



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>.

## Part II

# Models of Computation

This part of this book introduces a few of the [models of computation](#) used in system design, modeling, and simulation. This is not a comprehensive set, but rather a representative set. The particular MoCs described here are relatively mature, well understood, and fully implemented. Most of these are available in other tools besides Ptolemy II, though no other tool provides all of them.

# Dataflow

Edward A. Lee, Stephen Neuendorffer, Gang Zhou

## Contents

<b>3.1</b>	<b>Synchronous Dataflow</b>	<b>93</b>
3.1.1	Balance Equations	95
	<i>Sidebar: SDF Schedulers</i>	100
	<i>Sidebar: Frequency Analysis</i>	101
3.1.2	Feedback Loops	103
3.1.3	Time in Dataflow Models	105
	<i>Sidebar: Multirate Dataflow Actors</i>	106
	<i>Sidebar: Signal Processing Actors</i>	107
	<i>Sidebar: Dynamically Varying Rates</i>	108
	<i>Sidebar: StreamIt</i>	109
	<i>Sidebar: Other Variants of Dataflow</i>	110
	<i>Sidebar: Petri Nets</i>	111
	<i>Sidebar: Logic Actors</i>	112
<b>3.2</b>	<b>Dynamic Dataflow</b>	<b>113</b>
3.2.1	Firing Rules	113
3.2.2	Iterations in DDF	118
	<i>Sidebar: Token Flow Control Actors</i>	119
	<i>Sidebar: Structured Dataflow</i>	120
3.2.3	Combining DDF with Other Domains	123
	<i>Sidebar: Defining a DDF Iteration</i>	124
	<i>Sidebar: String Manipulation Actors</i>	125
	<i>Sidebar: Building Regression Tests</i>	126
<b>3.3</b>	<b>Summary</b>	<b>127</b>
	<i>Sidebar: IO Actors</i>	128
	<b>Exercises</b>	<b>129</b>

Ptolemy II was created to enable heterogeneous models to be developed and simulated together as part of an overall system model. As discussed in previous chapters, a key innovation in Ptolemy II is that, unlike other design and modeling environments, Ptolemy II supports multiple **models of computation** that are tailored to specific types of modeling problems. These models of computation define how the model will behave and are determined by the **director** that is used within that model. In Ptolemy II terminology, the director realizes a **domain**, which is an implementation of a model of computation. Thus, the director, domain, and model of computation are all tied together; when you construct a model that contains an SDFDirector (a synchronous dataflow director), for example, you have constructed a model “in the SDF domain,” using the SDF model of computation.

This chapter describes the **dataflow domains** that are currently available in Ptolemy II, which include synchronous (static) and dynamic dataflow models. Dataflow domains are appropriate for applications that involve processing streams of data values. These streams can flow through sequences of actors that transform them in some way. Such models are often called **pipe and filter** models, because the connections between actor are analogous to pipes that carry flows, and the actors are analogous to filters the change the flows in some way. Dataflow domains mostly ignore time, although SDF is capable of modeling streams with uniformly spaced time between iterations. Subsequent chapters discuss other domains and their selection and use.

## 3.1 Synchronous Dataflow

The **Synchronous dataflow (SDF)** domain, also called **static dataflow**,<sup>1</sup> was introduced by Lee and Messerschmitt (1987b), and is one of the first domains (or **models of computation**) developed for Ptolemy II. It is a specific type of **dataflow** model. In dataflow models, actors begin execution (they are **fired**) when their required data inputs become available. SDF is a relatively simple case of dataflow; the order in which actors are exe-

<sup>1</sup>The term “synchronous dataflow” can cause confusion because it is not synchronous in the sense of SR, considered in Chapter 5. There is no global clock in SDF models, and actors are fired asynchronously. For this reason, some authors prefer the term “static dataflow.” This does not avoid all confusion, however, because Dennis (1974) had previously coined the term “static dataflow” to refer to dataflow graphs where buffers could hold at most one token. Since there is no way to avoid a collision of terminology, we stick with the original “synchronous dataflow” terminology used in the literature. The term SDF arose from a signal processing concept, where two signals with sample rates that are related by a rational multiple are deemed to be synchronous.

cuted is static, and does not depend on the data that is processed (the values of the **tokens** that are passed between actors).

In a **homogeneous SDF** model, an actor fires when there is a token on each of its input ports and produces a token on each output port. In this case, the director simply has to ensure that each actor fires after the actors that supply it with data, and an **iteration** of the model consists of one firing of each actor. Most of the examples in Chapter 2 were homogeneous SDF models.

Not all actors produce and consume a single token each time they are fired, however; some require multiple input tokens before they can be fired and produce multiple output tokens. The SDF scheduler, which is responsible for determining the order in which actors are executed, supports more complex models than homogeneous SDF. It is capable of scheduling the execution of actors with arbitrary data rates, as long as these rates are given by specifying the number of tokens consumed and produced by the firing of each actor on each port.

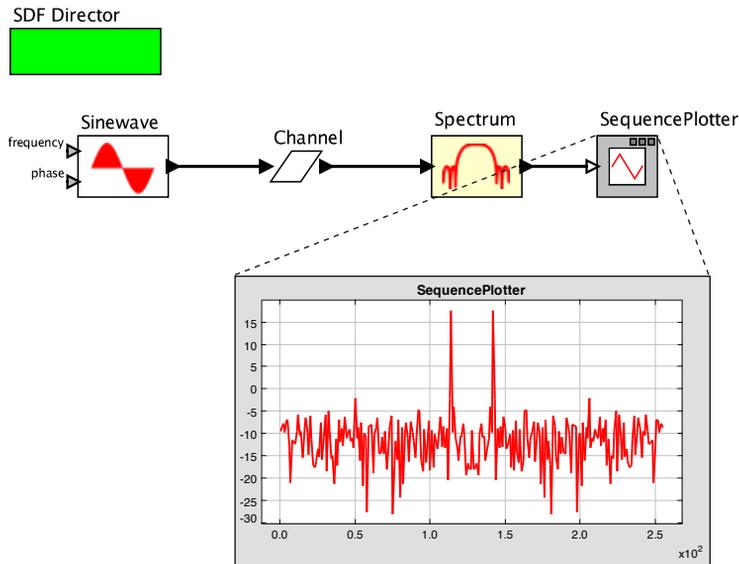


Figure 3.1: A multirate SDF model. The Spectrum actor requires 256 tokens to fire, so one iteration of this model requires 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum. [[online](#)]

**Example 3.1:** One example of an actor that requires multiple input tokens to fire is the **Spectrum** actor (see box on page 101). Figure 3.1 shows a system that computes the spectrum of the same noisy sine wave that we constructed in Figure 2.20. The Spectrum actor has a single parameter that specifies the order of the fast Fourier transform (**FFT**) used to calculate the spectrum. Figure 3.1 shows the output of the model with *order* set to 8 and the number of iterations set to 1. (See Chapter 17, Section 17.2 to improve the labeling of the plot.)

When the *order* parameter is set to 8, the Spectrum actor requires  $2^8$  (256) input samples to fire, and produces  $2^8$  output samples. In order for the Spectrum actor to fire once, the actors that supply its input data, Sinewave and Channel, must each fire 256 times. The SDF director extracts this relationship and defines one iteration of the model to consist of 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum.

This example implements a **multirate** model; that is, the firing rates of the actors are not identical. In particular, the Spectrum actor executes at a different rate than the other actors. It is common for the execution of a multirate model to consist of exactly one iteration. The director determines how many times to fire each actor in an iteration using balance equations, as described in the next section.

### 3.1.1 Balance Equations

Consider a single connection between two actors, *A* and *B*, as shown in Figure 3.2. The notation here means that when *A* fires, it produces *M* tokens on its output port, and when *B* fires, it consumes *N* tokens on its input port. *M* and *N* are nonnegative integers. Suppose that *A* fires  $q_A$  times and *B* fires  $q_B$  times. All tokens that *A* produces are

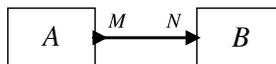


Figure 3.2: SDF actor *A* produces *M* tokens when it fires, and actor *B* consumes *N* tokens when it fires.

consumed by  $B$  if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \quad (3.1)$$

Given values  $q_A$  and  $q_B$  satisfying (3.1), the system remains in balance;  $A$  produces exactly as many tokens as  $B$  consumes.

Suppose we wish to process an arbitrarily large number of tokens, a situation that is typical of streaming applications. A naive strategy is to fire actor  $A$  an arbitrarily large number  $q_A$  times, and then fire actor  $B$   $q_B$  times, where  $q_A$  and  $q_B$  satisfy (3.1). This strategy is naive, however, because it requires storing an arbitrarily large number of unconsumed tokens in a buffer. A better strategy is to find the smallest positive  $q_A$  and  $q_B$  that satisfy (3.1). Then we can construct a schedule that fires actor  $A$   $q_A$  times and actor  $B$   $q_B$  times, and we can repeat this schedule as many times as we like without requiring any more memory to store unconsumed tokens. That is, we can achieve an **unbounded execution** (an execution processes an arbitrarily large number of tokens) with **bounded buffers** (buffers with a bound on the number of unconsumed tokens). In each round of the schedule, called an **iteration**, actor  $B$  consumes exactly as many tokens as actor  $A$  produces.

**Example 3.2:** Suppose that in Figure 3.2,  $M = 2$  and  $N = 3$ . There are many possible solutions to the corresponding balance equation, one of which is  $q_A = 3$  and  $q_B = 2$ . With these values, the following schedule can be repeated forever:

$$A, A, A, B, B.$$

An alternative schedule could also be used:

$$A, A, B, A, B.$$

In fact, the latter schedule has an advantage in that it requires less memory for storing intermediate tokens;  $B$  fires as soon as there are enough tokens, rather than waiting for  $A$  to complete its entire cycle.

Another solution to (3.1) is  $q_A = 6$  and  $q_B = 4$ . This solution includes more firings in the schedule than are strictly needed to keep the system in balance.

The equation is also satisfied by  $q_A = 0$  and  $q_B = 0$ , but if the number of firings of actors is zero, then no useful work is done. Clearly, this is not a solution we want. Negative solutions are also not meaningful.

The SDF director, by default, finds the least positive integer solution to the balance equations, and constructs a schedule that fires the actors in the model the requisite number of times, given by this solution. An execution sequence that fires the actors exactly as many times as specified by this solution is called a **complete iteration**.

In a more complicated SDF model, every connection between actors results in a balance equation. Hence, the model defines a system of equations, and finding the least positive integer solution is not entirely trivial.

**Example 3.3:** Figure 3.3 shows a network with three SDF actors. The connections result in the following system of balance equations:

$$\begin{aligned}q_A &= q_B \\2q_B &= q_C \\2q_A &= q_C.\end{aligned}$$

The least positive integer solution to these equations is  $q_A = q_B = 1$ , and  $q_C = 2$ , so the following schedule can be repeated forever to get an unbounded execution with bounded buffers,

$$A, B, C, C.$$

The balance equations do not always have a non-trivial solution, as illustrated in the following example.

**Example 3.4:** Figure 3.4 shows a network with three SDF actors where the only solution to the balance equations is the trivial one,  $q_A = q_B = q_C = 0$ . A conse-

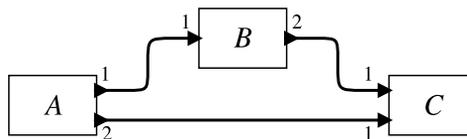


Figure 3.3: A consistent SDF model.

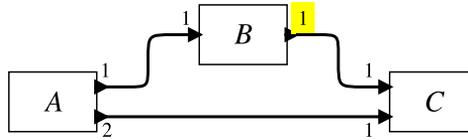


Figure 3.4: An inconsistent SDF model.

quence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

An SDF model that has a non-zero solution to the balance equations is said to be **consistent**. If the only solution is zero, then it is **inconsistent**. An inconsistent model has no unbounded execution with bounded buffers.

Lee and Messerschmitt (1987b) showed that if the balance equations have a non-zero solution, then they also have a solution where  $q_i$  is a nonnegative integer for all actors  $i$ . Moreover, for connected models (where there is a communication path between any two actors), they give a procedure for finding the least positive integer solution. Such a procedure forms the foundation for a scheduler for SDF models.

**Example 3.5:** Figure 3.5 shows an SDF model that makes extensive use of the multirate capabilities of SDF. The **AudioCapture** actor captures sound from the microphone on the machine on which the models run, producing a sequence of samples at a default rate of 8,000 samples per second. The **Chop** actor extracts chunks of 128 samples from each input block of 500 samples (see box on page 106). The **Spectrum** actor computes the power spectrum, which measures the power as a function of frequency (see box on page 101). The two **SequenceToArray** actors (box on page 106) construct arrays that are then plotted using **ArrayPlotter** actors (see Chapter 17). The particular plots that are shown are the response to a whistle. Notice the peaks in the spectrum at about 1,700 Hz and -1,700 Hz.

The SDF director in this model figures out that the AudioCapture actor needs to fire 500 times for each firing of Chop, Spectrum, and SequenceToArray, and the plotters.

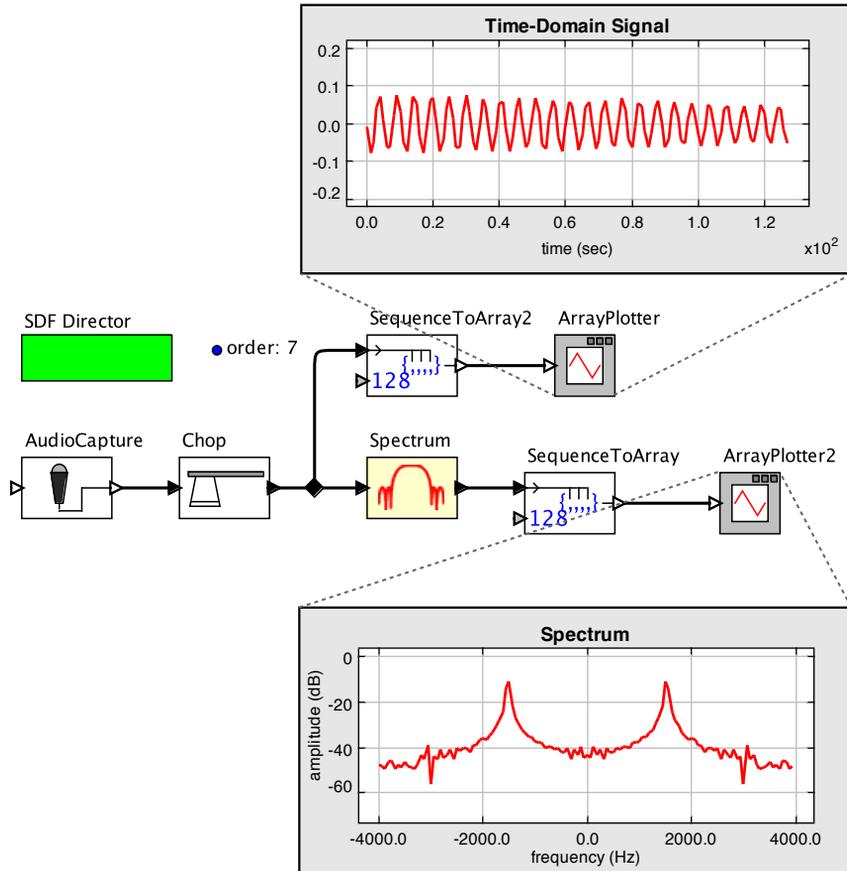


Figure 3.5: Model that computes the power spectrum of the audio signal captured from the microphone. The plots here show a whistle at about 1,700 Hz. [\[online\]](#)

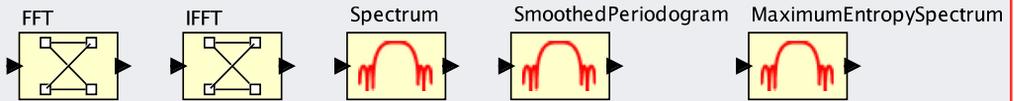
### Sidebar: SDF Schedulers

A key advantage of using SDF is that there may be many possible schedules for a given model, including some that execute actors in parallel. In this case, actors in the dataflow graph can be mapped onto different processors in a multicore or distributed architecture for improved performance. Lee and Messerschmitt (1987a) adapt classical job-shop scheduling algorithms (Coffman, 1976), particularly those introduced by Hu (1961), to SDF by converting the SDF graph into an acyclic precedence graph (APG). Lee and Ha (1989) classify scheduling strategies into **fully dynamic scheduling** (all scheduling decisions are made at run time), **static assignment scheduling** (all decisions except the assignment to processors are made at run time), **self-timed scheduling** (only the timing of an actor firing is determined at run time), and **fully-static scheduling** (every aspect of the schedule is determined before run time). Sih and Lee (1993a) extend the job-shop scheduling techniques to account for interprocessor communication costs (see also Sih and Lee (1993b)). Pino et al. (1994) show how to construct schedules for heterogeneous multiprocessors. Falk et al. (2008) give a parallel scheduling strategy based on clustering and demonstrate significant performance gains for multimedia applications.

In addition to parallel scheduling strategies, other scheduling optimizations are also useful (see Bhattacharyya et al. (1996b) for a collection of these). Ha and Lee (1991) relax the constraints of SDF to allow data-dependent iterative firing of actors (a technique called **quasi-static scheduling**). Bhattacharyya and Lee (1993) develop optimized schedules for iterated invocations of actors (see also Bhattacharyya et al. (1996a)). Bhattacharyya et al. (1993) optimize schedules to minimize memory usage and later apply these optimizations to code generation for embedded processors (Bhattacharyya et al., 1995). Murthy and Bhattacharyya (2006) collect algorithms that minimize the use of memory through scheduling and buffer sharing. Geilen et al. (2005) show that model checking techniques can be used to optimize memory. Stuijk et al. (2008) explore the tradeoff between throughput and buffering (see also Moreira et al. (2010)). Sriram and Bhattacharyya (2009) develop scheduling optimizations that minimize the number of synchronization operations in parallel SDF. Synchronization ensures that an actor does not fire before it receives the data it needs to fire. However, synchronization is not required if a previous synchronization already provides assurance that the data are present. By manipulating the schedule, one can minimize the number of required synchronization points.

### Sidebar: Frequency Analysis

The SDF domain is particularly useful for signal processing. One of the basic operations in signal processing is to convert a time domain signal into a frequency domain signal and vice versa (see [Lee and Varaiya \(2011\)](#)). Actors that support this operation are found in the `Actors→SignalProcessing→Spectrum` library, and shown below:



- **FFT** and **IFFT** calculate the discrete Fourier transform (DFT) and its inverse, respectively, of a signal using the fast Fourier transform algorithm. The *order* parameter specifies the number of input tokens that are used for each FFT calculation. It is a “radix two” algorithm, which implies that the number of tokens is required to be a power of two, and the *order* parameter specifies the exponent. For example, if *order*=10, then the number of input tokens used for each firing is  $2^{10} = 1024$ . The remaining actors implement various spectral estimation algorithms, and are all **composite actors** that use FFT as a component. These algorithms output signal power in decibels (dB) as a function of frequency. The output frequency ranges from  $-f_N$  to  $f_N$ , where  $f_N$  is the Nyquist frequency (half the sampling frequency). That is, the first half of the output represents negative frequencies and the second half represents positive frequencies.
- **Spectrum** is the simplest of the spectral estimators. It calculates the FFT of the input signal and converts the result to a power measurement in dB.
- **SmoothedPeriodogram** calculates a power spectrum by first estimating the autocorrelation of the input. This approach averages the inputs and is less sensitive to noise.
- **MaximumEntropySpectrum** is a parametric spectral estimator; it uses the Levinson-Durbin algorithm to construct the parameters of autoregressive (AR) models that could plausibly have generated the input signal. It then selects the model that maximizes the entropy (see [Kay \(1988\)](#)). It is the most sophisticated of the spectral estimators and typically produces the smoothest estimates.

Outputs from the three spectral estimators are compared in [Figure 3.6](#), where the input consists of three sinusoids in noise. Choosing the right spectral estimator for an application is a sophisticated topic, beyond the scope of this book.

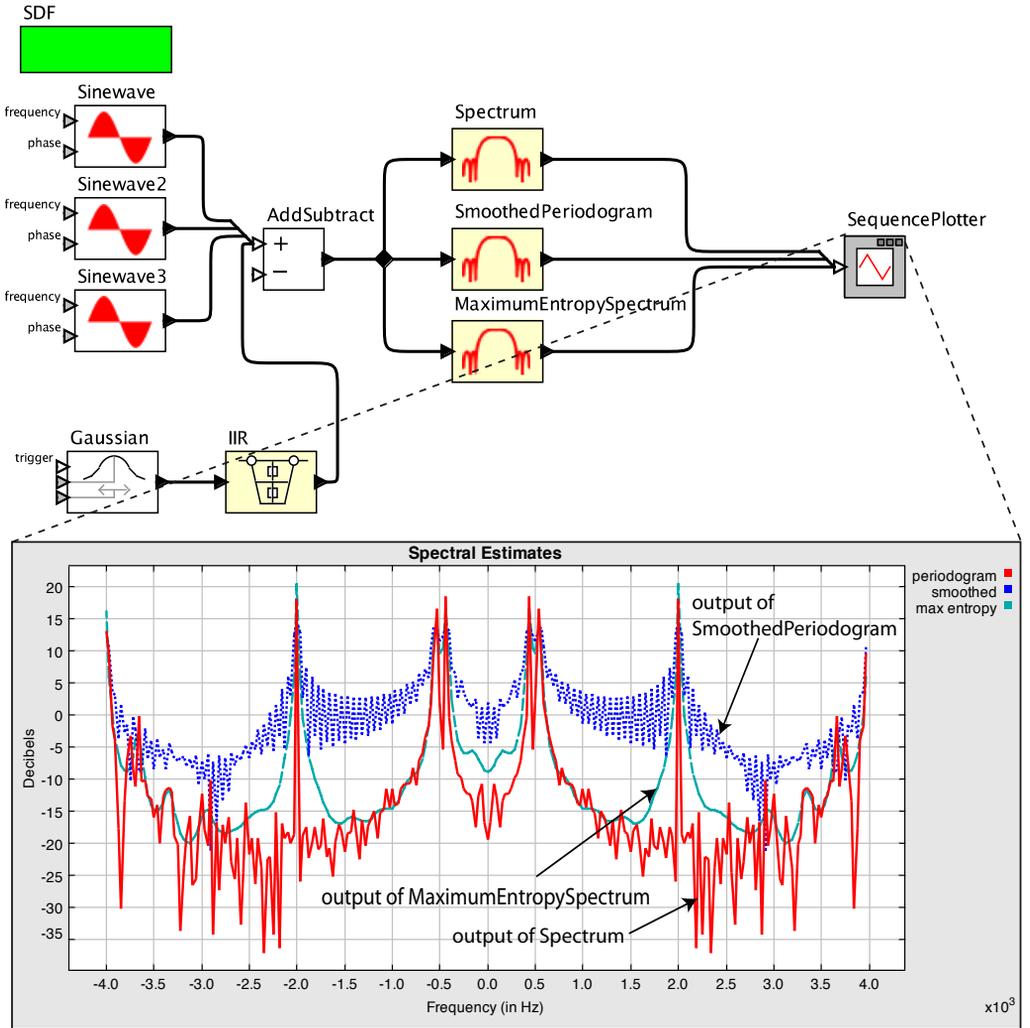


Figure 3.6: Comparison of three spectral estimation techniques described in the box on page 101. [online]

### 3.1.2 Feedback Loops

A feedback loop in SDF must include at least one instance of the **SampleDelay** actor (found in the `FlowControl`→`SequenceControl` sublibrary). Without this actor, the loop would **deadlock**; actors in the feedback loop would be unable to fire because they depend on each other for tokens. The `SampleDelay` actor resolves this problem by producing initial tokens on its output before the model begins firing. The initial tokens are specified by the *initialOutputs* parameter, which defines an array of tokens. These initial tokens enable downstream actors to fire and break the circular dependencies that would otherwise result from a feedback loop.

**Example 3.6:** Consider the model in Figure 3.7. This **homogeneous SDF** model generates a counting sequence using a feedback loop. The `SampleDelay` actor begins the process by producing a token with value of 0 on its output. This token, together with a token from the `Const` actor, enables the `AddSubtract` actor to fire. The output of that actor enables the next firing of `SampleDelay`. After the initial firing, the `SampleDelay` copies its input to its output unchanged.

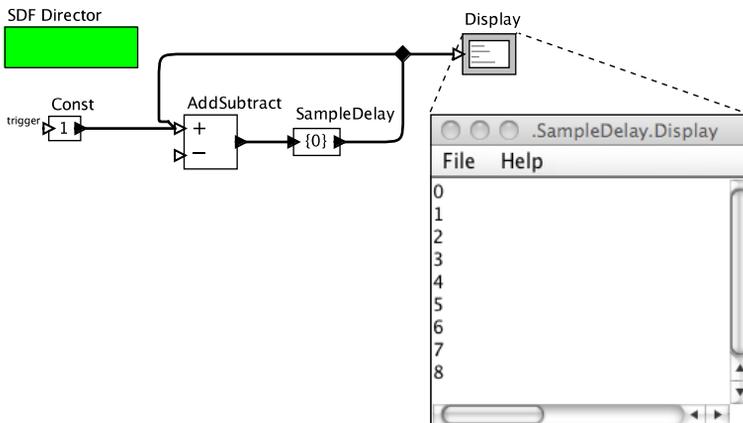


Figure 3.7: An SDF model with a feedback loop must have at least one instance of the `SampleDelay` actor in it. [[online](#)]

**Consistency** is sufficient to ensure **bounded buffers**, but it is not sufficient to ensure that an **unbounded execution** exists. The model may deadlock even if it is consistent. The SDF director analyzes a model for both consistency and deadlock. To allow feedback, it treats **delay actors** differently than other actors. A delay actor is able to produce initial output tokens before it receives any input tokens. It subsequently behaves like a normal SDF actor, consuming and producing a fixed number of tokens on each firing. In the SDF domain, the initial tokens are understood to be initial conditions for execution rather than part of the execution itself. Thus, the scheduler will ensure that all initial tokens are produced before the SDF execution begins. Conceptually, the SampleDelay actor could be replaced by initial tokens placed on a feedback connection.

**Example 3.7:** Figure 3.8 shows an SDF model with initial tokens on a feedback loop. The balance equations are

$$\begin{aligned} 3q_A &= 2q_B \\ 2q_B &= 3q_A. \end{aligned}$$

The least positive integer solution exists and is  $q_A = 2$ , and  $q_B = 3$ , so the model is consistent. With four initial tokens on the feedback connection, as shown, the following schedule can be repeated forever,

$$A, B, A, B, B.$$

This schedule starts with actor  $A$ , because at the start of execution, only actor  $A$  can fire, because actor  $B$  does not have sufficient tokens. When  $A$  fires, it consumes three tokens from the four initial tokens, leaving one behind. It sends three tokens to  $B$ . At this point, only  $B$  can fire, consuming two of the three tokens sent by  $A$ ,

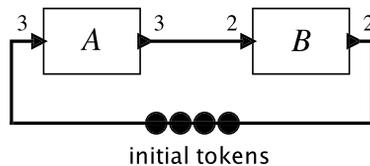


Figure 3.8: An SDF model with initial tokens on a feedback loop. In Ptolemy II, these initial tokens would be provided by a SampleDelay actor.

and producing two more tokens on its output. At this point, actor *A* can fire again, because there are exactly three tokens on its input. It will consume all of these and produce three tokens. At this point, *B* has four tokens on its input, enabling two firings. After those two firings, both actors have been fired the requisite number of times, and the buffer on the feedback arc again has four tokens. The schedule has therefore returned the dataflow graph to its initial condition.

Were there any fewer than four initial tokens on the feedback path, however, the model would deadlock. If there were only three tokens, for example, then *A* could fire, followed by *B*, but neither would have enough input tokens to fire again.

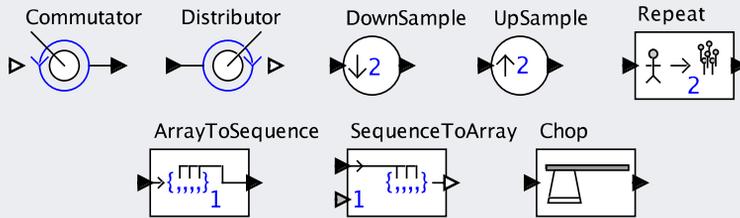
Lee and Messerschmitt (1987b) discuss the procedure for solving the balance equations, along with a procedure that will either provide a schedule for an unbounded execution or prove that no such schedule exists. Using these procedures, both bounded buffers and deadlock are **decidable** for SDF models (meaning that it is possible for Ptolemy to determine whether deadlock or unbounded buffers occur in any SDF model).

### 3.1.3 Time in Dataflow Models

In the SDF examples we have considered thus far, we have used the [SequencePlotter](#) actor but not the [TimedPlotter](#) actor (see Chapter 17). This is because the SDF domain does not generally use the notion of time in its models. By default, time does not advance as an SDF model executes (though the SDFDirector does contain a parameter, called *period*, that can be used to advance time by a fixed amount on each iteration of the model). Therefore, in most SDF models, the [TimedPlotter](#) actor would show the time axis as always being equal to zero. The [SequencePlotter](#) actor, in contrast, plots a sequence of values that are not time-based, and is therefore frequently used in SDF models. The [discrete event](#) (DE) and [Continuous](#) domains, discussed in Chapters 7 and 9, include a much stronger notion of time, and often use the [TimedPlotter](#).

### Sidebar: Multirate Dataflow Actors

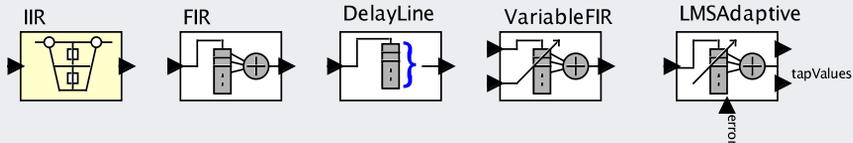
The Ptolemy II library offers a few actors that produce and/or consume multiple tokens per firing on a port. The most basic ones are shown below:



- **Commutator** and **Distributor**, in the `FlowControl`→`Aggregators` sublibrary, convert tokens arriving from multiple signals into a sequence of tokens and vice versa. Commutator has a **multiport** input, and on each firing, it reads a fixed number of tokens (given by its *blockSize* parameter) from each input channel, and outputs all the tokens from all the input channels as a sequence. Distributor reverses this process.
- **DownSample** and **UpSample**, in `SignalProcessing`→`Filtering`, discard or insert tokens. Downsample reads a fixed number of tokens (given by its *factor* parameter), and outputs one of those tokens (selected by its *phase* parameter). UpSample inserts a fixed number of zero-valued tokens between input tokens.
- **Repeat**, found under `FlowControl`→`SequenceControl`, is similar to UpSample except that instead of inserting zero-valued tokens, it repeats the input token.
- **ArrayToSequence** and **SequenceToArray**, found in the `Array` library, convert array tokens into sequences of tokens and vice versa. Both actors have an *arrayLength* parameter that specifies the length of the incoming (or outgoing) array. ArrayToSequence also has an *enforceArrayLength* parameter, which, if set to `true`, causes the actor to generate an error message if it receives an array of the wrong length. In SequenceToArray, *arrayLength* is a **PortParameter**, and hence the number of input tokens that are read can vary. These actors are SDF actors only when the array length is constant.
- **Chop**, in `FlowControl`→`SequenceControl`, reads a specified number of input tokens and produces a specified subset of those inputs, possibly padded with zero-valued tokens or previously consumed tokens.

## Sidebar: Signal Processing Actors

In addition to the spectral analysis actors described on page 101, Ptolemy II includes several other key signal processing actors, as shown below.



- **IIR** implements an infinite impulse response filter, also called a recursive filter (see [Lee and Varaiya \(2011\)](#)). Filter coefficients are provided in two arrays, one for the numerator and one for the denominator polynomial of the transfer function.
- **FIR** implements a finite impulse response filter, also called a tapped delay line, with coefficients specified by the *taps* parameter. Whereas IIR is a **homogeneous SDF** (single-rate) actor, FIR is potentially a **multirate** actor. When the *decimation* (*interpolation*) parameters are not equal to 1, the filter behaves as if it were followed (preceded) by a **DownSample** (**UpSample**) actor (see sidebar on page 106). However, the implementation is much more efficient than it would be using UpSample or DownSample actors; a polyphase structure is used internally, avoiding unnecessary use of memory and unnecessary multiplication by zero. Arbitrary sample-rate conversions by rational factors can be accomplished in this manner.
- **DelayLine** produces an array rather than the scalar produced by FIR. Instead of a weighted average of the contents of the delay line (which is what FIR produces), DelayLine simply outputs the contents of the delay line as an array.
- **VariableFIR** is identical to FIR except that the coefficients are provided as an array on an input port (and thus can vary) rather than being defined as actor parameters.
- **LMSAdaptive** is similar to FIR, except that the coefficients are adjusted on each firing using a gradient descent adaptive filter algorithm that attempts to minimize the power of the signal at the *error* input port.

In addition to the actors described here, the signal processing library includes fixed and adaptive lattice filters, statistical analysis actors, actors for communications systems (such as source and channel coders and decoders), audio capture and playback, and image and video processing actors. See the actor index on page 632.

### Sidebar: Dynamically Varying Rates

A variant of SDF that is called **parameterized SDF (PSDF)**, introduced by [Bhattacharya and Bhattacharyya \(2000\)](#), allows the production and consumption rates of ports to be given by a parameter rather than being a constant. The value of the parameter is permitted to change, but only between **complete iterations**. When the value of such a parameter changes, a new schedule must be used for the next complete iteration.

The example in Figure 3.9 illustrates how PSDF can be achieved with the SDF director in Ptolemy II. First, notice that the director's *allowRateChanges* parameter has been set to true. This indicates to the director that it may need to compute more than one schedule, since rate parameters may change during the execution of the model.

Second, notice that the **Repeat** actor's *numberOfTimes* parameter is set equal to the model parameter *rate*, which initially has value zero. Hence, when this model executes its first iteration, the **Repeat** actor will produce zero tokens, so the **Display** actor will not fire. The initial output from the **Ramp** actor, which has value 1, will not be displayed.

During this first iteration, the **Expression** and **SetVariable** actor both fire once. The **Expression** actor sets its output equal to input, unless the input is equal to the value of the *iterations* parameter (which it doesn't in this first iteration). The **SetVariable** actor sets the value of the *rate* parameter to 1. By default, **SetVariable** has a *delayed* parameter with value true, which means that the *rate* parameter changes only after the current iteration is complete.

In the second iteration, the value of the *rate* parameter is 1, so the **Repeat** actor copies its input (which has value 2) once to its output. The **Expression** and **SetVariable** actors set the *rate* parameter now to 2, so in the third iteration, the **Repeat** actor copies its input (which has value 3) twice to its output. The sequence of displayed outputs is therefore 2, 3, 3, 4, 4, 4, ...

To stop the model, the *iterations* parameter of the director is set to 5. In the last iteration of the execution, the **Expression** actor ensures that the *rate* parameter gets reset to 0. Hence, the next time the model executes, it will start again with the *rate* parameter set to 0.

In this example, each time the *rate* parameter changes, the SDF director recomputes the schedule. In a better implementation of PSDF, it would probably precompute schedules and/or cache previously computed schedules, but this implementation does not do that. It just recomputes the schedule between iterations.

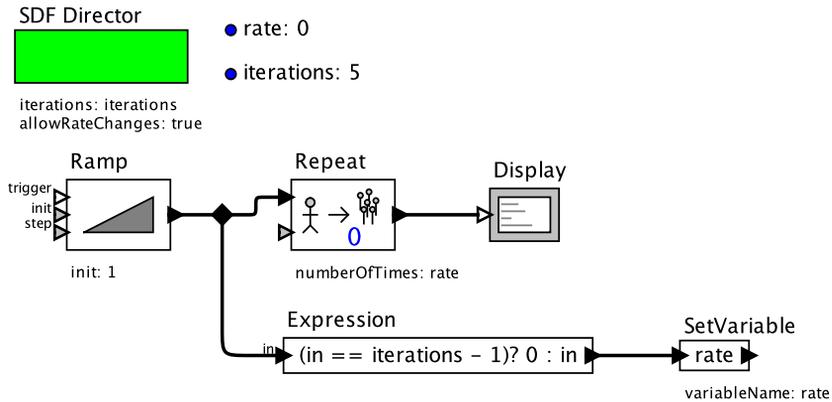


Figure 3.9: An SDF model with dynamically varying rates. [[online](#)]

### Sidebar: StreamIt

Thies et al. (2002) give a textual programming language, **StreamIt**, based on SDF and intended for use with streaming data applications such as multimedia. Software components (called **filters** rather than actors) produce and consume fixed amounts of data. The language provides compact structured constructs for common patterns of actor composition, such as chains of filters, parallel chains of filters, or feedback loops.

A key innovation in StreamIt is the notion of a **teleport message** (Thies et al., 2005). Teleport messages improve the expressiveness of SDF by allowing one actor to sporadically send a message to another; that is, rather than sending a message on every firing, only some firings send messages. The teleport message mechanism nonetheless ensures determinism by ensuring that the message is received by the receiving actor in exactly the same firing that it would have if the sending actor had sent messages on every firing. But it avoids the overhead of sending messages on every firing. This approach models a communication channel where tokens are sometimes, but not always, produced and consumed. But it preserves the determinism of SDF models, where the results of execution are the same for any valid schedule.

### Sidebar: Other Variants of Dataflow

A disadvantage of **SDF** is that every actor must produce and consume a fixed amount of data; the production and consumption rates cannot depend on the data. **DDF** (Section 3.2) relaxes this constraint at the cost of being able to statically precompute the firing schedule. In addition, as discussed earlier in the chapter, it is no longer possible to analyze all models for **deadlock** or **bounded buffers** (these questions are **undecidable**). Researchers have developed a number of variants of dataflow, however, that are more expressive than SDF but still amenable to some forms of static analysis.

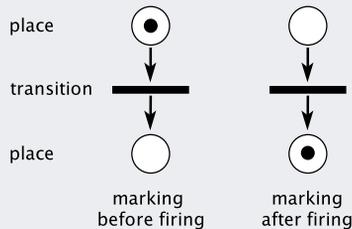
**Cyclo-static dataflow (CSDF)** allows production and consumption rates to vary in a periodic manner (Bilsen et al., 1996). An example is the **SingleTokenCommutator** actor (in `FlowControl`→`Aggregators`). This actor is similar to the **Commutator** actor (see sidebar on page 106), but instead of consuming all inputs in a single firing, it consumes inputs from only one channel in each firing, and rotates through the input channels on successive firings. For each input channel, the consumption rate alternates between zero and one. This actor is useful in feedback systems where the input to the second channel depends on the input to the first channel.

SDF can also be combined hierarchically with **finite state machines (FSMs)** to create **modal models**, described in Chapter 8. Each state of the FSM is associated with a submodel (a **mode refinement**, where each refinement can have different production and consumption rates). If the state transitions of the FSM are constrained to occur only at certain times, the model remains decidable. This combination was introduced by Girault et al. (1999), who called it **heterochronous dataflow (HDF)**. **SDF Scenarios** (Geilen and Stuijk, 2010) are similar to HDF in that they also use an FSM, but rather than having mode refinements, in SDF Scenarios each state of the FSM is associated with a set of production and consumption rates for a single SDF model. Bhattacharya and Bhattacharyya (2000) introduced **parameterized SDF (PSDF)**, where production and consumption rates can depend on input data as long as the same dependence can be represented in parameterized schedule.

Murthy and Lee (2002) introduced **multidimensional SDF (MDSDF)**. Whereas a channel in SDF carries a sequence of tokens, a channel in MDSDF carries a multi-dimensional array of tokens. That is, the history of tokens can grow along multiple dimensions. This model is effective for expressing certain kinds of signal processing applications, particularly image processing, video processing, radar and sonar.

### Sidebar: Petri Nets

**Petri nets**, named after Carl Adam Petri, are a popular modeling formalism related to [dataflow](#) (Murata, 1989). They have two types of elements, **places** and **transitions**, depicted as white circles and rectangles, respectively. A place can contain any number of tokens, depicted as black circles. A transition is **enabled** if all places connected to it as inputs contain at least one token.



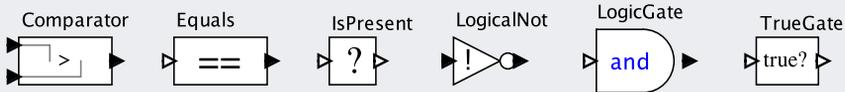
Once a transition is enabled, it can **fire**, consuming one token from each input place and depositing one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows the marking of a simple network before and after a firing. If a place provides inputs to more than one transition, then a token on that place may trigger a firing of either destination transition (one or the other fires, nondeterministically).

Petri net transitions are like dataflow actors; they fire when sufficient inputs are available. In basic Petri nets, tokens have no value, and firing of a transition does not involve any computation on tokens. A firing is just the act of moving tokens from one place to another. Also, places do not preserve token ordering, unlike dataflow buffers. Like [homogeneous SDF](#), transitions are enabled by a single token on each input place. Unlike SDF, a place may feed more than one transition, resulting in nondeterminism.

There are many variants of Petri nets, at least one of which is equivalent to SDF. In particular, tokens can have values (such tokens are called **colored tokens** in the literature, where the color of a token represents its value). Transitions can manipulate the color of tokens (analogous to the firing function of a dataflow actors). Arcs connecting places to transitions can be weighted to indicate that more than one token is required to fire a transition, or that a transition produces more than one token. This is analogous to SDF production and consumption rates. And finally, the Petri net graph structure can be constrained so that for each place, there is exactly one source transition and exactly one destination transition. With order-preserving places, such Petri nets are SDF graphs.

### Sidebar: Logic Actors

The actors found in the `Logic` library are useful for building control logic:



- The **Comparator** actor compares two values of type *double* (or of any type that can be losslessly converted to *double*, as explained in Chapter 14). The available comparisons are  $>$ ,  $>=$ ,  $<$ ,  $<=$ , and  $==$ . The output is a boolean.
- The **Equals** actor compares any number of input tokens of any type for equality and outputs a boolean true if they are equal and a false otherwise.
- The **IsPresent** actor outputs a boolean true if the input is present when it fires and false otherwise. In dataflow domains, the input is always present, so the output will always be true. This actor is more useful in the **SR** and **DE** domains (Chapters 5 and 7).
- **LogicalNot** accepts an input boolean and outputs the converse boolean.
- **LogicGate** implements the following six logic functions (the icon changes when you select the logic function):



- The **TrueGate** actor produces a boolean true output when the input is a boolean true. Otherwise, it produces no token at all. This is clearly not an SDF actor, but it can be used with **DDF**. It is also useful in **SR** (Chapter 5).

## 3.2 Dynamic Dataflow

Although the ability to guarantee [bounded buffers](#) and rule out [deadlock](#) is valuable, it comes at a price: SDF is not very expressive. It cannot directly express conditional firing, for example, such as when an actor fires only if a token has a particular value.

A number of dataflow variants have been developed that loosen the constraints of SDF; several of these are discussed in the sidebar on page [110](#). In this section, we describe a variant known as **dynamic dataflow (DDF)**. DDF is much more flexible than SDF, because actors can produce and consume a varying number of tokens on each firing.

### 3.2.1 Firing Rules

As in other dataflow [MoCs](#) (such as SDF) DDF actors begin execution when they have sufficient input data. For a given actor to fire, its **firing rule** (that is, the condition that must be met before an actor can fire) must be satisfied. In SDF, the actors' firing rule is constant. It simply specifies the fixed number of tokens that are required on each input port before the actor can fire. In the DDF domain, however, firing rules can be more complicated, and may specify a different number of tokens for each firing.

**Example 3.8:** The [SampleDelay](#) actor of [Example 3.6](#) is directly supported by the DDF MoC, without any need for special treatment of initial tokens. Specifically, the firing rule for [SampleDelay](#) states that on the first firing, it requires no input tokens. On subsequent firings, it requires one token.

Another difference is that, in SDF, actors produce a fixed number of tokens on each output port. In DDF, the number of tokens produced can vary.

**Example 3.9:** On its first firing, the [SampleDelay](#) actor produces the number of tokens specified in its *initialOutputs* parameter. On subsequent firings it produces a single token that is equal to the token it consumed.

The firing rules themselves need not be constant. Upon firing, a DDF actor may change the firing rules for the next firing.

A key DDF actor is the **BooleanSelect**, which merges two input streams into one stream according to a stream of boolean-valued control tokens (see sidebar on page 119). This actor has three firing rules. Initially, it requires one token on the *control* (bottom) port, and no tokens on the other two ports. When it fires, it records the value of the control token and changes its firing rule to require a token on one of the *trueInput* port (labeled *T*) or the *falseInput* port (labeled *F*), depending on the value of the control token. When the actor next fires, it consumes the token on the corresponding port and sends it to the output. Thus, it fires twice to produce one output. After producing an output, its firing rule reverts to requiring a single token on the *control* port.

A more general version of the **BooleanSelect** is the **Select** actor, which merges an arbitrary number of input streams into one stream according to a stream of integer-valued control tokens, rather than just two streams (see sidebar on page 119).

Whereas **BooleanSelect** and **Select** merge multiple input streams into one, **BooleanSwitch** and **Switch** do the converse; they split a single stream into multiple streams. Again, a stream of control tokens determines, for each input token, to which output stream that token should be sent. These **Switch** and **Select** actors accomplish conditional routing of tokens, as illustrated in the following examples.

**Example 3.10:** Figure 3.10 uses **BooleanSwitch** and **BooleanSelect** to accomplish conditional firing, the equivalent of if-then-else in an imperative programming language. In this figure, the **Bernoulli** actor produces a random stream of Boolean-valued tokens. This control stream controls the routing of tokens produced by the **Ramp** actor. When **Bernoulli** produces `true`, the output of the **Ramp** actor is multiplied by  $-1$  using the **Scale** actor. When **Bernoulli** produces `false`, **Scale2** is used; it passes its input through unchanged. The **BooleanSelect** uses the same control stream to select the appropriate **Scale** output.

**Example 3.11:** Figure 3.11 shows a DDF model that uses **BooleanSwitch** and **BooleanSelect** to realize data-dependent iteration using a feedback loop. The **Ramp**

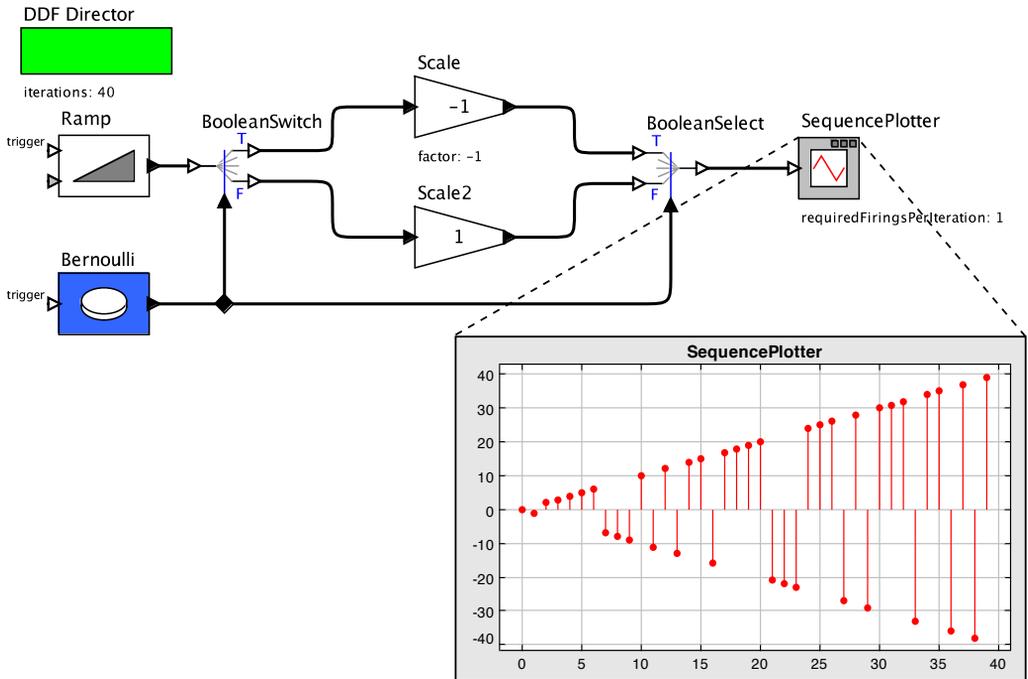


Figure 3.10: A DDF model that accomplishes conditional firing. [[online](#)]

actor feeds the loop with a sequence of increasing integers, 0, 1, 2, 3, etc. The [SampleDelay](#) initiates the loop by providing a *false* token to the *control* port of the [BooleanSelect](#). In the full cycle, each input integer is repeatedly multiplied by 0.5 until the resulting value is less than 0.5. The [Comparator](#) actor (found in the *Logic* library) controls whether the token is routed back around the loop for another iteration or routed out of the loop to the [Discard](#) actor (the one at the right with the icon that looks like a ground symbol on an electrical circuit diagram, found in *Sinks*→*GenericSinks*). The [Discard](#) actor receives and discards its input, but in this case, it is also used to control what an [iteration](#) means. The parameter *requiredFiringsPerIteration* has been added to the actor and assigned a value of 1 (see Section 3.2.2 below). Hence, one iteration of the model consists of as many iterations of the loop as needed to produce one firing of [Discard](#). This structure is analogous to a do-while loop in an imperative programming language.

