulated suspend time** or **lag** that is non-decreasing.

The time lag in a mode refinement is initially the difference between the start time of the environment of the modal model and the start time of the mode refinement (normally this difference is zero, but it can be non-zero, as explained below in Section 8.5.3). The lag increases each time the mode becomes inactive, but within the mode, time seems uninterrupted.

When an event crosses a hierarchical boundary into or out of the mode, its time stamp is adjusted by the amount of the lag. That is, when a mode refinement produces an output event, if the local time of that event is  $t$ , then the time of event that appears at the output of the modal model is  $t + \tau$ , where  $\tau$  is the accumulated suspend time.

A key observation is that when a submodel is inactive, it does not behave in the same manner as a submodel that receives inputs and then ignores them. This point is illustrated by the model of Figure 8.24. This model shows two instances of **DiscreteClock**, labeled **DiscreteClock1** and **DiscreteClock2**, which have the same parameter values. **DiscreteClock2** is inside a modal model labeled **ModalClock**, and **DiscreteClock1** is not inside a 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 the output of **DiscreteClock1**. 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 remember 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.

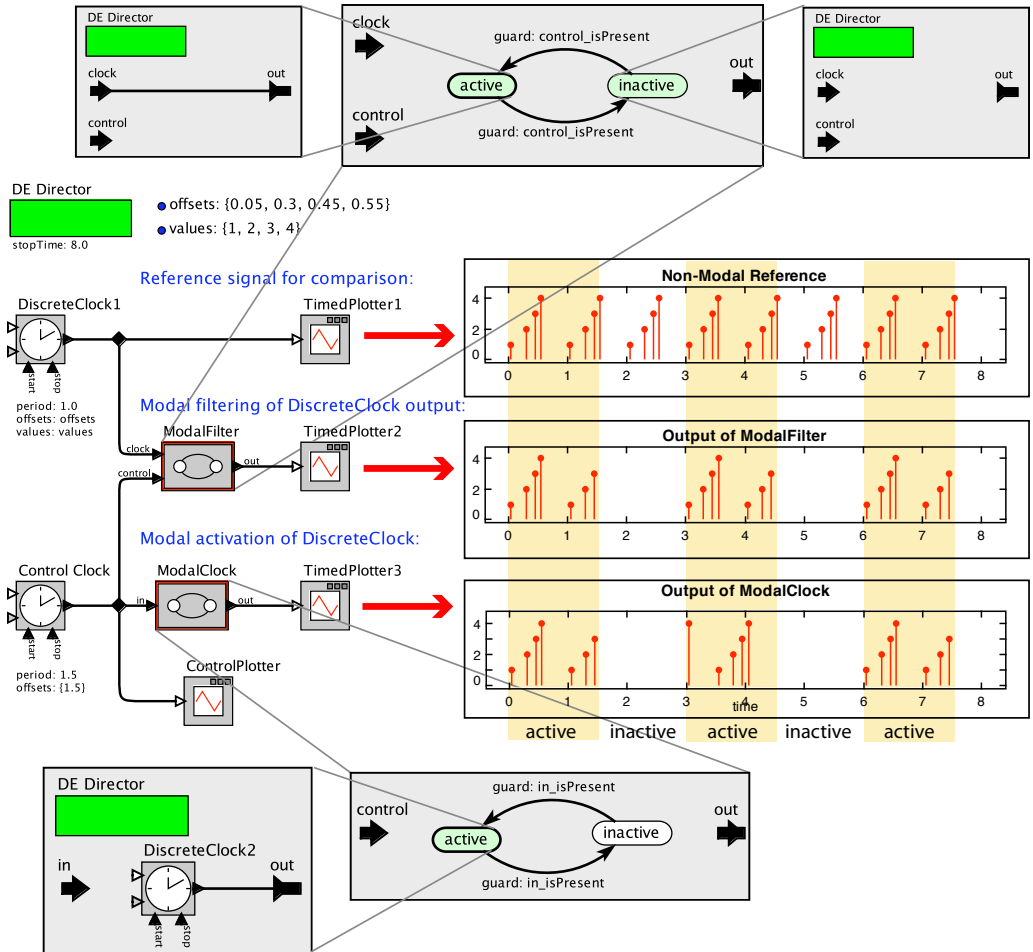


Figure 8.24: A model that illustrates that putting a timed actor such as DiscreteClock inside a modal model is not the same as switching between observing and not observing its output. [\[online\]](#)

### 8.5.3 Start Time in Mode Refinements

Usually, when we execute a timed model, we want it to execute over a specified time interval, from a start time to a stop time. By default, execution starts at time zero, but the *startTime* parameter of the director can specify a different start time.

When a DE model is inside a [mode refinement](#), however, by default, the start time in the submodel is the time at which is initialized. Normally, this is the same as the start time of the enclosing model, but when a [reset transition](#) is used, then the submodel may be reinitialized at an arbitrary time.

When a submodel is reinitialized by a reset transition, occasionally it is useful to restart execution at a particular time in the past. This can be accomplished by changing the *startTime* parameter of the inside DEDirector to something other than the default (which is blank, interpreted as the time of initialization).