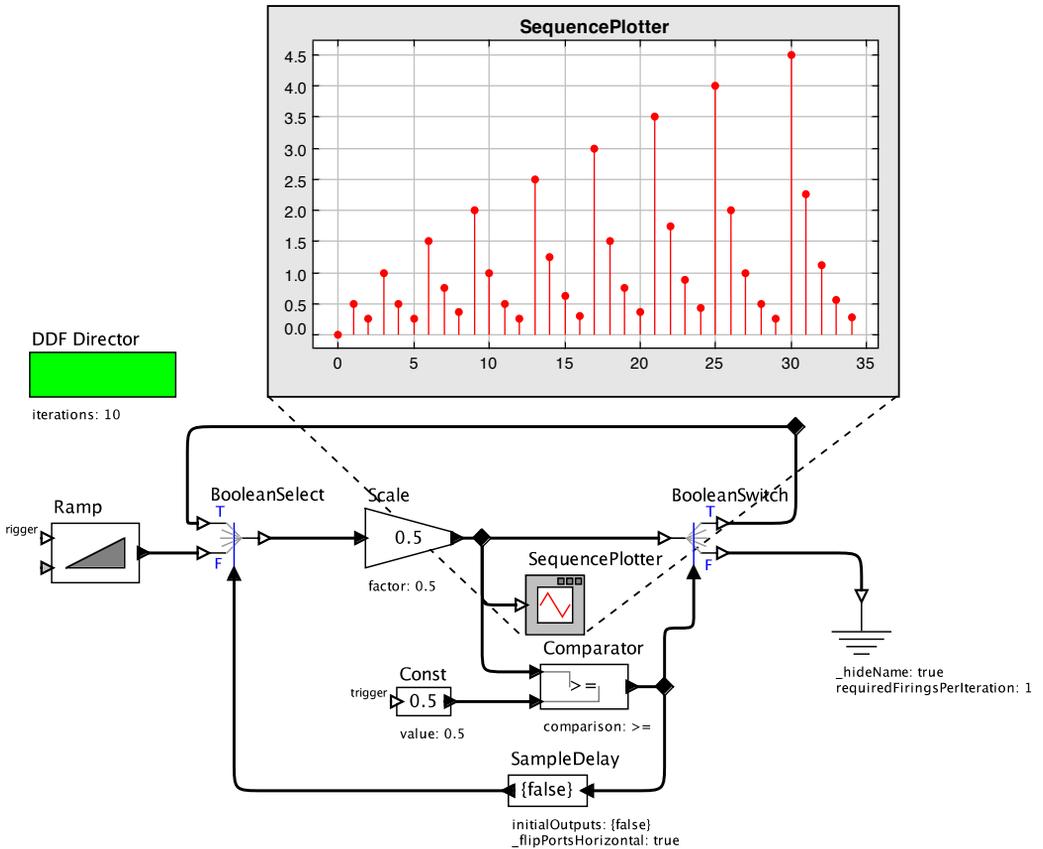


Figure 3.11: A DDF model that accomplishes data-dependent iteration. [\[online\]](#)

The pattern shown in Figure 3.11 is sufficiently useful that it might be used repeatedly. Fortunately, Ptolemy II includes a mechanism for storing and re-using a **design pattern**, created by Feng (2009). The pattern shown in Figure 3.12, for example, is available as a unit in the `MoreLibraries→DesignPatterns`. In fact, any Ptolemy II model can be exported to a library as a design pattern and reimported into another model as a unit by simply dragging it into the model.

The Switch and Select actors (and their boolean versions) that are part of the DDF domain provide increased flexibility and expressiveness relative to the SDF domain, but their use means that it may not be possible to determine a schedule with **bounded buffers**, nor is

Fire RepeatedAction for "count" times with Source as initial input, and send the result to Sink. ● count: 3

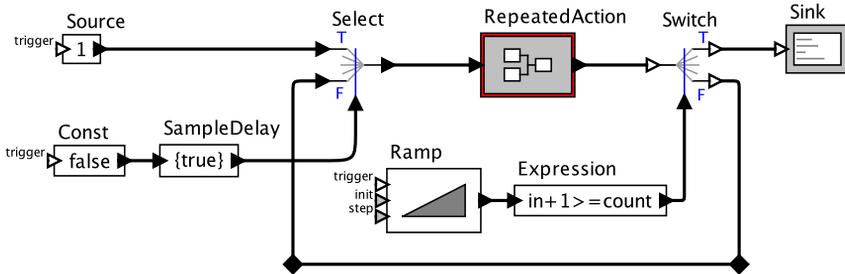


Figure 3.12: A design pattern stored as a unit in a library.

possible to ensure that the model will not [deadlock](#). In fact, [Buck \(1993\)](#) showed that bounded buffers and deadlock are [undecidable](#) for DDF models. For this reason, DDF models are not as readily analyzed.

**Example 3.12:** A variant of the if-then-else model shown in [Figure 3.10](#) is shown in [Figure 3.13](#). In this case, the inputs to the BooleanSelect have been reversed. Unlike the earlier model, this model has no schedule that assures [bounded buffers](#). The [Bernoulli](#) actor is capable of producing an arbitrarily long sequence of `true`-valued tokens, during which an arbitrarily long sequence of tokens may build up on input buffer for the `false` port of the BooleanSelect, thus potentially overflowing the buffer.

Switch and Select and their boolean cousins are dataflow analogs of the **goto** statement in imperative languages. They provide low-level control over execution by conditionally routing tokens. Like goto statements, their use can result in models that are difficult to understand. This problem is addressed using [structured dataflow](#), described in the sidebar on [page 120](#), and implemented in Ptolemy II using [higher-order actors](#), described in [Section 2.7](#).

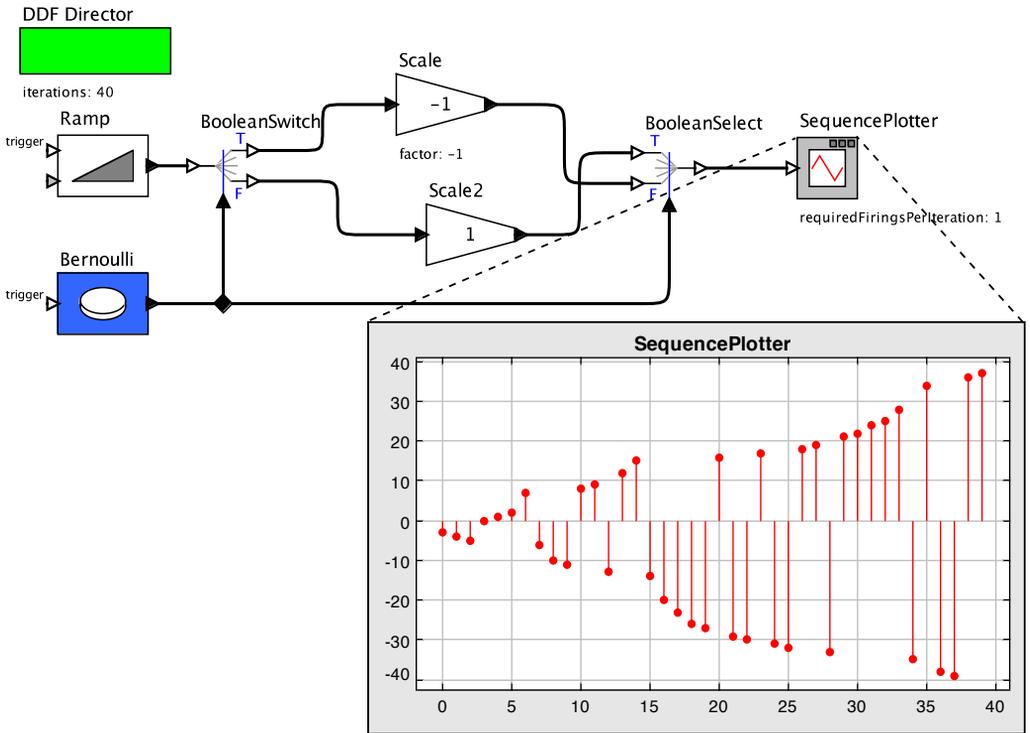


Figure 3.13: A DDF model that has no bounded buffer schedule. [\[online\]](#)

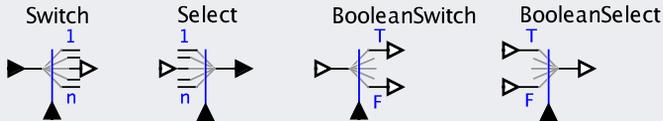
### 3.2.2 Iterations in DDF

One of the advantages of SDF is that a **complete iteration** is uniquely defined. It consists of a fixed number of firings of each of the actors in the model. It is therefore easy to control the duration of an overall execution of the model by setting the *iterations* parameter of the SDF director, which controls the number of times each actor will be executed.

The DDF director also has an *iterations* parameter, but defining an iteration is more difficult. An iteration can be defined by adding a parameter to one or more actors named *requiredFiringsPerIteration* and giving that parameter an integer value, as illustrated in the following example.

### Sidebar: Token Flow Control Actors

Ptolemy II provides a number of actors that can route tokens in a model. The most basic of these are **Switch** and **Select** (and their boolean variants), shown here:



On each firing, **Switch** consumes one token from the input and an integer-valued token from the *control* port (on the bottom) and routes the input token to the output channel specified by the control token. All other output channels produce no tokens on that firing. **Select** does the converse, consuming a single token from the channel specified by the control token and sending that token to the output. The other input channels consume no tokens. **BooleanSwitch** (**BooleanSelect**) are variants where the number of outputs (or inputs) is constrained to be two, and the control token is boolean rather than integer valued.

Switch and Select can be compared to the following actors with related functionality:



**ConfigurationSwitch** is similar to **BooleanSwitch** except that instead of a *control* input port it has a *parameter* that determines which output to send data to. If the value of the parameter does not change during execution of the model (normally this is the case with parameters), then this actor is an SDF actor that always produces zero tokens on one output and one token on the other. **ConfigurationSelect** is likewise similar to **BooleanSelect**.

**BooleanMultiplexor** and **Multiplexor** are similar to **BooleanSelect** and **Select** except that they consume one token from *all* input channels. These actors discard all but one of those input tokens, and send that one token to the output. Since these two actors consume and produce exactly one token on every channel, they are **homogeneous SDF** actors.

### Sidebar: Structured Dataflow

In an imperative language, structured programming replaces goto statements (which can be problematic, as described in [Dijkstra \(1968\)](#)) with nested `for` loops, `if-then-else`, `do-while`, and recursion. In structured dataflow, these concepts are adapted to the dataflow modeling environment.

Figure 3.14 shows an alternative way to accomplish the conditional firing of Figure 3.10. The result is an SDF model that has many advantages over the DDF model in Figure 3.10. The **Case** actor is an example of a [higher-order actor](#), like those discussed in Section 2.7. Inside, it contains two sub-models (refinements), one named `true` that contains a `Scale` actor with a parameter of `-1`, and one named `default` that contains a `Scale` actor with a parameter of `1`. When the control input to the `Case` actor is `true`, the `true` refinement executes one iteration. For any other control input, the `default` refinement executes.

This style of conditional firing is called **structured dataflow**, because, much like structured programming, control constructs are nested hierarchically. The approach avoids arbitrary data-dependent token routing (which is analogous to avoiding arbitrary branches using goto instructions). Moreover, the use of the `Case` actors enables the overall model to be an SDF model. In the example in Figure 3.14, every actor consumes and produces exactly one token on every port. Hence, the model is analyzable for deadlock and bounded buffers.

This style of structured dataflow was introduced in LabVIEW, a design tool developed by National Instruments ([Kodosky et al., 1991](#)). In addition to providing a conditional operation similar to that of Figure 3.14, LabVIEW provides structured dataflow constructs for iterations (analogous to `for` and `do-while` loops in an imperative language), and for sequences (which cycle through a finite set of submodels). Iterations can be achieved in Ptolemy II using the [higher-order actors](#) of Section 2.7. Sequences (and more complicated control constructs) can be implemented using [modal models](#), discussed in Chapter 8. Ptolemy II supports structured recursion using the **ActorRecursion** actor, found in `DomainSpecific→DynamicDataflow` (see Exercise 3). However, without careful constraints, boundedness again becomes undecidable with recursion ([Lee and Parks, 1995](#)).

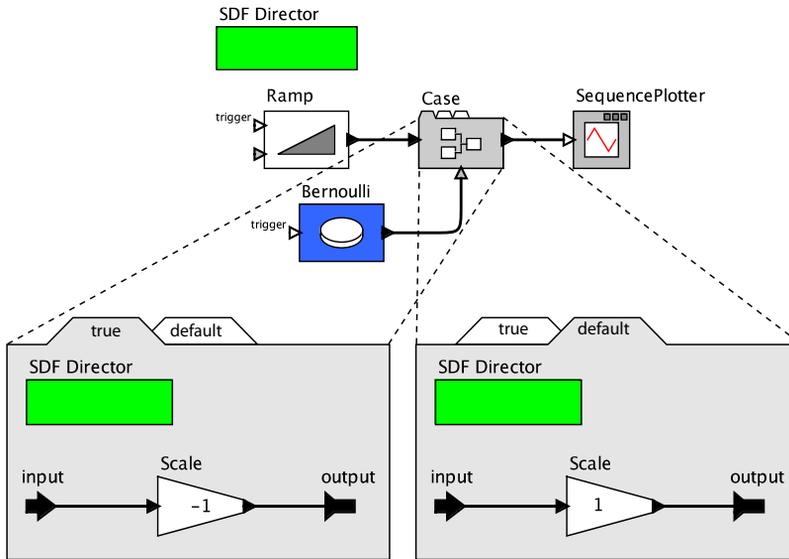


Figure 3.14: Structured dataflow approach to conditional firing. [[online](#)]

**Example 3.13:** Consider the if-then-else example in Figure 3.10, discussed in Example 3.10. The *iterations* parameter of the director is set to 40, and indeed the plot has 40 points. This is because a parameter named *requiredFiringsPerIteration* has been added to the SequencePlotter actor and assigned the value 1. As a consequence, each iteration must include at least one firing of the SequencePlotter. In this case, no other actor in the model has a parameter named *requiredFiringsPerIteration*, so this parameter ends up determining what constitutes an iteration.

When multiple actors within a model have parameters named *requiredFiringsPerIteration*, or when there are no such parameters, the situation is more subtle. In these cases, DDF still has a well-defined iteration, but the definition is complex, and can surprise the designer (see sidebar on page 124).

**Example 3.14:** Consider again the if-then-else example in Figure 3.10. If we remove the *requiredFiringsPerIteration* from the SequencePlotter, then 40 iterations of the model will produce only nine plotted points. Why? Recall from Section 3.2.1 that BooleanSelect fires twice for each output it produces. Absent any constraints in the model, the DDF director will not fire any actor more than once in an iteration.

**Example 3.15:** Figure 3.15 shows a DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory (see box on page 126 for why you might want to do this). This model uses the DirectoryListing actor (see box on page 128) to construct an array of file names for actors in a specified directory. Ptolemy models are identified by files whose names match the regular expression `.*.xml`, which matches any file name that ends with `.xml`. The *firingCountLimit* parameter of the DirectoryListing ensures that this actor fires only once. It will produce one array token on its output, and then will refuse to fire again. Once the data in that array have been processed, there are no more tokens to process, so the model **deadlocks**, and stops execution.

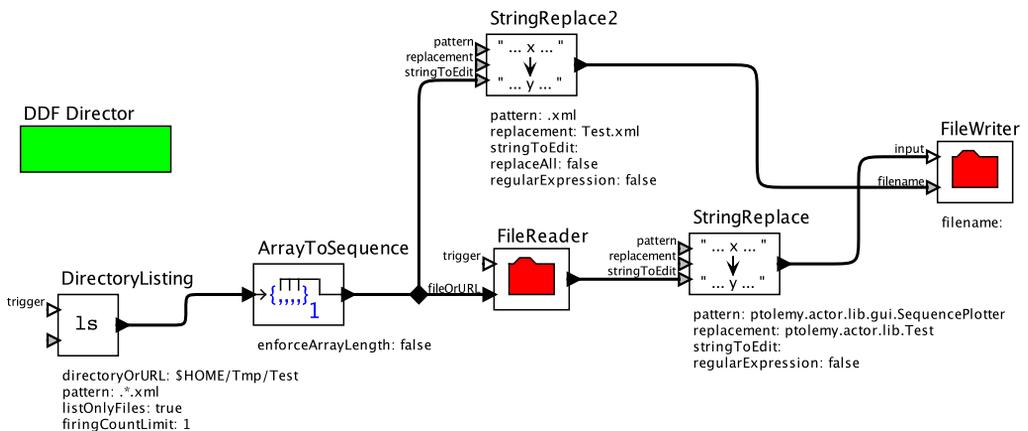


Figure 3.15: A DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory.

The [ArrayToSequence](#) actor (see box on page 106) converts the array of file names into a sequence of tokens, one string-valued token for each file name. Notice that the *enforceArrayLength* parameter is set to false for this actor. If we were to know exactly how many XML files were in the directory in question, we could have left this parameter with its default value true, set the *arrayLength* parameter to the number of files, and used the SDF director instead of the DDF director. The ArrayToSequence actor would consume one token and produce a fixed, known number of output tokens, and hence would be an SDF actor. But since we do not know in general how many matching files there will be in the directory, the DDF director is more useful.

The [FileReader](#) actor (see box on page 128) reads the XML file and outputs its contents as a single string. The [StringReplace](#) actor (see box on page 125) replaces all instances of the full classname for the SequencePlotter actor with the full classname for the Test actor.

The second [StringReplace](#) actor, named StringReplace2, is used to create a new filename from the original file name. For example, the filename `Foo.xml` will become `FooTest.xml`. The [FileWriter](#) actor then writes the modified filename to a file with the new file name.

Note that we could have used the [IterateOverArray](#) actor and the SDF director instead (see Section 2.7.2). We leave this as an exercise (see Problem 2 at the end of this chapter).

### 3.2.3 Combining DDF with Other Domains

Although a system may be best modeled as DDF overall, it may contain some subsystems that can be modeled as SDF. Thus, a DDF model may contain an [opaque](#) composite actor that has an [SDF](#) director. This approach can improve efficiency and provide better control over the amount of computation done in an iteration.

Conversely, a DDF model may be placed within an SDF model if it behaves like SDF at its input/output boundaries. That is, to be used within an SDF model, a DDF opaque composite actor should consume and produce a fixed number of tokens. It is not generally possible for the DDF director to determine from the model how many tokens are produced and consumed at the boundary (this question is [undecidable](#) in general) so it is

### Sidebar: Defining a DDF Iteration

An **iteration** in DDF consists of the minimum number of **basic iterations**, (defined below) that satisfy all constraints imposed by *requiredFiringsPerIteration* parameters.

In one **basic iteration**, the DDF director fires all **enabled** and **non-deferrable** actors once. An enabled actor is one that has sufficient data at its input ports, or has no input ports. A **deferrable actor** is one whose execution can be deferred because its execution is not currently required by a downstream actor. This is the case when the downstream actor either already has enough tokens on the channel connecting it to the deferrable actor, or the downstream actor is waiting for tokens on another channel or port. If there are no enabled and non-deferrable actors, then the director fires those enabled and deferrable actors that have the smallest maximum number of tokens on their output channels that will satisfy the demands of destination actors. If there are no enabled actors, then a **deadlock** has occurred. The above strategy was shown by Parks (1995) to guarantee that buffers remain bounded in an **unbounded execution** if there exists an unbounded execution with **bounded buffers**.

The algorithm that implements one basic iteration is as follows. Let  $E$  denote the set of enabled actors, and let  $D$  denote the set of deferrable enabled actors. One basic (default) iteration then consists of the following, where the notation  $E \setminus D$  means “the set of elements in  $E$  that are not in  $D$ ”:

```
if  $E \setminus D \neq \emptyset$  then  
    fire actors in  $(E \setminus D)$   
else if  $D \neq \emptyset$  then  
    fire actors in  $\text{minimax}(D)$   
else  
    declare deadlock  
end if
```

The function “ $\text{minimax}(D)$ ” returns a subset of  $D$  with the smallest maximum number of tokens on their output channels that satisfy the demand of destination actors. This will always include **sink actors** (actors with no output ports).

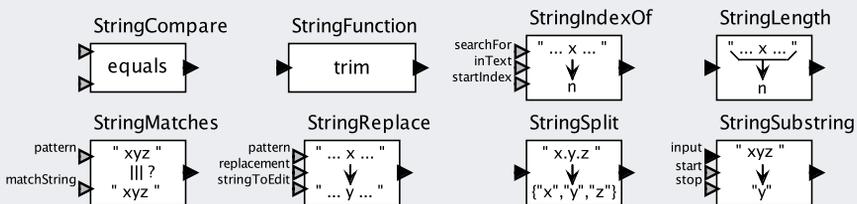
up to the model designer to assert the production and consumption rates. If they are not equal to a value of 1 (which need not be explicitly asserted), then the model designer can assert consumption and production rates by creating a parameter in each input port called *tokenConsumptionRate* and assigning it an integer value. Similarly, output ports should be given a parameter called *tokenProductionRate*.

Once the rates at the boundary are set, it is up to the model designer to ensure that they are respected at run time. This can be accomplished using the *requiredFiringsPerIteration* parameter, as explained above in Section 3.2.2. In addition, the DDF director has a parameter *runUntilDeadlockInOneIteration* that, when set to true, defines an iteration to be repeated invocations of a [basic iteration](#) (see sidebar on page 124). until deadlock is reached. If this parameter is used, it overrides any *requiredFiringsPerIteration* that may be present in the model.

DDF conforms with the [loose actor semantics](#), meaning that if a DDF director is used in [opaque](#) composite actor, its state changes when its `fire` method is invoked. In particular,

### Sidebar: String Manipulation Actors

The `String` library provides actor for manipulating strings:



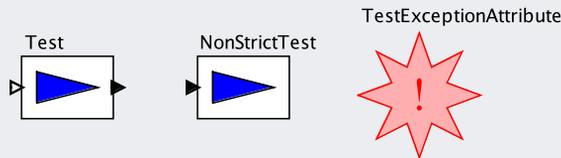
The **StringCompare** actor compares two strings, determine whether they are equal, or if one string starts with, ends with, or contains another string. The **StringMatches** actor checks whether a string matches a pattern given as a regular expression. The **StringFunction** actor can trim white space around a string or convert it to lower case or upper case. The **StringIndexOf** actor searches for a substring within a string a returns the index of that substring. The **StringLength** actor outputs the length of a string. **StringReplace** replaces a substring that matches a specified pattern with a specified replacement string. **StringSplit** divides a string at specified separators. **StringSubstring** extracts a substring, given a start and stop index.

dataflow actors **consume** input tokens in their `fire` method. Once the tokens have been consumed, they are no longer available in the input buffers. Thus, a second firing will see new data, regardless of whether `postfire` has been invoked. For this reason, DDF and SDF composite actors should not be used within domains that require **strict actor semantics**, such as **SR** and **Continuous**, unless the model builder can ensure that these

### Sidebar: Building Regression Tests

When developing nontrivial models and when extending Ptolemy II, good engineering practice requires creating **regression tests**. Regression tests guard against future changes that may change behavior in ways that can invalidate applications that were created earlier. Fortunately, in Ptolemy II, it is extremely easy to create regression tests.

Key components are found in `MoreLibraries`→`RegressionTest`:



The **Test** actor compares the inputs against the value specified by the `correctValues` parameter. The actor has a `trainingMode` parameter, which when set to true, simply records the inputs it receives. Therefore, a typical use is to put the actor in training mode, run the model, take the actor out training mode, and then save the model in some directory where all models are executed as part of daily testing. (This is how the vast majority of the rather extensive regression tests for the Ptolemy II itself are created.) The model will throw an exception if the Test actor receives any input that differs from the ones it recorded. Note that one of the key advantages of **determinate** models is the ability to construct such regression tests.

The **NonStrictTest** is similar, except that it tolerates (and ignores) absent inputs, and it tests the inputs in the **postfire** phase of execution rather than the **fire** phase. It is useful for domains such as **SR** and **Continuous**, which iterate to a **fixed point**.

Sometimes, a model is expected to throw an exception. A regression test for such a model should include an instance of **TestExceptionAttribute**, which also has a training mode. The presence of this attribute in a model causes the model to throw an exception if the execution of the model does *not* throw an exception, or if the exception it throws does not match the expected exception.

composite actors will not be fired more than once in an iteration of the SR of Continuous container.

Note that any SDF model can be run with a DDF Director. However, the notion of iteration may be different. Sometimes, a DDF model may be run with an SDF director even when there is data-dependent iteration. Figure 3.14 shows one example, where the `Case` actor facilitates this combination. However, it is sometimes possible to use this combination even when a `Switch` is used. The SDF scheduler will assume the `Switch` produces one token on every output channel, and will construct a schedule accordingly. While executing this schedule, the director may encounter actors that it expects to be ready to fire but which do not actually have sufficient input data to fire. Many actors can be safely iterated even if they have no input data. Their `prefire` method returns false, indicating to the director that they are not ready to fire. The SDF director will respect this, and will simply skip over that actor in a schedule. However, this technique is rather tricky and is not recommended. It can result in unintended sequences of actor execution.

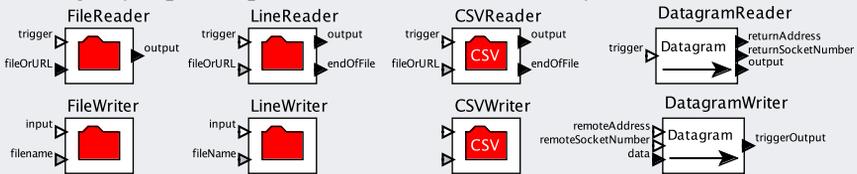
### 3.3 Summary

Dataflow is a simple and versatile model of computation in which the execution of actors is driven by the availability of input data. It is particularly useful for expressing [streaming](#) applications, where long sequences of data values are routed through computations, such as is common in signal processing and multimedia applications.

SDF is a simple (though restrictive) form of dataflow that enables extensive static analysis and efficient execution. DDF is more flexible, but also more challenging to control and more costly to execute, because scheduling decisions are made during run time. The two can be mixed within a single model, so the extra costs of DDF may be incurred only where they are absolutely required by the application. Both SDF and DDF are useful in [modal models](#), as explained in Chapter 8. Using SDF and DDF within modal models provides a versatile concurrent programming model.

## Sidebar: IO Actors

The following key input/output actors are in the `IO` library:



**FileReader** and **FileWriter** read and write files from the local disk or from a remote location specified via a URL or URI. For **FileReader**, the entire contents of the file is produced on a single output string token. For **FileWriter**, each input string token is written to a file, overwriting the previous contents of the file. In both cases, a new file name can be given for each firing. To read from standard input, specify `System.in` as the file name. To write to standard output, specify `System.out` as the file name. **LineReader** and **LineWriter** are similar, except that they read and write a line at a time.

**CSVReader** and **CSVWriter** read and write files or URLs that are in CSV format, or comma-separated values (actually, the separator can be anything; it need not be commas). CSV files are converted into `record` tokens, and record tokens are converted into CSV files. The first line of the file defines the field names of the records. To use **CSVReader**, you need to help the type system to determine the output type. The simplest way to do this is to enable `backward type inference` (see Section 14.1.4). This sets the data type of the output port of the **CSVReader** actor to be the most general type that is acceptable to the actors downstream. Thus, for example, if the actors downstream extract fields from the record, then the type constraints will automatically require those fields to be present and to have compatible types. You can also coerce the output type using the `[Customize→Ports]` context menu command.

The following actors are also in the `IO` library:



**ArrowKeySensor** produces outputs that respond to the arrow keys on the keyboard. **DirectoryListing** produces on its output an array of file names in a specified directory that match an (optional) pattern.

## Exercises

1. The multirate actors described in the box on page 106 and the array actors described in the boxes on page 88 and 86 are useful with SDF to construct **collective operations**, which are operations on arrays of data. This exercise explores the implementation of what is called an **all-to-all scatter/gather** using SDF. Specifically, construct a model that generates four arrays with values:

```
{ "a1", "a2", "a3", "a4" }
{ "b1", "b2", "b3", "b4" }
{ "c1", "c2", "c3", "c4" }
{ "d1", "d2", "d3", "d4" }
```

and converts them into arrays with values

```
{ "a1", "b1", "c1", "d1" }
{ "a2", "b2", "c2", "d2" }
{ "a3", "b3", "c3", "d3" }
{ "a4", "b4", "c4", "d4" }
```

Experiment with the use of [ArrayToElements](#) and [ElementsToArray](#), as well as [ArrayToSequence](#) and [SequenceToArray](#) (for the latter, you will also likely need [Commutator](#) and [Distributor](#)). Comment about the relative merits of your approaches. **Hint:** You may have to explicitly set the [channel widths](#) of the connections to 1. Double click on the wires and set the value. You may also experiment with [MultiInstanceComposite](#).

2. Consider the model in Figure 3.15, discussed in Example 3.15. Implement this same model using the [IterateOverArray](#) actor and only the SDF director instead of the DDF director (see Section 2.7.2).
3. The DDF director in Ptolemy II supports an actor called [ActorRecursion](#) that is a recursive reference to a composite actor that contains it. For example, the model shown in Figure 3.16 implements the sieve of Eratosthenes, which finds prime numbers, as described by [Kahn and MacQueen \(1977\)](#).

Use this actor to implement a composite actor that computes Fibonacci numbers. That is, a firing of your composite actor should implement the firing function

$f: \mathbb{N} \rightarrow \mathbb{N}$  defined by, for all  $n \in \mathbb{N}$ ,

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

When ActorRecursion fires, it clones the composite actor above it in the hierarchy (i.e., its container, or its container's container, etc.) whose name matches the value of its *recursionActor* parameter. The instance of ActorRecursion is populated with ports that match those of that container. This actor should be viewed as a highly experimental realization of a particular kind of **higher-order actor**. It is a higher-order actor because it is parameterized by an actor that contains it. Its implementation, however, is very inefficient. The cloning of the actor it references on each firing is expensive in terms of both memory and time. A better implementation would use an approach similar to the stack frame approach used in procedural programming languages. Instead, the approach it uses is more like copying the source code at run time and then interpreting it. In an attempt to make execution more efficient, this actor avoids creating the clone if it has previously created it. Also, the visual representation of the recursive reference is inadequate. There is no way, looking only at the image in Figure 3.16, to tell what composite actor the ActorRecursion instance references. Thus, you cannot really read the program from its visual representation.

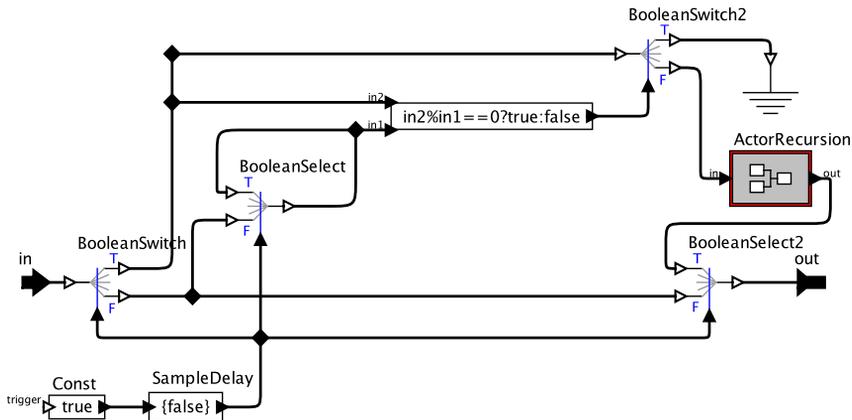


Figure 3.16: The sieve of Eratosthenes, using ActorRecursion in DDF. [\[online\]](#)

# Process Networks and Rendezvous

*Neil Smyth, John S. Davis II, Thomas Huining Feng, Mudit Goel, Edward A. Lee, Thomas M. Parks, Yang Zhao*

## Contents

---

<b>4.1 Kahn Process Networks</b> . . . . .	<b>132</b>
<i>Historical Notes: Process Networks</i> . . . . .	134
<i>Sidebar: Process Networks and Dataflow</i> . . . . .	135
4.1.1 Concurrent Firings . . . . .	136
<i>Sidebar: Actors With Unbounded Firings</i> . . . . .	138
<i>Sidebar: Useful Actors for PN Models</i> . . . . .	143
4.1.2 Stopping Execution of a PN Model . . . . .	144
<b>4.2 Rendezvous</b> . . . . .	<b>145</b>
<i>Sidebar: Useful Actors for Rendezvous Models</i> . . . . .	146
4.2.1 Multiway Rendezvous . . . . .	147
4.2.2 Conditional Rendezvous . . . . .	148
4.2.3 Resource Management . . . . .	150
<b>4.3 Summary</b> . . . . .	<b>151</b>
<b>Exercises</b> . . . . .	<b>152</b>

---

**Dataflow** models of computation are **concurrent**. The order of the firings is constrained only by data precedence, so the director has quite a bit of freedom to choose an ordering. There is no reason that two actors that do not depend on each other's data could not be fired simultaneously. Indeed, there is a long history of schedulers that do exactly that, as described in the sidebar on page 100. Nevertheless, the directors described in Chapter 3 fire actors one at a time. This chapter introduces two directors that fire actors concurrently. Both are similar to dataflow, semantically, but they are used for very different purposes.

The directors in this chapter execute each actor in its own thread. A **thread** is a sequential program that can potentially share variables with other threads that execute concurrently. On a multicore machine, two threads may execute in parallel on separate cores. On a single core, the instructions of each thread are arbitrarily interleaved. Understanding the interactions between threads is notoriously difficult (Lee, 2006) because of this arbitrary interleaving of instructions. The directors described in this chapter provide a way for threads to interact in more understandable and predictable ways.

One possible motivation for using the directors in this chapter is to exploit the parallelism offered by multiple cores. Every model described in the previous chapter, for example, could also be executed with a PN director, described below, and actors will fire simultaneously on multiple cores. One could expect models to execute faster than they would with an SDF director. However, this is not our experience. Our experience is that PN models almost always run slower than SDF models, regardless of the number of cores. The reason for this appears to be the cost of thread interactions associated with the mutual exclusion locks that enable Ptolemy II models to be edited while they run. In principle, this capability could be disabled, and one would then expect performance improvements. But as of this writing, no mechanism has been built in Ptolemy II to do this. Hence, improving performance is probably not an adequate reason to use the threaded directors of this chapter over SDF or DDF when SDF or DDF is suitable.

Instead, we focus in this chapter on models where concurrent firing of actors is required by the goals of the model. These examples show that these directors do in fact provide a fundamental increment in expressiveness, and not just a performance improvement.

## 4.1 Kahn Process Networks

A model of computation that is closely related to dataflow models is **Kahn process networks** (variously called **KPN**, **process networks** or **PN**), named after Gilles Kahn, who

introduced them (Kahn, 1974). The relationship between dataflow and PN is studied in detail by Lee and Parks (1995) and Lee and Matsikoudis (2009), but the short story is quite simple. In PN, each actor executes concurrently in its own thread. That is, instead of being defined by its **firing rules** and firing functions, a PN actor is defined by a (typically non-terminating) program that reads data tokens from input ports and writes data tokens to output ports. All actors execute simultaneously (conceptually; whether they actually execute simultaneously or are interleaved is irrelevant to the semantics).

In the original paper, Kahn (1974) gave very elegant mathematical conditions on the actors that ensure that a network of such actors is **determinate**. In this, case “determinate” means that the sequence of tokens on every connection between actors is uniquely defined, and specifically is independent of how the processes are scheduled. Every legal thread scheduling yields exactly the same data streams. Thus, Kahn showed that concurrent execution was possible without nondeterminism.

Three years later, Kahn and MacQueen (1977) gave a simple, easily implemented mechanism for programs that guarantees that the mathematical conditions are met to ensure determinism. A key part of the mechanism is to perform **blocking reads** on input ports whenever a process is to read input data. Specifically, blocking reads mean that if the process chooses to access data through an input port, it issues a read request and blocks until the data become available. It cannot test the input port for the availability of data and then perform a conditional branch based on whether data are available, because such a branch would introduce schedule-dependent behavior.

Blocking reads are closely related to firing rules. Firing rules specify the tokens required to continue computing (with a new firing function). Similarly, a blocking read specifies a single token required to continue computing (by continuing execution of the process). Kahn and MacQueen showed that if every actor implements a mathematical *function* from input sequences to output sequences (meaning that for each input sequence, the output sequence is uniquely defined), then blocking reads are sufficient to ensure determinism.

When a process writes to an output port, it performs a **nonblocking write**, meaning that the write succeeds immediately and returns. The process does not block to wait for the receiving process to be ready to receive data.<sup>1</sup> This is exactly how writes to output ports work in dataflow MoCs as well. Thus, the only material difference between dataflow and

---

<sup>1</sup>The Rendezvous director, discussed later in this chapter, differs at exactly this point, in that with that director, a write to an output port does not succeed until the actor with the corresponding input port is ready to read.

PN is that with PN, the actor is not broken down into firing functions. It is designed as a continuously executing program. The firing of an actor does not need to be finite.

### Historical Notes: Process Networks

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines** (Conway, 1963). Conway described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines *A* and *B* are connected so that *A* sends items to *B*, *B* runs for a while until it encounters a read command, which means it needs something from *A*. The control is then transferred to *A* until it wants to write, whereupon control is returned to *B* at the point where it left off."



Gilles Kahn (1946 – 2006)

The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a **CPO** (complete partial order). Kahn and MacQueen (1977) defined process interactions using non-blocking writes and blocking reads as a special case of continuous functions, and developed a programming language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974) includes the notion of **pipes**, which implement a limited form of process networks (pipelines only). Later, named pipes provided a more general form.

Kahn (1974) stated but did not prove what has come to be known as the **Kahn principle**, that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995).

### Sidebar: Process Networks and Dataflow

Three major variants of dataflow have emerged in the literature: Dennis dataflow (Dennis, 1974), Kahn process networks (KPN) (Kahn, 1974), and dataflow synchronous languages (Benveniste et al., 1994). The first two are closely related, while the third is quite different. This chapter deals with Kahn process networks, Dennis dataflow is addressed in Chapter 3, and dataflow synchronous languages are addressed in Chapter 5.

In **Dennis dataflow**, the behavior of an actor is given by a sequence of atomic firings that are enabled by the availability of input data. KPN, by contrast, has no notion of an atomic firing. An actor is a process that executes asynchronously and concurrently with others. Dennis dataflow can be viewed as a special case of KPN (Lee and Parks, 1995) by defining a firing to be the computation that occurs between accesses to inputs. But the style in which actors and models are designed is quite different. Dennis' approach is based on an operational notion of atomic firings driven by the satisfaction of firing rules. Kahn's approach is based on a denotational notion of processes as continuous functions on infinite streams.

Dennis' approach influenced computer architecture (Arvind et al., 1991; Srinivasan, 1986), compiler design, and concurrent programming languages (Johnston et al., 2004). Kahn's approach influenced process algebras (e.g. Broyl and Stefanescu (2001)) and concurrency semantics (e.g. Brock and Ackerman (1981); Matthews (1995)). It has had practical realizations in stream languages (Stephens, 1997) and operating systems (e.g. Unix pipes). Interest in these MoCs has grown with the resurgence of parallel computing, driven by multicore architectures (Creeger, 2005). Dataflow MoCs are being explored for programming parallel machines (Thies et al., 2002), distributed systems (Lzaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003), and embedded systems (Lin et al., 2006; Jantsch and Sander, 2005). There are improved execution policies (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) and standards (Object Management Group (OMG), 2007; Hsu et al., 2004).

Lee and Matsikoudis (2009) bridge the gap between Dennis and Kahn, showing that Kahn's methods extend naturally to Dennis dataflow, embracing the notion of firing. This is done by establishing the relationship between a firing function and the Kahn process implemented as a sequence of firings of that function. A consequence of this analysis is a formal characterization of firing rules and firing functions that preserve determinacy.

Kahn and MacQueen (1977) called the processes in a PN network *coroutines* for an interesting reason. A **routine** or **subroutine** is a program fragment that is “called” by another program. The subroutine executes to completion before the calling fragment can continue executing. The interactions between processes in a PN model are more symmetric, in that there is no caller and callee. When a process performs a blocking read, it is in a sense invoking a routine in the upstream process that provides the data. Similarly, when it performs a write, it is in a sense invoking a routine in the downstream process to process the data. But the relationship between the producer and consumer of the data is much more symmetric than with subroutines.

When using a conventional Ptolemy II actor with the PN director (vs. a custom-written actor), the actor is automatically wrapped in an infinite loop that fires it until either the model halts (see Section 4.1.2 below) or the actor terminates (by returning false from its *postfire* method). When the actor accesses an input, it blocks until input is available. When it sends an output, the output token goes into a **FIFO queue** (first-in, first-out) that is (conceptually) unbounded. The tokens will be eventually delivered in order to the destination actors.

An interesting subtlety arises when using actors that behave differently depending on whether input tokens are available on the inputs. For example, the *AddSubtract* actor will add all available tokens on its *plus* port, and subtract all available tokens on its *minus* port. When executing under the PN director, this actor will not complete its operation until it has received one token from every input channel. This is because accesses to the inputs are blocking. When the actor asks whether a token is available on the input (using the *hasToken* method), the answer is always yes! When it then goes to read that token (using the *get* method of the input port), it will block until there actually is a token available.

Just like dataflow, PN poses challenging questions about boundedness of buffers and about deadlock. PN is expressive enough that these questions are *undecidable*. An elegant solution to the boundedness question is given by Parks (1995) and elaborated by Geilen and Basten (2003). The solution given by Parks is the one implemented in Ptolemy II.

### 4.1.1 Concurrent Firings

PN models are particularly useful when there are actors that do not return immediately when fired. The **InteractiveShell** actor, for example, found in `Sources→SequenceSources`, opens a window (like the top and bottom windows in Figure 4.1) into which a user can enter information. When the actor fires, it displays in the window whatever input it has

received, followed by a prompt (which defaults to “>>”). It then waits for the user to enter a new value. When the user types a value and hits return, the actor outputs an event containing that value.

When this actor fires in response to an input, the firing will not complete until a user enters a value in the window that is opened. If a dataflow director is used (or any other director that fires actors one at a time), then the entire model will block waiting for user

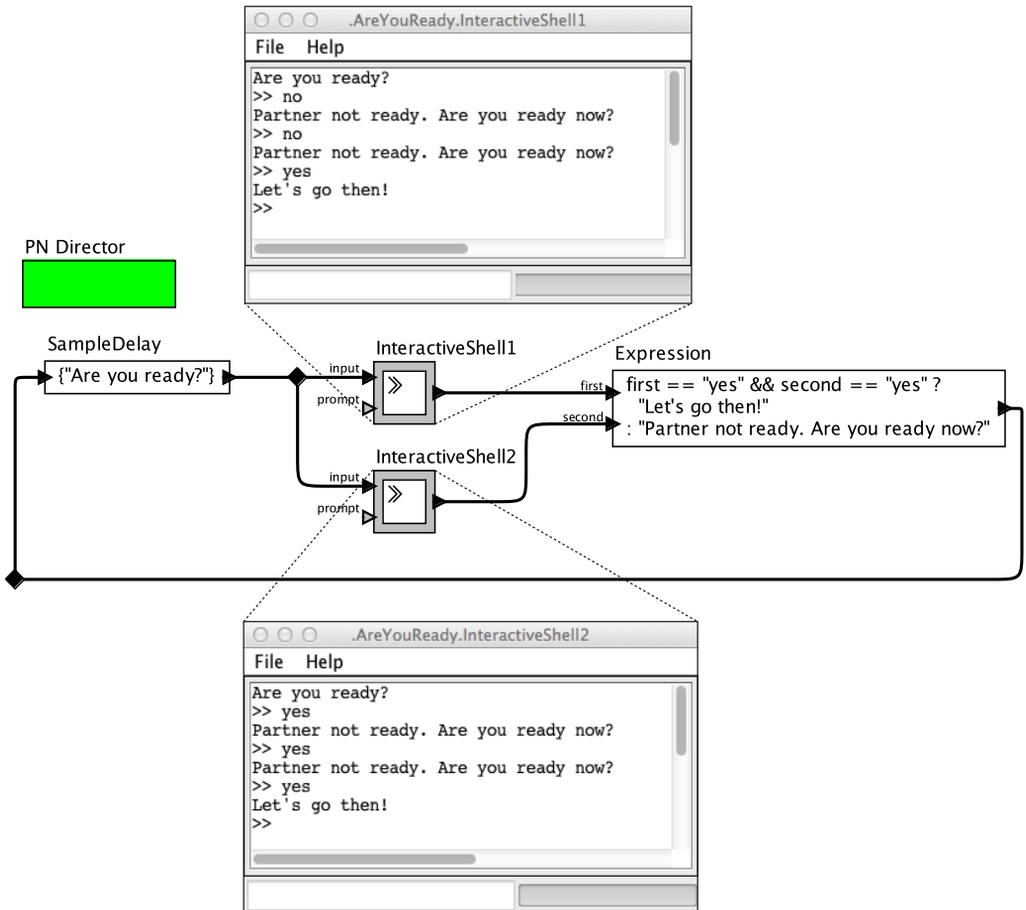


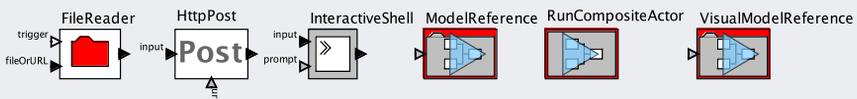
Figure 4.1: A model with two instances of `InteractiveShell`, each of which blocks waiting for user input. [\[online\]](#)

input from just one InteractiveShell. No other actor can fire. For some models, this will be a problem.

**Example 4.1:** Consider the model in Figure 4.1. This model emulates a dialog with two users where it is seeking concurrence from both users before proceeding with some other action. In this case, it simply asks each user “Are you ready?” It waits for one response from each user, and if both users have respond “yes,” then it responds “Let’s go then!” In this PN model, each instance of InteractiveShell executes in its own thread. If this model were executed instead using the SDF director, then the SDF director would fire one of these first, and would not even ask the second user the question until the first one has responded.

### Sidebar: Actors With Unbounded Firings

Below are a few actors that benefit particularly from use of the PN director because they do not (necessarily) return immediately when fired:



- **FileReader** reads a file on the local computer or a URL (uniform resource locator from the Internet). In the latter case, the actor may block for an indeterminate amount of time while the server responds.
- **HttpPost** posts a **record** to a URL on the Internet, emulating what a browser typically does when a user fills in an online form. The actor may block for an indeterminate amount of time while the server responds.
- **InteractiveShell** opens a window and waits for user input.
- **ModelReference** executes a referenced model to completion (or indefinitely, if the model does not terminate).
- **RunCompositeActor** executes a contained model to completion (or indefinitely, if the model does not terminate).
- **VisualModelReference** opens a Vergil window to display a referenced model and executes the model to completion (or indefinitely, if the model does not terminate).

In the previous example, the InteractiveShell actor interacts with the world outside the PN model. It performs I/O, and in particular, it blocks while performing the I/O, waiting for actions from the outside world. Another example that has this property is considered in Exercise 5, which considers actors that collect data from sensors.

As explained above, a key property of Kahn process networks is that models are **deterministic**. It may seem odd to assert that the model in Figure 4.1 is deterministic, however. Clearly, the sequence of values generated by the Expression actor, for example, differs from run to run, because it depends on what users type into the dialog windows that open. Indeed, determinism requires that every actor implement a mathematical *function* from input sequences to output sequences. Strictly speaking, InteractiveShell does not implement such a function. In two different runs, the same sequence of inputs will not generate the same sequence of outputs, because the outputs depend on what a user types. Nevertheless, the Kahn property is valuable, because it asserts that if you fix the user behavior, then the behavior of the model is unique and well defined. If on two successive runs, two users type exactly the same sequence of responses, then the model will behave exactly the same way. The behavior of the model depends *only* on the behavior of the users, and not on the behavior of the thread scheduler.

In Ptolemy II, it is possible to build process networks that are **nondeterminate** using a special actor called a **NondeterministicMerge** (see box on page 143). Such models can be quite useful.

**Example 4.2:** Consider the example in Figure 4.2. This model emulates a chat interaction between two users, where users can provide input in any order. In this model, users type their inputs into the windows opened by the InteractiveShell actors, and their inputs are merged and displayed by the **Display** actor.

In this model, it is *not* true that if on two successive runs, users type the same thing, that the results displayed will be the same. Indeed, if two users type things nearly simultaneously, then the order in which their text appears at the output of the NondeterministicMerge is undefined. Either order is correct. The order that results will depend on the thread scheduler, not on what the users have typed. This contrasts notably with the model in Figure 4.1.

The nondeterminism introduced in the previous model can be quite useful. But it comes at a cost. Models like this are much harder to test, for example. There is no single correct

result of execution. For this reason, `NondeterministicMerge` should be used with caution, and only when it is really needed. If a model can be built using deterministic mechanisms, then it should be built using deterministic mechanisms.

Although the previous example emulates a chat session between two clients, it is not a realistic implementation of a chat application. The two interactive shell windows are opened on the same screen and run in the same process, so two distinct users cannot prac-

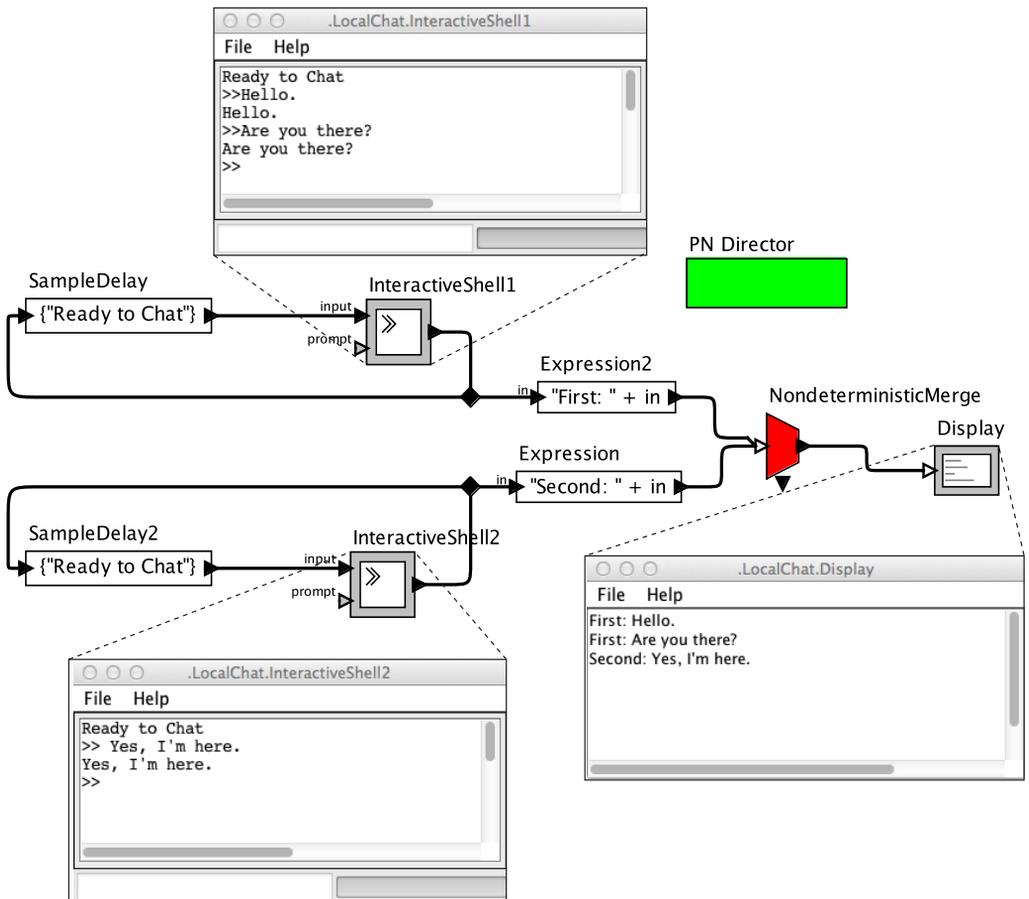


Figure 4.2: A model using a `NondeterministicMerge` to create a nondeterministic process network. [\[online\]](#)

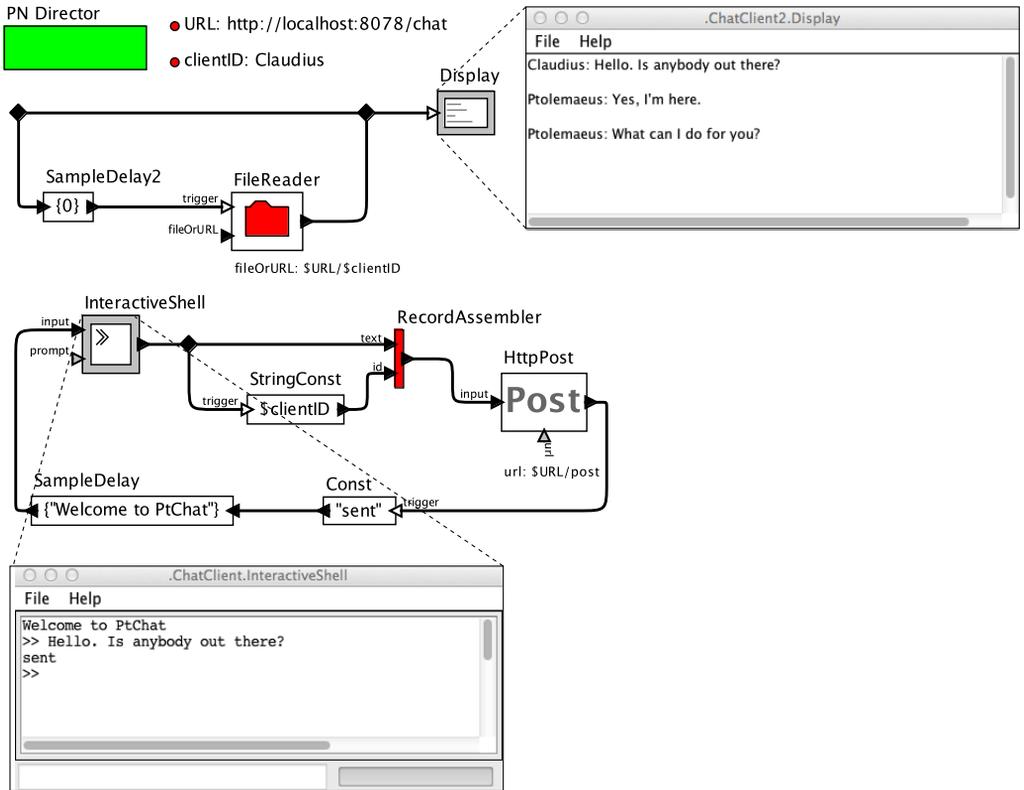


Figure 4.3: A model that implements a chat client. [\[online\]](#)

tically chat. PN can be used to build to build an actual chat client using only deterministic mechanisms.

**Example 4.3:** The model in Figure 4.3 implements a chat client. This model assumes that a server has been deployed at the URL specified by the *URL* parameter of the model. An implementation of such a server is developed in Exercise 1 of Chapter 16.

This client assumes that the server provides two interfaces. An HTTP Get request is used to retrieve chat text that will include text entered by the user of this model

interleaved with text provided by any other participant in the chat session. The upper feedback loop in Figure 4.3 issues this HTTP Get using a `FileReader` actor. Normally, the server will not respond immediately, but rather will wait until it has chat data to provide. Hence, the `FileReader` actor will block, waiting for a response. Upon receiving a response, the model will display the response using the `Display` actor, and then issue another HTTP Get request.

This technique, where Get requests block until there are data to provide, is called **long polling**. In effect, it provides a simple form of push technology, where the server pushes data to a client when there are data to provide. Of course, it only works if the client has issued a Get request, and if the client can patiently wait for a response, rather than, say, timing out.

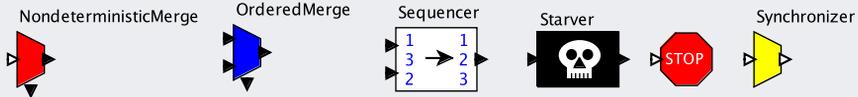
The lower feedback loop in Figure 4.3 is used for this client to supply chat text to the chat session. It uses an `InteractiveShell` to provide a window into which a user can enter text. When the user provides text, it assembles a `record` containing the text and user ID and posts the record to the server using an `HttpPost` actor. When the server replies, the model provides a confirmation to the user, “sent,” and waits for the user to enter new text.

In the meantime, the server will normally respond to the HTTP Post by issuing a reply to all pending instances of HTTP Get. In the example execution shown in Figure 4.3, the local user, Claudius, first types “Hello. Is anybody there?” Another user, Ptolemaeus, somewhere else on the Internet, replies “Yes, I’m here.” The remote user then further types “What can I do for you?” That text is “pushed” to this chat client using the long polling technique, and hence appears on the display spontaneously.

The previous example shows a use of PN to create two simultaneously executing tasks, each of which can block. This model is *deterministic*, in that for any sequence of outputs produced by the `FileReader` and `InteractiveShell`, all signals in the model are uniquely defined. The behavior of the model, therefore, depends *only* on user input, and not on thread scheduling, unlike the example in Figure 4.2. This form of determinism is very valuable. Among the many benefits is that models can be tested by defining input sequences and checking the responses.

### Sidebar: Useful Actors for PN Models

A few actors that are particularly useful in PN models are shown below:



All of these can be found in `DomainSpecific→ProcessNetworks`, but some of them also appear in `Actors→FlowControl`.

- **NondeterministicMerge**. Merge sequences of tokens by arbitrarily interleaving their tokens onto a single stream.
- **OrderedMerge**. Merge two monotonically increasing sequences of tokens into one monotonically increasing sequence of tokens.
- **Sequencer**. Take a stream of input tokens and stream of sequence numbers and re-order the input tokens according to the sequence numbers. On each iteration, this actor reads an input token and a sequence number. The sequence numbers are integers starting with zero. If the sequence number is the next one in the sequence, then the token read from the *input* port is produced on the *output* port. Otherwise, it is saved until its sequence number is the next one in the sequence.
- **Starver**. Pass input tokens unchanged to the output until a specified number of tokens have been passed. At that point, consume and discard all further input tokens. This can be used in a feedback loop to limit the execution to a finite data set.
- **Stop**. Stop execution of a model when a true token is received on any input channel.
- **Synchronizer**. Synchronize multiple streams so that they produce tokens at the same rate. That is, when at least one new token exists on every input channel, exactly one token is consumed from each input channel, and the tokens are output on the corresponding output channels.

### 4.1.2 Stopping Execution of a PN Model

A PN model in Ptolemy II executes until one of the following occurs:

- All actors have terminated. An actor terminates when its `postfire` method returns false. Many actors in the Ptolemy II library have a `firingCountLimit` parameter (e.g., the `Const` actor). Setting this parameter to a positive integer will cause the actor's process to terminate after the actor has fired the specified number of times.
- All processes are blocked on reads of input ports. This is a **deadlock**, similar to the ones that can occur in dataflow models. A deadlock may occur due to an error in the model, or it may be deliberate. For example, the `firingCountLimit` parameter mentioned above and the `Starver` actor (see box on page 143) can be used to create a **starvation** condition, where actors that have not terminated are blocked on reads that will never be satisfied. Despite the pejorative tone of the words “deadlock” and “starvation,” these are perfectly reasonable and useful techniques for terminating the execution of a PN model.
- The `Stop` actor reads a true-valued input token on its one input port. Upon reading this input, the `Stop` actor will coordinate with the director to stop all executing threads. Specifically, any thread that is blocked on a read or write of a port will terminate immediately, and any thread that is not blocked will terminate on its next attempt to perform a read or write. The `Stop` actor can be found in the `Actors` → `FlowControl` → `ExecutionControl` library.
- A buffer overflows. This occurs when the number of unconsumed tokens on a communication channel exceeds the value of the `maximumQueueCapacity` parameter of the director. Note that if you set `maximumQueueCapacity` to 0 (zero), then this will not occur until the operating system denies the Ptolemy system additional memory, which typically occurs when you have run out system memory.
- An exception occurs in some actor process. Other threads are terminated in a manner similar to what the `Stop` actor does.
- A user presses the stop button in Vergil. All threads are terminated in a manner similar to what the `Stop` actor does.

These are the only mechanisms for stopping an execution. How to use them is explored in Exercise 6.

Even so, there are some limitations. The `FileReader` actor, for example, used in Figure 4.3, sadly, cannot be interrupted if it is blocked on a read. Stopping the model in Figure 4.3, therefore, is difficult. If you hit the stop button, it will eventually time out, but the wait may be considerable. This appears to be a limitation in the underlying `java.net.URLConnection` class that is used by this actor.

## 4.2 Rendezvous

In the **Rendezvous** domain in Ptolemy II, like PN, each actor executes in its own `thread`. Unlike PN, communication between actors is by **rendezvous** rather than message passing with unbounded `FIFO queues`. Specifically, when an actor is ready to send a message via an output port, it blocks until the receiving actor is ready to receive it. Similarly if an actor is ready to receive a message via an input port, it blocks until the sending actor is ready to send it. Thus, this domain realizes both **blocking writes** and **blocking reads**.

This domain supports both conditional and multiway rendezvous. In **conditional rendezvous**, an actor is willing to rendezvous with any one of several other actors.<sup>2</sup> In **multiway rendezvous**, an actor requires rendezvous with multiple other actors at the same time.<sup>3</sup> When using conditional rendezvous, the choice of which rendezvous occurs is **nondeterministic**, in general, since which rendezvous occurs will generally depend on the thread scheduler.

The Rendezvous domain is based on the communicating sequential processes (**CSP**) model first proposed by [Hoare \(1978\)](#) and the calculus of communicating systems (**CCS**), given by [Milner \(1980\)](#). It also forms the foundation for the **Occam** programming language ([Galletly, 1996](#)), which enjoyed some success for a period of time in the 1980s and 1990s for programming parallel computers.

Rendezvous-based communication is also known as **synchronous message passing**, but we avoid this term to avoid confusion with the **SR** (synchronous-reactive) domain, described in Chapter 5, and the **SDF** domain, described in Chapter 3.

---

<sup>2</sup>For those familiar with the Ada language, this is similar to the `select` statement.

<sup>3</sup>Multiway rendezvous is also called **barrier synchronization**, since it blocks each participating thread until all participating threads have reached a barrier point, indicated by a `send` or a `get` via a port.

### Sidebar: Useful Actors for Rendezvous Models

A few particularly useful actors in Rendezvous models are shown below:



- **Barrier.** A [barrier synchronization](#) actor. This actor will accept inputs only when the sending actors on *all* input channels are ready to send.
- **Buffer.** A FIFO buffer. This actor buffers data provided at the input, sending it to the output when needed. The actor is willing to rendezvous with the output whenever the buffer is not empty. It is willing to rendezvous with the input as long as the buffer is not full. Inputs are delivered to the output in FIFO (first-in, first-out) order. You can specify a finite or infinite capacity for the buffer.
- **Merge.** A [conditional rendezvous](#). This actor merges any number of input sequences onto one output sequence. It begins with a rendezvous with any input. When it receives an input, it will then rendezvous with the output. After successfully delivering the input token to the output, it returns again to being willing to rendezvous with any input.
- **ResourcePool.** A resource contention manager. This actor manages a pool of resources, where each resource is represented by a token with an arbitrary value. Resources are granted on the *grant* output port and released on the *release* input port. These ports are both [multiports](#), so resources can be granted to multiple users of the resources, and released by multiple actors. The initial pool of resources is provided by the *initialPool* parameter, which is an array of arbitrary type. At all times during execution, this actor is ready to rendezvous with any other actor connected to its *release* input port. When such a rendezvous occurs, the token provided at that input is added to the resource pool. In addition, whenever the resource pool is non-empty, this actor is ready to rendezvous with any actor connected to its *grant* output port. If there are multiple such actors, this will be a [conditional rendezvous](#), and hence may introduce [nondeterminism](#) into the model. When such an output rendezvous occurs, the actor sends the first token in the resource pool to that output port and removes that token from the resource pool.

## 4.2.1 Multiway Rendezvous

In **multiway rendezvous**, an actor performs a rendezvous with multiple other actors at the same time. Two mechanisms are provided by the Rendezvous director for this. First, if an actor sends an output to multiple other actors, the communication forms a multiway rendezvous. All destination actors must be ready to receive the token before any destination actor will receive it. And the sender will block until all destination actors are ready to receive the token.

**Example 4.4:** The model in Figure 4.4 illustrates multiway rendezvous. In this example, the **Ramp** is sending an increasing sequence of integers to the **Display**. However, the transfer is constrained to occur only when the **Sleep** actor reads inputs, because in the Rendezvous domain, sending data to multiple recipients via a **relation** is accomplished via a multiway rendezvous. The **Sleep** actor reads data, then sleeps an amount of time given by its bottom input before reading the next input data token. In this case, it will wait a random amount of time (given by the **Uniform** random-number generator) between input readings. This has the side effect of constraining the transfers from the **Ramp** to the **Display** to occur with the same random intervals.

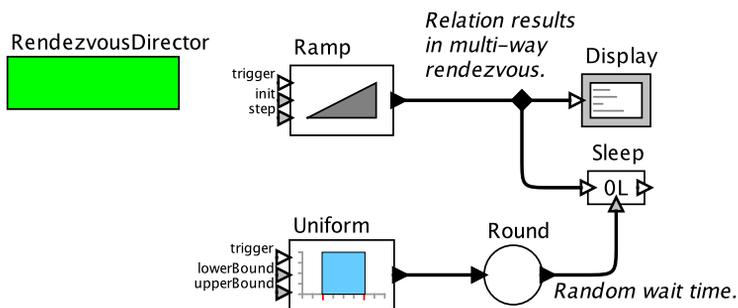


Figure 4.4: An illustration of multiway rendezvous, where the timing of the communication between the Ramp and the Display is controlled by the Sleep actor. [\[online\]](#)

The **Barrier** actor (see box on page 146) also performs multiway rendezvous.

**Example 4.5:** The model in Figure 4.5 illustrates multiway rendezvous using the Barrier actor. In this example, the two Ramp actors are sending increasing sequences of integers to the Display actors. Again, the transfer is constrained to occur only when both the Barrier actor and the Sleep actor read inputs. Thus, a multiway rendezvous between the two Ramp actors, the two Display actors, the Barrier actor, and the Sleep actor constrains the two transfers to the Display actors to occur simultaneously.

## 4.2.2 Conditional Rendezvous

In **conditional rendezvous**, an actor is willing to rendezvous with any one of several other actors. Usually, this results in **nondeterministic** models.

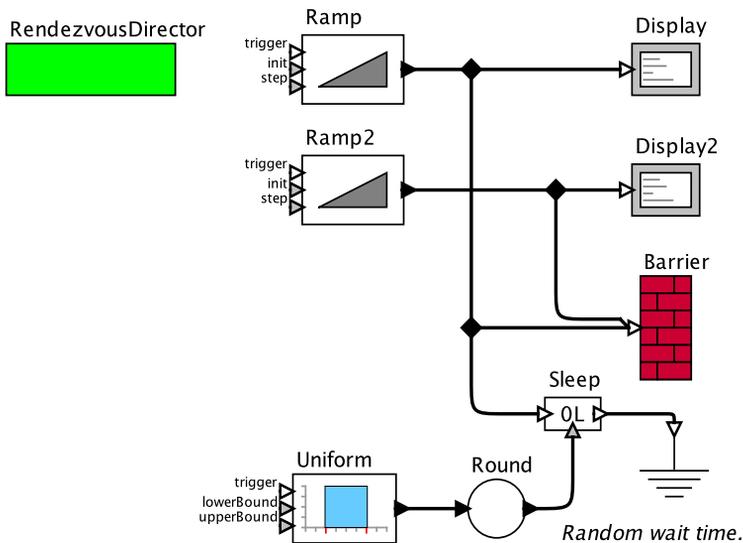


Figure 4.5: An illustration of multiway rendezvous using the Barrier actor. [[online](#)]

**Example 4.6:** The model in Figure 4.6 illustrates conditional rendezvous. This model uses the [Merge](#) actor (see box on page 146). The top Ramp actor will produce the sequence 0, 1, 2, 3, 4, 5, 6, 7 on its output. The bottom Ramp will produce the sequence -1, -2, -3, -4, -5, -6, -7, -8. The Display actor will display a nondeterministic merging of these two sequences.

The example in Figure 4.6 includes a parameter *SuppressDeadlockReporting* with value true. The Ramp actors starve the model by specifying a finite *firingCountLimit*, thus providing a stopping condition similar to the ones we would use with PN (see Section 4.1.2). By default, the director will report the [deadlock](#), but with the *SuppressDeadlockReporting* parameter, it silently stops the execution of the model. This parameter indicates that the deadlock is a normal termination and not an error condition.

Suppose that, unlike the previous example, we wish to deterministically interleave the outputs of the two Ramp actors, alternating their outputs to get the sequence 0, -1, 1, -2, 2, ... One way to accomplish that, due to [Arbab \(2006\)](#), is shown in Figure 4.7. This model relies on the fact that the input to the [Buffer](#) actor (see box on page 146) participates in a multiway rendezvous with both instances of Ramp and the top channel of the Merge actor. Since it has capacity one, it forces this rendezvous to occur before it provides an input to

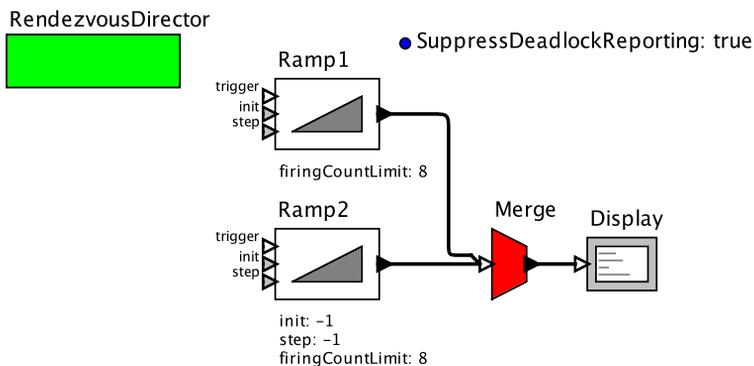


Figure 4.6: An illustration of conditional rendezvous for nondeterministic merge. [\[online\]](#)

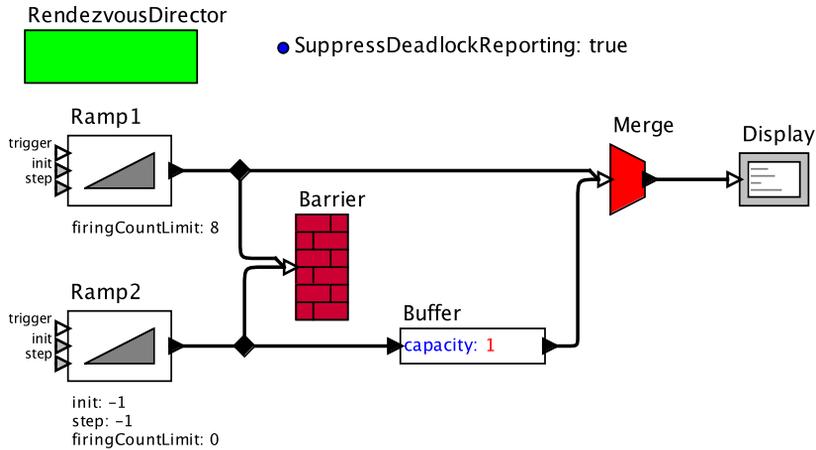


Figure 4.7: An illustration of conditional rendezvous used to create a deterministic merge. [\[online\]](#)

the bottom channel of the merge, and it blocks subsequent instances of this rendezvous until after it has provided the bottom input to the Merge.

Although this model is extremely clever, it is using *nondeterministic* mechanisms to accomplish deterministic aims. In fact, it is easy to construct a much simpler model that accomplishes the same goal without any nondeterministic mechanisms (see Exercise 7).

### 4.2.3 Resource Management

The *conditional rendezvous* mechanism provided by the Rendezvous director is ideally suited to resource management problems, where actors compete for shared resources. Generally, nondeterminism is acceptable and expected for such models. The *Resource-Pool* actor (see box on page 146) is ideal for such applications.

**Example 4.7:** The model in Figure 4.8 illustrates resource management where a pool (in this case containing only one resource) provides that resource nondeterministically to one of two *Sleep* actors. The resource is represented by an integer that initially has value 0. This value will be incremented each time the resource is

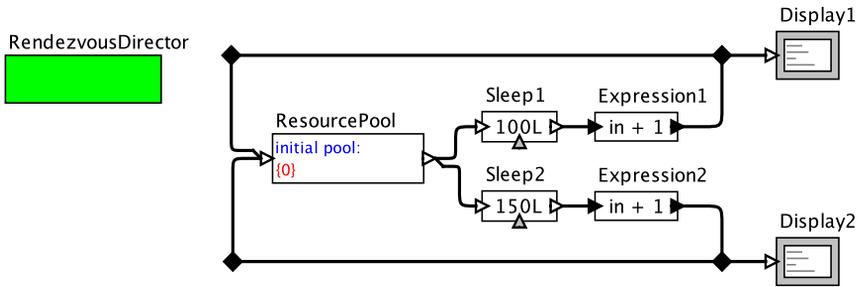


Figure 4.8: An illustration of conditional rendezvous for resource management. [\[online\]](#)

used. The Sleep actor that gets the resource holds it for a fixed amount of time (100 and 150 ms, respectively). After this time, it releases the resource, sending it to an **Expression** actor, which increments the value of the resource and then returns it to the resource pool.

The input and output ports of the ResourcePool actor both realize a **conditional rendezvous**. Hence, it is nondeterministic which Sleep actor will acquire the resource when both are ready for the resource. Note that there is no assurance of fairness in this system, and in fact it is possible for only one of the two Sleep actors to get access to the resource.

### 4.3 Summary

The two domains described in this chapter, PN and Rendezvous, both execute each actor in a model in its own thread. A PN model is deterministic as long as it does not include an instance of **NondeterministicMerge**. The Rendezvous domain is not generally deterministic. PN is particularly useful when models include actors that may block for indeterminate amounts of time, and where we don't wish to block the entire the model when this occurs. Rendezvous is particularly useful for resource management problems, where there is contention for limited resources, and for models where the timing of concurrent actions needs to be synchronized.

## Exercises

The purpose of these exercises is to develop some intuition about the process networks model of computation and how programming with it differs from programming with an imperative model.<sup>4</sup> For all of the following exercises, you should use the PN Director and “simple” actors to accomplish the task. In particular, the following actors are sufficient:

- [Ramp](#) and [Const](#) (in `Sources`)
- [Display](#) and [Discard](#) (in `Sinks`)
- [BooleanSwitch](#) and [BooleanSelect](#) (in `FlowControl`→`BooleanFlowControl`)
- [SampleDelay](#) (in `FlowControl`→`SequenceControl`)
- [Comparator](#), [Equals](#), [LogicalNot](#), or [LogicGate](#) (in `Logic`)

Feel free to use any other actor that you believe to be “simple.” Also, feel free to use any other actors, simple or not, for testing your composite actors, but stick to simple ones for the implementation of the composite actors.

1. The [SampleDelay](#) actor produces initial tokens. In this exercise, you will create a composite actor that consumes initial tokens, and hence be thought of as a negative delay.
  - (a) Create a PN model containing a [composite actor](#) with one input port and one output port, where the output sequence is the same as the input sequence except that the first token is missing. That is, the composite actor should discard the first token and then act like an identity function. Demonstrate by some suitable means that your model works as required.
  - (b) Augment your model so that the number of initial tokens that are discarded is given by a parameter of the composite actor. **Hint:** It may be useful to know that the expression language<sup>5</sup> (see Chapter 13) includes a built-in function `repeat`, where, for example,

---

<sup>4</sup>You may want to run `vergil` with the `-pn` option, which gives you a subset of Ptolemy II that is more than adequate to do these exercises. To do this on the command line, simply type “`vergil -pn`”. If you are running Ptolemy II from Eclipse, then in the toolbar of the Java perspective, select Run Configurations. In the Arguments tab, enter `-pn`.

<sup>5</sup>Note that you can easily explore the expression language by opening an ExpressionEvaluator window, available in the [File→New] menu. Also, clicking on Help in any parameter editor window will provide documentation for the expression language.

`repeat(5, 1) = {1, 1, 1, 1, 1}`

2. This problem explores operations on a stream of data that depend on the data in the stream.
  - (a) Create a PN model containing a composite actor with one input port and one output port, where the output sequence is the same as the input sequence except that any consecutive sequence of identical tokens is replaced by just one token with the same value. That is, redundant tokens are removed. Demonstrate by some suitable means that your model works as required.
  - (b) Can your implementation run forever with bounded buffers? Give an argument that it can, or explain why there is no implementation that can run with bounded buffers.
3. Create an implementation of an `OrderedMerge` actor using only “simple” PN actors. (Note that there is a Java implementation of `OrderedMerge`, which you should not use, see box on page 143.) Your implementation should be a composite actor with two input ports and one output port. Given any two numerically increasing sequences of tokens on the input ports, your actor should merge these sequences into one numerically increasing sequence without losing any tokens. If the two sequences contain tokens that are identical, then the order in which they come out does not matter.
4. In Figure 4.9 is a model that generates a sequence of numbers known as the **Hamming numbers**. These have the form  $2^n 3^m 5^k$ , and they are generated in numerically increasing order, with no redundancies. This model can be found in the PN demos (labeled as `OrderedMerge`). Can this model run forever with bounded buffers? Why or why not?

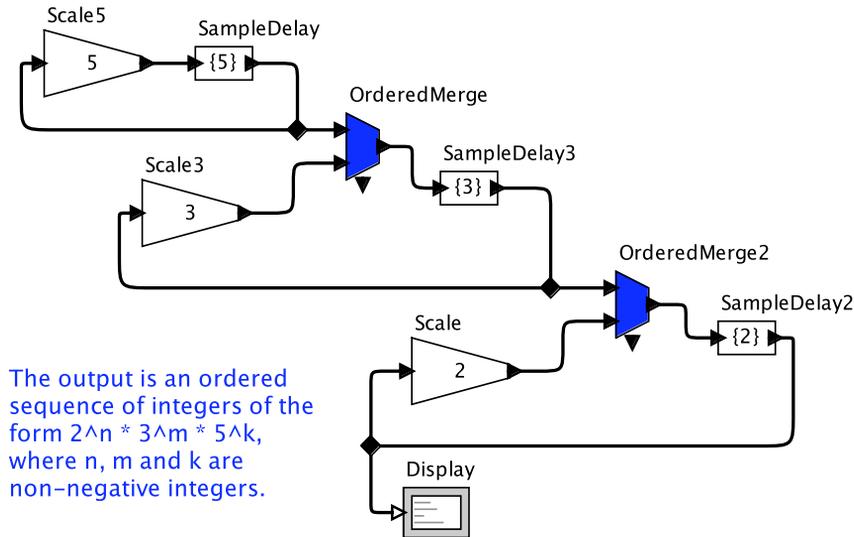
For this problem, assume that the data type being used is unbounded integers, rather than what is actually used, which is 32 bit integers. With 32 bit integers, the numbers will quickly overflow the representable range of the numbers, and wrap around to negative numbers.

5. A common scenario in embedded systems is that multiple sensors provide data at different rates, and the data must be combined to form a coherent view of the physical world. In general, this problem is called **sensor fusion**. The signal processing involved in forming a coherent view from noisy sensor data can be quite sophisticated, but in this exercise we will focus not on the signal processing, but rather on the concurrency and logical control flow. At a low level, sensors are connected to

PN Director



This model, whose structure is due to Kahn and MacQueen, calculates integers whose prime factors are only 2, 3, and 5, with no redundancies. It uses the OrderedMerge actor, which takes two monotonically increasing input sequences and merges them into one monotonically increasing output sequence.



The output is an ordered sequence of integers of the form  $2^n * 3^m * 5^k$ , where  $n$ ,  $m$  and  $k$  are non-negative integers.

Figure 4.9: Model that generates a sequence of Hamming numbers. [\[online\]](#)

embedded processors by hardware that will typically trigger processor interrupts, and interrupt service routines will read the sensor data and store it in buffers in memory. The difficulties arise when the rates at which the data are provided are different (they may not even be related by a rational multiple, or may vary over time, or may even be highly irregular).

Assume we have two sensors, SensorA and SensorB, both making measurements of the same physical phenomenon that happens to be a sinusoidal function of time, as follows:

$$\forall t \in \mathbb{R}, \quad x(t) = \sin(2\pi t/10)$$

Assuming time  $t$  is in seconds, this has a frequency of 0.1 Hertz. Assume further that the two sensors sample the signal with distinct sampling intervals to yield the

SDF Director

Model of a sensor sensing a sinusoidal signal with the specified frequency and phase at the specified sampling frequency. This composite actor simulates real-time behavior by sleeping the amount of time given by the samplingPeriod (in seconds) before producing an output.

- frequency: 1.0
- phase: 0.0
- samplingPeriod: 0.1
- noiseStandardDeviation: 0.1

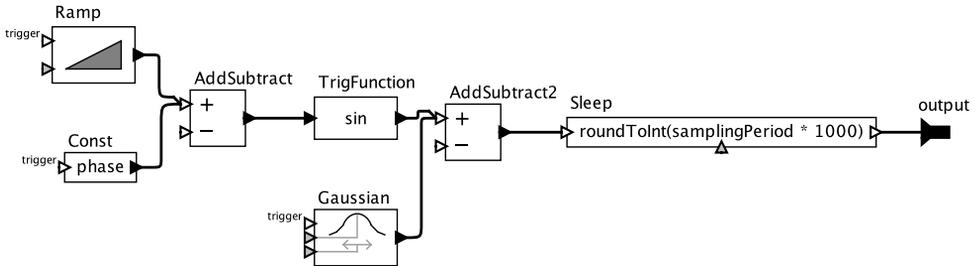


Figure 4.10: Model of a real-time sensor. [\[online\]](#)

following measurements:

$$\forall n \in \mathbb{Z}, \quad x_A(n) = x(nT_A) = \sin(2\pi nT_A/10),$$

where  $T_A$  is sampling interval of SensorA. A similar set of measurements is taken by SensorB, which samples with period  $T_B$ .

A model of such a sensor for use with the PN director of Ptolemy II is shown in figure 4.10. You can create an instance of that sensor in Vergil by invoking the [Graph→Instantiate Entity] menu command, and filling in the boxes as follows:

```
class: SensorModel
location (URL): http://embedded.eecs.berkeley.edu/
concurrency/models/SensorModel.xml
```

Create two instances of the sensor in a Ptolemy II model with a PN director.

The sensor has some parameters. The *frequency* you should set to 0.1 to match the equations above. The *samplingPeriod* you should set to 0.5 seconds for one of the sensor instances, and 0.75 seconds for the other. You are to perform the following experiments.<sup>6</sup>

<sup>6</sup>You may find the actors described in the sidebars on pages 106, 107, and 119 useful.

- (a) Connect each sensor instance to its own instance of the [SequencePlotter](#). Execute the model. You will likely want to change the parameters of the [SequencePlotter](#) so that *fillOnWrapup* is `false`, and you will want to set the *X Range* of the plot to, say, “0.0, 50.0” (do this by clicking on the second button from the right at the upper right of each plot). Describe what you see. Do the two sensors accurately reflect the sinusoidal signal? Why do they appear to have different frequencies?
  - (b) A simple technique for sensor fusion is to simply average sensor data. Construct a model that averages the data from the two sensors by simply adding the samples together and multiplying by 0.5. Plot the resulting signal. Is this signal an improved measurement of the signal? Why or why not? Will this model be able to run forever with bounded memory? Why or why not?
  - (c) The sensor fusion strategy of averaging the samples can be improved by normalizing the sample rates. For the sample periods given, 0.5 and 0.75, find a way to do this in PN. Comment about whether this technique would work effectively if the sample periods did not bear such a simple relationship to one another. For example, suppose that instead of 0.5 seconds, the period on the first sensor was 0.500001.
  - (d) When sensor data is collected at different rates without a simple relationship, one technique that can prove useful is to create [time stamps](#) for the data and to use those time stamps to improve the measurements. Construct a model that does this, with the objective simply of creating a plot that combines the data from the two sensors in a sensible way.
6. In this problem, we explore how to use the mechanisms of Section [4.1.2](#) to deterministically halt the execution of a PN model. Specifically, in each case, we consider a [Source](#) actor feeding a potentially infinite sequence of data tokens to a [Display](#) actor. We wish to make this sequence finite with a specific length, and we wish to ensure that the [Display](#) actor displays every element of the sequence.
- (a) Suppose that you have a [Source](#) actor with one output port and no parameters whose process iterates forever producing outputs. Suppose that its outputs are read by a [Display](#) actor, which has one input port and no output ports. Find a way to use the [Stop](#) actor to deterministically stop the execution, or argue that there is no way to do so. Specifically, the [Source](#) actor should produce a specified number of outputs, and every one of these outputs should be consumed and displayed by the [Display](#) actor before execution halts.

- (b) Most Source actors in Ptolemy II have a *firingCountLimit* parameter that limits the number of outputs they produce. Show that this can be used to deterministically halt the execution without the help of a **Stop** actor.
  - (c) Many Source actors in Ptolemy II have *trigger* input ports. If these inputs are connected, then the actor process will read a value from that input port before producing each output. Show how to use this mechanism, with or without the **Stop** actor, to achieve our goal of deterministically halting execution, or argue that it is not possible to do so. Again, the Source should produce a pre-specified amount of data, and the Display should consume and display all of that data. You may use **Switch**, **Select**, or any other reasonably simple actor. Be sure to explain each actor you use, unless you are sure it is exactly the actor provided in the Vergil library.
7. Figure 4.7 shows a model that deterministically interleaves the outputs of two Ramp actors. That model uses a nondeterministic mechanism (the **conditional rendezvous** of the **Merge** actor), and then carefully regulates the nondeterminism using a **multiway rendezvous** and a **Buffer** actor. The end result is deterministic. However, the same objective (deterministically interleaving two streams in an alternating, round-robin fashion) can be accomplished with purely deterministic mechanisms. Construct a Rendezvous model that does this. **Hint:** Your model will probably work unchanged with using the PN or SDF directors instead of Rendezvous.

# Synchronous-Reactive Models

*Stephen A. Edwards, Edward A. Lee, Stavros Tripakis, Paul Whitaker*  
*In memory of Paul Caspi*

## Contents

---

<b>5.1</b>	<b>Fixed-Point Semantics</b> . . . . .	<b>160</b>
<b>5.2</b>	<b>SR Examples</b> . . . . .	<b>163</b>
5.2.1	Acyclic Models . . . . .	163
5.2.2	Feedback . . . . .	164
	<i>Sidebar: About Synchrony</i> . . . . .	165
	<i>Sidebar: Synchronous-Reactive Languages</i> . . . . .	166
	<i>Sidebar: Domain-Specific SR Actors</i> . . . . .	167
5.2.3	Causality Loops . . . . .	174
5.2.4	Multiclock Models . . . . .	175
<b>5.3</b>	<b>Finding the Fixed-Point</b> . . . . .	<b>176</b>
<b>5.4</b>	<b>The Logic of Fixed Points</b> . . . . .	<b>178</b>
	<i>Sidebar: Causality in Synchronous Languages</i> . . . . .	179
	<i>Sidebar: CPOs, Continuous Functions and Fixed Points</i> . . . . .	182
<b>5.5</b>	<b>Summary</b> . . . . .	<b>184</b>
	<b>Exercises</b> . . . . .	<b>185</b>

---

The **synchronous-reactive (SR)** model of computation is designed for modeling systems that involve synchrony, a fundamental concept in concurrent systems (see sidebar on page 165). It is an appropriate choice for modeling applications with complicated control logic where many things are happening at once (concurrently) and yet **determinism** and precise control are important. Such applications include embedded control systems, where safety must be preserved. SR systems are good at orchestrating concurrent actions, managing shared resources, and detecting and adapting to faults in a system. Whereas **dataflow** models are good for managing streams of data, SR systems are good at managing sporadic data, where events may be present or absent, and where the absence of events has meaning (more than just transport delay). For example, detecting the absence of an event may be an essential part of a **fault management** system. SR is also a good model of computation for coordinating **finite state machines**, described in Chapters 6 and 8, which can be used to express the control logic of the individual actors that are concurrently executed.

The Ptolemy II SR domain has been influenced by the family of so-called **synchronous languages** (see sidebar on page 166) and in particular **dataflow synchronous languages** such as **Lustre** (Halbwachs et al., 1991) and **Signal** (Benveniste and Le Guernic, 1990). SR primarily realizes the model of **synchronous block diagrams** as described by Edwards and Lee (2003b). The model of computation is closely related to synchronous digital circuits. In fact, this chapter will illustrate some of the ideas using circuit analogies, although the SR domain is intended more for modeling embedded software than circuits.

SR can be viewed as describing **logically timed** systems. In such systems, time proceeds as a sequence of discrete steps, called **reactions** or **ticks**. Although the steps are ordered, there is not necessarily a notion of “time delay” between steps like there is in discrete time systems; and there is no *a priori* notion of real time. Thus, we refer to time in this domain as **logical time** rather than discrete time.

The similarities and differences with **dataflow** models are:

1. Like **homogeneous SDF**, an iteration of an SR model consists of one iteration of each actor in the model. Each iteration of the model corresponds to one tick of the logical clock. Indeed, most of the SDF models considered in Chapter 2 could just as easily have been SR models. For example, the behavior of the channel model in Figure 2.29 and all of its variants would behave identically under the SR director.
2. Unlike dataflow and **process networks**, there is no buffering on the communication between actors. In SR, and output produced by one actor is observed by the destination

actors in the same tick. Unlike *rendezvous*, which also does not have buffered communication, SR is *determinate*.

3. Unlike dataflow, an input or output may be **absent** at a tick. In dataflow, the absence of an input means simply that the input hasn't arrived yet. In SR, however, an absent signal has more meaning. Its absence is not a consequence of accidents of scheduling or of the time that computation or communication may take. Instead, the absence of a signal at a tick is *defined* deterministically by the model. As a consequence, in SR, actors may react to the *absence* of a signal. This is quite different from dataflow, where actors react only the *presence* of a signal.
4. As we will explain below, in SR, an actor may be fired multiple times between invocations of *postfire*. That is, one *iteration* of an actor may consist of more than invocation of the *fire* method. For simple models, particularly those without feedback, you will never notice this. Sometimes, however, significant subtleties arise. We focus on such models in this chapter.

## 5.1 Fixed-Point Semantics

Consider a model with three actors with the structure shown in Figure 5.1(a). Let  $n$  denote the tick number. The first tick of the local clock corresponds to  $n = 0$ , the second to  $n = 1$ , etc. At each tick, each actor implements an input-output function (which typically changes from tick to tick, possibly in ways that depend on previous inputs). For example, actor 1, in tick 0, implements the function  $f_1(0)$ . That is, given an input value  $s_1(0)$  on port  $p1$ , it will produce output value  $s_2(0) = (f_1(0))(s_1(0))$  on output port  $p5$ .

At any tick, an input may be absent; in this view, “absent” is treated like any other value. The actor can respond to an absent input, and it may assert an absent output or assign the output some value compatible with the data type of the output port.

Each actor thus produces a sequence of values (or absent values), one at each tick. Actor 1 produces values  $s_2(0), s_2(1), \dots$ , while actor 2 produces  $s_1(0), s_2(1), \dots$ , and actor 3 produces  $s_3(0), s_3(1), \dots$ , where any of these can be absent. The job of the SR director is to find these values (and absences). This is what it means to execute the model.

As illustrated in Figures 5.1(b) through (d), any SR model may be rearranged so that it becomes a single actor with function  $f(n)$  at tick  $n$ . The domain of this function is a tuple of values (or absences)  $s(n) = (s_1(n), s_2(n), s_3(n))$ . So is the codomain. Therefore, at



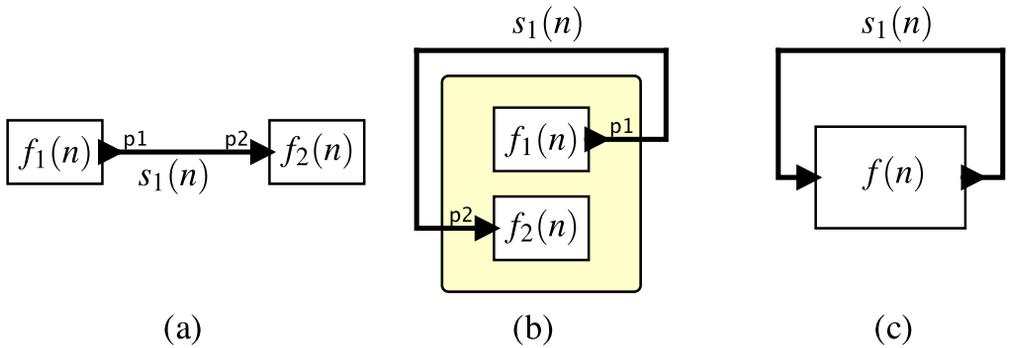


Figure 5.2: Even an SR model without feedback is reducible to a fixed point problem at each tick of the logical clock.

because when there is feedback, an actor may be reacting to an input *that is a function of its own output*. Clearly, one can get trapped in **causality** problems, where the input is not known until the output is known, and the output can't be known until the input is known. Indeed, such causality problems are the major source of subtlety in SR models.

A simple case of SR is a model without feedback, as shown in Figure 5.2. Even such a model is reducible to a fixed-point problem, but in this case it becomes a rather simple problem. The function  $f_1(n)$  at tick  $n$  only needs to be evaluated once at each tick, and it immediately finds the fixed point. The function  $f_2(n)$  never needs to be evaluated (from the perspective of the SR director), but the SR director fires and postfixes actor 2 anyway because of the side effects it may have (e.g. updating a display). But actor 2 plays no role in finding the fixed point.

Once the director has found the fixed point, it can then allow each actor to update its function to  $f(n + 1)$  in preparation for the next tick. Indeed, this is what an actor does in its **postfire** phase of execution. An iteration of the model, therefore, consists of some number of firings of the actors, until a fixed point is found, followed by one invocation of postfire, allowing the actor update its state in reaction to the inputs provided by the fixed point that was found. The details of how this execution is carried out are described below in Section 5.3, but first, we consider some examples.

## 5.2 SR Examples

### 5.2.1 Acyclic Models

SR models without feedback are much like [homogeneous SDF](#) models without feedback, except that signals may be absent. The ability to have absent signals can be convenient for controlling the execution of actors.

**Example 5.1:** Recall the if-then-else of [Figure 3.10](#), which uses [dynamic dataflow](#) to conditionally route tokens to the computations to be done. A similar effect can be achieved in SR using [When](#) and [Default](#) (see sidebar on page 167), as shown in [Figure 5.3](#). This model operates on a stream produced by the [Ramp](#) actor in one of

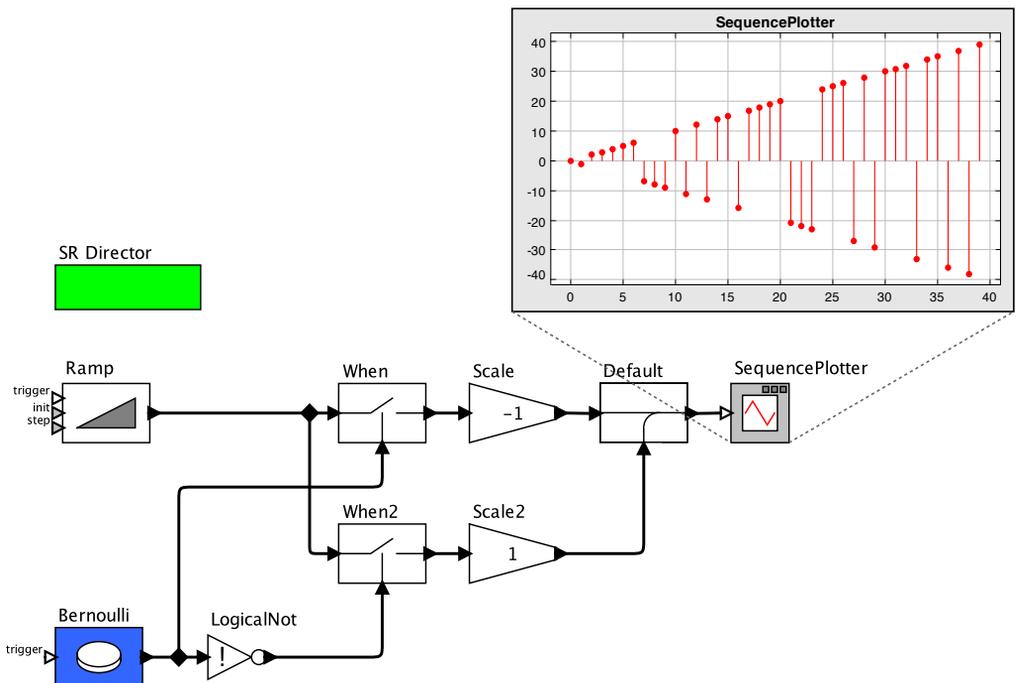


Figure 5.3: A model accomplishing conditional execution using SR. [\[online\]](#)

two (rather trivial) ways. Along the top path, it multiplies the stream by  $-1$ . Along the bottom path, it multiplies by  $1$ . Such a pattern might be used, for example, to model intermittent failures in a system.

The **Bernoulli** actor generates a random boolean that is used to control two instances of **When**. The top **When** actor will convey the output from the **Ramp** to its output when the boolean is true. The bottom **When** actor will convey the output from the **Ramp** to its output when the boolean is false. When the output of a **When** actor is **absent**, then the downstream **Scale** actor will also have an absent output. Hence, the **Default** actor will have only one present input in each tick, and it will convey that input to its output. Finally, the **SequencePlotter** plots the result.

Whereas with dataflow models, it is possible to make wiring errors that will result in unbounded buffers, as for example in Figure 3.13, in SR, execution is always bounded. Every connection between actors stores at most one token on each tick of the clock. Hence, there is no mechanism for memory usage to become unbounded (unless, of course, an actor does so internally).

## 5.2.2 Feedback

More interesting SR models involve feedback (directed cycles in the graph), as in Figure 5.1. With such feedback systems, **causality** becomes a concern. Consider in particular the relationship between actors 1 and 2 in Figure 5.1(a). At a tick  $n$  of the logical clock, it would seem that we need to know  $s_1(n)$  in order to evaluate function  $f_1(n)$ . But to know  $s_1(n)$ , it seems we need to evaluate  $f_2(n)$ . But to evaluate  $f_2(n)$ , it seems we need to know  $s_2(n)$ , which requires evaluating  $f_1(n)$ . We appear to have gotten stuck in a **causality loop**.

Causality loops must be broken by **non-strict actors**. An actor is said to be **strict** if it requires knowledge of all its inputs in order to provide outputs. If it can provide outputs without full knowledge of the inputs, then it is non-strict. The simplest non-strict actor is the **NonStrictDelay** (see box on page 167). It can be used to break causality loops, as illustrated in the following example.

### Sidebar: About Synchrony

The general definitions of the term **synchronous** are (1) occurring or existing at the same time or (2) moving or operating at the same rate. In engineering and computer science, the term has a number of meanings that are mostly consistent with these definitions, but oddly inconsistent with one another. In referring to concurrent software using threads or processes, synchronous communication refers to a **rendezvous** style of communication, where the sender of a message must wait for the receiver to be ready to receive, and the receiver must wait for the sender. Conceptually, the communication occurs at the same time from the perspective of each of the two threads, consistent with definition (1). In Java, however, the keyword `synchronized` defines blocks of code that are *not* permitted to execute simultaneously, which is inconsistent with both definitions.

There is yet a third meaning of the word synchronous, which is the definition we use in this chapter. This third meaning underlies **synchronous languages** (see box on page 166). Two key ideas govern these languages. First, the outputs of components in a program are (conceptually) simultaneous with their inputs (this is called the **synchrony hypothesis**). Second, components in a program execute (conceptually) **simultaneously and instantaneously**. Even though this cannot occur in reality, a correct execution must behave as though it did. This interpretation is consistent with *both* definitions (1) and (2) above, since executions of components occur at the same time and operate at the same rate.

In circuit design, the word synchronous refers to a style where a clock signal that is distributed throughout a circuit causes circuit components called “latches” to record their inputs on the rising or falling edges of the clock. The time between clock edges needs to be sufficient for circuit gates between latches to settle. Conceptually, this model is very similar to the model in synchronous languages. Assuming that the gates between latches have zero delay is equivalent to the synchrony hypothesis, and global clock distribution gives simultaneous and instantaneous execution of those gates. Hence, the SR domain is often useful for modeling digital circuits.

In power systems engineering, synchronous means that electrical waveforms have the same frequency and phase. In signal processing, synchronous means that signals have the same sample rate, or that their sample rates are fixed multiples of one another. The term **synchronous dataflow**, described in Chapter 3.1, is based on this latter meaning of the word synchronous. This usage is consistent with definition (2).

### Sidebar: Synchronous-Reactive Languages

The synchronous-reactive model of computation dates back to at least the mid-1980s, when a number of programming languages were developed. The term “reactive” comes from a distinction in computational systems between **transformational systems**, which accept input data, perform a computation, and produce output data, and **reactive systems**, which engage in an ongoing dialog with their environment (Harel and Pnueli, 1985). Manna and Pnueli (1992) state

“The role of a reactive program ... is not to produce a final result but to maintain some ongoing interaction with its environment.”

The distinctions between transformational and reactive systems led to the development of a number of innovative programming languages. The **synchronous languages** (Benveniste and Berry, 1991) take a particular approach to the design of reactive systems, in which pieces of the program react **simultaneously and instantaneously** at each tick of a global clock. Primary among these languages are Lustre (Halbwachs et al., 1991), Esterel (Berry and Gonthier, 1992), and Signal (Le Guernic et al., 1991). Statecharts (Harel, 1987) and its implementation in Statemate (Harel et al., 1990) also have a strongly synchronous flavor.

SCADE (Berry, 2003) (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on Lustre, borrows concepts from Esterel, and provides a graphical syntax in which state machines similar to those in Chapter 6 are drawn and actor models are composed synchronously. One of the main attractions of synchronous languages is their strong formal properties that facilitate formal analysis and verification techniques. For this reason, SCADE models are used in the design of safety-critical flight control software systems for commercial aircraft made by Airbus.