**Example 8.15:** This use of the *startTime* parameter is illustrated in Figure 8.25, which implements a **resettable timer**. This example has a modal model with a single mode and a single reset transition. The *startTime* of the inside DEDirector is set to 0.0, so that each time the reset transition is taken, the execution of the submodel begins again at time 0.0.

In this example, a [PoissonClock](#) generates random reset events that cause the reset transition to be taken. The refinement of the mode has a [SingleEvent](#) actor that is set to produce an event at time 0.5 with value 2.0. As shown in the plot, this modal model produces an output event 0.5 time units after receiving an input event, unless it receives a second input event during the 0.5 time unit interval. The second event resets the timer to start over. Thus, the event at time 1.1 does not result in any output because the event at time 1.4 resets the timer.

When a reset transition is taken and the destination mode refinement has a specified *start-Time*, the [accumulated suspend time](#) increases by  $t$ , where  $t$  is the current time of the enclosing model. After the reset transition is taken, the lag between local time and global time is larger by  $t$  than it was before the transition was taken.



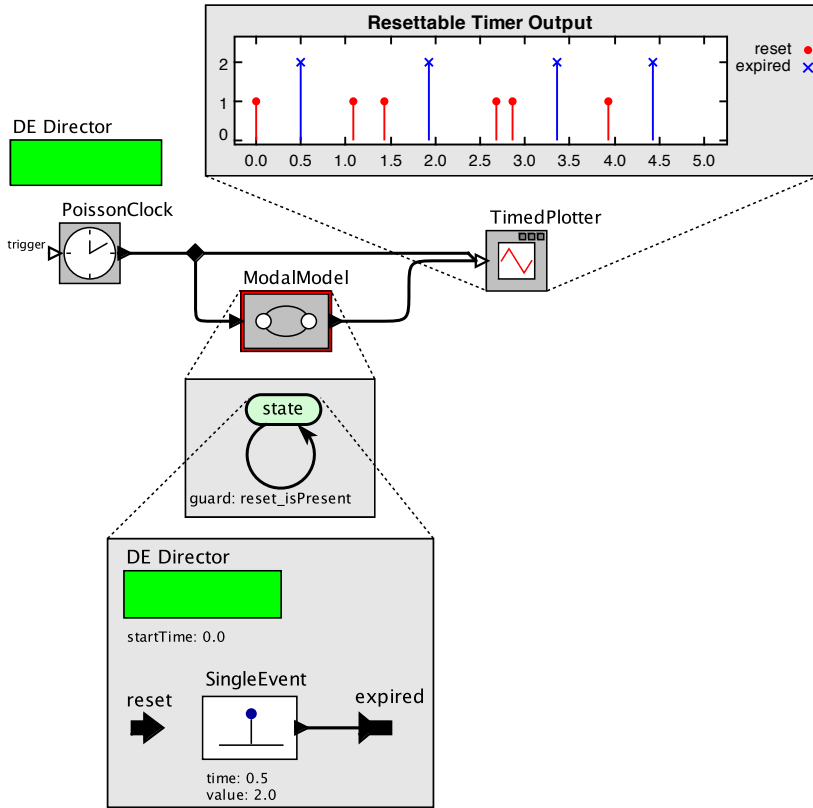


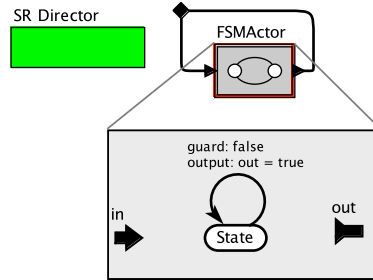
Figure 8.25: A resettable timer implemented by using a reset transition to restart a submodel at time zero. [\[online\]](#)

## 8.6 Summary

FSMs and modal models in Ptolemy II provide a very expressive way to build up complex modal behaviors. As a consequence of this expressiveness, it takes some practice to learn to use them well. This chapter 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 documents are accessible when running Vergil by right clicking and selecting *Documentation*.

## Exercises

1. In the following model, the only signal (going from the output of **FSMActor** back to its input) has value `absent` at all ticks.



Explain why this is correct.

2. Construct a variant of the example in Figure 8.1 where the *clean* and *noisy* states share the same refinement, yet the behavior is the same.
3. This problem explores the use of the SDF model of computation together with modal models to improve expressiveness. In particular, you are to implement a simple run-length coder using no director other than SDF, leveraging modal models with state refinements. Specifically, given an input sequence, such as

$$(1, 1, 2, 3, 3, 3, 3, 4, 4, 4)$$

you are to display a sequence of pairs  $(i, n)$ , where  $i$  is a number from the input sequence and  $n$  is the number of times that number is repeated consecutively. For the above sequence, your output should be

$$((1, 2), (2, 1), (3, 4), (4, 3)).$$

Make sure your solution conforms with SDF semantics. Do not use the non-SDF techniques of section 3.2.3. Note that this pattern arises commonly in many coding applications, including image and video coding.