In Ptolemy II, SR is a form of **coordination language** rather than a programming language, (see also ForSyDe (Sander and Jantsch, 2004), which also uses synchrony in a coordination language). This allows for “primitives” in a system to be complex components rather than built-in language primitives. This, in turn, enables heterogeneous combinations of MoCs, since the complex components may themselves include components developed under another model of computation.

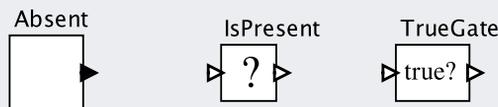
### Sidebar: Domain-Specific SR Actors

The SR actors in `DomainSpecific`→`SynchronousReactive` below are inspired by the corresponding operators of the synchronous languages Lustre and Signal.



- **Current** outputs the most recently received non-absent input. If no input has been received, then the output is absent.
- **Default** merges two signals with a priority. If the preferred input (on the left) is present, then the output is equal to that input. If the preferred input is absent, then the output is equal to the alternate input (on the bottom, whether it is absent or not).
- **NonStrictDelay** provides a one-tick delay. On each firing, it produces on the output port whatever value it read on the input port in the previous tick. If the input was absent on the previous tick of the clock, then the output will be absent. On the first tick, the value may be given by the *initialValue* parameter. If no value is given, the first output will be absent.
- **Pre** outputs the previously received (non-absent) input. When the input is absent, the output is absent. The first time the input is present, the output is given by the *initialValue* parameter of the actor (which by default is absent). It is worth noting that, contrary to `NonStrictDelay`, `Pre` is *strict*, meaning that the input must be known before the output can be determined. Thus, it will not break a [causality loop](#). To break a causality loop, use `NonStrictDelay`.
- **When** filters a signal based on another. If the control input (on the bottom) is present and true, then the data input (on the left) is copied to the output. If control is absent, false, or true with the data input being absent, then the output is absent.

The Ptolemy II library also offers several actors to manipulate absent values:



- **Absent**. Output is always absent.
- **IsPresent** outputs true if its input is present and false otherwise.
- **TrueGate** outputs true if its input is present and true; otherwise, absent.

**Example 5.2:** A simple model of a digital circuit is shown in Figure 5.4. It is a model of a 2-bit, modulo-4 counter that produces the integer sequence 0, 1, 2, 3, 0, 1, .... The feedback loops use `NonStrictDelay` actors, each of which models a latch (a latch is a circuit element that captures a value and holds it for some period of time). It also includes two actors that model logic gates, the `LogicalNot` and `LogicGate` (see box on page 112).

The upper loop, containing the `LogicalNot`, models the low-order bit (**LOB**) of the counter. It starts with value false, the initial output of the `NonStrictDelay`, and the alternates between true and false in each subsequent tick.

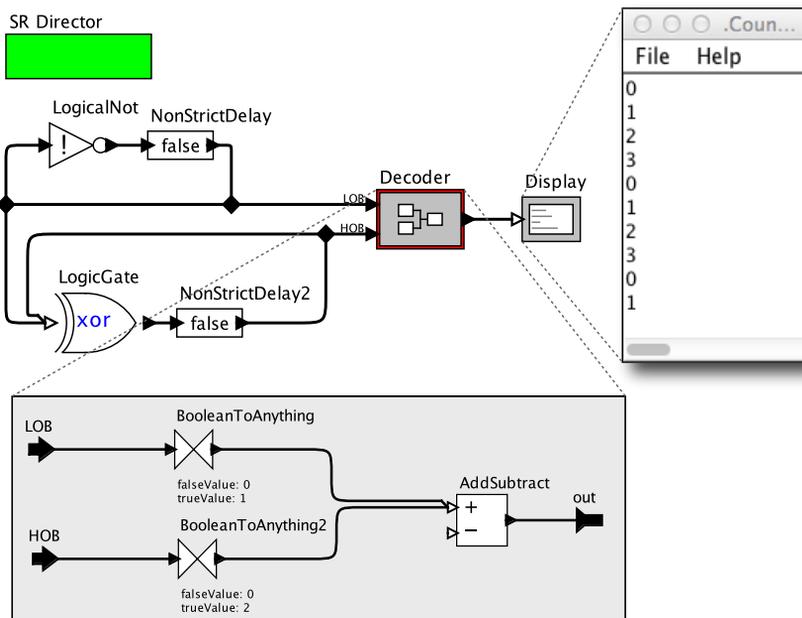


Figure 5.4: A model of a 2-bit counter in SR. The top-level model includes a `Decoder` composite actor that translates the boolean data into integers. [\[online\]](#)

The lower loop, containing the LogicGate, models a carry circuit, implementing the high-order bit (**HOB**) of the counter. It also starts with false, and toggles between true and false in each tick where the LOB is true.

The Decoder is a composite actor provided just to generate a more readable display. It converts the two Boolean values into a numerical value from 0 to 3 by assigning values to the LOB. It contains two **BooleanToAnything** actors that convert the Boolean values to the values of the LOB and HOB, which are then added together.

The **NonStrictDelay** actors in Figure 5.4 are non-strict. They are able to produce outputs without knowing the inputs. On the first tick, the values of the outputs are given by the *initialValue* parameters of the actors. In subsequent ticks, the values of the outputs are given by the input *from the previous tick*, which has been found by identifying the fixed point. Thus, these actors break the potential causality loops.

Notice that it would not work to use **Pre** instead of **NonStrictDelay** (see box on page 5.2.1). The **Pre** actor is strict, because it has to know whether the input is present or not in order to determine whether the output is present or not.\*

The model of Figure 5.4 is rather simple and does not illustrate the full power of SR. In fact, the same model would work with an **SDF** director, provided that **NonStrictDelay** actors are replaced by **SampleDelay** actors.†

Every directed cycle in SR is required to contain at least one non-strict actor. But **NonStrictDelay** is not the only non-strict actor. Another example of a non-strict actor is the **NonStrictLogicGate** actor, which can be parameterized to implement functions such as non-strict logical AND, also called a **parallel AND**. The truth table of the non-strict AND with two inputs is shown below (the actor can in fact accept an arbitrary number of in-

---

\*The Lustre synchronous language (Halbwachs et al., 1991) is able to make **Pre** non-strict by using a **clock calculus**, which analyzes the model to determine in which ticks the inputs will be present. Thus, in Lustre, **Pre** does not execute in ticks where its input is absent. As a consequence, when it does execute, it knows that the input is present, and even though it does not know the value of the input, it is able to produce an output. The SR director in Ptolemy II does not implement a clock calculus, adopting instead the simpler clocking scheme of Esterel (Berry and Gonthier, 1992).

†**SampleDelay** produces initial outputs during the initialize phase of execution. In dataflow domains, those initial outputs are buffered and made available during the execution phase. In SR, however, there is no buffering of data, and any outputs produced during initialize are lost. Hence, **SampleDelay** is not useful in SR.

puts):

inputs	$\perp$	<i>true</i>	<i>false</i>
$\perp$	$\perp$	$\perp$	<i>false</i>
<i>true</i>	$\perp$	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>

Here, the symbol  $\perp$  means **unknown**. Observe that when one input is known to be false, the output is false, even if the other input is unknown.

**Example 5.3:** The model shown in Figure 5.5 results in non-ambiguous semantics despite its feedback loop. The NonStrictLogicGate implements the AND logic function, and outputs a Boolean “false” value at every tick because one of the inputs is always false.

A practical example that also has cycles without NonStrictDelay is next.

**Example 5.4:** Figure 5.6 shows an SR realization of **token-ring media access control (MAC)** protocol given by Edwards and Lee (2003b). The top-level model has three instances of an **Arbiter** class connected in a cycle. It also has a ComposeDisplay composite actor used to construct a human-readable display of the results of execution, shown at the bottom.

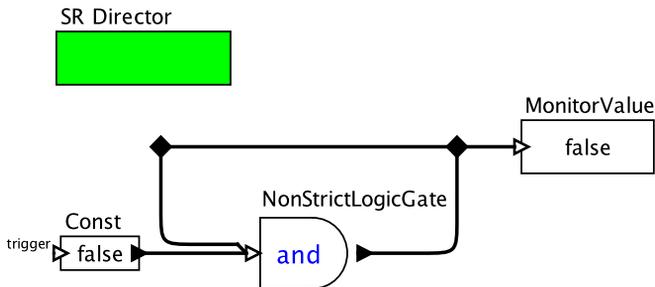


Figure 5.5: A non-ambiguous model which uses a non-strict logical AND. [online]

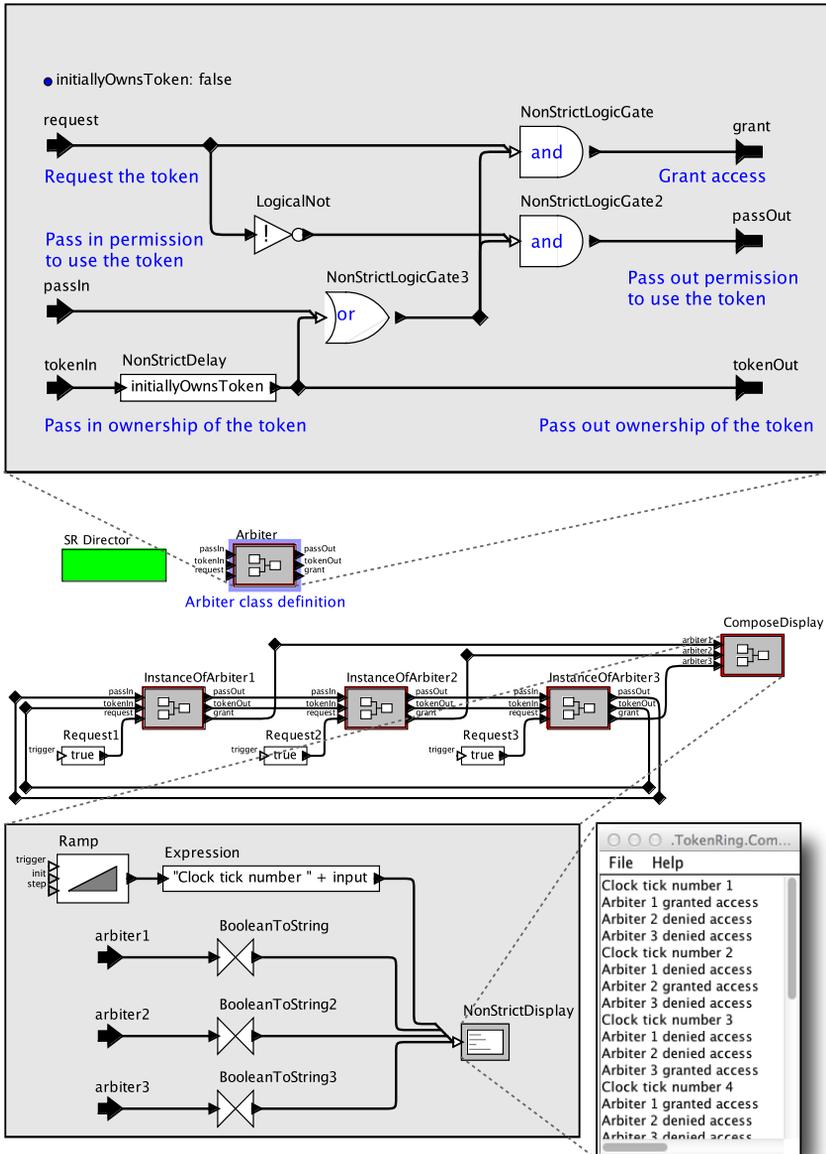


Figure 5.6: A token-ring media access protocol implemented using SR. From Edwards and Lee (2003b). [online]

The goal of this system is to arbitrate fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each tick of the logical clock, the arbiter grants access to requestor holding the token, if it requests access. If it does not request access, then the model grants access to the first requestor downstream of the block with the token that requests access. In the figure, all three requestors are always requesting access, and in the display at the bottom, you can see that access is granted fairly in a round-robin fashion. In this model, `InstanceOfArbiter1` starts with the token (see the parameter of the instance).

The three arbiters are instances of the [actor-oriented class](#) shown at the top of the figure. This class has three inputs and three outputs. It has an instance of [NonStrictDelay](#) that outputs true for the arbiter that currently holds the token. Exactly one of the three is initialized with value true. At each tick of the clock, the arbiter passes the token down to the next arbiter. This forms a cycle that include three instances of `NonStrictDelay`.

However, there are another cycles that have no instances of `NonStrictDelay`, for example the cycle passing through each *request* input and *grant* output. This cycle has three instances of [NonStrictLogicGate](#), configured to implement the parallel AND. This logic gate will grant access to the requestor if it has a request and it either holds the token or its *passIn* input is true (meaning that the upstream arbiter has the token but does not have a request). Although it is far from trivial to see at glance, every cycle of logical gates can be resolved without full knowledge of the inputs, so the model does not suffer from a causality loop.

Another example of a non-strict actor is the [Multiplexor](#) or [BooleanMultiplexor](#) (see box on page 119). These require their control input (at the bottom of the icon) to be known; the value of this input then determines which of the data inputs are to be forwarded to the output. Only that one data input needs to be known for the actor to able to produce an output.

**Example 5.5:** An interesting example, shown in Figure 5.7, calculates either  $\sin(\exp(x))$  or  $\exp(\sin(x))$ , depending on a coin toss from the [Bernoulli](#) actor. [Malik \(1994\)](#) called examples like this **cyclic combinational circuits**, because, although there is feedback, there is actually no state stored in the system. The output (each value plotted) depends only on the current inputs (the data from the Ramp

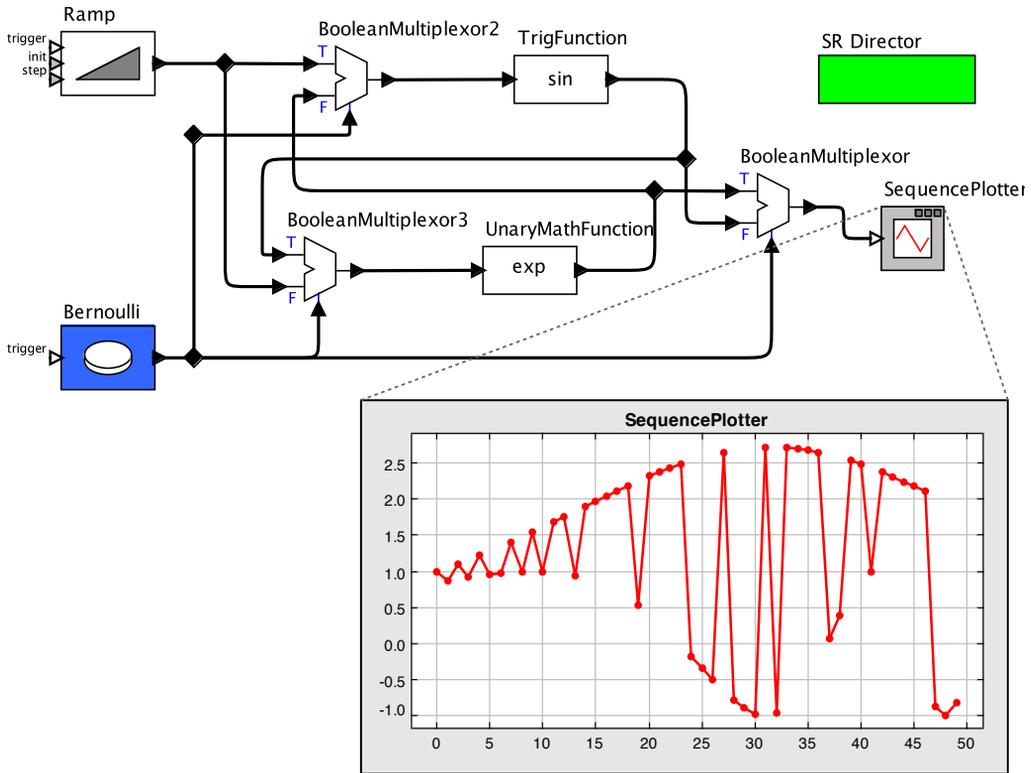


Figure 5.7: A model of a cyclic combinational circuit in SR. From Malik (1994). [online]

and Bernoulli actors). A circuit whose output depends only on the current inputs and not on the past history of inputs is called a **combinational** circuit. Most circuits with feedback are not combinational. The output depends not only on the current inputs, but also on the current state, and the current state changes over time.

In this case, feedback is being used to avoid having to have two copies of the actors that do the actual computation, the **TrigFunction** and **UnaryMathFunction** (see box on page 58). An equivalent model that uses two copies of these actors is shown in Figure 5.8. If these models are literally implemented in circuits, with a separate cir-

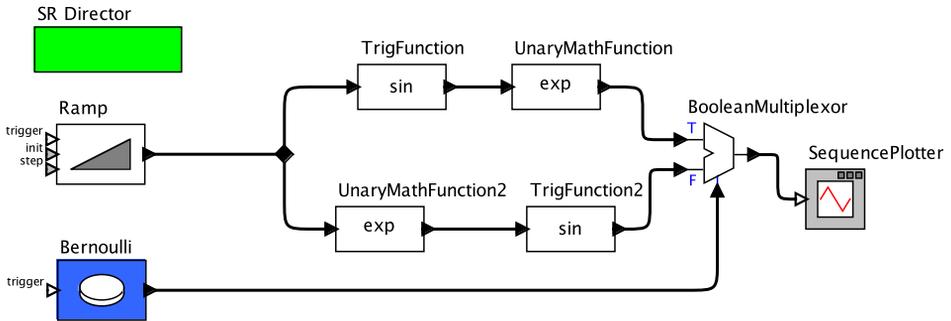


Figure 5.8: Acyclic version of the model of Figure 5.7 that uses two copies of each of the math actors. [\[online\]](#)

cuit for each actor, then the model in Figure 5.7 may be considerably less expensive than the one in Figure 5.8.

The model has three BooleanMultiplexor actors. These actors send either their “T” or “F” input value to the output port depending on whether the control input (at the bottom of the actor) is set to true or false. At each tick, one of the two BooleanMultiplexor actors on the left will be able to provide an output (once it is provided with an input from the Ramp). That one BooleanMultiplexor, therefore, breaks the causality loop and enables finding a fixed point.

### 5.2.3 Causality Loops

Not all SR models are executable. In particular, it is possible to construct feedback models that exhibit a [causality loop](#), as illustrated by the following examples.

**Example 5.6:** Two examples of loops with unresolvable cyclic dependencies are shown in Figure 5.9. Both the [Scale](#) and the [LogicalNot](#) actors are strict, and hence their inputs must be known for the outputs to be determined. But the outputs are equal to the inputs in these models, so the inputs cannot be known. The SR director will reject these models, reporting an exception

**IllegalActionException:** Unknown inputs remain. Possible causality loop:  
*in Display.input*

## 5.2.4 Multiclock Models

The logical clock in the SR domain is a single, global clock. Every actor under the control of an SR director will be fired on every tick of this clock. But what we want some actors to be fired more or less frequently? Fortunately, the [hierarchy](#) mechanism in Ptolemy II makes it relatively easy to construct models with multiple clocks proceeding at different rates. The **EnabledComposite** actor is particularly useful for building such **multiclock** models.

**Example 5.7:** Consider the **guarded count** model of Figure 5.10, which counts down to zero from some initial value and then restarts the count from some new value. At the top level, the model has two composite actors and two Display actors. The Countdown composite actor uses SR primitive actors to implement the following count-down behavior: whenever it receives a non-absent value  $n$  (an integer) at its *start* input port, it (re)starts a count-down from  $n$ ; that is, it outputs the sequence of values  $n, n-1, \dots, 0$  at its *count* output port. When the count reaches 0, the *ready* port outputs a value true, signaling that the actor is ready for a new count down.

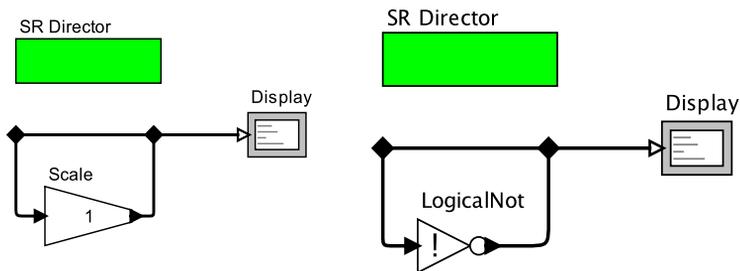


Figure 5.9: Two SR models with invalid loops.

The ready signal controls the firing of the EnabledComposite actor. Within this composite, a reaction only occurs when a true value is provided on the *enable* input port (the port at the bottom of the actor). Note that the ready signal is initially true, due to the NonStrictDelay actor used inside Countdown.

The clock of the SR director inside EnabledComposite progresses at a slower rate than the clock of the top-level SR director. In fact, the relationship between these rates is determined dynamically by the data provided by the Sequence actor.

## 5.3 Finding the Fixed-Point

For acyclic models (such as the one shown Figure 5.8) or cyclic models where every cycle is “broken” by a NonStrictDelay actor (such as the model shown in Figure 5.4), executing the model efficiently is easy. The actors of the model can be ordered according to their dependencies (e.g., using a topological sorting algorithm) and then fired according to that order. In this case, each actor only needs to be fired once at each tick of the logical clock.

However, this strategy will not work with models like those in Figures 5.6 or 5.7, because the order in which the actors have to be fired depends on data computed by some of the actors. Fortunately, there is a simple execution strategy that works. The key is to start each tick of the logical clock by assigning a special value called unknown, denoted  $\perp$ , to all signals. The director can then simply evaluate actors in arbitrary order until no more progress is made. For strict actors, if there are any unknown inputs, then the outputs will remain unknown. For non-strict actors, even when some inputs are unknown, some outputs may become known. This procedure is said to have converged when no firing of any actor changes the state of any signal. If the actors all follow the [strict actor semantics](#) (see box on page 433), then it can be proven that this procedure converges in a finite number of steps (see, for example, [Edwards and Lee \(2003b\)](#)).

Upon convergence, either all signals will be known, or some signals will remain unknown. If every iteration results in all signals being known for all possible inputs, then the model is said to be **constructive** (that is, a solution can be “constructed” in a finite number of steps). Otherwise, the model is declared to be **non-constructive**, and it is rejected.

Note that in the Ptolemy II SR domain, the causality analysis is performed dynamically, at run-time. This is in contrast to languages such as Esterel, where the compiler attempts

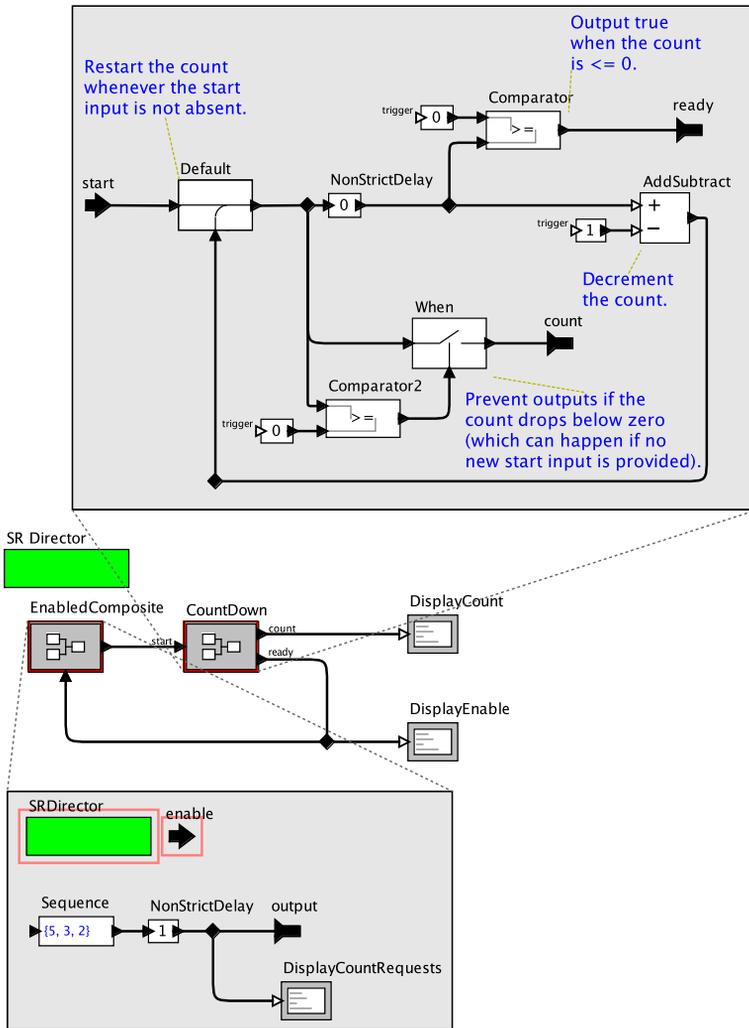


Figure 5.10: Multiclock model in SR. [online]

to prove statically (i.e., at compile-time) that the program is constructive (see the sidebar on page 179).

SR can only work correctly with actors that follow the [strict actor semantics](#). To understand this, we can model an actor as a state machine. Let  $\vec{x}$ ,  $\vec{y}$  and  $\vec{s}$  denote the vectors of inputs, outputs, and states, respectively. Then the behavior of the state machine can be described as

$$\vec{y}(n) = f(\vec{x}(n), \vec{s}(n)) \quad (5.1)$$

$$\vec{s}(n+1) = g(\vec{x}(n), \vec{s}(n)), \quad (5.2)$$

where  $n$  indexes the ticks of the logical clock,  $f$  models the [fire](#) method that computes outputs from current inputs and current state, and  $g$  models the [postfire](#) method that computes the next state from current inputs and current state. The key here is that the fire method does not change the state of the actor. Hence, the fire method can be invoked repeatedly, and each time, given the same inputs, it will produce the same outputs.

An additional condition on actors is that they be [monotonic](#) (see box on page 182). Although the mathematical underpinnings of this constraint are quite sophisticated, the practical manifestation of the constraint is simple. An actor is monotonic if it does not change its mind about outputs given more information about inputs. Specifically, if the fire method is invoked with some inputs unknown, then if the actor is non-strict, it may be able to produce outputs. Suppose that it does. Then the actor is monotonic if given more information about the inputs (fewer inputs are unknown) does not cause it to produce a *different* output than the one it produced with less information.

Most Ptolemy II actors conform to the strict actor semantics and are monotonic and therefore can be used in SR.

## 5.4 The Logic of Fixed Points

Recall the two models of Figure 5.9, both of which exhibit causality loops. These models, however, are different from one another in an interesting way. They exhibit the difference between a deterministic and a [constructive](#) semantics of synchronous models. The constructive semantics is based on ideas from **intuitionistic logic**, and although it is also deterministic, it rejects some models that would be accepted by a broader deterministic semantics based on classic logic.

### Sidebar: Causality in Synchronous Languages

The problem of how to resolve cyclic dependencies, the **causality problem**, is one of the major challenges in synchronous languages. We briefly summarize several solutions here, and refer the reader to research literature and survey articles such as [Caspi et al. \(2007\)](#) for more details.

The most straightforward solution to the causality problem is to forbid cyclic dependencies altogether. This is the solution adopted by the Lustre language, which requires that every dataflow loop must contain at least one **pre** operator. The same effect could be achieved by the Ptolemy II SR director by requiring that every loop contain at least one **NonStrictDelay** actor. This actor breaks the instantaneous cyclic dependency. The Lustre compiler statically checks this condition and rejects those programs that violate it. The same policy is followed in SCADE.

Another approach is to accept a broader set of **constructive** programs, as is the case with the Ptolemy II SR domain. This approach was pioneered by [Berry \(1999\)](#) for Esterel. A key difference between Esterel and SR is that in SR a fixed-point is computed at run-time (at each tick of the logical clock), while the Esterel compiler attempts to prove that a program is constructive at compile-time. The latter is generally more difficult since the inputs to the program are generally unknown at compile-time. On the other hand, statically proving that a program is constructive has two key benefits. First, it is essential for safety-critical systems, where run-time exceptions are to be avoided. Second, it allows generation of implementations that minimize the run-time overhead of fixed-point iteration.

Yet another approach is to accept only deterministic programs, or conversely, reject programs that, when interpreted as a set of constraints, do not yield unique solutions. This approach is followed in Signal ([Benveniste and Le Guernic, 1990](#)) and Argos ([Maraninchi and Rémond, 2001](#)). One drawback with this approach is that it sometimes accepts dubious programs. For instance, consider a program representing the system of equations

$$Y = X \wedge \neg Y.$$

Although this system admits a unique solution in classic two-valued logic, namely,  $X = Y = \text{false}$ , it is unclear whether the corresponding implementation is meaningful. In fact, a straightforward **combinational** circuit implementation is unstable; it oscillates.

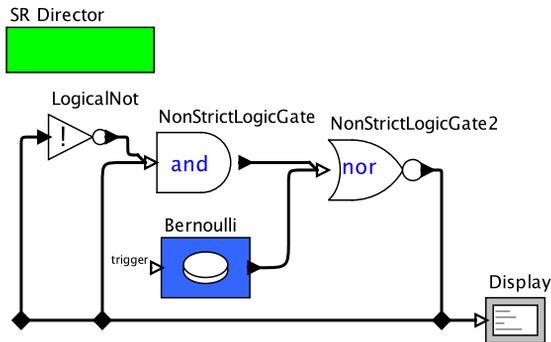


Figure 5.11: Non-constructive example with a unique fixed point. [online]

In particular, we could have interpreted the left model of Figure 5.9 as defining an equation between the input and output of the Scale actor, say  $x$ , as follows:

$$x = 1 \cdot x$$

In the classic logic interpretation, the above equation has multiple solutions, e.g.,  $x = 0$ ,  $x = 1$ , and so on. A non-deterministic semantics based on classic logic would accept any of these solutions as a valid behavior of the system. A deterministic semantics would declare the model ambiguous, and thus invalid. In the SR semantics, the above equation has a unique least fixed-point solution, namely,  $x = \perp$ , unknown. Hence, SR also rejects this model.

The right model of Figure 5.9 can be seen as defining the equation

$$x = \neg x$$

where  $\neg$  denotes logical negation. In this case, in the classic logical interpretation, there is no solution at all, quite a different situation. A deterministic semantics may again reject this model. In the case of SR, the solution is again  $x = \perp$ , unknown, resulting in rejection of the model.

A third situation, due to Malik (1994) and shown in Figure 5.11, however, could be accepted by a deterministic semantics, but is rejected by a constructive semantics. Logically, the output of the AND gate should always be false, and hence the output sent to the Display actor should be equal to the negation of the input value produced by the Bernoulli

actor. Hence, there is a single unique behavior for all possible inputs. The model, however, is rejected by the Ptolemy II SR director as **non-constructive** whenever the Bernoulli actor produces a false. In that case, all signals in the loop remain unknown. In the constructive SR semantics, this solution with unknowns is the *least* fixed point, and hence is the behavior selected, even though there is a unique fixed point with no unknowns.

Even though the circuit in Figure 5.11 seems to have a logically consistent behavior for every input, there are good reasons for rejecting it. If this were actually implemented as a circuit, then time delays in the logic gates would cause the circuit to oscillate. It would not, in fact, realize the logic specified by the model. To realize such circuits in software, the only known technique for finding the unique fixed point and verifying that it is unique, in general, is to exhaustively search over all possible signal assignments. In a small model like this, such an exhaustive search is possible, but it becomes intractable for larger models, and it becomes impossible if the data types have an infinite number of possible values. Thus, the fact that the model is non-constructive reveals very real practical problems with the model.

We now give a brief introduction to the theoretical foundation of the SR semantics. This is a rather deep topic, and our coverage here is meant only to whet the appetite of the reader to learn more. The SR semantics is based on the theory of continuous functions over complete partially ordered sets (CPOs) (see box on page 182). In the case of SR, the key CPO is a so-called flat CPO shown in Figure 5.12. This CPO consists of the minimal element  $\perp$  and all “legal” values of Ptolemy models, such as booleans, integer and real numbers, but also tuples, records, lists, and so on (see Chapter 14). Any legal value is considered to be greater than  $\perp$  in the CPO order, but the legal values are incomparable among themselves, leading to the term “flat”.

Now, consider an SR model. The output of every actor in the model can be seen as a variable taking values in the above flat CPO. The vector of all output variables can be seen as taking values in the product CPO obtained by forming the cartesian product of all individual CPOs, with element-wise ordering. For simplicity, let us suppose that

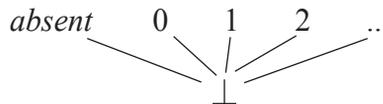


Figure 5.12: The flat CPO ensuring existence of a unique least fixed-point in SR.

### Sidebar: CPOs, Continuous Functions and Fixed Points

The SR semantics is based on order theory, which we summarize here; see [Davey and Priestly \(2002\)](#) for a more thorough explanation

Consider a set  $S$ . A **binary relation** on  $S$  is a subset  $\sim \subseteq S \times S$ . We often write  $x \sim y$  instead of  $(x, y) \in \sim$ . A **partial order** on  $S$  is a binary relation  $\sqsubseteq$  which is **reflexive** (i.e.,  $\forall x \in S : x \sqsubseteq x$ ), **antisymmetric** (i.e.,  $\forall x, y \in S : x \sqsubseteq y$  and  $y \sqsubseteq x$  implies  $x = y$ ), and **transitive** (i.e.,  $\forall x, y, z \in S : x \sqsubseteq y$  and  $y \sqsubseteq z$  implies  $x \sqsubseteq z$ ). A **partially ordered set** or **poset** is a set equipped with a partial order.

Let  $X \subseteq S$ . An **upper bound** of  $X$  is an element  $u \in S$  such that  $\forall x \in X : x \sqsubseteq u$ . A **least upper bound** of  $X$ , denoted  $\sqcup X$ , is an element  $\ell \in S$  such that  $\ell \sqsubseteq u$  for all upper bounds  $u$  of  $X$ . A **chain** of  $S$  is a subset  $C \subseteq S$  which is **totally ordered**:  $\forall x, y \in C : x \sqsubseteq y$  or  $y \sqsubseteq x$ . A **complete partial order** or **CPO** is a poset  $S$  such that every chain of  $S$  has a least upper bound in  $S$ . This condition also guarantees that every CPO  $S$  has a **bottom element**  $\perp$ , such that  $\forall x \in S : \perp \sqsubseteq x$ . (Indeed, the empty chain must have a least upper bound in  $S$ , and the set of upper bounds of the empty subset of  $S$  is the entire  $S$ .)

To illustrate the above concepts, consider the set of natural numbers  $\mathbb{N} = \{0, 1, 2, \dots\}$ .  $\mathbb{N}$  is a poset with the usual (total, and therefore also partial) order  $\leq$ . Because  $\leq$  is a total order,  $\mathbb{N}$  is a chain. The least upper bound of  $\mathbb{N}$  can be defined to be a new number  $\omega$  such that  $n < \omega$  for all  $n \in \mathbb{N}$ .  $\omega$  is not a natural number, therefore,  $\mathbb{N}$  is not a CPO. On the other hand, the set  $\mathbb{N}^\omega = \mathbb{N} \cup \{\omega\}$  is a CPO. The bottom element of  $\mathbb{N}^\omega$  is 0.

Every poset whose chains are all finite is a CPO. This is because the greatest element in a chain is also the least upper bound of the chain. This is why the “flat” poset of Figure 5.12 is a CPO.

Consider two CPOs  $X$  and  $Y$ . A function  $f : X \rightarrow Y$  is **Scott-continuous** or simply **continuous** if for all chains  $C \subseteq X$ ,  $f(\sqcup C) = \sqcup\{f(c) \mid c \in C\}$ . It can be shown that every continuous function is also **monotonic**, i.e. it satisfies:  $\forall x, y \in X : x \sqsubseteq y$  implies  $f(x) \sqsubseteq f(y)$ . However, not all monotonic functions are continuous. For example, consider the function  $f : \mathbb{N}^\omega \rightarrow \mathbb{N}^\omega$  such that  $f(n) = 0$  for all  $n \in \mathbb{N}$  and  $f(\omega) = \omega$ . Then  $f(\sqcup \mathbb{N}) = f(\omega) = \omega$ , whereas  $\sqcup\{f(n) \mid n \in \mathbb{N}\} = \sqcup\{0\} = 0$ . The following **fixed-point theorems** are well-known results of order theory: (A) Every monotonic function  $f : X \rightarrow X$  on a CPO  $X$  has a least fixed-point  $x^*$ . (B) If  $f$  is also continuous then  $x^* = \bigsqcup_{i \geq 0} f^i(\perp)$ , where  $f^0(\perp) = \perp$  and  $f^{i+1}(\perp) = f(f^i(\perp))$ . (B) is used to obtain an effective procedure for computing the semantics of an SR model.

the model is *closed*, in the sense that every input port of every actor in the model is connected to some output port (the theory also works for open models, but is slightly more complicated; we refer the reader to [Edwards and Lee \(2003b\)](#) for a more detailed explanation). The SR model then defines a function  $F$  which has both as domain and co-domain this product CPO: this is because the model is closed, so every input is also an output. Thus,  $F$  takes as input a vector  $\vec{x}$  and returns as output another vector  $\vec{y}$ . The latter is obtained by firing all actors in the model once. Given this interpretation, a closed SR model defines the equation

$$\vec{x} = F(\vec{x})$$

This equation has a unique least solution  $\vec{x}^*$ , provided that  $F$  is [monotonic](#); that is, provided that  $\vec{x} \leq \vec{y}$  implies  $F(\vec{x}) \leq F(\vec{y})$ . (The precise condition is for the function to be continuous, but in the case of flat CPOs, monotonicity is equivalent to continuity.) The solution  $\vec{x}^*$  is called a fixed-point because it satisfies  $\vec{x}^* = F(\vec{x}^*)$ . It is ‘least’ in the sense that it is smaller in the CPO ordering than every other solution of the above equation. That is, for any  $\vec{y}$  such that  $\vec{y} = F(\vec{y})$ , it must be  $\vec{x}^* \leq \vec{y}$ .

Moreover, the least fixed-point can be computed effectively in a finite number of iterations, in fact, at most  $N$  iterations, where  $N$  is the total number of outputs in the model. Indeed, starting with all outputs set to  $\perp$ , every iteration that fires all actors without reaching the fixed-point is guaranteed to update at least one output. The first time an output is updated, it changes from  $\perp$  to some legal value  $v$ . Because  $F$  is monotonic, the same output can no longer change from  $v$  to  $\perp$  or any other  $v'$ , since  $v > \perp$  and  $v$  is incomparable with any  $v' \neq v$ . Therefore, each output can be updated at most once. As a result, the fixed-point must be reached after at most  $n$  iterations.

The monotonicity of  $F$  is ensured by ensuring that every individual actor is monotonic; that is, that its fire method is monotonic. Monotonicity of  $F$  then follows from the fact that composition of monotonic functions results in a monotonic function. Monotonicity of atomic actors is ensured in Ptolemy by construction. The key is to ensure that if an actor outputs a known value, say  $v$ , in the presence of unknown inputs, then if those inputs become known, the actor will not “change its mind” and output a different value  $v'$ . A straightforward way to ensure this property is by making an actor [strict](#), in the sense that it requires all inputs to be known, otherwise, it produces unknown outputs. Most actors in

Ptolemy are strict, but a few key ones that we have discussed are non-strict. Every cycle in an SR model requires some non-strict actors.

## 5.5 Summary

This chapter has introduced the SR domain in Ptolemy II. In SR, execution is governed by a logical clock, and at each tick of the clock, actors execute, conceptually, simultaneously and instantaneously. We have explained how this results in a fixed-point semantics, and have given examples of both cyclic and acyclic models. We have shown that SR admits multiple clock domains, where clocks progress at different rates. Finally, we have given a brief introduction to the (rather deep) mathematical foundations behind the semantics of SR models.

## Exercises

1. This exercise studies the use of absent events in SR.
  - (a) As a warmup, use [Sequence](#) and [When](#) to construct an SR model that generates a sequence of values *true* interspersed with *absent*. For example, produce the sequence

*(true, absent, absent, true, absent, true, true, true, absent)* .

Make sure your model adequately displays the output. In particular, *absent* should be visible in the display.<sup>‡</sup>

- (b) Use [Default](#) and [When](#) to create a composite actor **IsAbsent** that given any input sequence, produces an output *true* at every tick when the input is *absent*, and otherwise produces the output *absent*.<sup>§</sup>
  - (c) Create a composite actor that can recognize the difference between single and double mouse clicks. Your 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*, your actor should produce output `true` on *singleClick*, where *N* is a parameter of your actor. If instead a second `true` input occurs within *N* ticks of the first, then your actor should output a `true` on *doubleClick*.

How does your model behave if given three values *true* within *N* ticks on input port *click*?

- (d) **Extra credit:** Redo (a)-(c) by writing a custom a Java actor for each of the three functions above. How does this design compare with the design implemented using primitive SR actors? Is it more or less understandable? Complex?
2. The token-ring model of Figure 5.6 is [constructive](#) under the assumption that exactly one of the instances of the [Arbiter](#) initially owns the token (it has its *initiallyOwnsToken* parameter set to `true`). If no instance of [Arbiter](#) initially owns the token, then is the model still constructive? If so, explain why. If not, given a set of values of the [Request](#) actors that exhibits a causality loop.

---

<sup>‡</sup>Note that you could use [TrueGate](#) to implement this more simply, but part of the goal of this exercise is to fully understand [When](#).

<sup>§</sup>Again, a simpler implementation is available using [IsPresent](#), but the goal of this exercise is to fully understand [Default](#).

# 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.

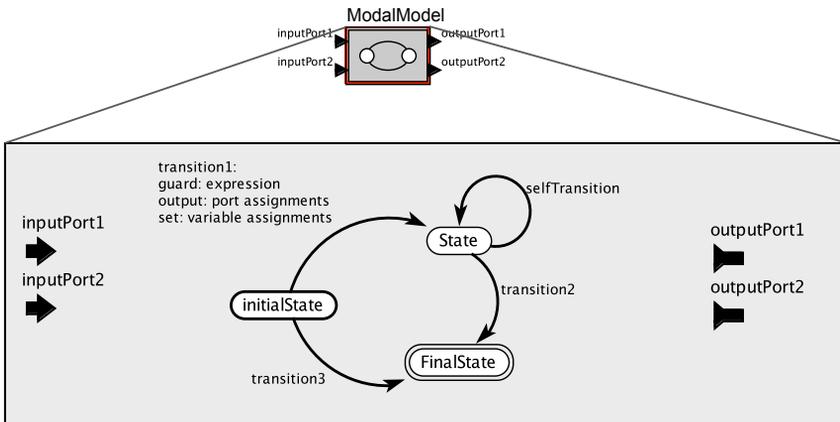


Figure 6.1: Visual notation for state machines in Ptolemy II.

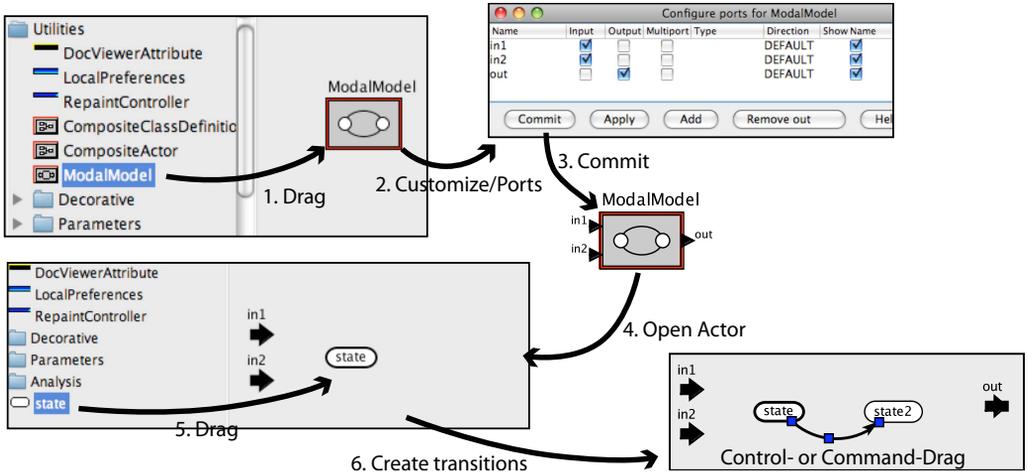


Figure 6.2: Creating FSMs in Vergil, using the ModalModel actor (a similar procedure applies to using the FSMActor).

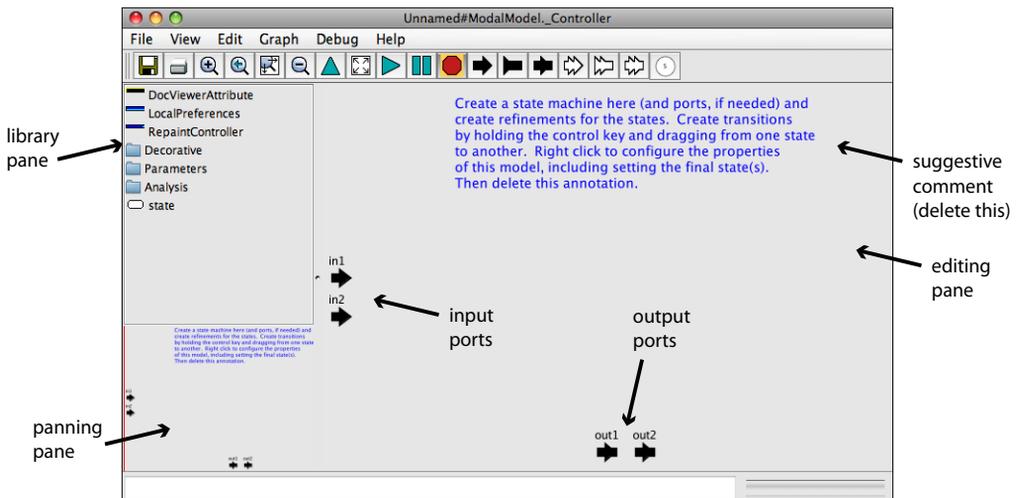


Figure 6.3: Editor for FSMs in Vergil, showing two input and two output ports, before being populated with an FSM.

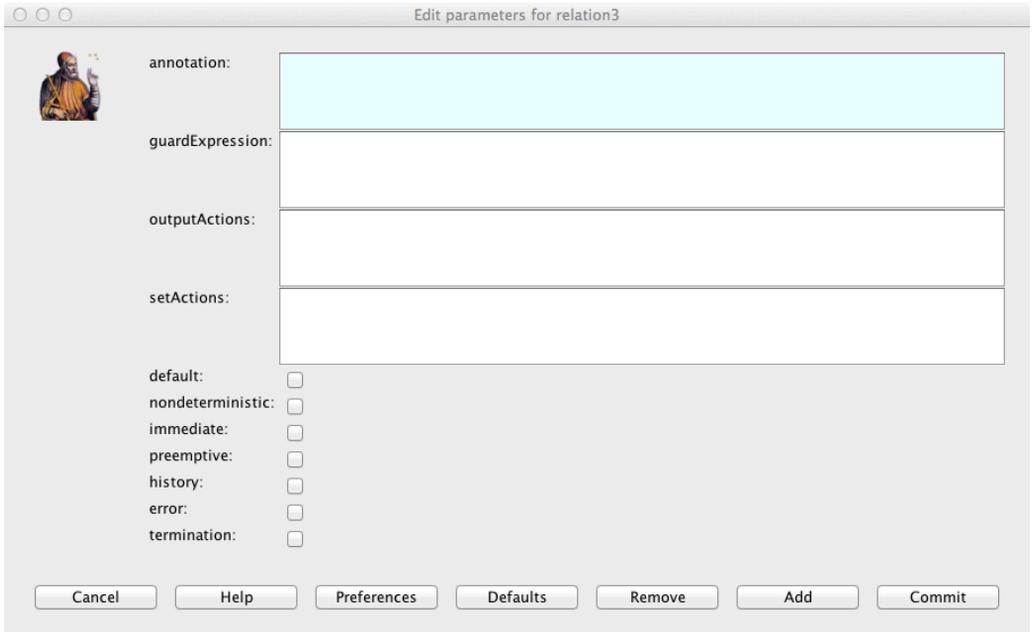


Figure 6.4: Dialog box for configuring a transition in an FSM.

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  $\Sigma = \{heating, cooling\}$ . If the state  $s = heating$ , then the heater is on. If  $s = cooling$ , 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,  $\Sigma = \{heating, cooling\}$ . There are four transitions, each of which has a guard that specifies the conditions under which the transition

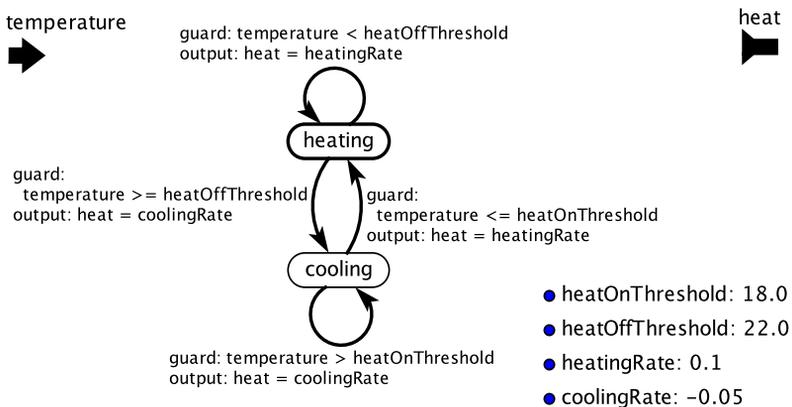


Figure 6.5: FSM model of a thermostat.

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.

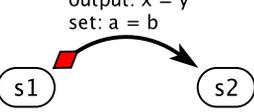
notation	description
<p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p> 	<p>An <b>ordinary transition</b>. Upon firing, if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM will choose the transition and produce the value <math>y</math> 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.

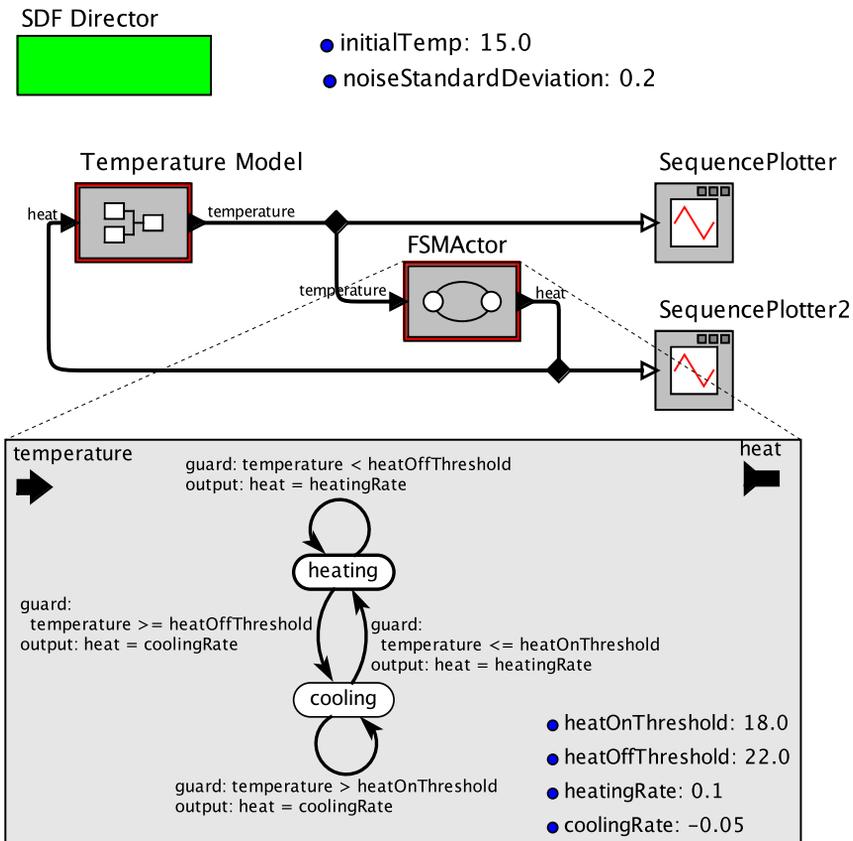


Figure 6.6: The FSM model of a thermostat of Figure 6.5 embedded in an SDF model. The Temperature Model actor is shown in Figure 6.7. [online]

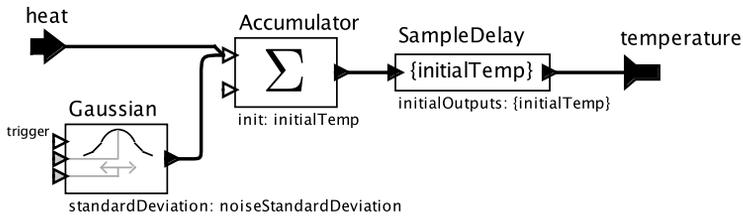


Figure 6.7: The Temperature Model composite actor of Figure 6.6.

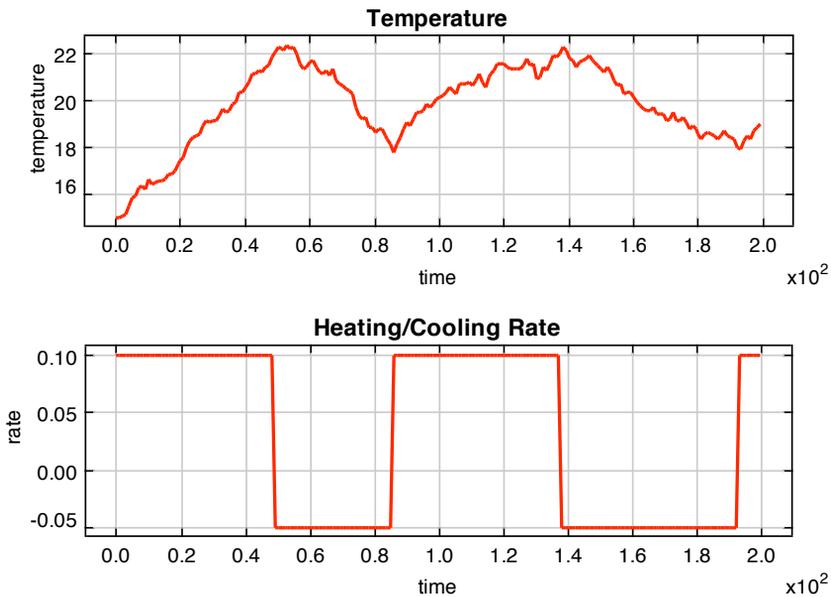


Figure 6.8: Two plots generated by Figure 6.6, showing the temperature (above) and the heating rate (below), which reflects whether the heater is on or off. Both are shown as a function of time.

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, 5,  $\dots$ .

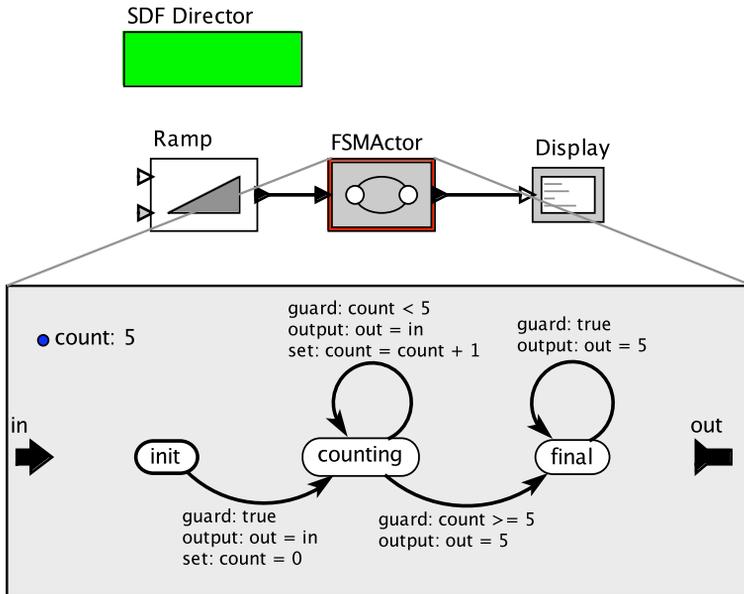


Figure 6.9: An extended state machine, where the *count* variable is part of the state of the system. [\[online\]](#)

## 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.

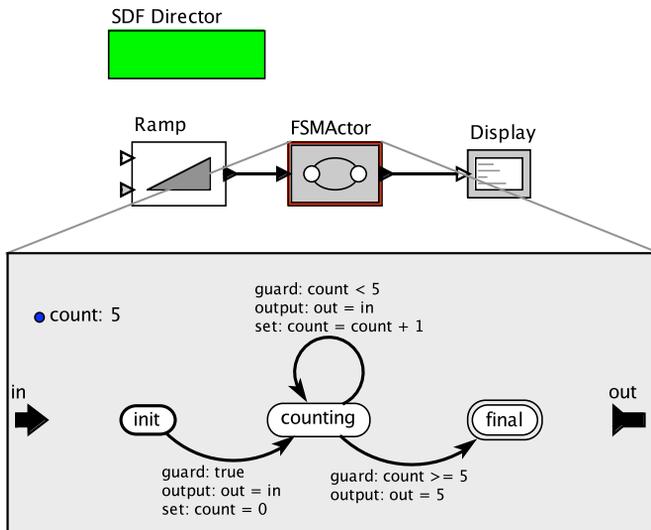


Figure 6.10: A state machine with a final state, which indicates the end of execution of the state machine. [[online](#)]

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

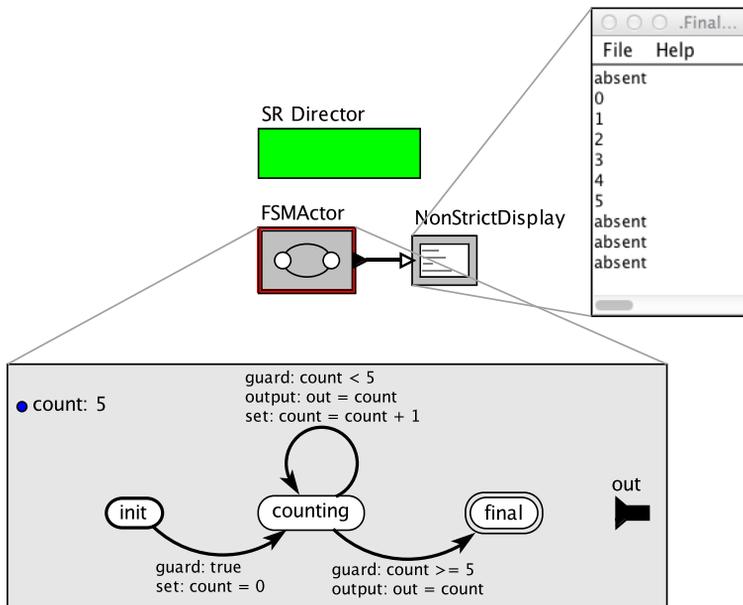


Figure 6.11: A state machine with a final state in an SR model. [[online](#)]

*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).

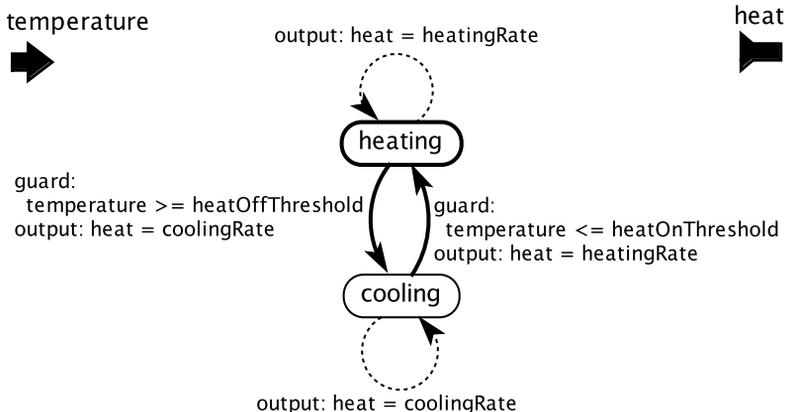


Figure 6.12: An FSM equivalent to that shown in Figure 6.5, but using default self-transitions (indicated with dashed lines). These are taken if the other outgoing transition is not enabled. [\[online\]](#)

### 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 ...*

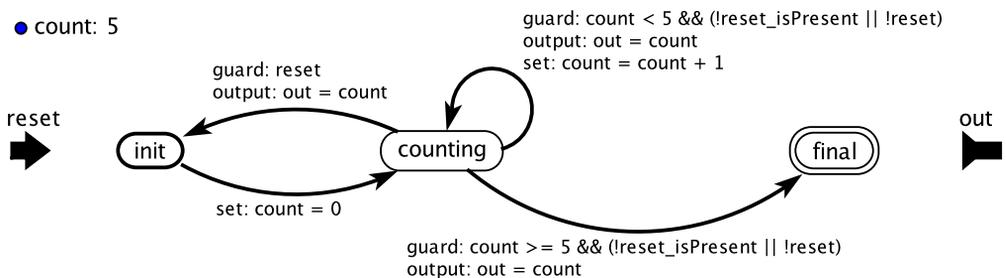


Figure 6.13: A state machine like that in Figure 6.11, but with an additional *reset* input port. [[online](#)]

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 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

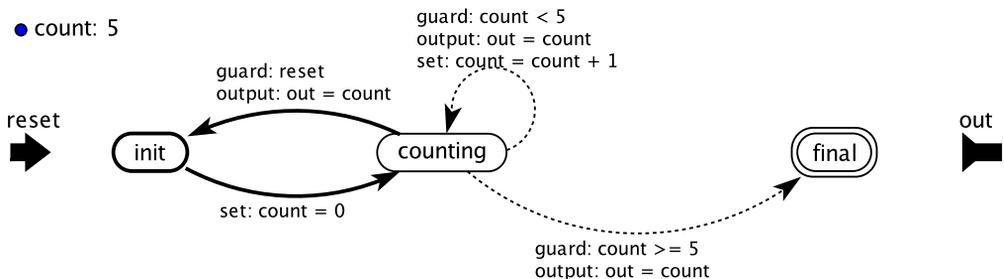


Figure 6.14: A state machine like that in Figure 6.13, but using default transitions to simplify the guard expressions. [\[online\]](#)

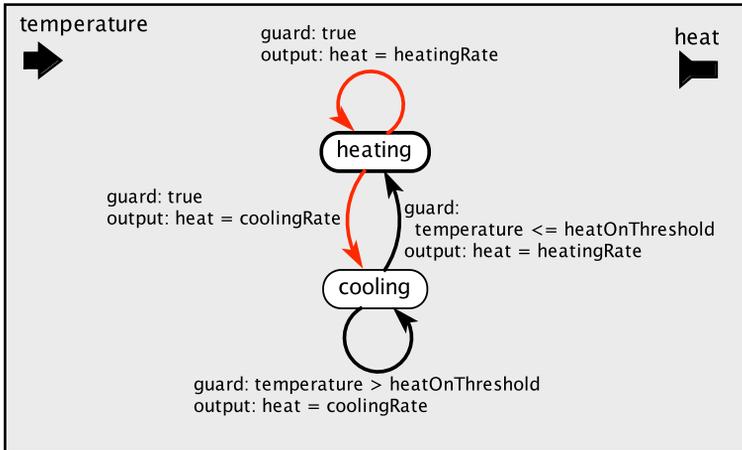


Figure 6.15: A model of a faulty thermostat that nondeterministically switches from heating to cooling. [\[online\]](#)

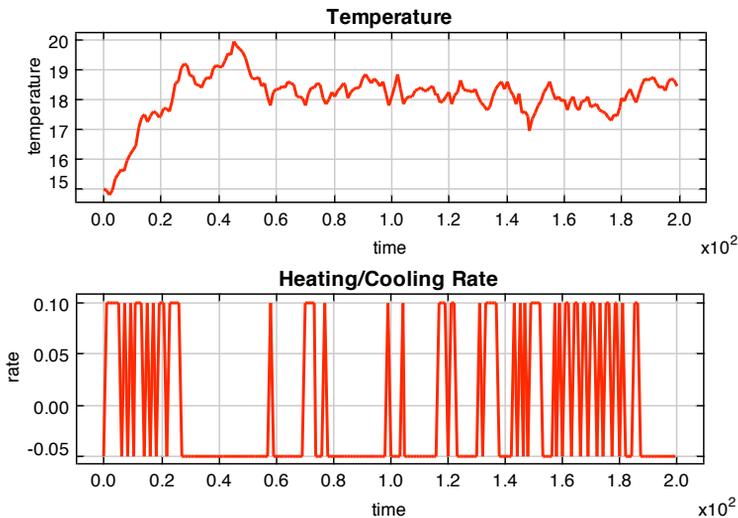


Figure 6.16: Plot of the thermostat FSM of Figure 6.15, a variant of Figure 6.5.

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.

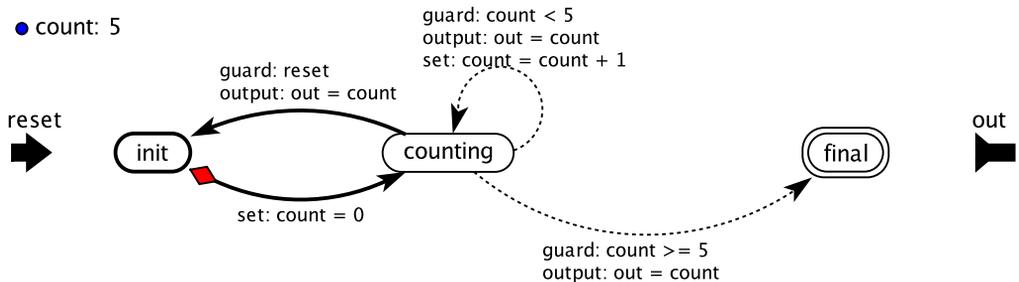


Figure 6.17: A state machine like that in Figure 6.14, but using an immediate transition to prevent absent outputs before counting. The immediate transition is indicated by a red diamond. [online]

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.

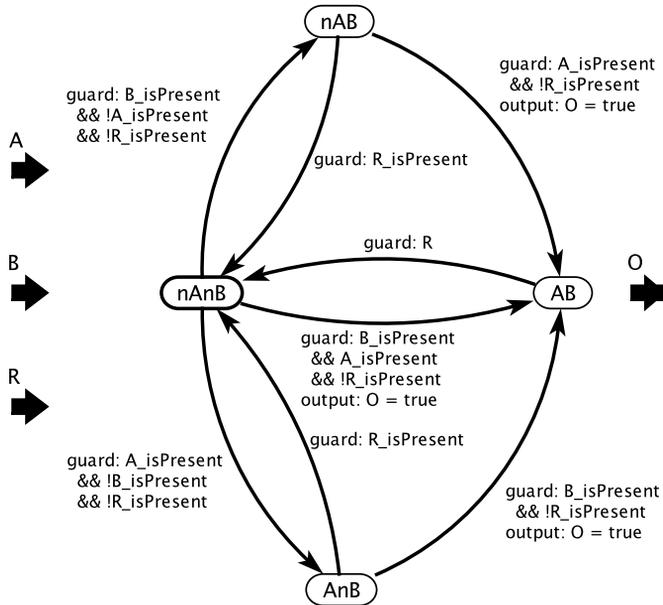


Figure 6.18: A brute-force implementation of the classic ABRO state machine. [\[online\]](#)

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-

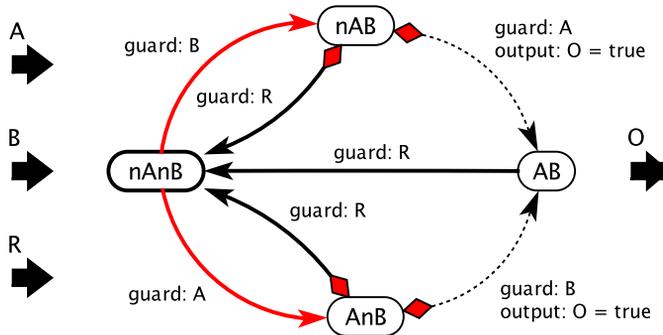


Figure 6.19: An implementation of the ABRO state machine that leverages default transitions, immediate transitions, and nondeterministic transitions to simplify the guard expressions. [\[online\]](#)

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.

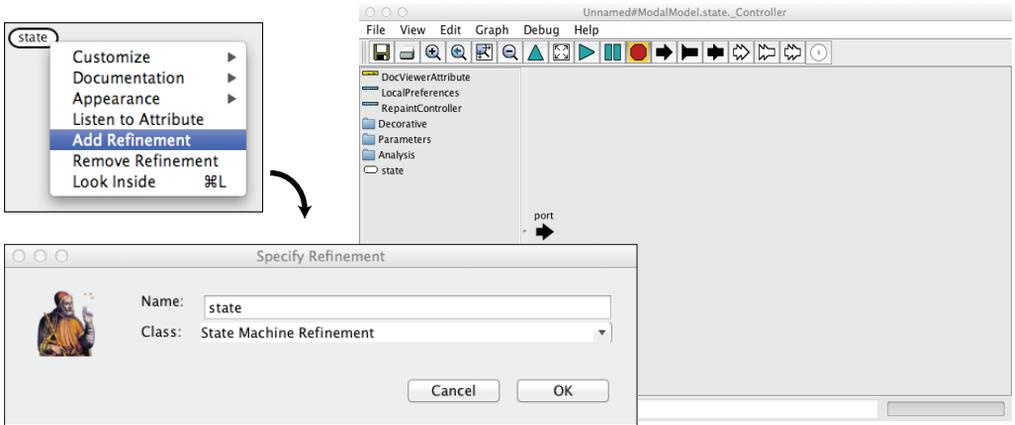


Figure 6.20: How to add a refinement to the state of a ModalModel.

### 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.

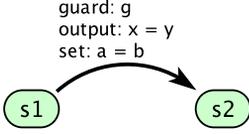
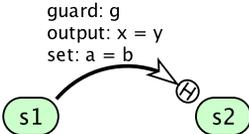
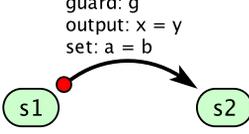
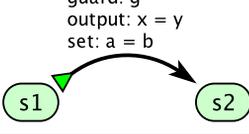
notation	description
 <p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p>	<p>An <b>ordinary transition</b>. Upon firing, the refinement of the source state is fired first, and then if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM 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>s_2</math> are reset to their initial states. For this reason, these transitions are sometimes called <b>reset transitions</b>.</p>
 <p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p>	<p>A <b>history transition</b>. This is similar to an ordinary transition, except that when entering state <math>s_2</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>s_2</math>, of course, the refinements will start from their initial states.</p>
 <p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p>	<p>A <b>preemptive transition</b>. If the current state is <math>s_1</math> and the guard is true, then the state refinement (the FSM sub-model) for <math>s_1</math> will not be invoked prior to the transition.</p>
 <p>guard: <math>g</math> output: <math>x = y</math> set: <math>a = b</math></p>	<p>A <b>termination transition</b>. If all refinements of state <math>s_1</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 nondeterministic FSM also has random behavior, but no probability model is provided.

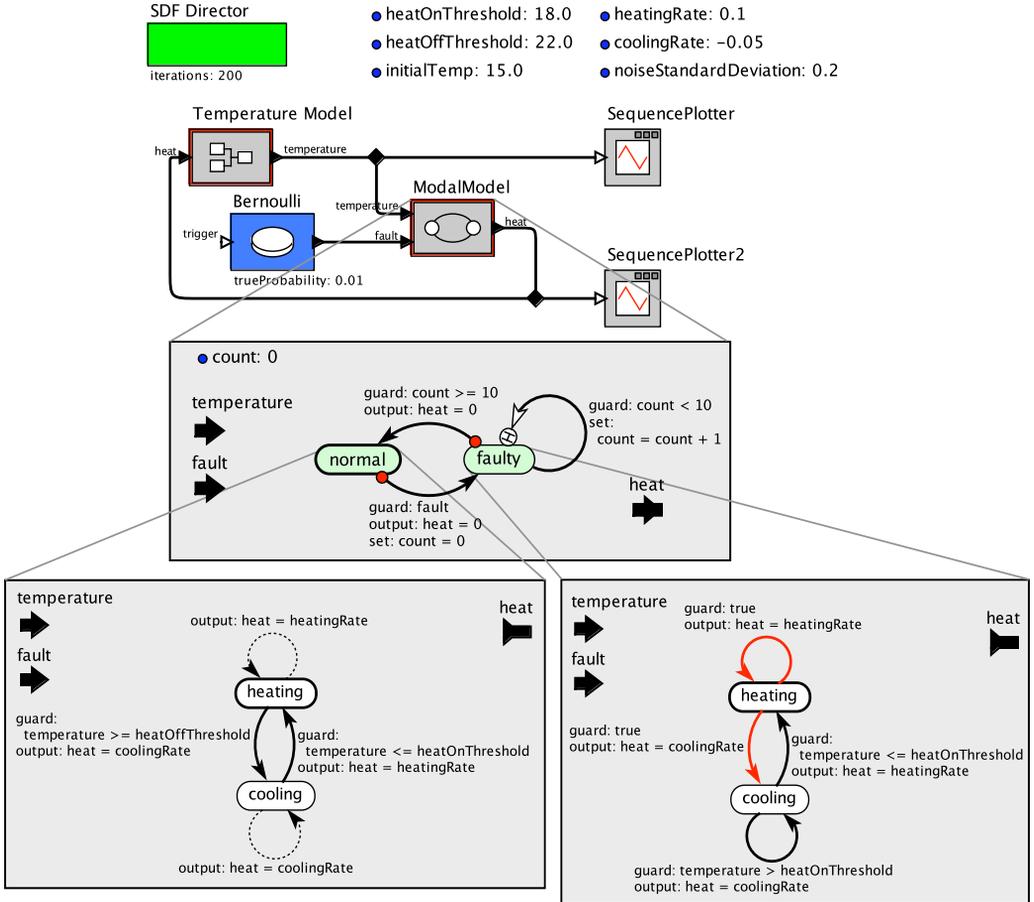


Figure 6.21: A hierarchical FSM that combines the normal and faulty thermostats of Examples 6.1 and 6.8. [online]

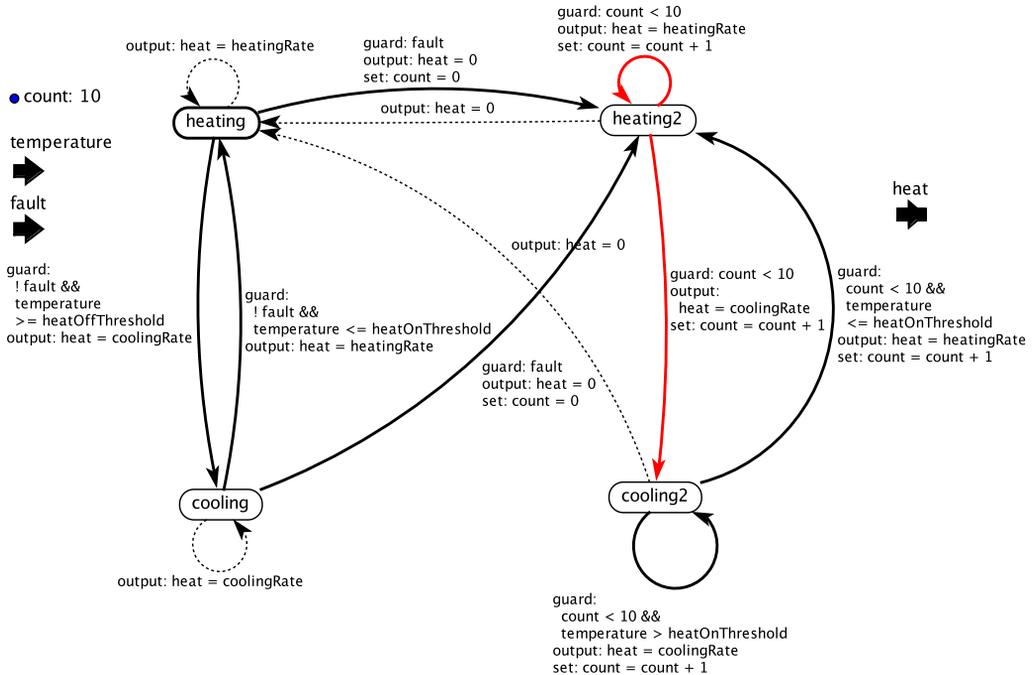


Figure 6.22: A flat FSM version of the hierarchical FSM of Figure 6.21. [online]

### 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.

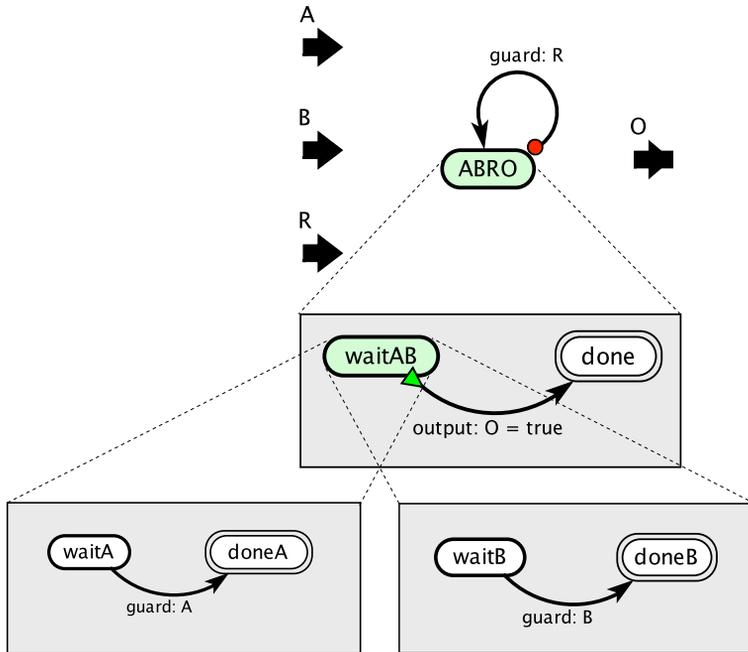


Figure 6.23: A hierarchical version of the ABRO model of Figure 6.19. [online]

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 chooses 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.

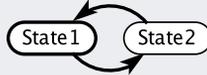
The transitions out of a state are checked in the following order:

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 state1 = 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 state1.incomingPort.link(relation2);
10 state1.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*.

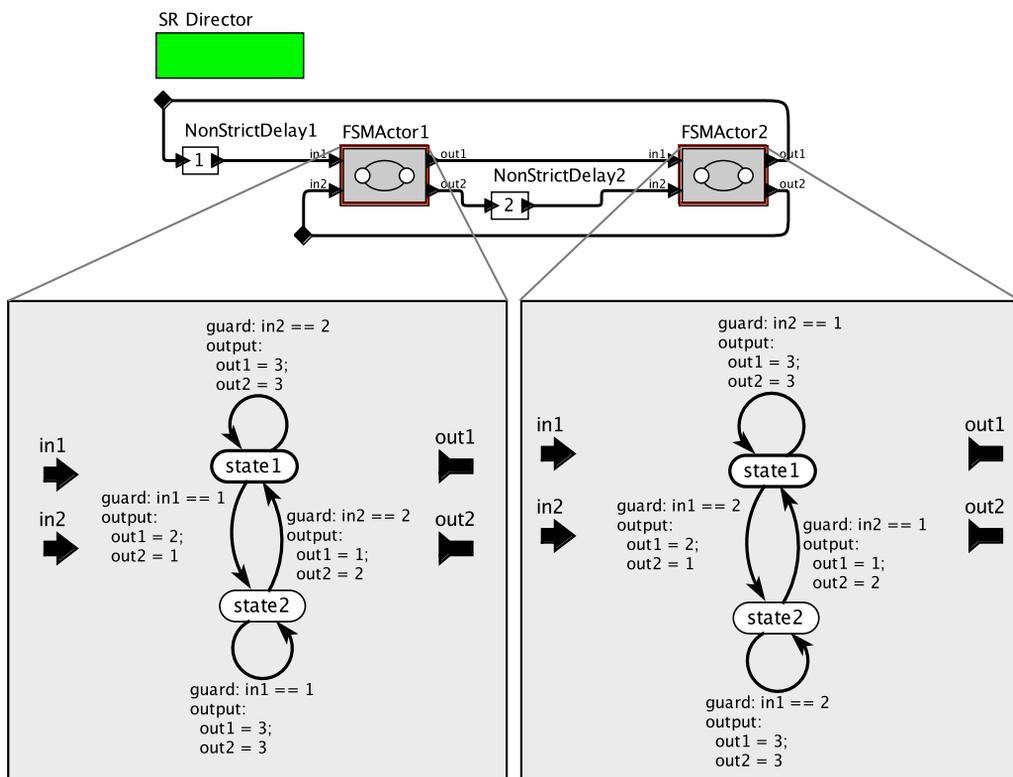


Figure 6.24: A model that requires separation of actions between the `fire` method and the `postfire` method in order to be able to converge to a fixed point. [\[online\]](#)

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*.

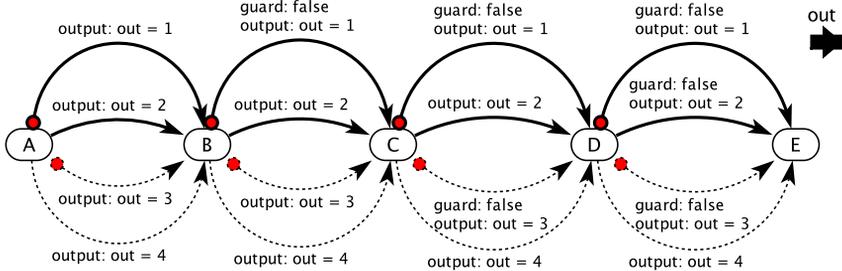
- (a) Create an implementation of this actor using an [extended state machine](#).
  - (b) How does your model behave if given three values *true* within  $N$  ticks on input port *click*, followed by at least  $N$  *absent* ticks.
  - (c) Discuss the feasibility and attractiveness of implementing this 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?

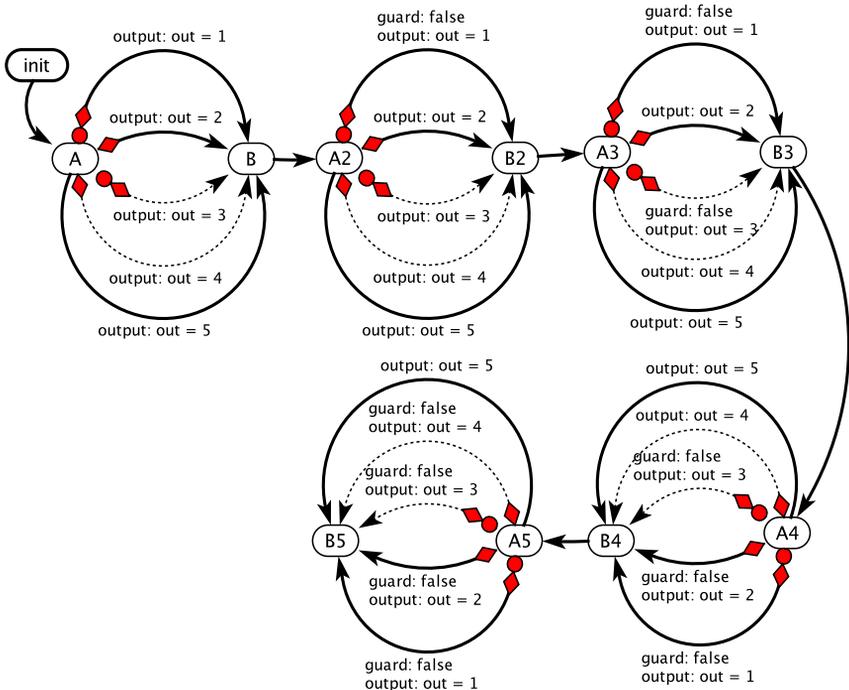
9. This exercise studies relative priorities of transitions.

(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.

# Discrete-Event Models

*Edward A. Lee, Jie Liu, Lukito Muliadi, and Haiyang Zheng*

## Contents

---

<b>7.1</b>	<b>Model of Time in DE Domain</b> . . . . .	<b>235</b>
7.1.1	Model Time vs. Real Time . . . . .	235
7.1.2	Simultaneous Events . . . . .	237
7.1.3	Synchronizing Events . . . . .	239
	<i>Sidebar: Clock Actors</i> . . . . .	241
	<i>Sidebar: Time Measurement</i> . . . . .	242
	<i>Sidebar: Time Delays</i> . . . . .	243
	<i>Sidebar: Samplers and Synchronizers</i> . . . . .	244
	<i>Probing Further: DE Semantics</i> . . . . .	245
	<i>Sidebar: Queue and Server Actors</i> . . . . .	246
<b>7.2</b>	<b>Queueing Systems</b> . . . . .	<b>247</b>
<b>7.3</b>	<b>Scheduling</b> . . . . .	<b>251</b>
	<i>Sidebar: Random Number Generators</i> . . . . .	252
	<i>Sidebar: Record Actors</i> . . . . .	253
7.3.1	Priorities . . . . .	254
7.3.2	Feedback Loops . . . . .	256
7.3.3	Multithreaded Execution . . . . .	258
7.3.4	Scheduling Limitations . . . . .	263
<b>7.4</b>	<b>Zeno Models</b> . . . . .	<b>264</b>
<b>7.5</b>	<b>Using DE with other Models of Computation</b> . . . . .	<b>265</b>

7.5.1	State Machines and DE . . . . .	266
7.5.2	Using DE with Dataflow . . . . .	267
7.6	Wireless and Sensor Network Systems . . . . .	268
7.7	Summary . . . . .	270
	Exercises . . . . .	271

The **discrete-event (DE) domain** is used to model timed, discrete interactions between concurrent actors. Each interaction is called an **event**, and is conceptually understood to be an instantaneous message sent from one actor to another. An event in Ptolemy II is a **token** (encapsulating the message) arriving at a port at a particular **model time**. The key idea in DE is that each actor reacts to input events in temporal order. That is, every time an actor fires, it will react to input events that occur later than events from previous firings. Because this domain relies on temporal sequencing, its model of time (discussed later in this chapter) is essential to its operation.

**Example 7.1:** An example of a discrete-event model is shown in Figure 7.1. This example illustrates a common application of DE, modeling faults or error conditions that occur at random times. This example models what is called a **stuck at fault**, where at a random time, a signal become stuck at a fixed value and no longer varies with the input. This example illustrates why it is important that events be processed in time-stamp order.

Specifically, the *StuckAtFault* actor is a state machine (see Chapter 6) with two states, *normal* and *faulty*. When it is in the *normal* state, then when an input arrives on the *in* port, the value of the input is copied to the *out* port and stored in the *previousIn* parameter. When an *error* event arrives, the state machine switches to the *faulty* state, and henceforth produces a constant output.

In the plot in Figure 7.2, we see that the model switches to the faulty state between times 7 and 8. The events at the *in* port are triggered in this model by the **DiscreteClock** actor, which produces events that are regularly spaced in time, while the **PoissonClock**, which triggers the error condition, produces events that are irregularly spaced in time (representing the occurrence of an error). (See the sidebar on page 241 for a description of these clock actors.) These actors are common in discrete-event models, where typically the only importance of their output is the time at which the output is produced. The value of the output does not matter much.

The *meanTime* parameter of the **PoissonClock** actor, which specifies the expected time between events, is set to 10.0 in the model, so the actual time of the error in

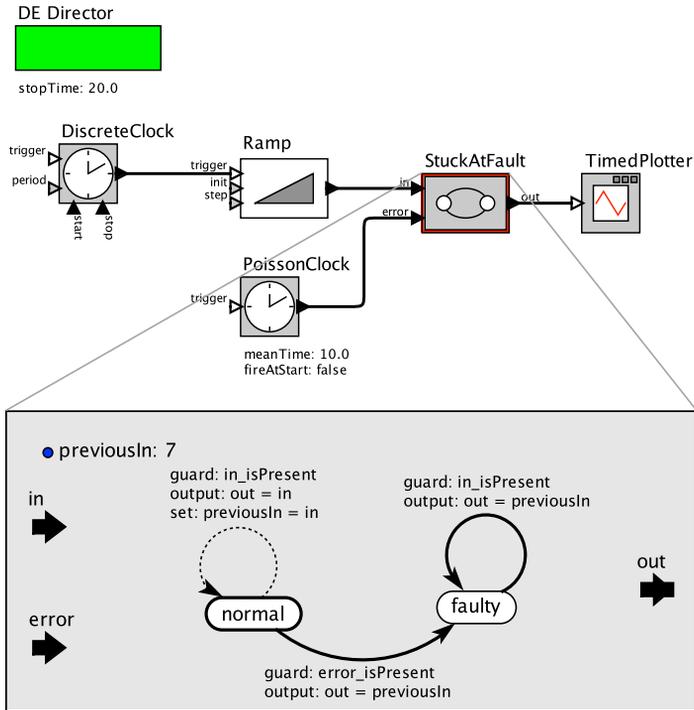


Figure 7.1: Simple example of a DE model. [[online](#)]

this run, approximately 7.48, is reasonably close to the expected time. Note that as with all random actors in Ptolemy II (see sidebar on page 252 for more such actors), you can control the *seed* of the random number generator in order to get reproducible simulations.

In general, for any actor that changes state in reaction to input events, as the Stuck-AtFault actor does, it is important to react to events in temporal order. The behavior of the actor depends on its state. In this case, once it has made the transition to the *faulty* state, its input-output behavior is very different. All subsequent input values will be ignored.

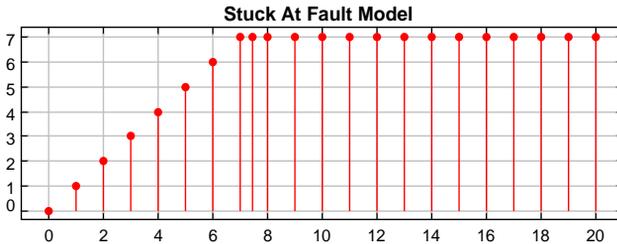


Figure 7.2: Sample output from the model in Figure 7.1.

In this chapter, we examine the mechanics and subtleties of DE models. We begin with a discussion of the model of time, and then show how DE provides a [determinate MoC](#). We discuss the subtleties that arise when events occur simultaneously and when models include feedback loops.

## 7.1 Model of Time in DE Domain

In DE models, connections between actors carry [signals](#) that consist of events placed on a time line. Each event has both a value and a [time stamp](#), where the time stamp defines a global ordering between events. This is different from [dataflow](#), where a signal consists of a sequence of tokens, and there is no time significance in the signal. A time stamp is a [superdense time](#) value, consisting of a [model time](#) and a [microstep](#).\*

### 7.1.1 Model Time vs. Real Time

A DE model executes chronologically, processing the oldest events (those with earlier time stamps) first. Time advances as events are processed, both in the model and in the outside world. There is potential confusion, therefore, between [model time](#), which

---

\*Do not confuse “model time” with “model of time.” In Ptolemy nomenclature, “model time” is a time value (e.g. 10:15 AM), whereas the “model of time” encompasses the overall approach to handling time sequencing.

is the time that evolves in the model, and **real time**, which is the time that elapses in the real world while the model executes (also called **wall-clock time**). Model time may advance more rapidly or more slowly than real time. The DE director has a parameter, *synchronizeToRealTime*, that, when set to `true`, synchronizes the two notions of time, to the extent possible. It does this by delaying execution of the model (when necessary) to allow real time to “catch up” with model time. Of course, this only works if the computer executing the model is fast enough that model time is advancing more rapidly than real time. When this parameter is set to `true`, model-time values are interpreted as being in units of seconds, but otherwise the units are arbitrary.

**Example 7.2:** Consider the DE model shown in Figure 7.3. This model includes a **PoissonClock** actor, a **CurrentTime** actor, and a **WallClockTime** actor (see sidebars on pages 241 and 242). The plot shows that wall-clock time barely advances during execution, whereas model time advances quite far (to about 9 seconds). Since the horizontal axis in this plot is model time, the model time plot increases linearly. If you set the

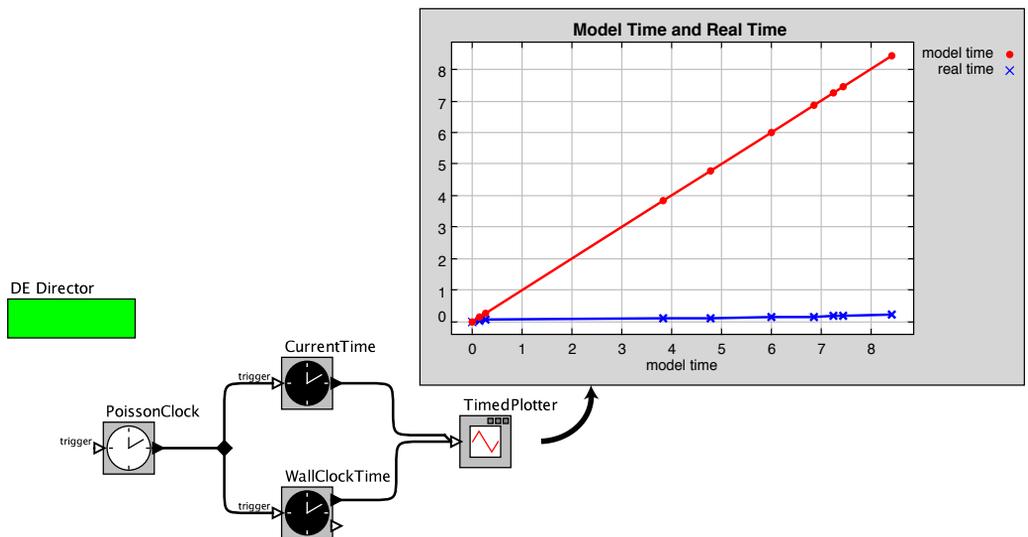


Figure 7.3: Model time vs. real time (wall clock time). [[online](#)]

*synchronizeToRealTime* parameter of the director to `true`, then you will find that the two plots coincide almost perfectly.

The ability to synchronize model time and real time is useful when you want a model to accurately display to a user the timing of events in the model. For example, a model that generates sounds according to some rhythm will not be very satisfying if it just executes as fast as possible.

### 7.1.2 Simultaneous Events

A question that arises in the DE domain is how [simultaneous](#) events are handled. As previously described, strongly simultaneous events have the same time stamp (model time and microstep), whereas weakly simultaneous events have the same model time, but not necessarily the same microstep. We have stated that events are processed in chronological order, but if two events have the same time stamp, which one should be processed first?

**Example 7.3:** Consider the model shown in Figure 7.4, which produces a histogram of the interarrival times of events from a `PoissonClock` actor. This model calculates the difference between the current event time and the previous event time, resulting in the plot that is shown in the figure. The `Previous` actor is a **zero-delay actor**, meaning that it produces an output with the same time stamp as the input (except on the first firing, where in this case it produces no output—see sidebar on 243). Thus, when the `PoissonClock` actor produces an output, there will be two (strongly) simultaneous events, one at the input to the *plus* port of the `AddSubtract` actor, and one at the input of the `Previous` actor.

At this point, should the director fire the `AddSubtract` actor or the `Previous` actor first? Either approach appears to respect the chronological order of the events, but intuitively we might expect that the `Previous` actor should be fired first. And indeed, in this example, the director will fire the `Previous` actor first, for reasons described below.

To ensure [determinism](#), the order in which actors are fired must be well defined. The order is governed by a **topological sort** of the actors in the model, which is a list of the actors in

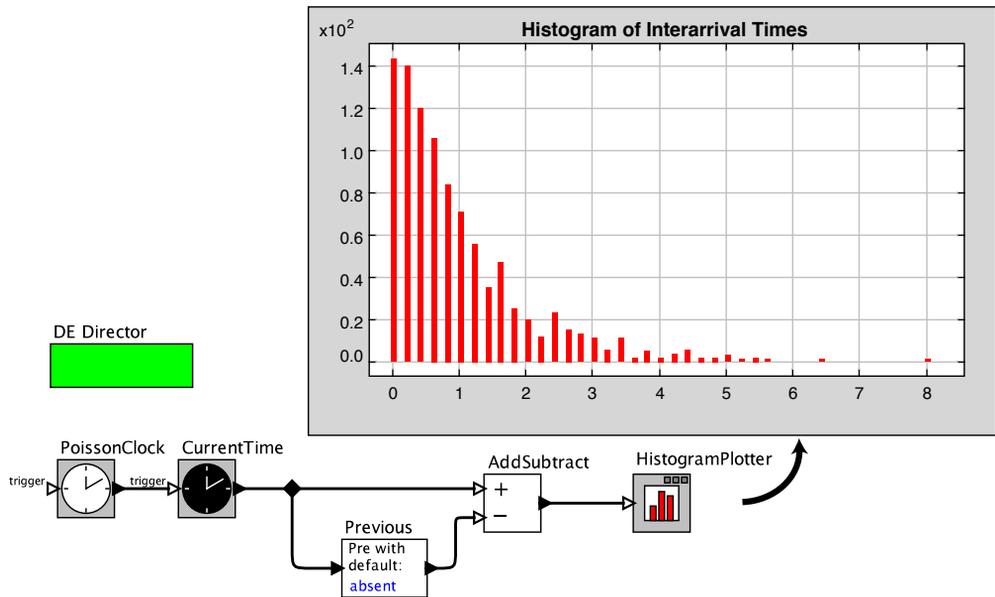


Figure 7.4: Histogram of interarrival times, illustrating handling of simultaneous events. There is a subtle bug in this model, corrected in Figures 7.5 and 7.6. [\[online\]](#)

data-precedence order. For a given model, there may be multiple valid topological sorts, but all adhere to the rule that any actor *A* that sends events to another actor *B* will always occur earlier in the topological sort. Thus, the DE director, by analyzing the structure of the model, delivers determinate behavior, where actors that produce data are fired before actors that consume their results, even in the presence of simultaneous events. All valid sorts yield the same events.

In the example above, it is helpful to know how the `AddSubtract` actor works. When it fires, it will add all (strongly simultaneous) available tokens on the *plus* port, and subtract all (strongly simultaneous) available tokens on the *minus* port. If there is a token at the *plus* port, but none at the *minus* port, then the output will equal the token at the *plus* port. Conversely, if there is a token at the *minus* port, but none at the *plus* port, then the output will equal the *negative* of the token at the *minus* port.

Based on this behavior, there is only one valid topological sort here: PoissonClock, CurrentTime, Previous, AddSubtract, and HistogramPlotter. In this list, AddSubtract appears after Previous, because Previous sends its event to AddSubtract. Therefore, given strongly simultaneous events at the inputs of AddSubtract and Previous as described in the example above, the director will always fire the Previous actor first.

### 7.1.3 Synchronizing Events

Although the example given in Figure 7.4 provides a good overview of how actor firings are sequenced, it has a subtle problem. Each output of the AddSubtract actor is supposed to be the time between arrivals of two successive events from the PoissonClock actor (the **interarrival time**). However, the very first event produced by the CurrentTime actor has a value equal to the time stamp of the first event produced by the PoissonClock (which in this case is 0.0 because the PoissonClock by default produces an initial event when it begins execution). The Previous actor, however, will not produce any output at this time, because (by default) its output is absent upon arrival of its first input (see sidebar on page 243). Hence, the first output of the AddSubtract will have value 0.0, which is not, in fact, an interarrival time! Thus, the plotted histogram includes a spurious value 0.0.

To fix this problem, we would like to ensure that the AddSubtract actor receives exactly one event on each input port, and only receives events when there is one available for each port. We can accomplish this using the **Sampler** actor, as shown in Figure 7.5 (see sidebar on page 244). This actor produces an output event only if there is an input event

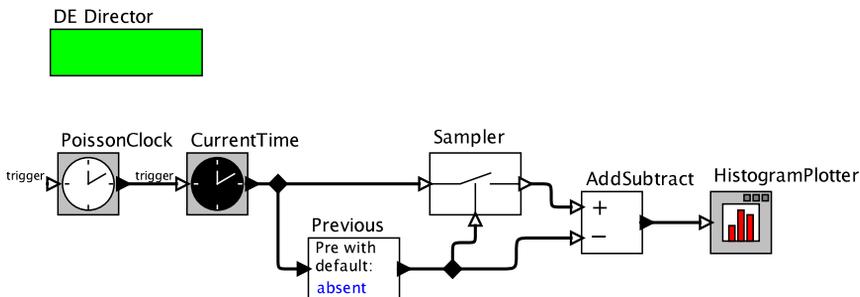


Figure 7.5: Histogram of interarrival times, correcting the subtle bug in Figure 7.4 using the Sampler actor. [online]

on its *trigger* input (the bottom port). Upon receiving a trigger input, it passes to its output whatever (strongly simultaneous) events are available on its input port. So in this example, the first output from the CurrentTime actor will be discarded, because, at that time, there is no event on the *trigger* input.

There are a number of other ways to accomplish the same goal. For example, the Synchronizer actor described in the sidebar on page 244 can also be used in place of the Sampler. More directly, the TimeGap actor can also be used to solve this problem (see sidebar on page 242), as shown in Figure 7.6. It combines the functionality of the Previous, Sampler, and AddSubtract actors.

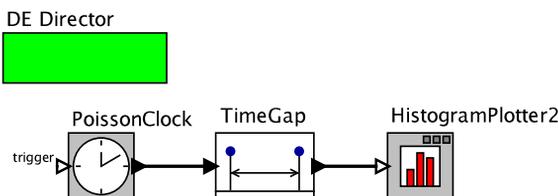
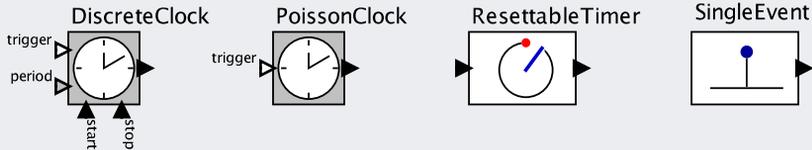


Figure 7.6: Histogram of interarrival times using the TimeGap actor. [\[online\]](#)

### Sidebar: Clock Actors

**Clock** actors generate timed events. Four such actors are shown below:

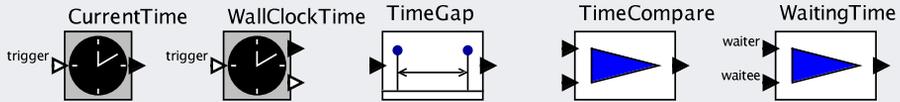


These actors are found in the `Sources→TimedSources` library, except for `ResettableTimer`, which is in `DomainSpecific→DiscreteEvent`.

- **DiscreteClock** is a versatile event generator. Its simplest use is to generate a sequence of events that are regularly spaced in time. Its default parameters cause the actor to produce tokens with value 1 (an *int*) spaced one time unit apart, starting at execution start time (typically time zero). This default setting is used to produce the plot in Figure 7.2. But `DiscreteClock` can generate much more complex event patterns that cycle through a finite sequence of values and have periodically repeating time offsets from the start of each period. It can also generate a one-time, finite sequence of events (instead of a periodic pattern) by setting the *period* to `Infinity`. These events can be arbitrarily placed in time. See the actor documentation for details.
- **PoissonClock**, in contrast, produces events at random times. The time between events is given by independent and identically distributed (**IID**) exponential random variables. An output signal with these characteristics is called a **Poisson process**. Like the `DiscreteClock` actor, the `PoissonClock` actor can cycle through a finite sequence of values, or it can produce events with the same value each time.
- **ResettableTimer** produces an output event after the time specified by the input event has elapsed (in **model time**, not **real time**). That is, an output event is produced at the model time of the input plus the value of the input. If the input value is zero, then the output will be produced at the next **microstep**. This actor allows a pending event to be canceled, and also allows a new input event to preempt a previously scheduled output. See the actor documentation for details.
- **SingleEvent** is not really needed, since `DiscreteClock` is capable of producing single events. Nonetheless, this actor is sometimes useful because it visually emphasizes in a model that only a single event is produced.

### Sidebar: Time Measurement

The following actors provide access to the [model time](#) of events:



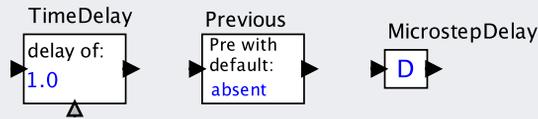
The **CurrentTime** actor, which is found in `Sources→TimedSources`, observes the model time of events. It produces an output event whenever an input event arrives, where the value of the output event is the model time of the input event. (The time value on the output of the **CurrentTime** actor is represented as a *double*, so it may not be an exact representation of the internal model time, which, as explained in Section 1.7.3, has a fixed resolution that does not change as the value of time increases. The resolution of a *double*, by contrast, decreases as the magnitude increases.) The output event has the same [time stamp](#) as the input event, so the actor's reaction is conceptually instantaneous. (**CurrentMicrostep**, found in same library and not shown above, is useful primarily for debugging.)

The **WallClockTime** actor, which is found in `RealTime`, observes the real time of events. When it fires in reaction to a trigger event, it outputs a *double* representing the amount of [real time](#) that has passed (in seconds) since the actor was initialized. Since this value depends on arbitrary scheduling decisions that the DE director makes, it is [nondeterminate](#). Nonetheless, it can be useful for measuring performance, for example by firing it at the beginning and ending of a part of the model's execution. The inputs that arrive on the *trigger* input also pass through to an output port, a feature that makes it somewhat easier to control the scheduling of downstream actors in some domains.

The `DomainSpecific→DiscreteEvent` library has the other time-related actors, which are used to measure model-time differences between events. **TimeGap** measures the difference between successive events in a signal. **TimeCompare** measures the time gap between events in two signals. It is triggered by the arrival of an event at either input, and outputs the difference between the model time of this event and that of the last event on the other port. **WaitingTime** also measures the gap between two signals, but in a somewhat different manner. When an event arrives on the *waiter* port, the actor begins waiting for an event to arrive on the *waitee* port. When the first such event arrives, the actor outputs the model time difference.

### Sidebar: Time Delays

The following actors provide mechanisms for delaying events by manipulating their time stamps:



The most useful of these is **TimeDelay**, which increments the **model time** of the input event by a specified amount. The amount of the increment is displayed in the icon and defaults to 1.0. This actor delays an event (in model time, not in real time). Only the model time is incremented; the microstep of the output is the same as the microstep of the input.

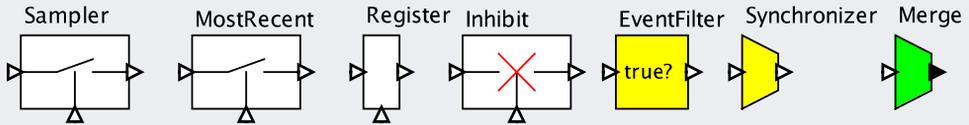
The amount of the delay can optionally be given on the bottom input port. If an event is provided on the bottom input port, then the value of that event specifies the delay for input events that arrive at the same time or later (i.e., that have a time stamp greater than or equal to that of the event arriving on the bottom input port). The **TimeDelay** actor can be found in `DomainSpecific→DiscreteEvent`.

Occasionally, it is useful to set a model-time delay to zero. In this case, the model time of the output event is the same as that of the input, but its **microstep** is incremented. This setting can be useful for feedback loops, as discussed in the main text. A **TimeDelay** with delay equal to 0.0 is equivalent to the **MicrostepDelay** actor.

The **Previous** actor, upon receiving an input event, produces an output event with the same time stamp as the input event, but with the value of the previously received input event. When it receives the first event (i.e., there has been no previous event) it outputs a default value if one is given, or it outputs nothing (absent output). This actor, therefore, delays each input event until the arrival of the next input event.

## Sidebar: Samplers and Synchronizers

The following actors provide mechanisms for synchronizing events:



The Sampler and Synchronizer are in `FlowControl`→`Aggregators`, whereas the rest are in `DomainSpecific`→`DiscreteEvent`.

- **Sampler** copies selected events from its input port (a [multiport](#)) to its output port when it receives an event on the *trigger* port (at the bottom of the icon). Only those input events that are strongly simultaneous with the trigger are copied; these events are sent to the corresponding output [channel](#).
- **MostRecent** is similar to Sampler, except that, upon receiving a trigger, it copies the *most recent* input event (which may or may not be simultaneous with the trigger event) to the corresponding output channel. It provides an optional *initialValue* parameter, which specifies the output value if no input has arrived upon triggering.
- **Register** is similar to MostRecent, except that upon receiving a trigger, it copies the most recent *strictly earlier* input event to the corresponding output channel. This actor, unlike Sampler or MostRecent, always introduces delay, and hence is also similar to the delay actors described in the sidebar on page 243. The delay can be as small as one [microstep](#), in which case the input and output will be weakly [simultaneous](#). Because it introduces delay, this actor is useful in [feedback](#) loops (see Section 7.3.2).
- **Inhibit** is the converse of the Sampler. It copies all inputs to the output *unless* it receives a *trigger* input.
- **EventFilter** accepts only *boolean* inputs, and copies only true-valued inputs to the output.
- **Synchronizer** copies inputs to outputs only if every input [channel](#) has a strongly [simultaneous](#) event.
- **Merge** merges events on any number of input channels into a single signal in time-stamp order. If it receives strongly simultaneous events, then it either discards all but the first one (if the *discard* parameter is `true`), or it outputs the events with increasing microsteps (if the *discard* parameter is `false`).

## Probing Further: DE Semantics

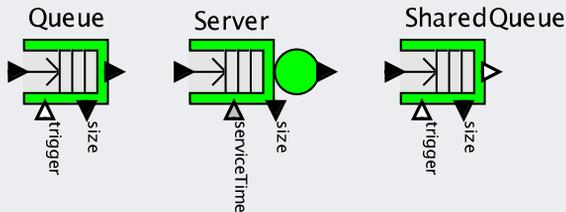
Discrete-event modeling based on time stamps has been around for a long time. One of the earliest complete discrete-event formalisms is called DEVS, for Discrete Event System Specification (Zeigler, 1976). Many subtly different variants have appeared over time (see for example Ramadge and Wonham (1989), Cassandras (1993), Baccelli et al. (1992), and Zeigler et al. (2000)). Variants of DE also form the foundation for the widely used hardware description languages VHDL, Verilog, and SystemC, and a number of network simulation tools such as OPNET Modeler (from OPNET Technologies, Inc.), ns-2 (a collaborative open-source effort led by the Virtual Internetwork Testbed Project VINT), and ns-3 (<http://www.nsnam.org/>). The SysML implementation in IBM Rational's **Rhapsody** tool is also a variant of a DE model. The variant of DE described in this chapter, particularly its use of **superdense time**, appears to be unique.

The formal **semantics** of DE is a fascinating and deep topic. The oldest approaches describe execution of DE models in terms of state machines (Zeigler, 1976). More recent approaches define a metric space (Bryant, 1985) in which actors become contraction maps and the meaning of a model becomes a fixed point of these maps (Reed and Roscoe, 1988; Yates, 1993; Lee, 1999; Liu et al., 2006). DE semantics have also been given as a generalization of the semantics of **synchronous-reactive** languages (Lee and Zheng, 2007), and as fixed points of monotonic functions over a complete partial order (CPO) (Broy, 1983; Liu and Lee, 2008). These latter approaches are denotational (Baier and Majster-Cederbaum, 1994), whereas the state machine approaches have a more operational flavor.

DE models can be large and complex, so execution performance is important. There has also been extensive work on simulation strategies for DE models. A particularly interesting challenge is exploiting parallel hardware. The strong ordering imposed by time stamps makes parallel execution difficult (Chandy and Misra, 1979; Misra, 1986; Jefferson, 1985; Fujimoto, 2000). A recently proposed strategy called **PTIDES** (for programming temporally integrated distributed embedded systems), leverages network time synchronization to provide efficient distributed execution (Zhao et al., 2007; Lee et al., 2009b; Eidson et al., 2012). In PTIDES, DE is used not only as a simulation technology, but also as an implementation technology. That is, the DE event queue and execution engine become part of the deployed embedded software.

### Sidebar: Queue and Server Actors

The following actors provide queueing and are particularly useful for modeling behavior in communication networks, manufacturing systems, service systems, and many other queueing systems. These actors are found in the `DomainSpecific→DiscreteEvent` library.



- Queue** takes a token received on its input port and stores it in the queue. When the *trigger* port receives an **event**, the oldest element in the queue is produced on the output. If there is no element in the queue when a token is received on the trigger port, then no output is produced. In this case, if the *persistentTrigger* parameter is true, then the next input that is received will be sent immediately to the output. A *capacity* parameter limits the capacity of the queue; inputs received when the queue is full are discarded. The *size* output produces the size of queue after each input is handled.
- Server** models a server with a fixed or variable service time. A server is either busy (serving a customer) or not busy at any given time. If an input arrives when the server is not busy, then the input token is produced on the output with a delay given by the *serviceTime* parameter. If an input arrives while the server is busy, then that input is queued until the server becomes free, at which point it is produced on the output with an additional delay given by the *serviceTime* parameter. If several inputs arrive while the server is busy, then they are served on a first-come, first-served basis. The *serviceTime* may be provided on an input port rather than in the parameter. A *serviceTime* received on the input port applies to all events that arrive at the same or later times, until another *serviceTime* value is received. The *size* output produces the size of queue after each input is handled.
- SharedQueue** is similar to Queue but supports multiple outputs, each of which draws tokens from the same queue.

## 7.2 Queueing Systems

A common use of DE is to model **queueing systems**, which are networks of queues and servers. These models are typically paired with models of random arrivals and service times. One of the most basic queueing systems is the **M/M/1 queue**, where events (such as customers) arrive according to a Poisson process and enter a queue. When they reach the head of the queue, they are served by a single server with a random service time. In an M/M/1 queue, the service time has an exponential distribution, as illustrated by the next example.

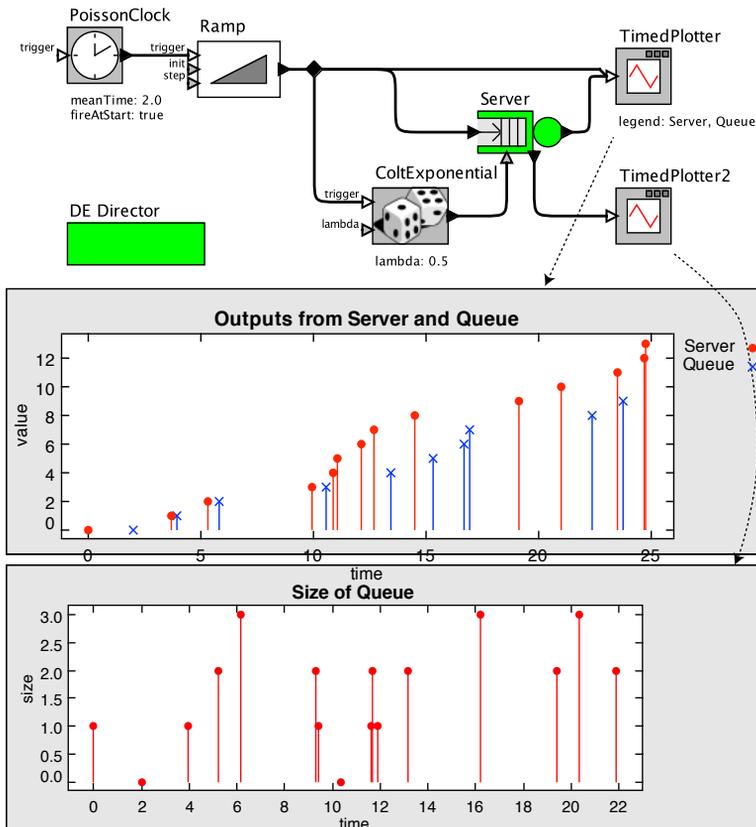


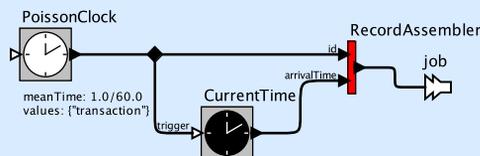
Figure 7.7: A model of an M/M/1 queue. [online]

**Example 7.4:** A Ptolemy II model of an M/M/1 queue is shown in Figure 7.7. The **PoissonClock** simulates the arrival of customers. In this case, the average interarrival time is set to 2.0. The **Ramp** actor is used to label the customers with distinct integers for identification. The **ColtExponential** shown in the figure is one of many **random number generators** in Ptolemy II (see box on page 252). For each customer arrival, it generates a new random number according to an exponential distribution. The *lambda* parameter of the **ColtExponential** actor is set to 0.5, which results in a mean value of  $1/0.5$  or 2.0. The **Server** actor is a queue with a server (see box on page 246). In this case, a new service time is specified for each customer that arrives. The **Server** outputs both the customer number, shown in the upper plot, and the size of the queue when a customer arrives or departs, shown in the lower plot. Note that three customers arrive in a burst around time 4, resulting in a buildup of the queue size at that time.

More interesting examples use networks of queues and servers.

**Example 7.5:** Figure 7.8 shows a model of a storage system where jobs arrive at random and are processed by a CPU (central processing unit). The CPU then writes to disk 1, performs additional processing, writes to disk 2, and finally performs a third round of processing. This particular model is given by **Simitci (2003)**, who analyzes the model using queueing theory and predicts an average latency through the network of 0.057 seconds. The experimental result, shown in the **MonitorValue** actor, is very close to the predicted value for this particular Monte Carlo run.

To allow easy measurement of job latencies, incoming jobs are time stamped with the time of arrival. Specifically, the **EventGenerator** composite actor is implemented as follows:



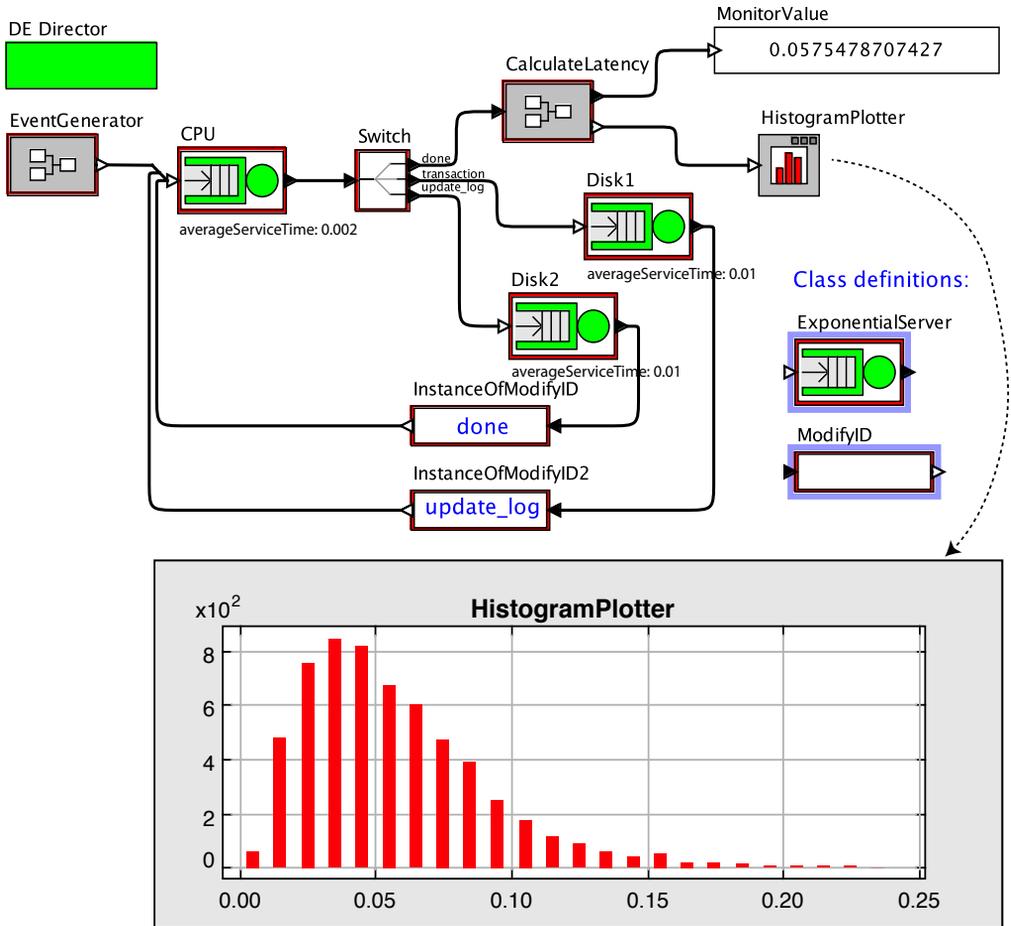
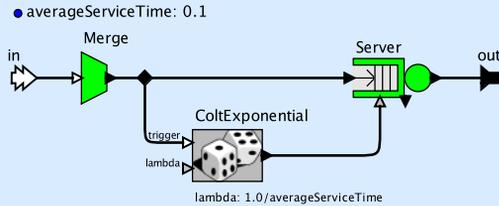


Figure 7.8: A queueing model of a transaction processor that writes to two disks. [\[online\]](#)

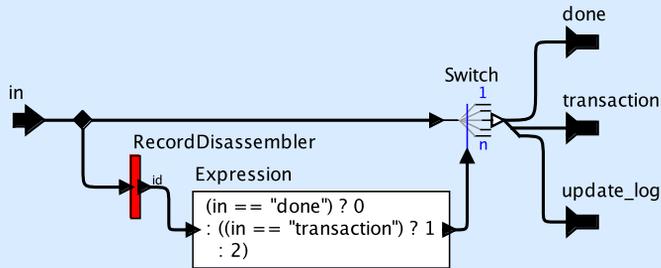
This composite actor produces events according to a Poisson process with average interarrival time of 1/60-th of a second. Each event is a `record` with two fields: an `id` and an `arrivalTime` (see box on page 253). The `arrivalTime` carries the `time stamp` of the event, which is provided by the `CurrentTime` actor. The `id` is a constant string “transaction” that will be used to route the event to the appropriate disk. Referring

again to Figure 7.8, the CPU, Disk1, and Disk2 actors are also composite actors, each of which is an instance of the actor-oriented class **ExponentialServer**, defined as follows:



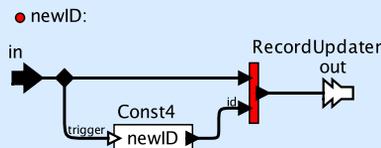
This composite actor has a single parameter, *averageServiceTime*. It implements a server with a random service time. Upon arrival of an event on any input channel, the **ColtExponential** actor generates a new random number from an exponential distribution. This random number specifies the service time that the newly arrived event will experience when it is served. If the queue is empty, it will be served immediately. Otherwise, it will be served when the server has served all events that precede it in the queue.

Referring again to Figure 7.8, the Switch is defined as follows:

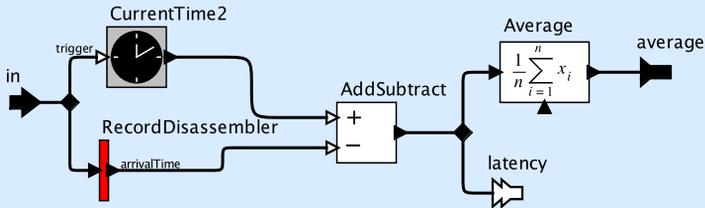


This actor extracts the *id* field from the incoming record, and, using an **Expression** and a **Switch** actor, routes the input event to one of three output ports depending on the *id*.

The two feedback paths contain instances of the actor-oriented class **ModifyID**, which is defined as follows:



This actor constructs a new record from the input record by replacing the *id* field of the record using a [RecordUpdater](#) (see box on page 253). The event with the new *id* is processed by the CPU again. When the *id* becomes “done,” the job event is routed to the CalculateLatency composite actor, implemented as follows:



This submodel measures the overall latency of the job. It extracts the *arrivalTime* from the incoming record and subtracts it from the time stamp of the completed job. It then outputs both the calculated latency and a running average of the latency calculated with the [Average](#) actor. A [HistogramPlotter](#) actor is used to plot a histogram of the latencies, and the [MonitorValue](#) is used to display the running average in the model itself.

## 7.3 Scheduling

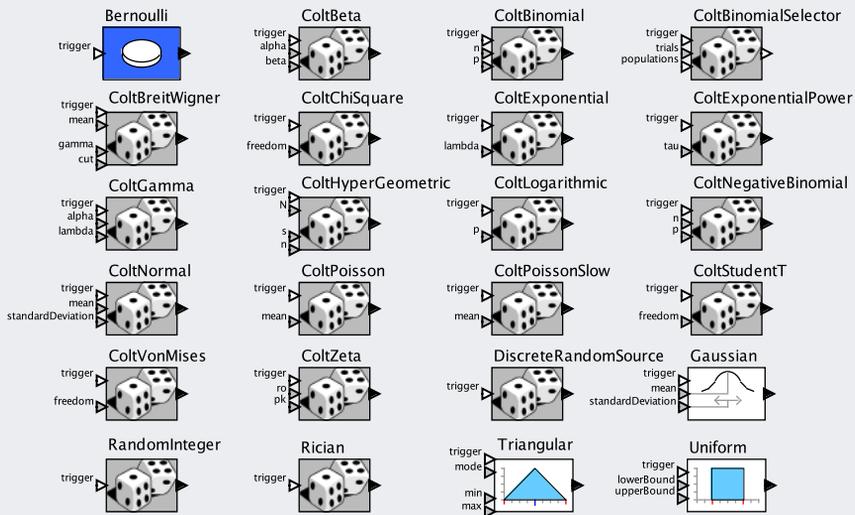
The main task of the DE director is to fire actors in time-stamp order. The DE director maintains an event queue that is sorted by time stamp, first by [model time](#) and then by [microstep](#). The DE director controls the execution order by selecting the earliest event in the event queue and making its time stamp the **current model time**. It then determines the actor the event is destined for, and finds all other events in the event queue with the same time stamp that are destined for the same actor. It sends those events to the actor’s input ports and then fires the actor. If after firing the actor it has not consumed all these events, then the director will fire the actor again until all input events have been consumed.<sup>†</sup>

<sup>†</sup>Note that if an actor is written incorrectly and does not consume input events, then this strategy results in an infinite loop, where the actor will be repeatedly fired forever. This reflects a bug in the actor itself. Actors are expected to read inputs, and in the DE domain, reading an input consumes it.

The DE director uses a specific rule to ensure determinism. Specifically, it guarantees that it has provided *all* events that have a **time stamp** equal to the current time stamp before it fires the actor. It cannot subsequently provide an additional event with the same time stamp. Such an additional event may possibly have the same model time, but in that case it does not have the same microstep.

### Sidebar: Random Number Generators

Ptolemy II includes a number of **random number generators**, shown below:



The actors all produce a random number on their output port on each firing. Most of them have parameters that can also be set on input ports, so that parameters can vary on each firing. See the documentation of each actor for details.

All of these actors provide a *seed* parameter, which can be used to ensure repeatable random sequences. When you set the seed parameter of one random actor, then you set it for all random actors (it is a **shared parameter**, meaning that that its value is shared by all actors in a model that have this parameter). All random actors also share a single underlying random number generator, so reproducible experiments are easy to control.

The actors named with the prefix **Colt** were created by David Bauer and Kostas Oikonomou, with contributions from others, using an open source library called Colt, originally developed by CERN (the European Organization for Nuclear Research).

### Sidebar: Record Actors

The `record` type in Ptolemy II is similar to a `struct` in C. It can contain any number of named fields, each of which can have an arbitrary data type (it can even be a record). There are several actors that manipulate records, found in [Actors→FlowControl→Aggregators] and shown below.



All of these actors require the addition of new ports using the [Configure→Ports] menu item on the actor. We recommend making the field name visible using the Show Name column; see Figure 2.18.

- **RecordAssembler** outputs a record that contains one field for each input port, where the name of the field is the same as the name of the input port.
- **RecordDisassembler** extracts fields from a record. The name of the output port must match the name of the field; the type system (see Chapter 14) will report a type error if the input record does not have a field that matches the output port name.
- **OrderedRecordAssembler** constructs a record token in which the order of the fields in the record matches the order of the input ports (top to bottom). This actor is probably not useful unless you are writing Java code that iterates over the fields of the record and depends on the order.
- **RecordUpdater** adds or modifies fields of a record. The icon's built-in input port provides the original record. Additional input ports must be added and named with the field name you wish to add or modify. The output record will be a modified record.

There are two other actors that are useful for constructing records; these are found in [Actors→Conversions] and shown below:



Both of these actors accept string inputs. **ExpressionToToken** parses the input string, which can be any string accepted by the expression language described in Chapter 13, including records. If the input string specifies a record, then the output token will be a record token. **JSONToRecord** accepts any string in the widely used Internet **JSON** format and produces a record.

As discussed earlier, the DE director performs a [topological sort](#) of the actors. Once this sort has been performed, each actor can be assigned a **level**. The level is largest number of upstream actors along a path from either a source actor (which has no upstream actors) or a delay actor.

**Example 7.6:** For example, the actors in [Figure 7.5](#) have the following levels:

- PoissonClock: 0
- CurrentTime: 1
- Previous: 2
- Sampler: 3
- AddSubtract: 4
- HistogramPlotter: 5

When two events with the same time stamp are inserted into the event queue, the event that is destined for the actor with the lower level will appear earlier in the queue. This ensures, for example, that in [Figure 7.5](#), Previous will fire before Sampler, and Sampler will fire before AddSubtract. Thus, when AddSubtract fires, it is assured of seeing all input events at the current model time.

#### 7.3.1 Priorities

The DE director sorts events by model time, then by microstep, and then by level. But it is still possible to have events that have the same time stamp and level.

**Example 7.7:** Consider the model shown in [Figure 7.9](#). Here, the two Ramp actors have the same level (1) and the two FileWriter actors have the same level (2). When the clock produces an event, there will be two events in the event queue with the same time stamp and level, one destined for Ramp and one destined for Ramp2. Since these two actors do not communicate, the order in which they are fired does not matter. However, the two FileWriter actors might be writing to the same file (or to standard out). In this case, the firing order does matter, but the order cannot

be determined by the time stamp and level alone. We may not want the director to arbitrarily choose an order. In this case, the designer may choose to use priority parameters, as described below.

The previous example illustrates an unusual scenario: two actors can affect each other even though there is no direct communication between them in the model. They are interacting **under the table**, in a manner that is invisible to the DE director. When there is such interaction, the model builder may wish to exercise some control over the order of execution.

The `Utilities`→`Parameters` library contains a *Priority* parameter that can be dragged and dropped onto actors. The value can be set for each actor independently by double clicking on the actor. A lower value is interpreted as a higher priority. When events have the same time stamp and the same level, the DE director consults the priority of the destination actors, and places events destined to actors whose priorities are higher (*Priority* has a lower value) earlier in the event queue.

**Example 7.8:** For the example in Figure 7.9, Ramp2 and FileWriter2 have *Priority* zero, so they will fire before Ramp and FileWriter, which both have *Priority* one. Without the use of priorities, the order would be *nondeterminate*.

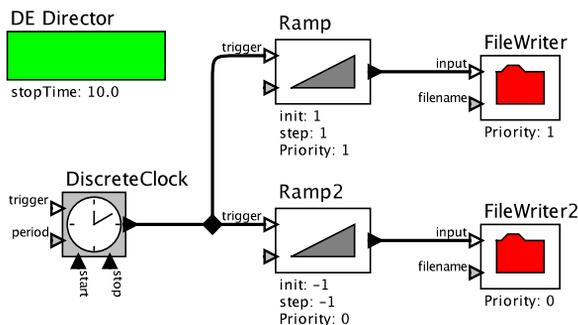


Figure 7.9: Although it is rarely necessary, it may sometimes be useful to set the priorities of actors in DE. This only has an effect when the firing order is otherwise not determined by time stamps or by data precedences (levels). [[online](#)]

In DE modeling it is rare to need priorities; because their value has global meaning (i.e., the priorities are honored throughout the model), this mechanism is not very modular.

## 7.3.2 Feedback Loops

If the model has a directed loop (called a **feedback** loop), then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor that introduces a time delay, such as [TimeDelay](#), [Register](#), or queues or servers (see sidebars on pages [243](#), [244](#), and [246](#)).

**Example 7.9:** Consider the model shown in Figure [7.10](#). That model has a [DiscreteClock](#) actor that produces events every 1.0 time units. Those events trigger the [Ramp](#) actor, which produces outputs that start at 0 and increase by 1 on each firing. In this model, the output of the Ramp goes into an [AddSubtract](#) actor, which subtracts from the Ramp output its own prior output delayed by one time unit. The result is shown in the plot in the figure.

To ensure that models with feedback are determinate, the DE director assigns delay actors a level of zero, but delays the time at which they read their inputs until the [postfire](#) stage, which occurs after firing any other actors that are scheduled to run at the same time.

**Example 7.10:** In the example in Figure [7.10](#), [TimeDelay](#) has level 0, whereas [AddSubtract](#) has level 1, so [TimeDelay](#) will fire before [AddSubtract](#) at any time stamp when it is scheduled to produce an output. However, it does not read its input until after the [AddSubtract](#) actor has fired in response to its own output event. [TimeDelay](#) reads its input in the postfire phase, at which time it simply records the input and requests a new firing at the current time plus its time delay (1.0, in this case).

Occasionally, it is necessary to put a [TimeDelay](#) actor in a feedback loop with a delay of 0.0. This has the effect of incrementing the microstep without incrementing the model time, which allows iteration without time advancing.

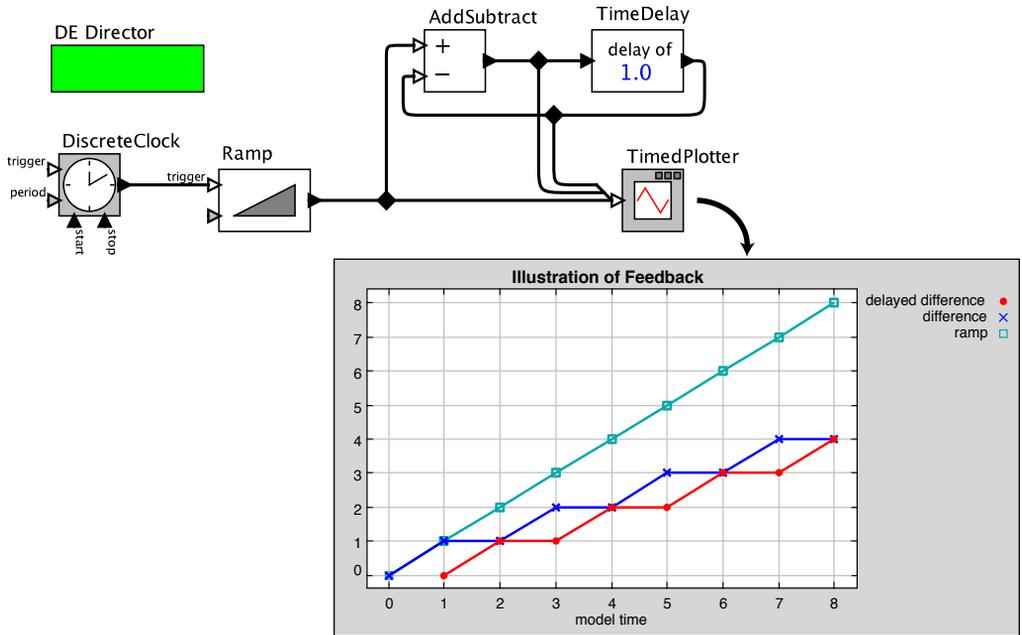


Figure 7.10: Discrete-event model with feedback, which requires a delay actor such as TimeDelay. [\[online\]](#)

**Example 7.11:** Consider the model in Figure 7.11, which produces the plot in Figure 7.12. This model produces a variable number of events at each integer model time using a feedback loop that has a TimeDelay actor with the delay set to 0.0. This causes the events that are fed back to use incremented microsteps at the same model time. This model uses a BooleanSwitch (see sidebar on page 119) to feed back a token only if its value is non-negative.

The previous example illustrates iteration in DE using a construct similar to the feedback iteration in dataflow illustrated in Example 3.11. In DE, we can use a Merge actor rather than a BooleanSelect because the use of time stamps makes the Merge determinate.

### 7.3.3 Multithreaded Execution

The DE director fires one actor at a time and does not fire the next actor until the previous one finishes firing. This approach creates two potential problems. First, if an actor does not return from its `fire` method, then the entire model will be blocked. This issue can arise if the actor is attempting to perform I/O. Second, the execution of the model is unable to exploit multicore architectures. Both of these problems can be solved using the **ThreadedComposite** actor, found in the `HigherOrderActors` library. The following example illustrates the first problem.

**Example 7.12:** Consider the example in Figure 7.13, which uses the `InteractiveShell` actor, previously considered in Section 4.1.1. In this model, the `Expression` actor is used to format a string for display (see Section 13.2.4 in Chapter 13).

Note that the time stamps in this model are not particularly meaningful. They do not accurately reflect the time at which the user types a value, but they do represent the order of what the user typed. That is, the time stamp is bigger for values that are entered later. In addition, this model cannot do anything while the `InteractiveShell` actor is waiting for the user to type something.

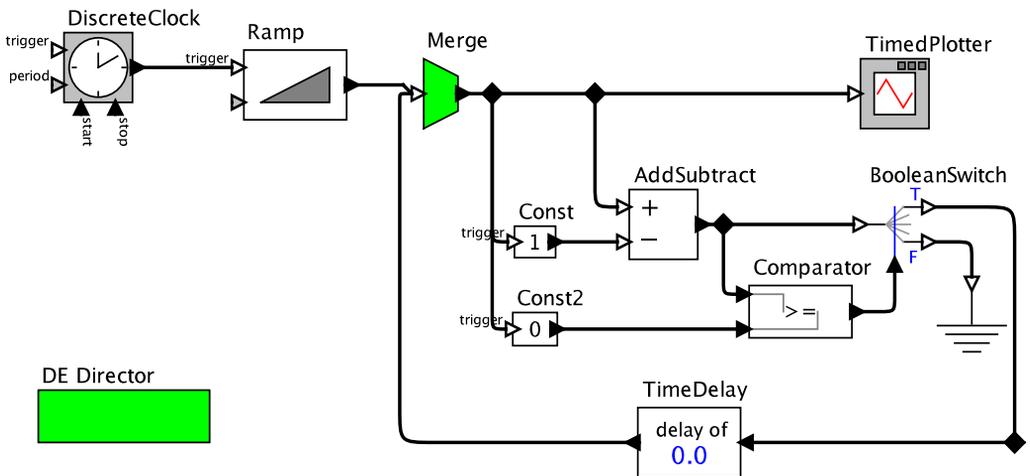


Figure 7.11: Illustration of `TimeDelay` with delay value of zero. [[online](#)]

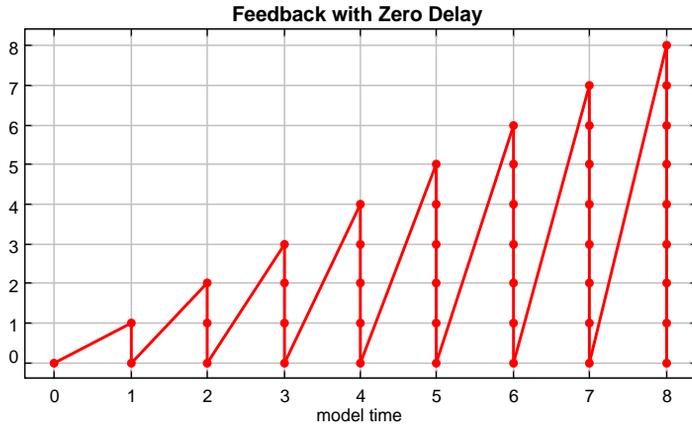


Figure 7.12: Result of executing the model in Figure 7.11.

The `ThreadedComposite` actor is an example of a **higher-order component**, which is an actor that has another actor as a parameter or an input. `ThreadedComposite` is parameterized by another actor, and when it fires, instead of performing any functionality itself, it begins the execution of the other actor in another thread, and returns immediately from the `fire` method. Since the actor's functionality executes in another thread, the model is not blocked. Most interestingly, the `ThreadedComposite` is able to perform such concurrent execution while ensuring **determinate** results. This capability can be used to create a much more useful interactive model, as shown in the next example.

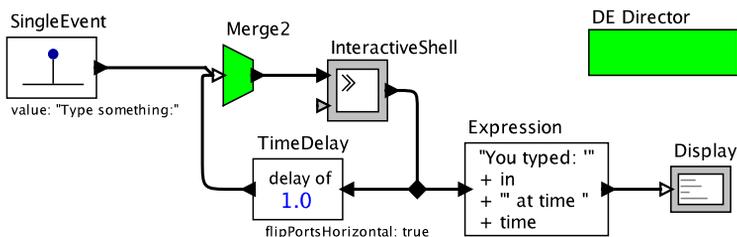


Figure 7.13: A DE model using the `InteractiveShell` actor, whose execution is stalled until a user types something. [\[online\]](#)

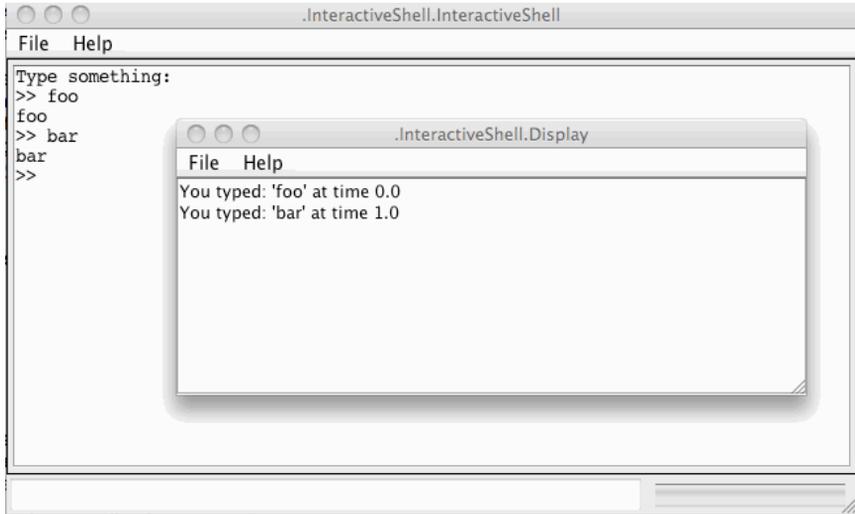


Figure 7.14: An execution of the model in Figure 7.13.

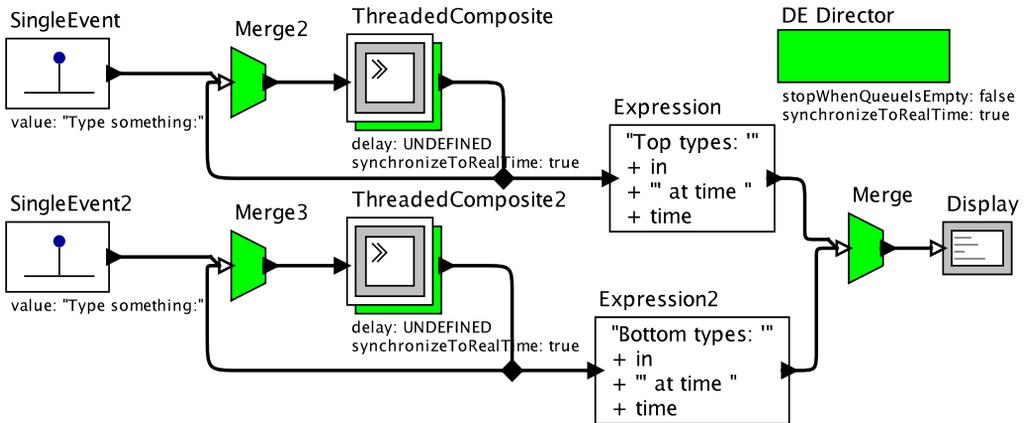


Figure 7.15: A model where two instances of InteractiveShell are executed in separate threads by the ThreadedComposite actor. [\[online\]](#)

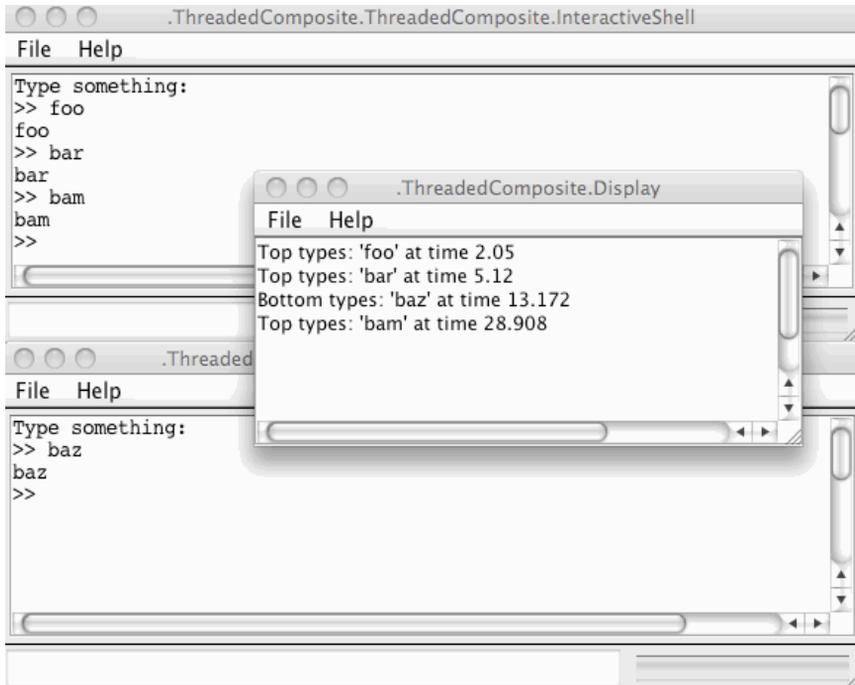


Figure 7.16: An execution of the model in Figure 7.15.

**Example 7.13:** Consider the model in Figure 7.15. This model opens two interactive shell windows into which users can type, as shown in Figure 7.16. The model will not block if the user does not type something, as it would without the ThreadedComposite.

This model is created by dragging two instances of ThreadedComposite, and then dropping instances of InteractiveShell onto the ThreadedComposite instances. The icons of the ThreadedComposite actors become like those of InteractiveShell, decorated with small green shadow.

In this model, the two instances of InteractiveShell execute asynchronously in threads that are separate from that of the DE director; when the model is running, there are three threads executing. The threads running the InteractiveShell actors block while waiting for user input. When the user types something and hits return,

the InteractiveShell actor produces an output, causing its containing ThreadedComposite to produce an output.

There are a number of subtle points about this model. The *delay* parameter of the ThreadedComposite actors is set to UNDEFINED, which instructs the ThreadedComposite actors to assign the current model time, whatever that time may be, to their output event time stamps. The time stamps of the outputs are therefore *nondeterminate*.

If instead we had set this parameter to some number  $\tau$ , then the output events would be assigned model time  $t + \tau$ , where  $t$  is the model of time of the triggering input event. This makes the output time stamps determinate. However, it also limits concurrency. When *delay* is set to a number  $\tau$ , the model will block when current model time reaches  $t + \tau$ . Otherwise, the ThreadedComposite would attempt to produce output events with time stamps in the past.

Another subtle effect is that the *synchronizeToRealTime* parameter of the director is set to true. This ensures that “current model time” advances no faster than real time. Thus, in the output traces shown in Figure 7.16, the reported times can be interpreted as a measure of the elapsed time in seconds from the start of execution until the user typed something. This gives the time stamps a physical meaning. It also justifies the nondeterminacy in the model, since the time at which a user types something is certainly nondeterminate (it is not specified by the model).

A third subtlety is that the *stopWhenQueueIsEmpty* parameter of the director is set to false. By default, the DE director will stop executing when there are no more events to process. But in this model, events can still appear later on, because the user types something. Thus, we do not want the model to stop when the queue becomes empty.

The ThreadedComposite actor offers a mechanism for executing models concurrently in multiple threads, but the mechanism is much more determinate and controllable than using threads directly, which is fraught with difficulties (Lee, 2006). The actor contained by a ThreadedComposite need not be an atomic actor; it can be an arbitrarily complex *composite actor*. The fact that the contained actor executes in a separate thread enables the use of actors that may get stalled waiting for I/O, and it also enables parallel execution on multicore machines for improved performance. The subtleties of this actor and further details on its usage are described by Lee (2008b).

### 7.3.4 Scheduling Limitations

As of this writing, the DE director in Ptolemy II implements an approximation of the semantics described by [Lee and Zheng \(2007\)](#). In particular, the current implementation is not able to execute all models that can, in theory, be executed by an exact implementation of the semantics.

**Example 7.14:** Consider the model shown in Figure 7.17. This model has an [opaque](#) composite actor in a feedback loop. The *clock* output of the composite actor does not depend on the input. Hence, at any given time stamp, the composite actor should be able to produce an event on the *clock* output without knowing whether an event is present on the input port. However, attempting to execute this model results in an exception:

```
IllegalActionException: Found a zero delay loop containing
OpaqueComposite
in FixedPointLimitation
```

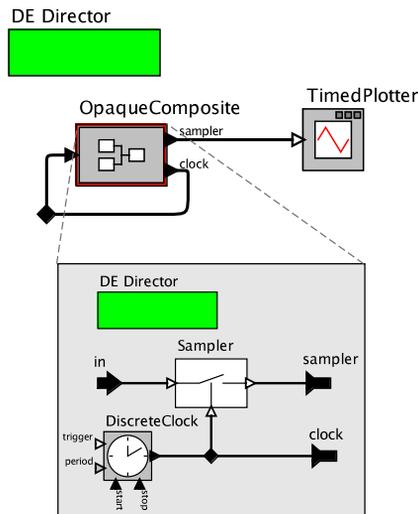


Figure 7.17: A model that should, in theory, be executable, but is not executable in the current implementation of DE.

In the current implementation, at each time stamp (model time and microstep), the DE director will fire an actor at most once, and when it fires that actor, it guarantees that all input events at that time stamp are available. As a consequence, the composite actor in Figure 7.17 can never be fired, because the director cannot ensure that an input event is present or absent at the current time stamp until it has already fired the composite actor once at that time stamp. Note that this problem cannot be corrected by putting a zero-delay [TimeDelay](#) actor in the feedback loop (see Exercise 1). It *can* be corrected by using a director with a [fixed point](#) semantics, as shown in Example `refexample:FixedPointNoLimitation`.

## 7.4 Zeno Models

It is possible to create DE models where time fails to advance, as illustrated by the example below.

**Example 7.15:** Suppose that if in Figure 7.11 we were to omit the `BooleanSwitch` and unconditionally feed back the tokens. Then time would never advance; only the microsteps would advance.

A model where model time stops advancing and only the microsteps advance is called a **chattering Zeno** model. Microsteps are implemented by the DE director as a Java `int`, so the increment of the microstep will eventually overflow, causing the director to report an exception.

It is also possible to create models, called **Zeno** models, where the time advances, but will not advance past some finite value.

**Example 7.16:** An example of a Zeno model is shown in Figure 7.18. This model triggers a feedback loop using a [SingleEvent](#) actor (see sidebar on page 241). The feedback loop contains a [TimeDelay](#) actor (see sidebar on page 243) whose delay value is given by  $1/n^2$ , where  $n$  begins at  $n = 1$  and is incremented each time a token cycles around the loop. The delay, therefore, approaches zero, and this model can produce an infinite number of events before time 2.0.

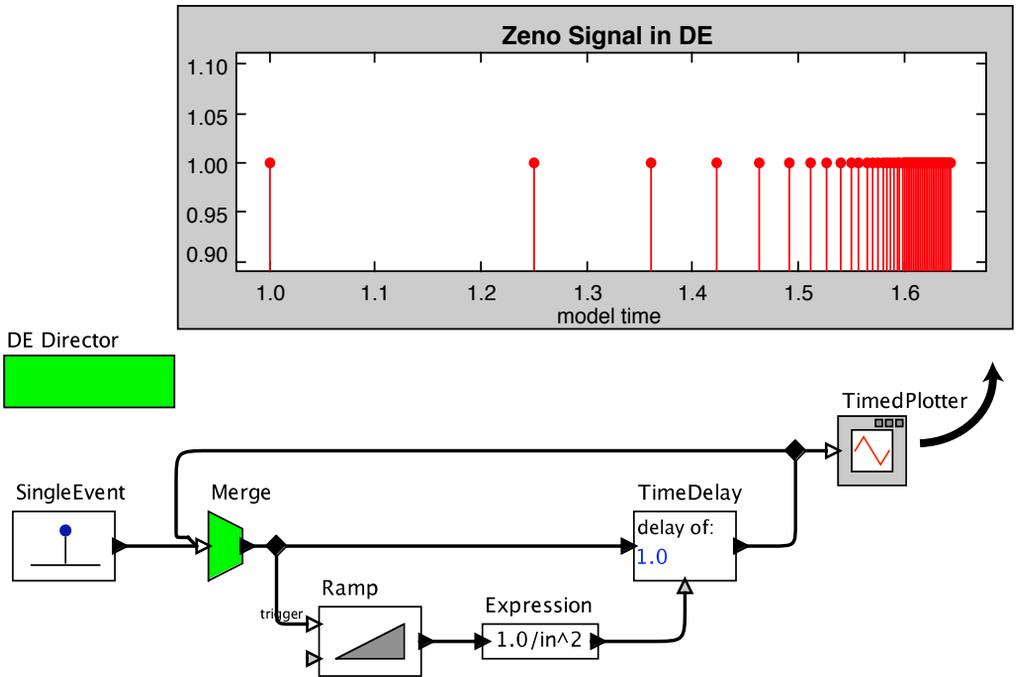


Figure 7.18: Example of a Zeno model, where the model time fails to advance. [\[online\]](#)

In DE, because time has finite precision (see Section 1.7.3), eventually a Zeno model becomes a chattering Zeno model. When the time increment falls below the [time resolution](#), then time stops advancing. The particular model in the previous example, however, fails before that happens because when  $n$  becomes sufficiently large, floating point errors in calculating  $1/n^2$  (oddly) yield a negative number, causing the TimeDelay actor to report an exception.

## 7.5 Using DE with other Models of Computation

DE models can be usefully combined with other models of computation. Here we highlight some of the most useful combinations.

### 7.5.1 State Machines and DE

As shown in Figure 7.1, an actor in a DE model may be defined by a state machine. Such a state machine may be capable of initiating a feedback loop without requiring an external stimulus event, as illustrated by the next example.

**Example 7.17:** Figure 7.19 shows a simple DE model containing a single **FSMAc**tor. In this example, the initial state has an enabled transition (with guard `true`), which causes the FSMActor to fire at the execution start time and produce an output. This output, in turn, initiates the feedback loop. This actor does not need an input event to trigger the first firing and initiate feedback.

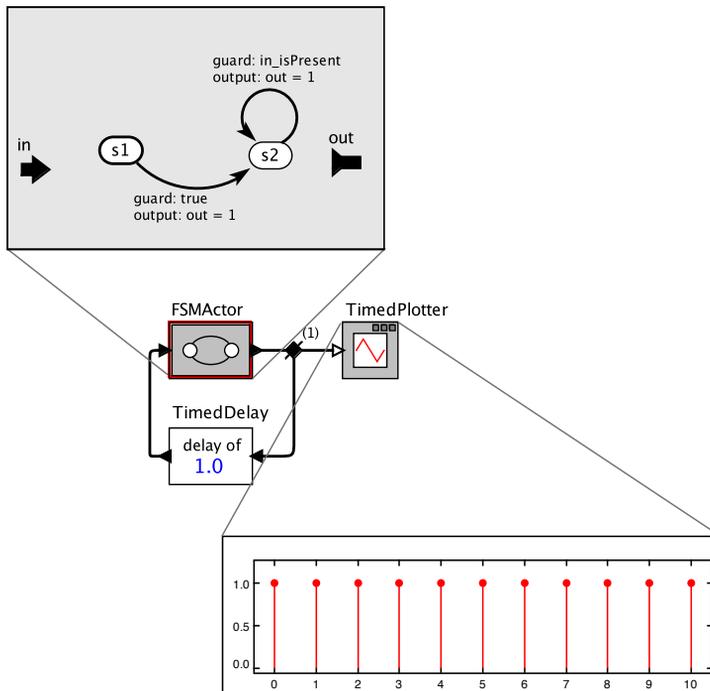


Figure 7.19: A simple DE model containing an FSM. [\[online\]](#)

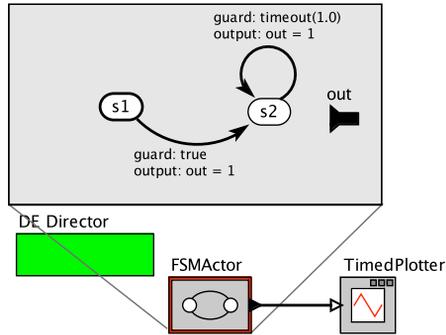


Figure 7.20: A DE model with identical behavior to that in Figure 7.19, but containing an FSM that controls timing with the timeout function. [online]

After the first firing, the actor is in state  $s_2$ , where the outgoing transition has guard `in_isPresent`. Hence, all subsequent firings require an input event. The net result is an unbounded sequence of events one time unit apart, as shown in the plot.

The same effect can be achieved by an FSM that directly controls timing using the `timeout` function (see Table 6.2), as shown in Figure 7.20.

## 7.5.2 Using DE with Dataflow

It is possible to use a dataflow director within a DE model. The SDF director, for example, can be quite useful within DE models. If a composite actor in a DE model contains an SDF director, then the internal dataflow model will execute one **complete iteration** each time an event is received on an input port. This approach can be useful when the overall DE system model includes complex computations that do not involve timed events and are conveniently described by a dataflow model.

The DDF director can also be used within DE models, although as discussed in Section 3.2, controlling the definition of an iteration in DDF is more difficult than with SDF. Using the dataflow PN (Process Network) director within DE rarely makes sense, because, as discussed in Section 4.1, it is quite difficult to deterministically bound an iteration.

The dataflow directors described in Chapter 3 are generally untimed, except that (as described earlier) the SDF director has a *period* parameter. This parameter can be useful within a DE model. If the parameter is set to something other than zero, then the SDF submodel will execute periodically within the DE model, regardless of whether any input events are provided. This approach can be used to design [clock](#) actors with much more complicated output patterns than are easily specified using [DiscreteClock](#). Note, however, that the submodel will execute *only* at multiples of the *period*, and not whenever an input event is provided. If at some multiple of the period there are insufficient inputs to execute a complete iteration, then the SDF model will not fire at that time. Moreover, if inputs are provided faster than the SDF submodel consumes them, then they will queue up, possibly eventually exhausting available memory. See [Exercise 2](#) for an example.

It is rarely useful to put a DE model inside an SDF model (or any other dataflow model). The DE submodel will expect to be fired at model times determined by its internal actors, but the SDF model will only fire at multiples of the *period*. Hence, this combination will typically trigger an exception similar to the following:

```
IllegalActionException: SDF Director is unable to fire CompositeActor
at the requested time: ... . It responds it will fire it at: ...
in .DEwithinSDF.CompositeActor.DE Director
```

It is possible, however, to put a DE submodel within an SDF model if the SDF model is itself nested within a DE model, and the *period* parameter of the SDF model is set to zero. In this case, the SDF director will delegate firing requests to the higher-level DE director, ignoring time advancements itself.

## 7.6 Wireless and Sensor Network Systems

The **wireless domain** builds on the DE domain to support modeling of wireless networks. In the wireless domain, channel models mediate communication between actors, and the [visual syntax](#) (i.e., the graphical representation of the model) does not require wiring between components. The visual representation of models in the wireless domain is more important than in other Ptolemy II domains because the location of icons forms a two-dimensional map of the wireless system being modeled. The positions of icons on the screen, the distance between them and the objects in between them, affect their communication.

**Example 7.18:** The top level of the model in Figure 7.21 contains a **WirelessDirector**, two instances of **WirelessComposite**, and a **DelayChannel**. **WirelessComposite1** has an output port, and **WirelessComposite2** has an input port. Although these ports are not directly connected, they do communicate with each other. Each port has a parameter *outsideChannel* that names the wireless channel through which it communicates, which in this case is **DelayChannel**.

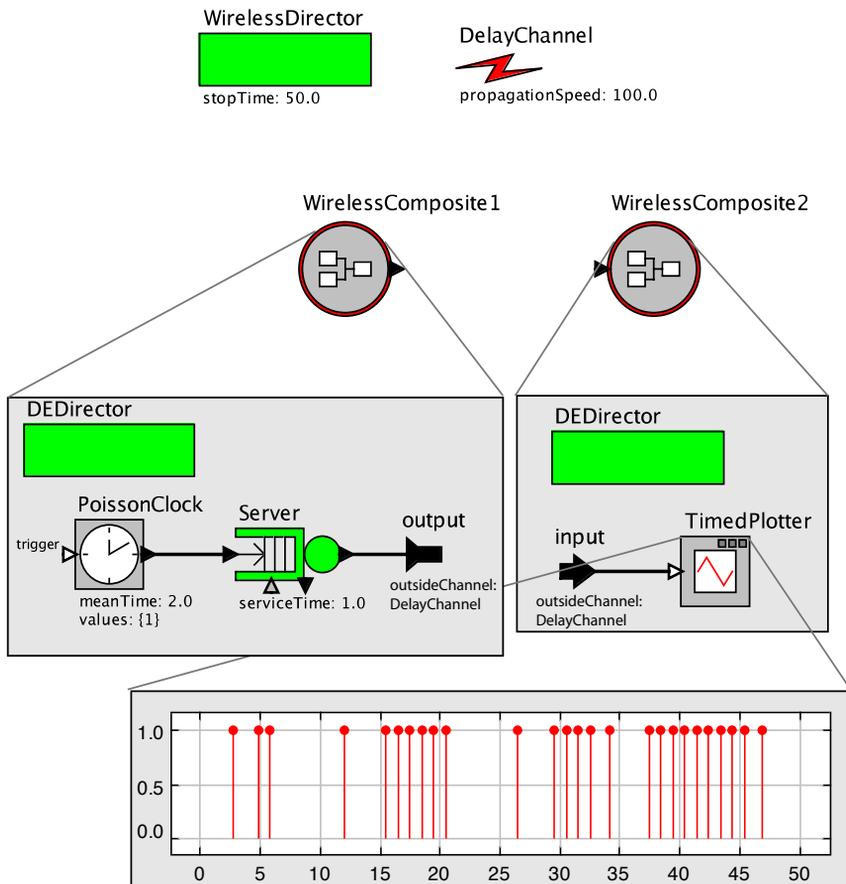


Figure 7.21: A model of a wireless system. [[online](#)]

The `DelayChannel` component models a wireless communication channel that, in this simple example, delays the message by an amount of time that is proportional to the distance between the two `WirelessComposite` icons in the model. The proportionality constant is given by the *propagationSpeed* parameter (in arbitrary units of distance/time). In this case, the icons are about 175 units apart, so a propagation speed of 100 translates into a delay of approximately 1.75 time units.

In this model, `WirelessComposite1` is a **sporadic source** of events. A sporadic source is a random source where there is a fixed lower bound on the time between events. In this case, the lower bound is enforced by the `Server` actor (see sidebar on page 246) and the randomness is provided by the `PoissonClock` actor.

`WirelessComposite2` in this model simply plots the received events as a function of time. The first event is sent by `WirelessComposite1` at time 1.0 and received by `WirelessComposite2` at approximately 2.75. The plot shows that no two events are more closely spaced than one time unit apart.

In addition to modeling channel delays, the wireless domain can model power loss, interference, noise, and occlusion. It can also model directional antenna gain and mobile transmitters and receivers. See [Baldwin et al. \(2004\)](#) and [Baldwin et al. \(2005\)](#) for details, and see the demos included in the Ptolemy II package for examples.

## 7.7 Summary

The DE domain provides a solid foundation for modeling discrete timed behavior. Mastering its use requires an understanding of the model of time, simultaneity, and feedback, each of which is covered in this chapter. A key characteristic of this domain is that execution is determinate even when events are simultaneous.

## Exercises

1. Consider the model in Figure 7.22. Unlike the model Figure 7.17, the DE director can execute this model because of the `TimeDelay` in the feedback loop.
  - (a) Explain why the model in Figure 7.22 produces no output events.
  - (b) Explain why the model in Figure 7.22 is not equivalent to the model in Figure 7.17, were the DE director able to execute it.
2. Consider the model in Figure 7.23. It has an SDF submodel within a DE model.
  - (a) Suppose the *period* parameter of the SDF director is 1.5 and the *period* of the `DiscreteClock` is 1.0. What output do you expect from the `CompositeActor`? Does this model execute with bounded memory?

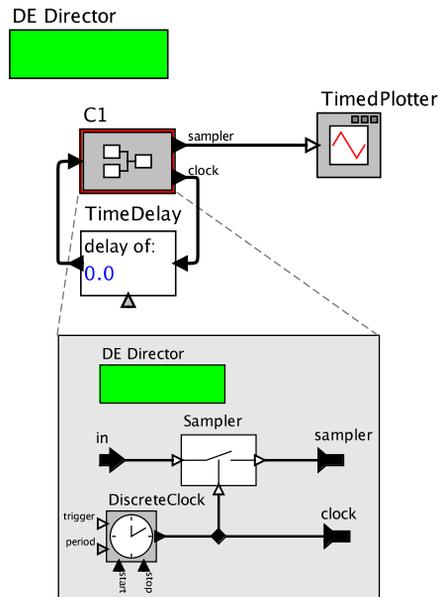


Figure 7.22: A model that is executable but is not equivalent to the model in Figure 7.17. [online]

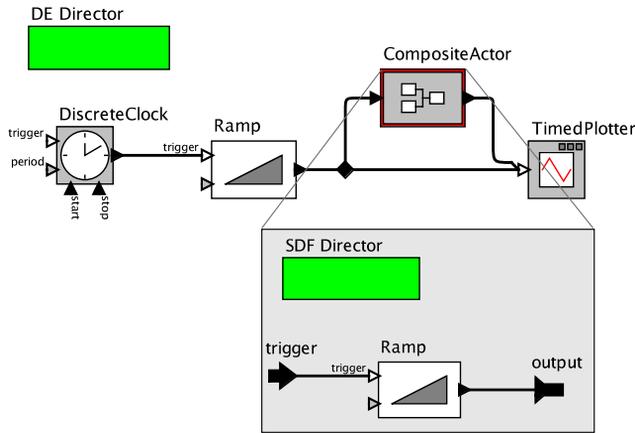


Figure 7.23: Simple example of an SDF submodel within a DE model. [\[online\]](#)

- (b) Find values of the *period* parameter of the SDF director and of the [Discrete-Clock](#) actor that will generate the plot in Figure 7.24. Explain why this output makes sense.
3. This problem explores some subtleties of combining FSMs with DE models. Construct a DE model consisting of a [PoissonClock](#) that triggers a [Ramp](#) that provides

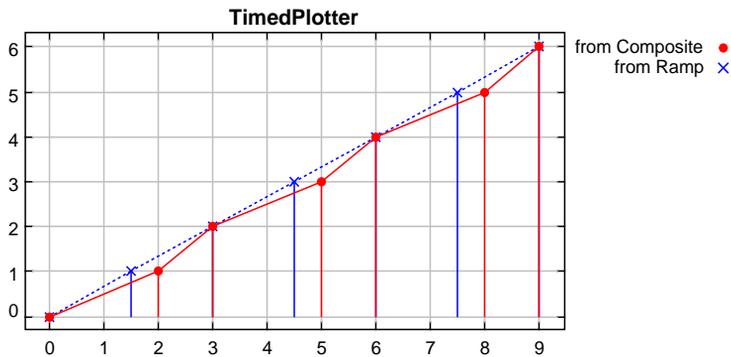


Figure 7.24: A plot that can be generated by the model in Figure 7.23.

input to an FSM (see Chapter 6). Set the *fireAtStart* parameter of the `PoissonClock` actor to `false`) so that it does not produce an output at time zero.

- (a) Construct an FSM that will produce an output at time zero even with no input event at time zero.
- (b) Modify your FSM so that, in addition, when it does receive an input event, it produces two outputs at the `model time` of the input event. The first such output should have the same value as the input event. The second such output should occur one `microstep` later and should have a value that is twice the value of the input event.

# 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 [SignalSource](#) 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

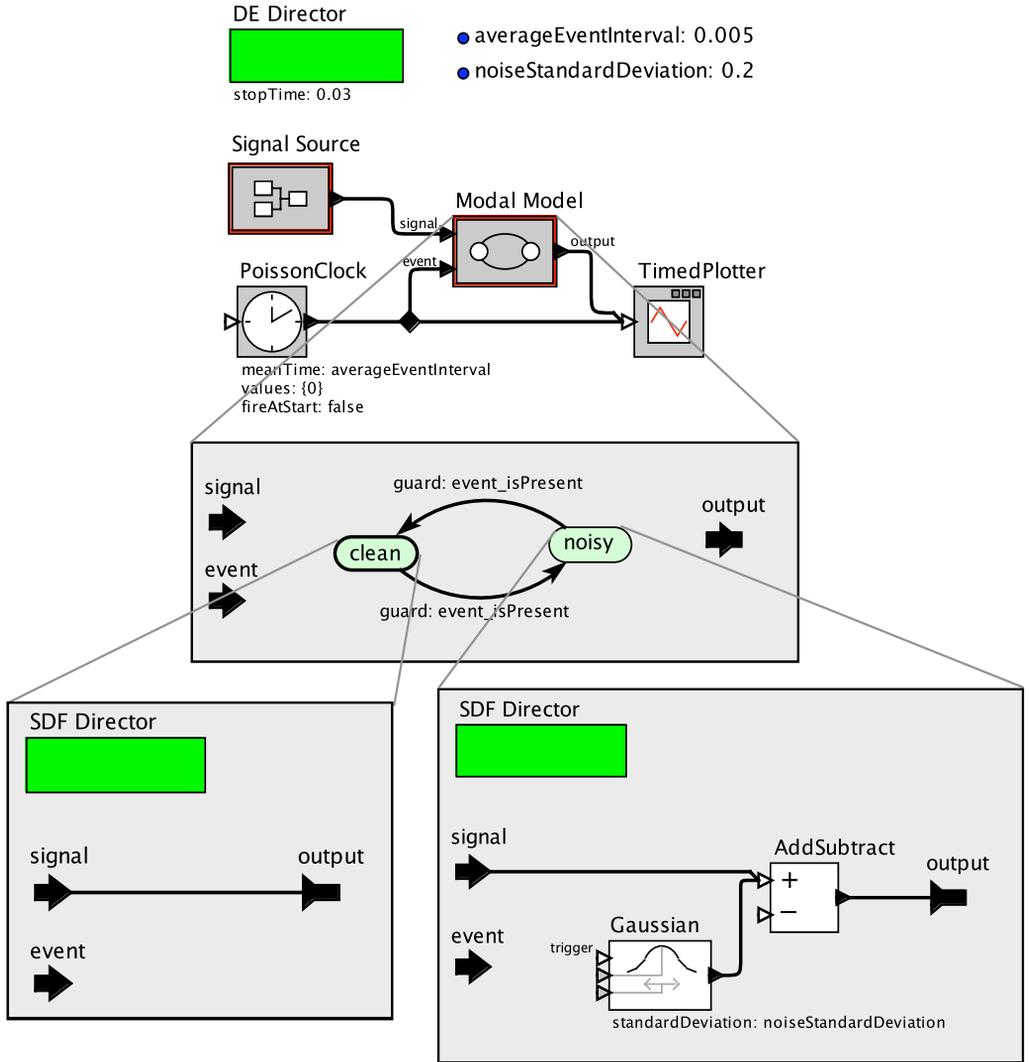


Figure 8.1: Simple modal model that has a normal (clean) operating mode, in which it passes inputs to the output unchanged, and a faulty mode, in which it adds Gaussian noise. It switches between these modes at random times determined by the PoissonClock actor. [\[online\]](#)

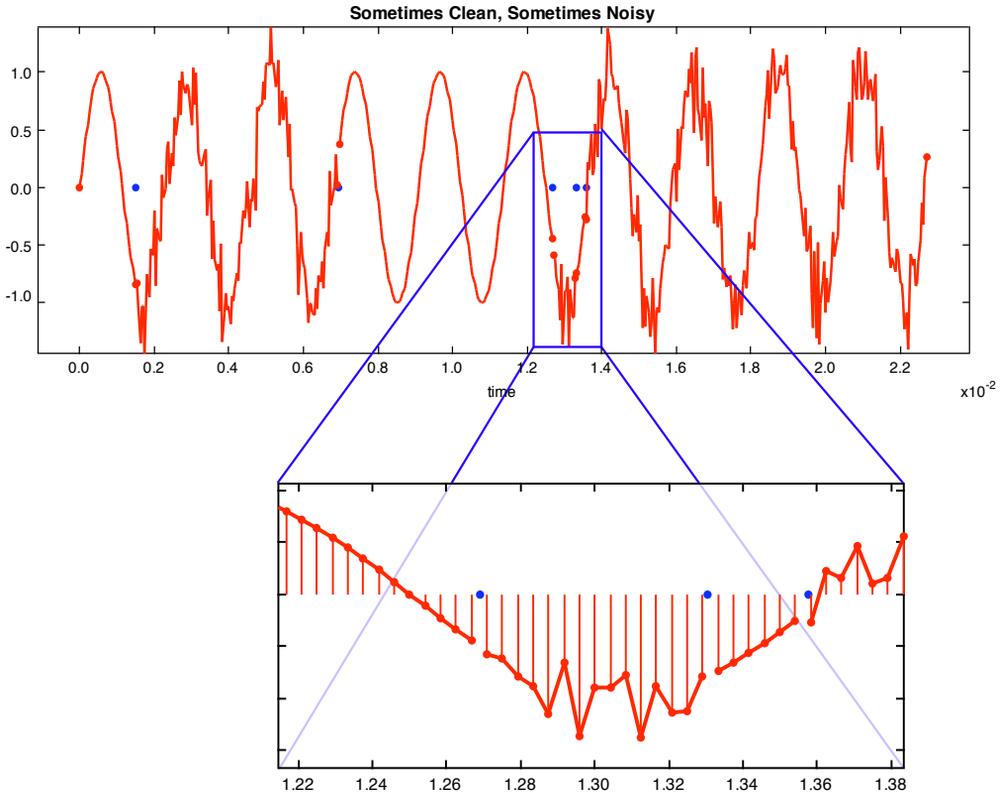


Figure 8.2: Plot generated by the model in Figure 8.1.

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.

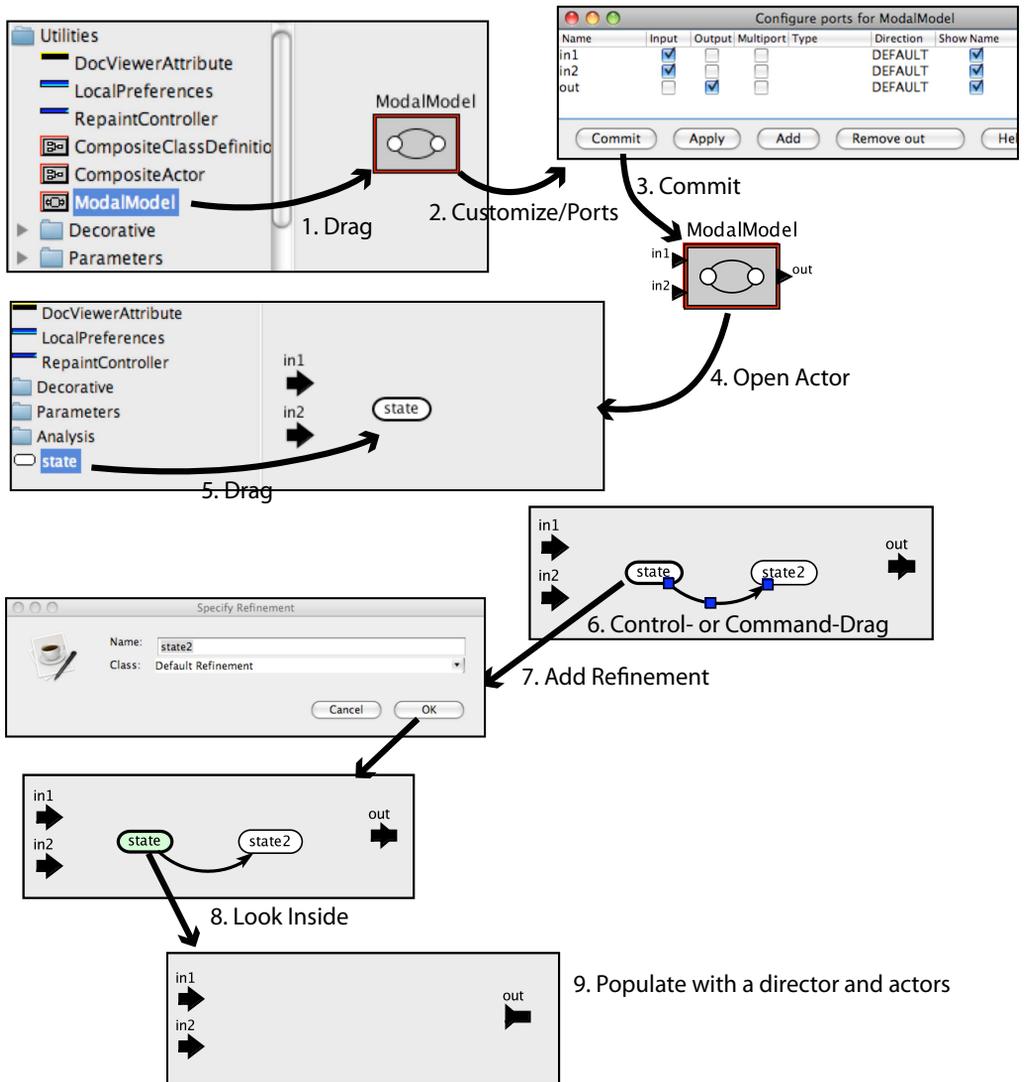


Figure 8.3: How to create modal models.

## 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.

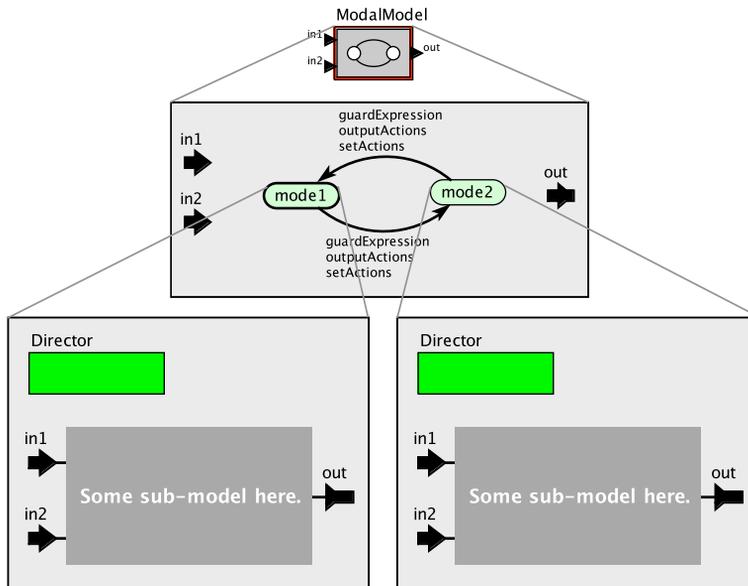


Figure 8.4: General pattern of a modal model with two modes, each with its own refinement.

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

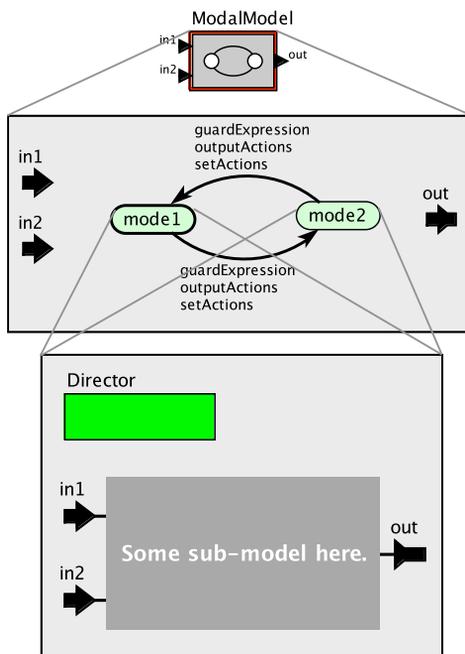


Figure 8.5: Variant of the pattern in Figure 8.4 where two modes share the same refinement.

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.

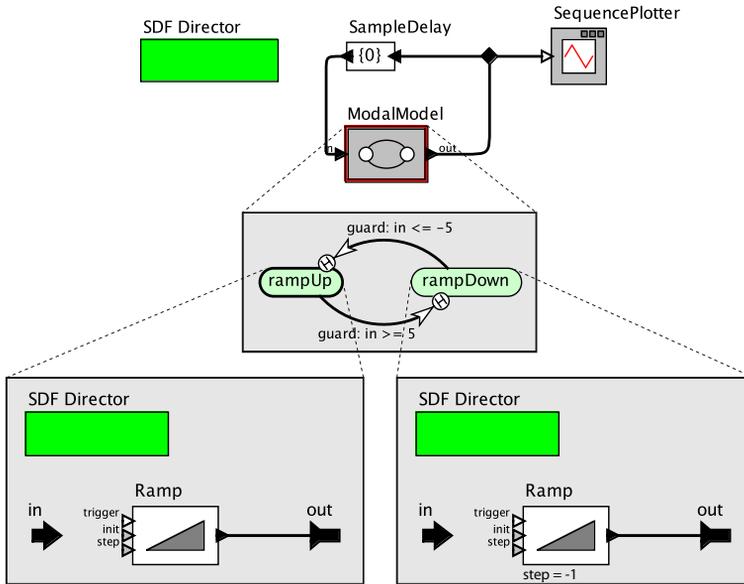


Figure 8.6: A modal model whose behavior depends on whether transitions are reset transitions or history transitions. [[online](#)]

## 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.

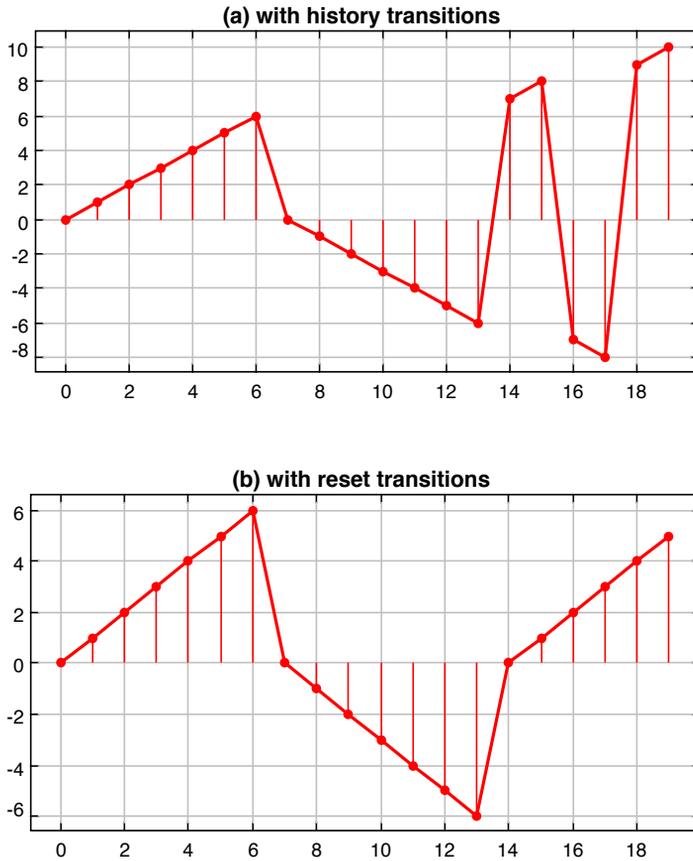


Figure 8.7: (a) The plot resulting from executing the model in Figure 8.6, which has history transitions. (b) The plot that would result from from changing the transitions to reset transitions.

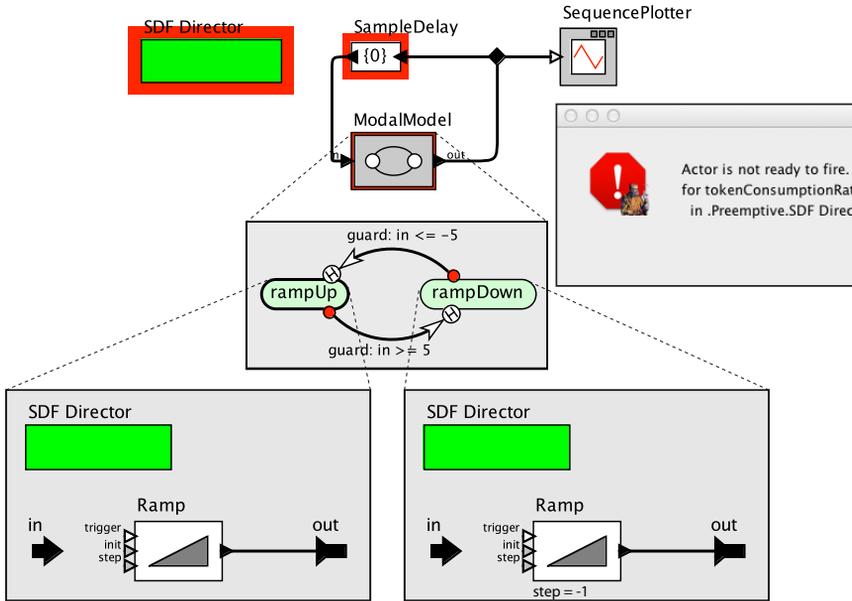
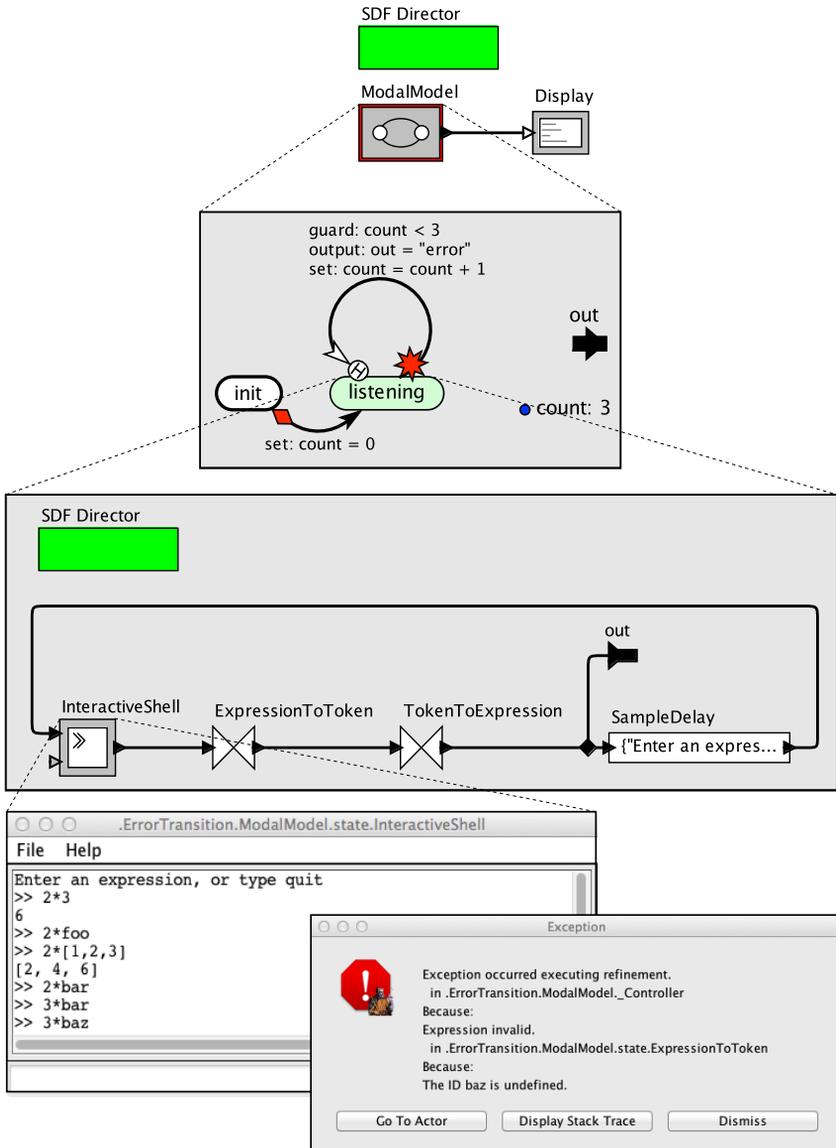


Figure 8.8: A modal model where preemptive transitions prevent the refinements from producing outputs that are expected by the SDF director. [online]

### 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*f00`.” This is invalid because there is no variable named “`f00`” 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

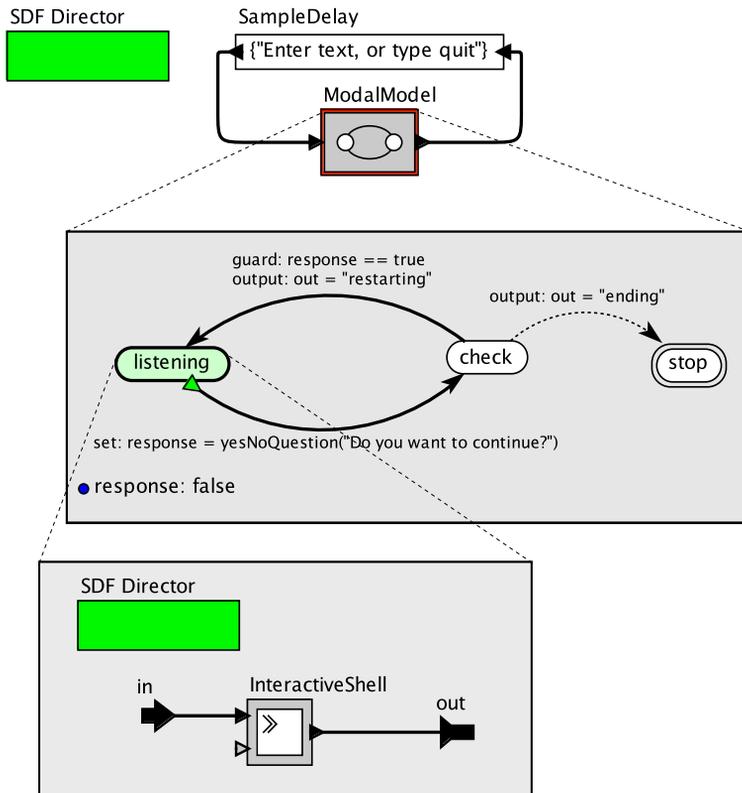


Figure 8.10: A modal model with a termination transition. [[online](#)]

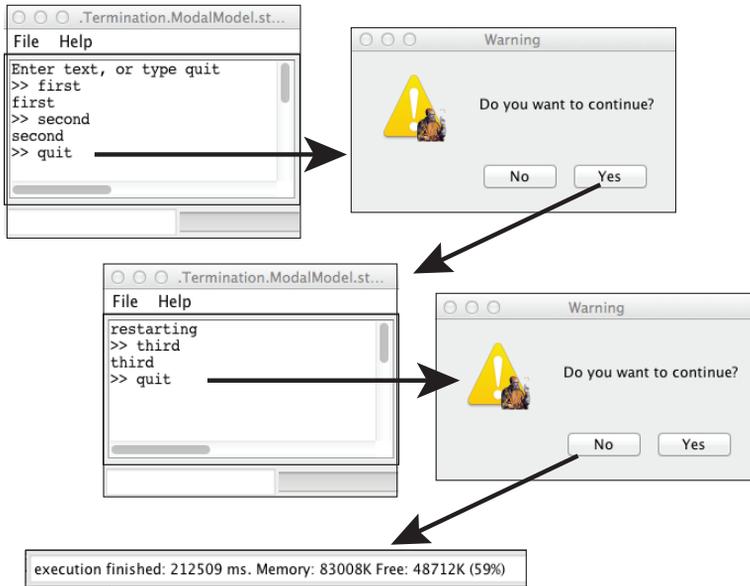


Figure 8.11: An execution of the model in Figure 8.10.

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.

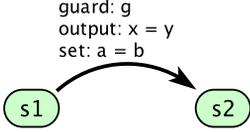
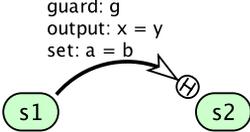
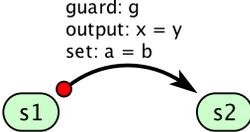
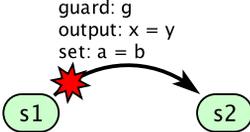
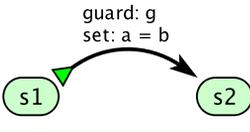
notation	description
	<p>An <b>ordinary transition</b>. Upon firing, the refinement of the source state is fired first, and then if the guard <math>g</math> is <code>true</code> (or if no guard is specified), then the FSM will choose the transition. It will produce the value <math>y</math> on output 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. There could even be

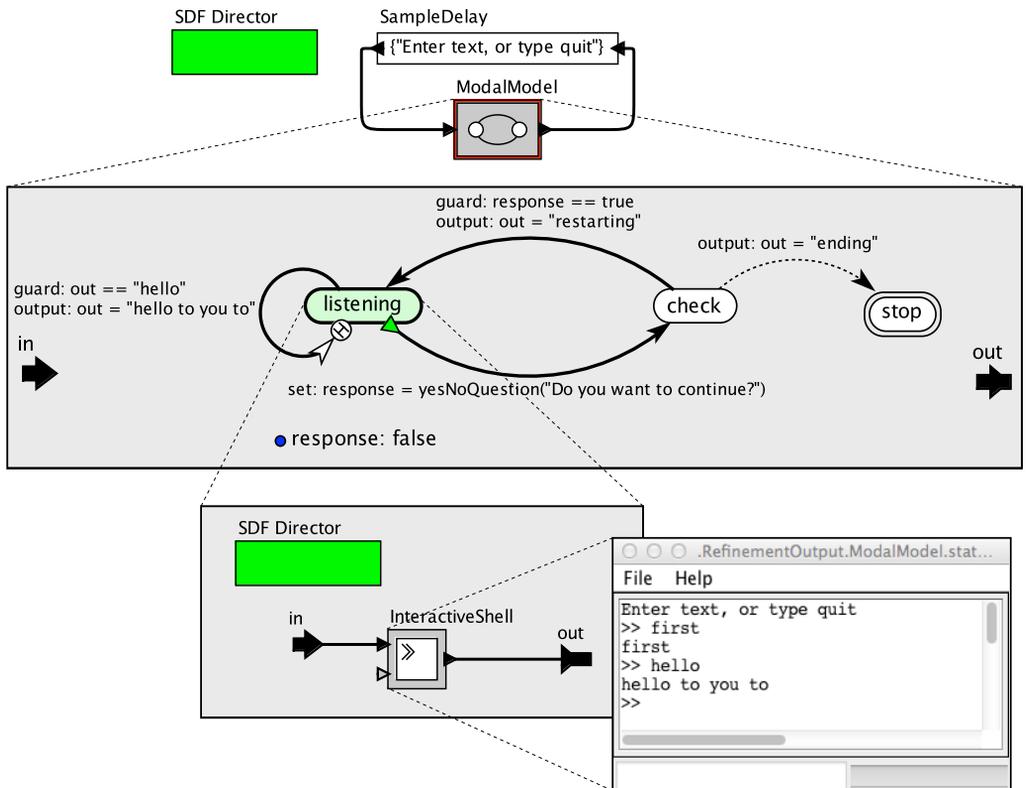


Figure 8.12: A variant of the model in Figure 8.10 that includes a guard that references an output from a refinement. [\[online\]](#)

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.

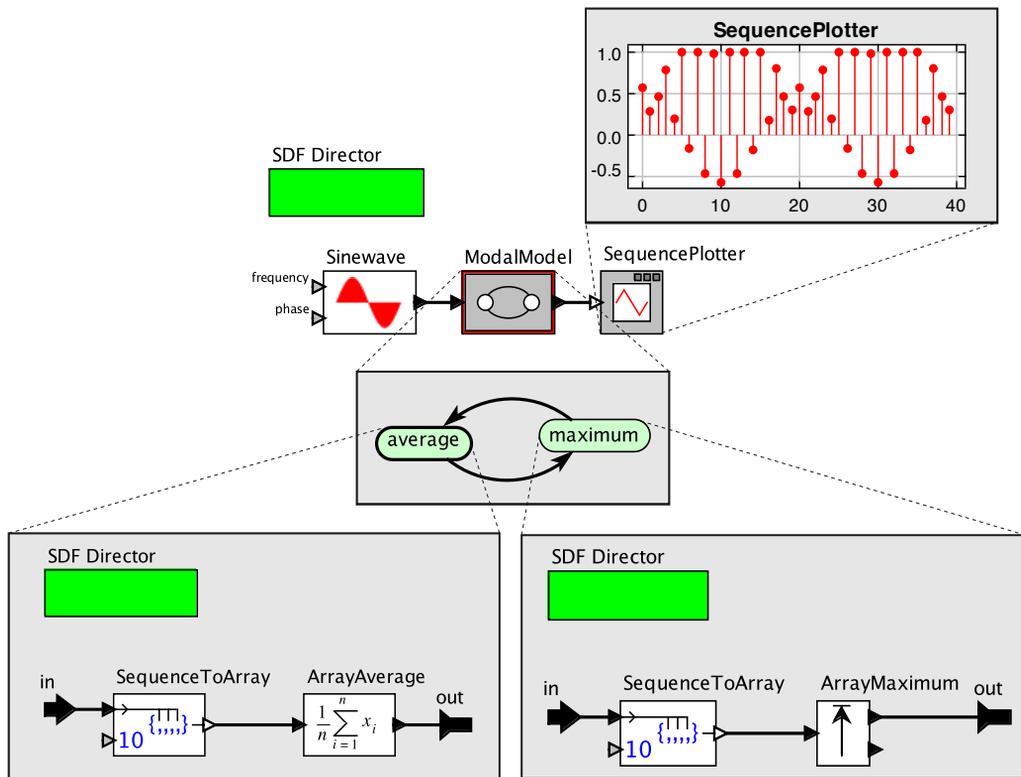


Figure 8.13: An SDF model where the ModalModel requires more than one token on its input in order to fire. [\[online\]](#)

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

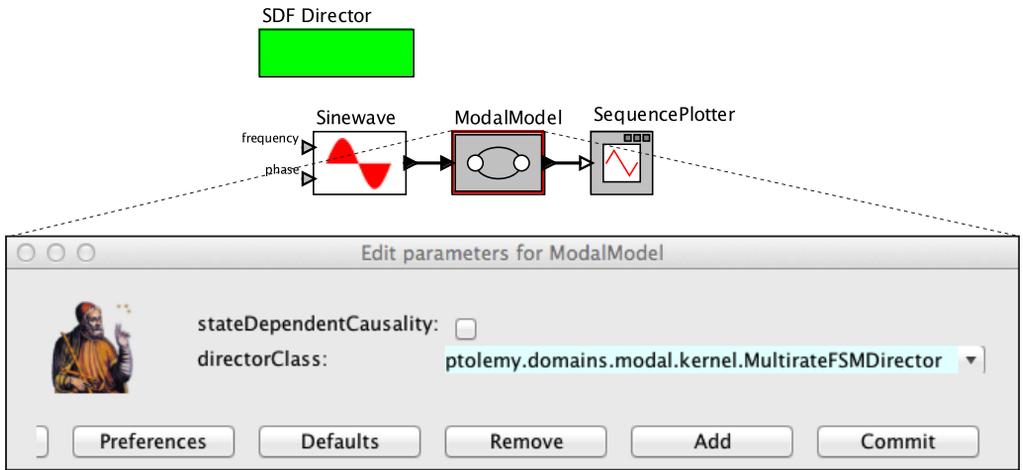


Figure 8.14: In order for the ModalModel to become an SDF actor with non-unit consumption and production on its inputs and outputs, it has to use the specialized MultirateFSMDirector.

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.

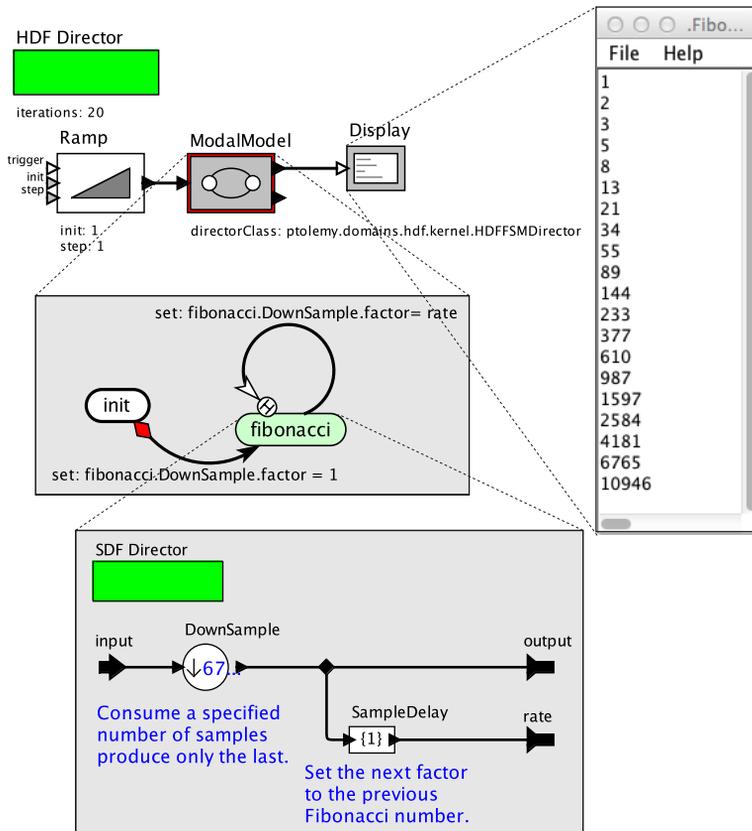


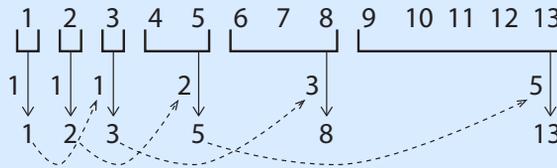
Figure 8.15: A model that calculates the Fibonacci sequence using a heterochronous dataflow model. This model is due to Ye Zhou. [\[online\]](#)

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.

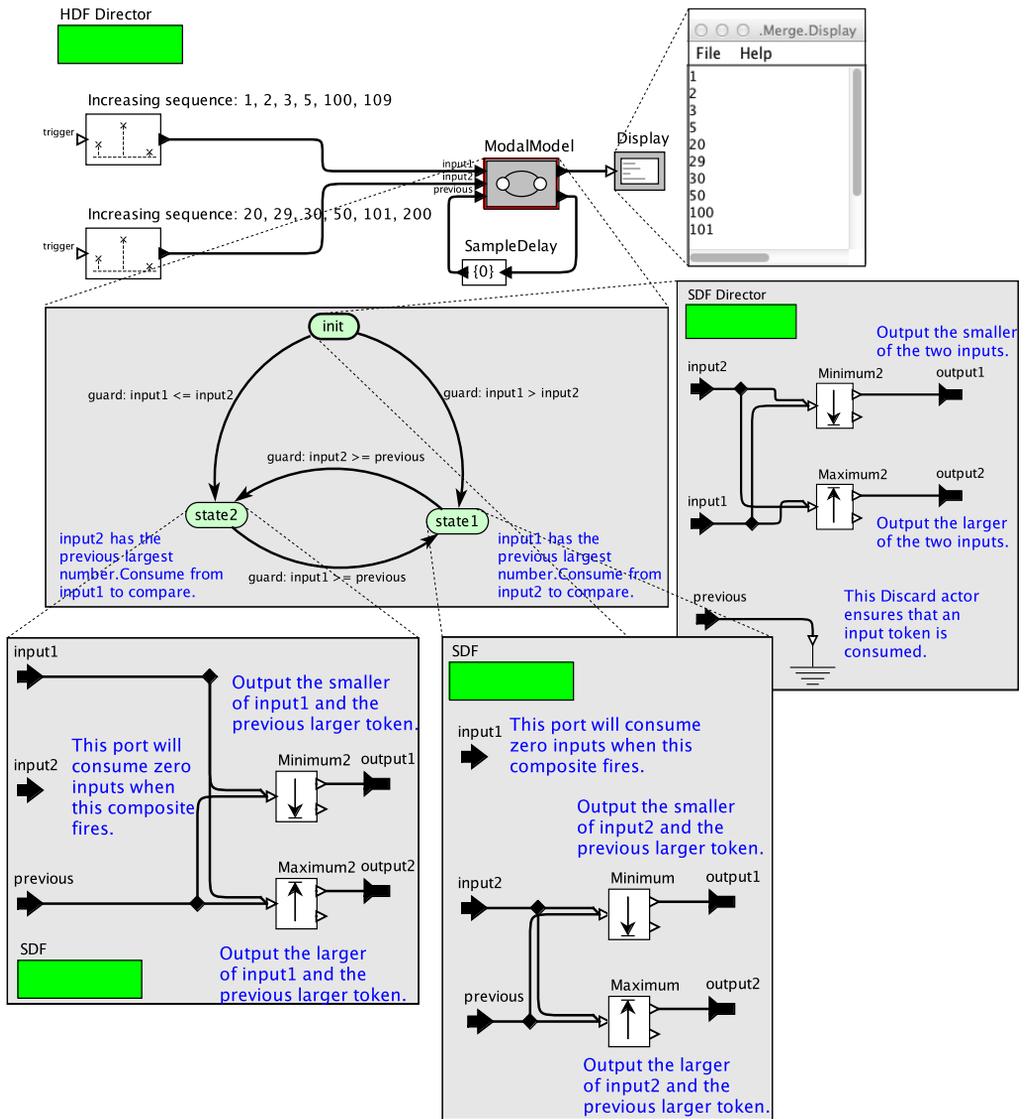


Figure 8.16: A heterochronous dataflow model that merges two numerically increasing streams into one numerically increasing stream (due to Ye Zhou and Brian Vogel). [\[online\]](#)

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

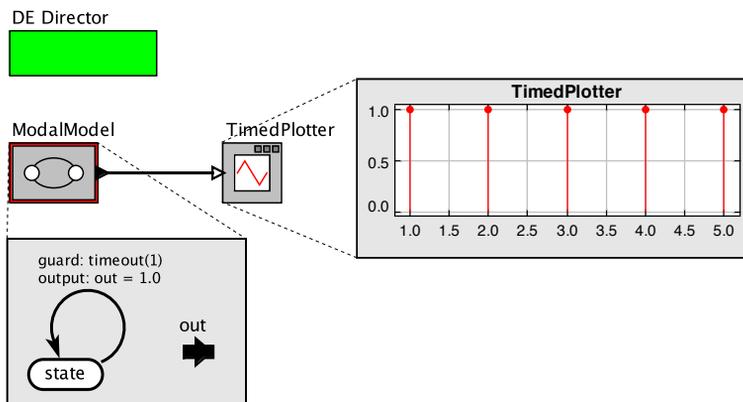


Figure 8.17: A spontaneous FSM, which produces output events that are not triggered by input events. [[online](#)]

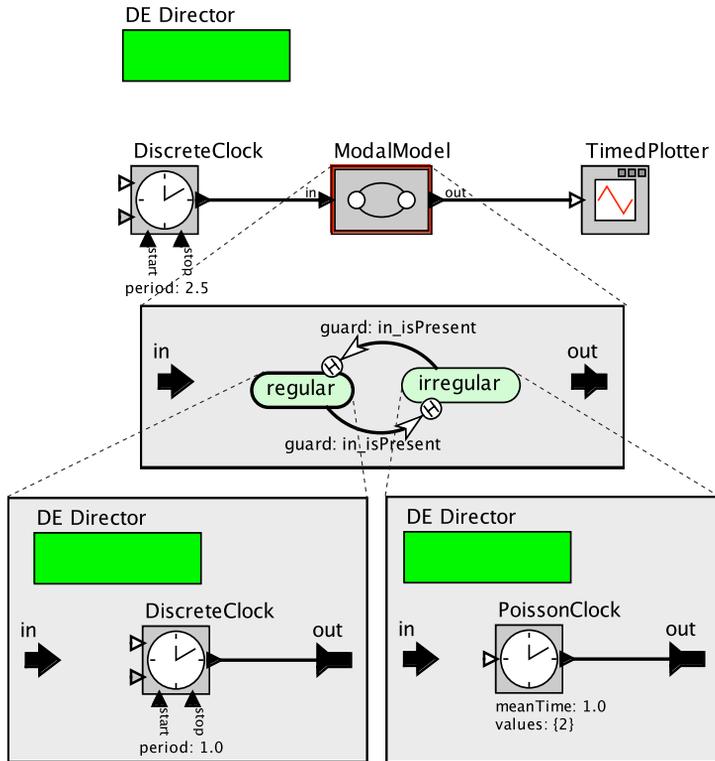


Figure 8.18: Another spontaneous modal model, which produces output events that are not triggered by input events. [\[online\]](#)

*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

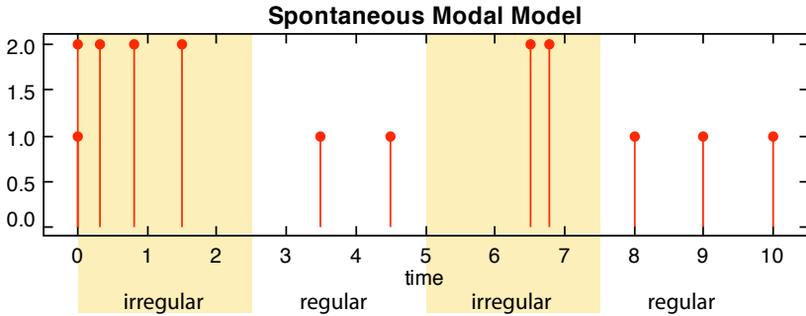


Figure 8.19: A plot of the output from one run of the model in Figure 8.18.

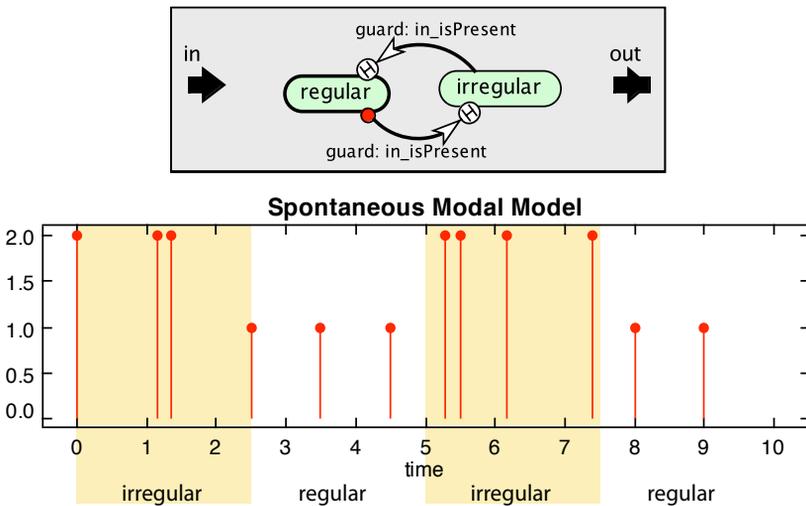


Figure 8.20: A variant of Figure 8.18 where a preemptive transition prevents the initial firing of the DiscreteClock.

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.

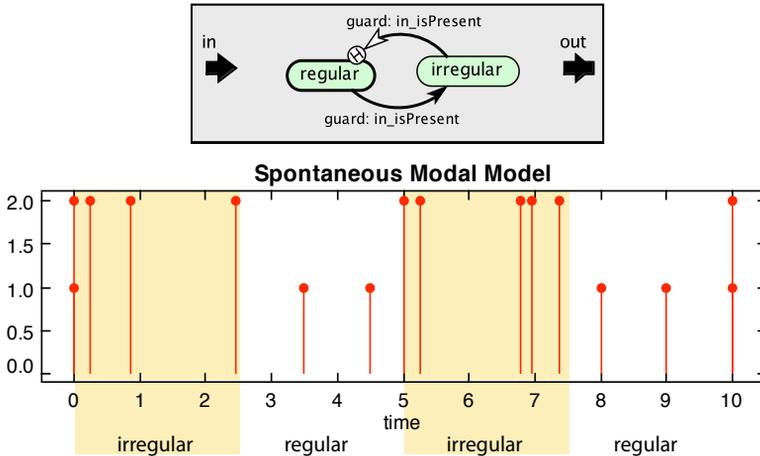


Figure 8.21: A variant of Figure 8.18 in which a reset transition causes the PoissonClock to produce events when the *irregular* mode is reactivated.

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.

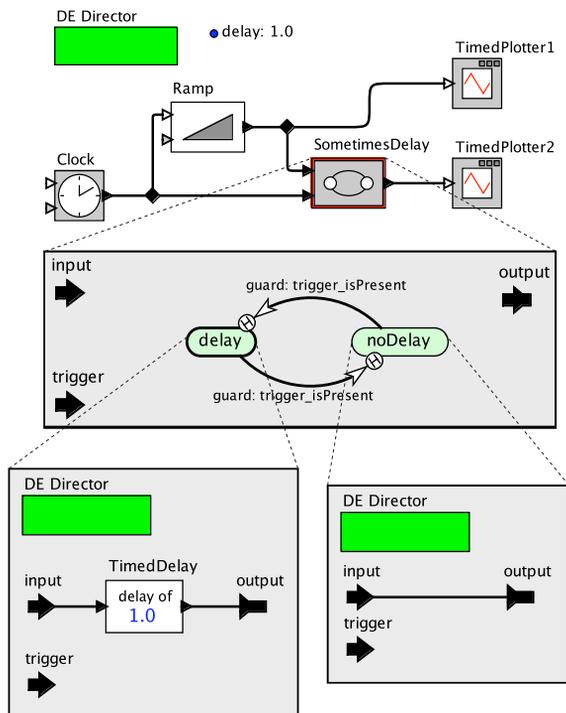


Figure 8.22: A modal model that switches between delaying the input by one time unit and not delaying it. [\[online\]](#)

### 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?

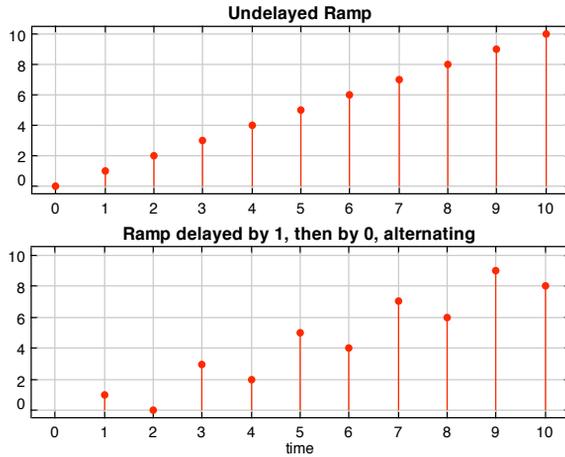


Figure 8.23: The result of executing the model in Figure 8.22.

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.

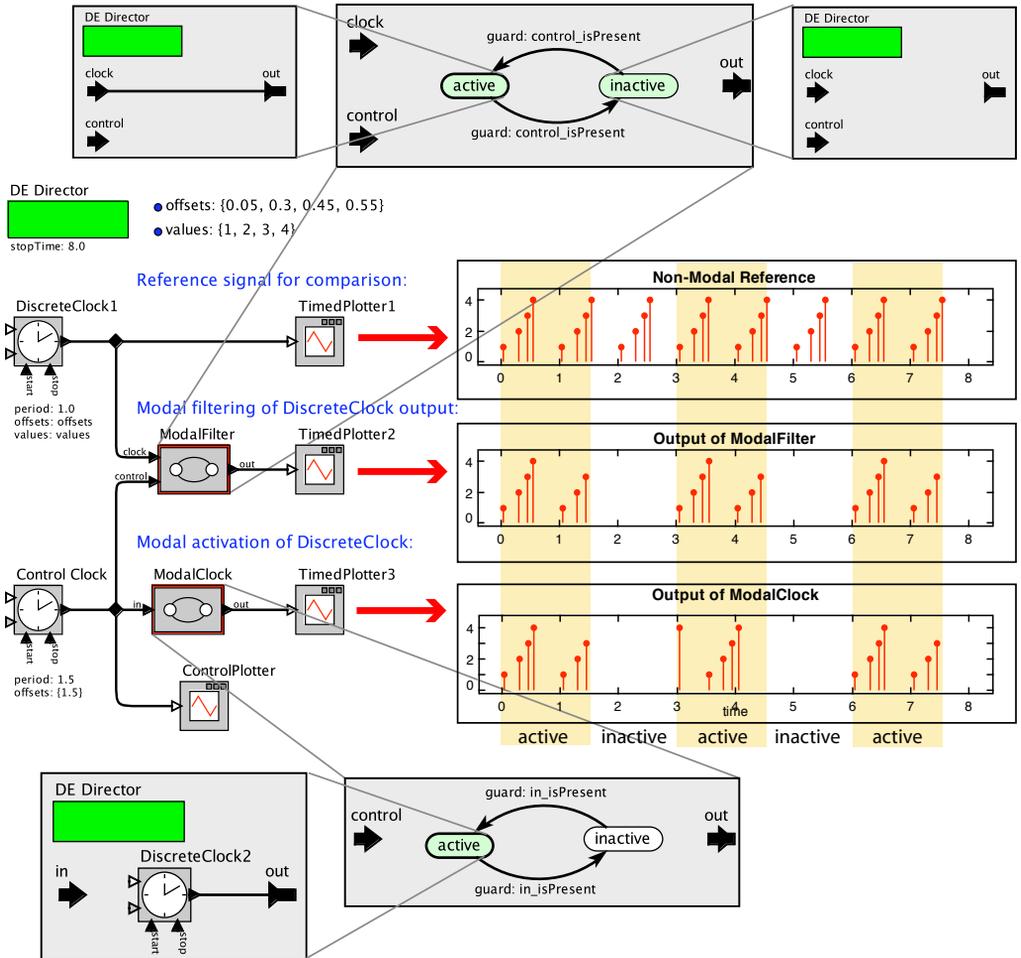


Figure 8.24: A model that illustrates that putting a timed actor such as Discrete-Clock inside a modal model is not the same as switching between observing and not observing its output. [online]

### 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.

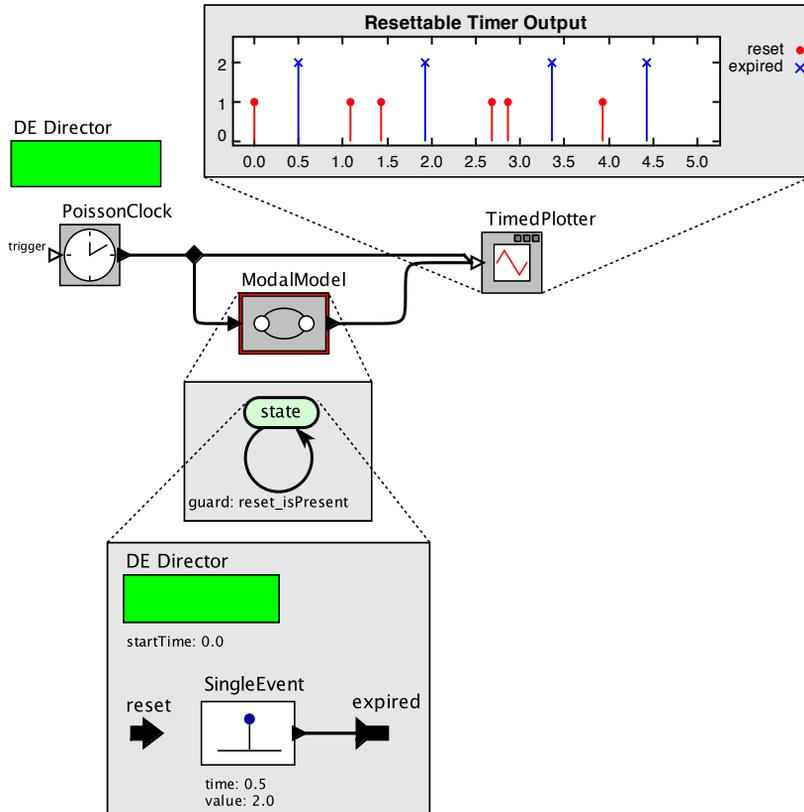


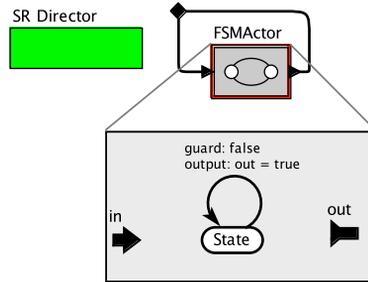
Figure 8.25: A resettable timer implemented by using a reset transition to restart a submodel at time zero. [[online](#)]

## 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.

# Continuous-Time Models

Janette Cardoso, Edward A. Lee, Jie Liu, and Haiyang Zheng

## Contents

---

<b>9.1</b>	<b>Ordinary Differential Equations</b>	<b>316</b>
9.1.1	Integrator	316
9.1.2	Transfer Functions	320
9.1.3	Solvers	322
	<i>Sidebar: Continuous-Time Signal Generators</i>	326
	<i>Sidebar: Actors for Modeling Dynamics</i>	327
	<i>Sidebar: Runge-Kutta Methods</i>	328
	<i>Sidebar: Runge-Kutta Methods - Continued</i>	329
<b>9.2</b>	<b>Mixed Discrete and Continuous Systems</b>	<b>330</b>
9.2.1	Piecewise Continuous Signals	330
	<i>Sidebar: Probing Further: Discrete Sets</i>	334
9.2.2	Discrete-Event Signals in the Continuous Domain	335
9.2.3	Resetting Integrators at Discrete Times	336
9.2.4	Dirac Delta Functions	338
	<i>Sidebar: Continuous Signals from Discrete Events</i>	338
	<i>Sidebar: Generating Discrete Events</i>	341
9.2.5	Interoperating with DE	343
9.2.6	Fixed-Point Semantics	345
<b>9.3</b>	<b>Hybrid Systems and Modal Models</b>	<b>346</b>
9.3.1	Hybrid Systems and Discontinuous Signals	351
<b>9.4</b>	<b>Summary</b>	<b>353</b>
	<b>Exercises</b>	<b>354</b>

---

Continuous-time models are realized using the Ptolemy II **continuous-time (CT)** domain (also called the **Continuous** domain), which models physical processes. This domain is particularly useful for **cyber-physical systems**, which are characterized by their mixture of computational and physical processes.

The CT domain conceptually models time as a continuum. It exploits the **superdense time** model in Ptolemy II to process signals with discontinuities, signals that mix discrete and continuous portions, and purely discrete signals. The resulting models can be combined hierarchically with **discrete event** models, and **modal models** can be used to develop **hybrid systems**.

## 9.1 Ordinary Differential Equations

The continuous dynamics of physical processes are represented using **ordinary differential equations (ODEs)**, which are differential equations over a time variable. The Ptolemy II models of continuous-time systems are similar to those used in Simulink (from The MathWorks), but Ptolemy's use of superdense time provides cleaner modeling of mixed signal and hybrid systems (Lee and Zheng, 2007). This section focuses on how continuous dynamics are specified in a Ptolemy II model and how the Continuous director executes the resulting models.

### 9.1.1 Integrator

In Ptolemy II, differential equations are represented using **Integrator** actors in feedback loops. At time  $t$ , the output of an Integrator actor is given by

$$x(t) = x_0 + \int_{t_0}^t \dot{x}(\tau) d\tau, \quad (9.1)$$

where  $x_0$  is the *initialState* of the Integrator,  $t_0$  is the *startTime* of the director, and  $\dot{x}$  is the input signal to the Integrator. Note that since the output  $x$  of the Integrator is the integral of its input  $\dot{x}$ , then at any given time, the input  $\dot{x}$  is the derivative of the output  $x$ ,

$$\dot{x}(t) = \frac{d}{dt}x(t). \quad (9.2)$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use. ODEs can be represented using Integrator actors, as illustrated by the following example.

**Example 9.1:** The well-known **Lorenz attractor** is a non-linear feedback system that exhibits a style of chaotic behavior known as a **strange attractor**. The model in Figure 9.1 is a block diagram representation of the set of nonlinear ODEs that govern the behavior of this system. Let the output of the top integrator be  $x_1$ , the output of the middle integrator be  $x_2$ , and the output of the bottom integrator be  $x_3$ .

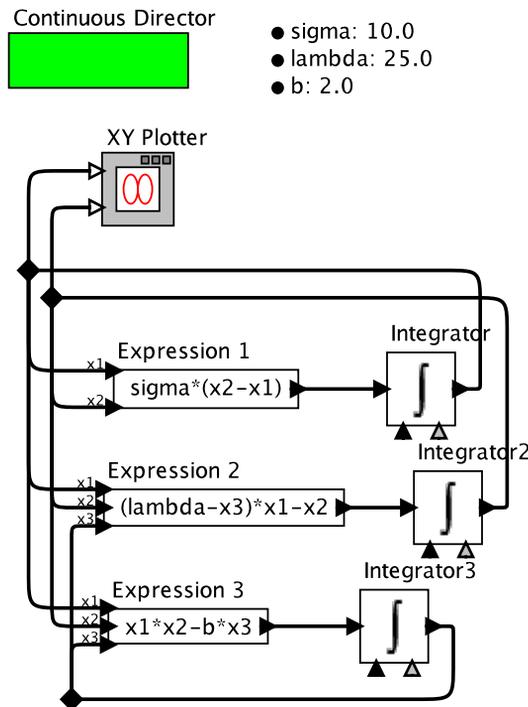


Figure 9.1: A model describing a set of nonlinear ordinary differential equations. [\[online\]](#)

Then the equations described by Figure 9.1 are

$$\begin{aligned}\dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t) \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)\end{aligned}\tag{9.3}$$

where  $\sigma$ ,  $\lambda$ , and  $b$  are real-valued constants. For each equation, the expression on the right side of the equals sign is implemented by an **Expression** actor, whose icon shows the expression. Each expression refers to parameters (such as *lambda* for  $\lambda$  and *sigma* for  $\sigma$ ) and input ports of the actor (such as *x1* for  $x_1$  and *x2* for  $x_2$ ). The expression in each **Expression** actor can be edited by double clicking on the actor, and the parameter values can be edited by double clicking on the parameters, which are shown next to bullets at the top.

The three integrators specify initial values for  $x_1$ ,  $x_2$ , and  $x_3$ ; these values can be changed by double-clicking on the corresponding Integrator icon. In this example, all three initial values are set to 1.0 (not shown in the figure).

The Continuous Director, shown at the upper left, manages the simulation of the model. It contains a sophisticated ODE solver with several key parameters. These parameters can be accessed by double clicking on the director, which results in the dialog box shown in Figure 9.2.

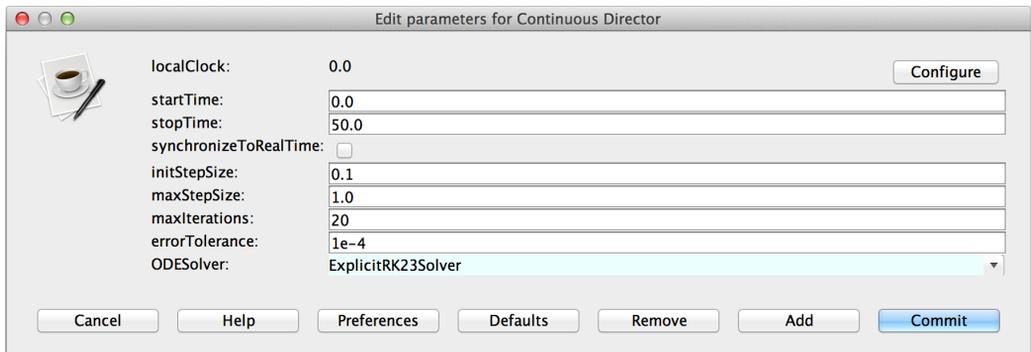


Figure 9.2: Dialog box showing director parameters for the model in Figure 9.1.

The simplest parameters are *startTime* and *stopTime*, which define the region of the time line over which the simulation will execute. The effects of the other parameters are explored in Exercise 1.

The output of the Lorenz model is shown in Figure 9.3. The XY Plotter displays  $x_1(t)$  vs.  $x_2(t)$  for values of  $t$  in between *startTime* and *stopTime*.

Like the Lorenz model, many continuous-time models contain integrators in feedback loops. Instead of using Integrator actors, however, it is possible to use more elaborate blocks that implement linear and non-linear dynamics, as described below.

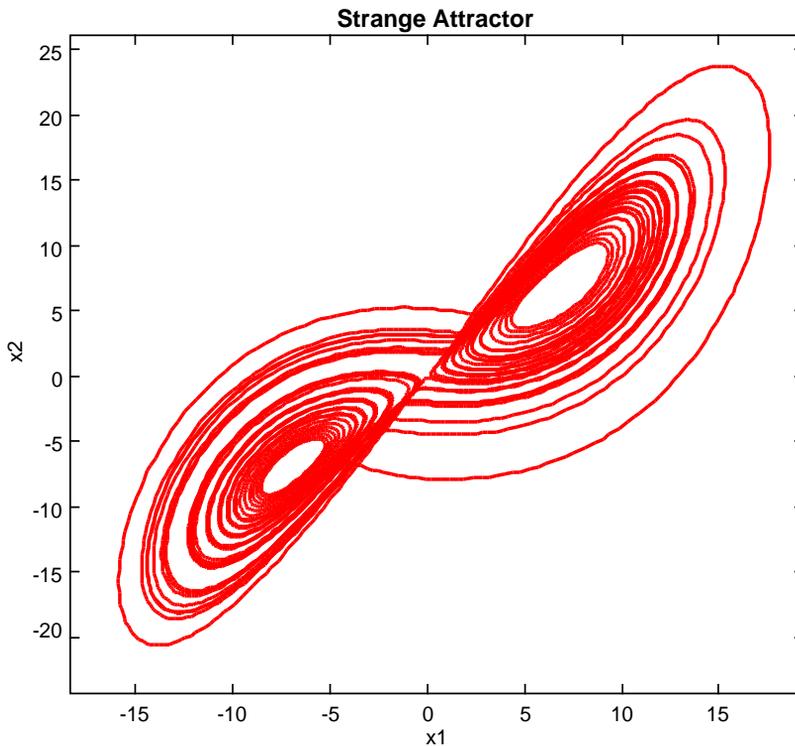


Figure 9.3: Result of running the Lorenz model.

## 9.1.2 Transfer Functions

When representing continuous-time systems, it is often more convenient to use a higher-level description than individual integrators. For example, for linear time invariant (**LTI**) systems, it is common to characterize their input output behavior in terms of a **transfer function**, which is the Laplace transform of the impulse response. Specifically, for an input  $x$  and output  $y$ , the transfer function may be given as a function of a complex variable  $s$ :

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \dots + b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \dots + a_n} \quad (9.4)$$

where  $Y$  and  $X$  are the Laplace transforms of  $y$  and  $x$ , respectively. The number  $n$  of denominator coefficients is strictly greater than the number  $m$  of numerator coefficients. A system that is described by a transfer function can be constructed using individual integrators, but is more conveniently implemented using the **ContinuousTransferFunction** actor, as illustrated by the following example.

**Example 9.2:** Consider the model in Figure 9.4, which produces the plot in Figure 9.5. This model generates a square wave using a **ContinuousClock** actor (see sidebar on page 326) and feeds that square wave into a **ContinuousTransferFunction** actor. The transfer function implemented by the **ContinuousTransferFunction** actor is given by the following:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{0.001s^2 + 0.01s + 1}.$$

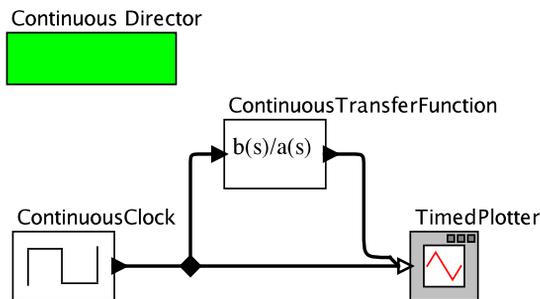


Figure 9.4: Model illustrating the use of **ContinuousTransferFunction**. [[online](#)]

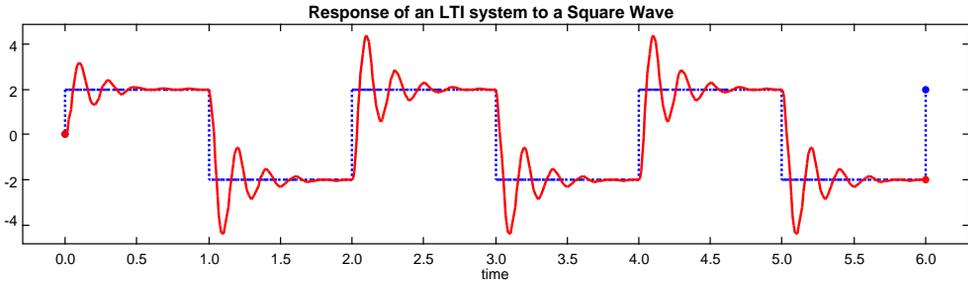


Figure 9.5: Result of running the model in Figure 9.4.

Comparing the equation above to Equation (9.4), we see that  $m = 1$  and  $n = 3$ , with additional parameters as follows:

$$\begin{aligned} b_1 &= 1 \\ a_1 &= 0.001, \quad a_2 = 0.01, \quad a_3 = 1.0 \end{aligned}$$

The parameters of the actor are therefore set to

$$\begin{aligned} \text{numerator} &= \{1.0\} \\ \text{denominator} &= \{0.001, 0.01, 1.0\} \end{aligned}$$

An equivalent model constructed with individual integrators is shown in Figure 9.6 (see Exercise 2 to explore why these are equivalent).

The previous example shows that a complex network of integrators, gains, and adders can be represented compactly using the ContinuousTransferFunction actor. In fact, this actor uses the specified parameter values to construct a hierarchical model similar to the one shown in Figure 9.6. It is possible to view this hierarchical model by right clicking on the ContinuousTransferFunction actor and selecting `Open Actor`. (Select `[Graph→Automatic Layout]` so that the actors are shown in a more readable layout.) ContinuousTransferFunction is an example of a **higher-order actor**, where the parameters specify an actor network that implements the functionality of the actor.

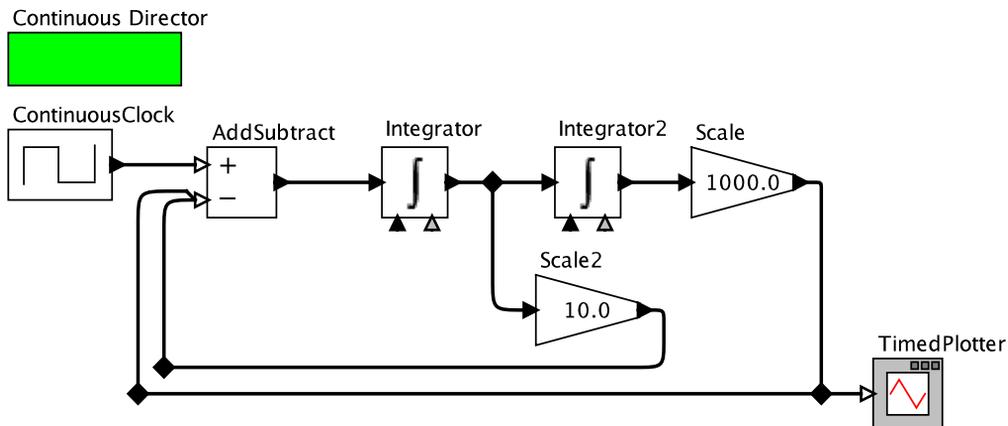


Figure 9.6: A model equivalent to the one in Figure 9.4 assuming the parameters of Example 9.2. [\[online\]](#)

The ContinuousTransferFunction actor and other actors that support higher-level descriptions of dynamics are summarized in the sidebar on page 327.

### 9.1.3 Solvers

Numerical integration is an old, complex, and deep topic (Press et al., 1992). A complete treatment of this topic is beyond the scope of this text, but it is useful to understand the basic concepts in order to make effective use of the Ptolemy solver functions (which use numerical integration to find solutions to equations). In this section, we will give a brief overview of the solver mechanisms that are implemented in the Ptolemy II Continuous director.

Suppose that  $w$  is a continuous-time signal. For the moment, let us ignore the [superdense time](#) model used in Ptolemy II, and assume that  $w$  is an integrable function of the form  $w: \mathbb{R} \rightarrow \mathbb{R}$ . Assume further that for any  $t \in \mathbb{R}$ , we have a procedure to evaluate  $w(t)$ . Suppose further that  $x$  is a continuous-time signal given by

$$x(t) = x_0 + \int_0^t w(\tau) d\tau,$$

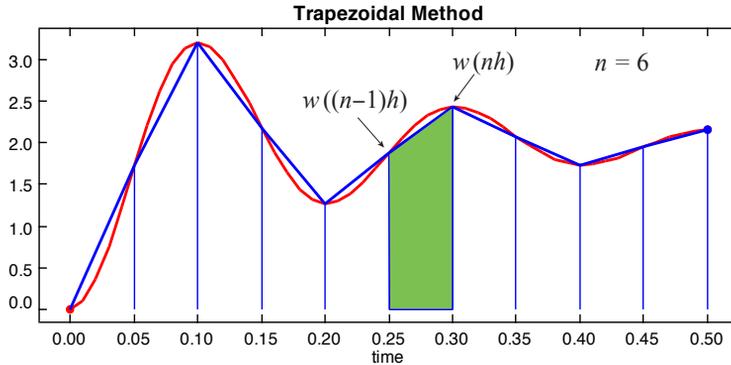


Figure 9.7: Illustration of the trapezoidal method. The area under the curve is approximated by the sum of the areas of the trapezoids. One of the trapezoids is shaded.

where  $x_0$  is a constant. Equivalently,  $x(t)$  is the area under the curve formed by  $w(\tau)$  from  $\tau = 0$  to  $\tau = t$ , plus an initial value  $x_0$ . Note that, consequently,  $w$  is the derivative of  $x$ , or  $w(t) = \dot{x}(t)$ . Given  $w$ , we can construct  $x$  by providing  $w$  as the input to an [Integrator](#) actor with *initialState* set to  $x_0$ ;  $x$  will then be the output.

**Numerical integration** is the process of evaluating  $x$  at enough points  $t \in \mathbb{R}$  to accurately deduce the shape of the function. Of course, the meaning of “accurate” may depend on the application, but one of the key criteria is that the value of  $x$  is sufficiently accurate at sufficiently many points that those values of  $x$  can be used to calculate values of  $x$  at additional points  $t \in \mathbb{R}$  in time. A **solver** is a realization of a numerical integration algorithm. The simplest solvers are **fixed step size solvers**. They define a **step size**  $h$ , and calculate  $x$  at intervals of  $h$ , specifically  $x(h)$ ,  $x(2h)$ ,  $x(3h)$ , etc.

A reasonably accurate fixed step size solver uses the **trapezoidal method**, where it approximates  $x$  as follows:

$$x(nh) = \begin{cases} x_0, & \text{if } n = 0 \\ x((n-1)h) + h(w((n-1)h) + w(nh))/2, & \text{if } n \geq 1 \end{cases}$$

This approach is illustrated in Figure 9.7. As defined by the equations above, the area under the curve  $w$  from 0 to  $nh$  is approximated by the sum of the areas of trapezoids of width  $h$ , where the heights of the sides of the trapezoids are given by  $w(mh)$  for integers

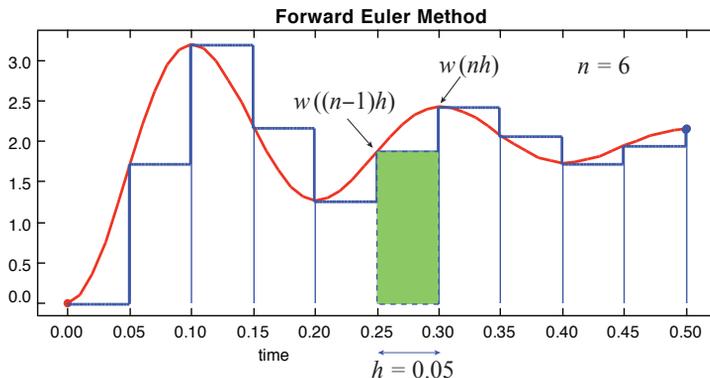


Figure 9.8: Illustration of the forward Euler method. The area under the curve is approximated by the sum of the areas of the rectangles, like the shaded one.

$m$ . The shaded trapezoid in the figure approximates the area under the curve from time  $(n - 1)h$  to time  $nh$ , where  $n = 6$  and  $h = 0.05$ .

A trapezoidal method solver is difficult to use within feedback systems like the one shown in Figure 9.1, however, because the solver needs to compute the outputs of the Integrator actors based on the inputs. To compute the output of an Integrator at time  $t_n = nh$ , the solver needs to know the value of the input at both  $t_{n-1} = (n - 1)h$  and  $t_n = nh$ . But in Figure 9.1, the input to the integrators at any time  $t_n$  depends on the output of the same integrators at that same time  $t_n$ ; there is a circular dependency. Solvers that exhibit a circular dependency are called **implicit method** solvers. One way to use them within feedback systems is to “guess” the feedback value and iteratively refine the guess until some desired accuracy is achieved, but in general there is no assurance that such strategies yield unique answers.

In contrast to implicit method solvers, the **forward Euler method** is an **explicit method** solver. It is similar to the trapezoidal method but is easier to apply to feedback systems. It approximates  $x$  by

$$x(nh) = x((n - 1)h) + hw((n - 1)h).$$

This approach is illustrated in Figure 9.8. The area under each step is approximated as a rectangle rather than as a trapezoid. This method is less accurate, usually, and errors accumulate faster, but it does not require the solver to know the input at time  $nh$ .

In general, using a smaller step size  $h$  increases the accuracy of the solution, but increases the amount of computation required. The step size required to meet a target level of accuracy depends on how rapidly the signal is varying. Both the trapezoidal method and the forward Euler method can be generalized to become **variable step size solvers** that dynamically adjust their step size based on the variability of the signal. Such solvers evaluate the integral at time instants  $t_1$ ,  $t_2$ , etc., using an algorithm to determine the increment to use between time instants. This algorithm first chooses a step size, then performs the numerical integration, then estimates the error. If the estimate of the error is above some threshold (controlled by the *errorTolerance* parameter of the director), then the director redoes the numerical integration with a smaller step size.

A variable-step-size forward Euler solver will first determine a time increment  $h_n$  to define  $t_n = t_{n-1} + h_n$  and then calculate

$$x(t_n) = x(t_{n-1}) + h_n w(t_{n-1}).$$

The variable-step-size forward Euler method is a special case of the widely used **Runge-Kutta (RK)** methods. The Continuous director offers two variants of RK solvers, `ExplicitRK23Solver` and `ExplicitRK45Solver`, selected using the *ODESolver* parameter of the director. These variants are described in more detail in the sidebars on pages 328 and 329. The plot in Figure 9.5 is generated using the `ExplicitRK23Solver`. A closeup with stems that indicate where the solver chose to calculate signal values is shown in Figure 9.9. This figure shows that the solver uses smaller step sizes in regions where the signal is varying more rapidly.

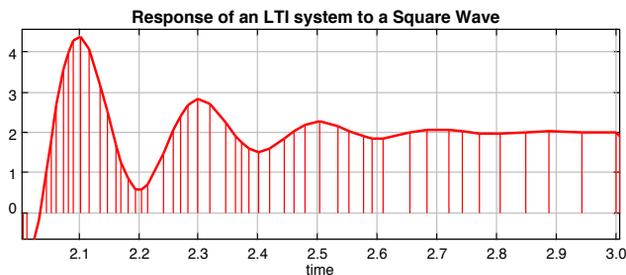
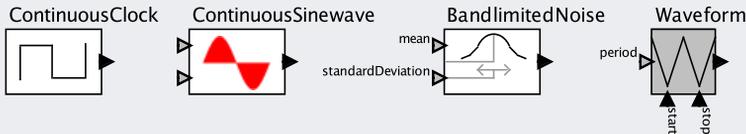


Figure 9.9: A closeup of the plot in Figure 9.5 (with stems showing) reveals that the solver uses smaller step sizes in regions where the signal varies more rapidly.

### Sidebar: Continuous-Time Signal Generators

The continuous domain provides several actors that generate [continuous-time signals](#).

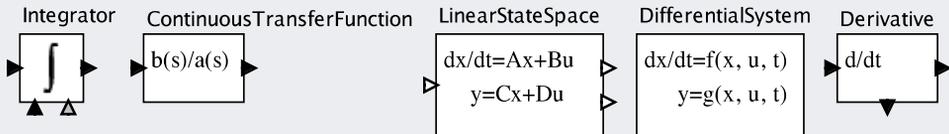


These actors are located in `DomainSpecific→Continuous→SignalGenerators`, except `BandlimitedNoise`, which is in `DomainSpecific→Continuous→Random`.

- **ContinuousClock** has parameters similar to [DiscreteClock](#), but produces a **piecewise constant** signal. A square wave, like that shown in Figure 9.5, is a simple example of such a signal; the actor is also capable of producing complex repeating or non-repeating patterns.
- **ContinuousSinewave**, as the name implies, produces a sine wave. The frequency, phase, and amplitude of the sine wave are set via parameters. This actor constrains the [step size](#) of the [solver](#) to ensure that its output is reasonably smooth; the step size is set to be no greater than one tenth of a period of the sine wave.
- **BandlimitedNoise** is the most sophisticated of these signal generators. This actor generates continuous-time noise with a Gaussian distribution and controlled bandwidth. Although a full discussion of the topic is beyond the scope of this text, we note that it is not theoretically possible for a causal system to produce perfectly bandlimited noise (see [Lee and Varaiya \(2011\)](#)). This actor implements a reasonable approximation. Like `ContinuousSinewave`, this actor affects the step size chosen by the solver; it ensures that the solver samples the signal at least as frequently as twice the specified bandwidth. This is nominally the Nyquist frequency of an ideally bandlimited noise signal.
- **Waveform** produces a periodic waveform from a finite set of samples. It provides two interpolation methods, linear and Hermite, where the latter uses a third-order interpolation technique based on the Hermite curves in Chapter 11 of [Foley et al. \(1996\)](#). Hermite interpolation is useful for generating smooth curves of arbitrary shape. The interpolation assumes that the waveform is periodic. Note that this actor also affects the step sizes taken by the solver. In particular, it ensures that the solver includes the specified samples, though it does not require the solver to include any samples between them.

### Sidebar: Actors for Modeling Dynamics

Ptolemy II provides several actors that can be used to model continuous-time systems with complicated **dynamics** (behavior over time). These actors are shown below, and can be found in `DomainSpecific→Continuous→Dynamics`:



The fundamental actor is the **Integrator**, described in Sections 9.1.1 and 9.2.4. The others are **higher-order actors** that construct submodels using instances of Integrator.

- **ContinuousTransferFunction**, as explained in Section 9.1.2, can realize a continuous-time system based on a **transfer function** specified as a ratio of two polynomials. The ContinuousTransferFunction actor does not support non-zero initial conditions for the Integrators.
- **LinearStateSpace** specifies the input-output relationship of a system with a set of matrices and vectors that describe a linear constant-coefficient difference equation (**LCCDE**). Unlike ContinuousTransferFunction, this actor supports non-zero initial conditions, but it is similarly constrained to model systems that can be characterized by linear functions.
- **DifferentialSystem** can be used to model complicated nonlinear dynamics. For example, it can be used to specify the **Lorenz attractor** of Example 9.1, as shown in Exercise 3. See the actor documentation for details.
- **Derivative** provides a crude estimate of the derivative of its input. Use of this actor is discouraged, however, because its output can be very noisy, even if the input is continuous and differentiable. The output is simply the difference between the current input and the previous input divided by the step size. If the input is not differentiable, however, the output is not **piecewise continuous**, which may force the solver to use the smallest allowed step size. Note that this actor has *two* outputs. The bottom output produces a discrete event when the input has a discontinuity. This event represents a **Dirac delta function**, explained in Section 9.2.4.

### Sidebar: Runge-Kutta Methods

In general, an ODE can be represented by a system of differential equations on a vector-valued state

$$\begin{aligned}\dot{x}(t) &= g(x(t), u(t), t), \\ y(t) &= f(x(t), u(t), t),\end{aligned}$$

where  $x : \mathbb{R} \rightarrow \mathbb{R}^n$ ,  $y : \mathbb{R} \rightarrow \mathbb{R}^m$ , and  $u : \mathbb{R} \rightarrow \mathbb{R}^l$  are state, output, and input signals. The functions  $g : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \rightarrow \mathbb{R}^n$  and  $f : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \rightarrow \mathbb{R}^m$  are state functions and output functions respectively. The state function  $g$  is represented by the Expression actors in the feedback path in Figure 9.1. This function gives the inputs  $\dot{x}(t)$  of the Integrator actors as a function of their outputs  $x(t)$ , external inputs  $u(t)$  (of which there are none in Figure 9.1), and the current time  $t$  (which is also not used in Figure 9.1).

Given this formulation, an explicit  $k$ -stage RK method has the form

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i, \quad (9.5)$$

where

$$\begin{aligned}K_0 &= h_n g(x(t_{n-1}), u(t_{n-1}), t_{n-1}), \\ K_i &= h_n g\left(x(t_{n-1}) + \sum_{j=0}^{i-1} A_{i,j} K_j, u(t_{n-1} + hb_i), \right. \\ &\quad \left. t_{n-1} + hb_i\right), \quad i \in \{1, \dots, k-1\}\end{aligned}$$

and  $A_{i,j}$ ,  $b_i$  and  $c_i$  are algorithm parameters calculated by comparing the form of a Taylor series expansion of  $x$  with (9.5).

The first-order RK method, also called the **forward Euler method**, has the (much simpler) form

$$x(t_n) = x(t_{n-1}) + h_n \dot{x}(t_{n-1}).$$

This method is conceptually important but not as accurate as other available methods.

**Continued on page 329.**

### Sidebar: Runge-Kutta Methods - Continued

Continued from page 328.

More accurate Runge-Kutta methods have three or four stages, and also control the step size for each integration step. The `ExplicitRK23Solver` implemented by the Continuous director is a  $k = 3$  (three-stage) method and is given by

$$x(t_n) = x(t_{n-1}) + \frac{2}{9}K_0 + \frac{3}{9}K_1 + \frac{4}{9}K_2, \quad (9.6)$$

where

$$K_0 = h_n g(x(t_{n-1}), t_{n-1}), \quad (9.7)$$

$$K_1 = h_n g(x(t_{n-1}) + 0.5K_0, u(t_{n-1} + 0.5h_n), t_{n-1} + 0.5h_n), \quad (9.8)$$

$$K_2 = h_n g(x(t_{n-1}) + 0.75K_1, u(t_{n-1} + 0.75h_n), t_{n-1} + 0.75h_n). \quad (9.9)$$

Notice that in order to complete one integration step, this method requires evaluation of the function  $g$  at intermediate times  $t_{n-1} + 0.5h_n$  and  $t_{n-1} + 0.75h_n$ , in addition to the times  $t_{n-1}$ , where  $h_n$  is the step size. This fact significantly complicates the design of actors, because they have to tolerate multiple evaluations (firings) that are speculative and may have to be redone with a smaller step size if the required accuracy is not achieved. The validity of a step size  $h_n$  is not known until the full integration step has been completed. In fact, any method that requires intermediate evaluations of the state function  $g$ , such as the classical fourth-order RK method, linear multi-step methods (LMS), and BulirschStoer methods, will encounter the same issue.

In the Continuous domain, the RK solvers speculatively execute the model at intermediate points, invoking the `fire` method of actors but not their `postfire` method. As a consequence, actors used in the Continuous domain must all conform to the [strict actor semantics](#); they must not change state in their `fire` method.

## 9.2 Mixed Discrete and Continuous Systems

The continuous domain supports mixtures of discrete and continuous behaviors. The simplest such mixtures produce [piecewise continuous](#) signals, which vary smoothly over time except at particular points in time, where they vary abruptly. Piecewise continuous signals are explained in Section [9.2.1](#).

In addition, signals can be genuinely discrete. In particular, as with the [DE](#) domain, a signal in the Continuous domain can be absent at a time stamp. A signal that is *never* absent is a true **continuous-time signal**. A signal that is always absent except at a [discrete set](#) of time stamps is a **discrete-event signal**, explained in Section [9.2.2](#). A model that mixes both types of signals is a **mixed signal** model. It is possible to have signals that are continuous-time over a range of time stamps, and discrete-event over another range. These are called **mixed signals**.

### 9.2.1 Piecewise Continuous Signals

As shown in Figure [9.9](#), variable step-size solvers produce more samples per unit time when a signal is varying rapidly. These solvers do not, however, directly support discontinuous signals, such as the square wave shown in Figure [9.5](#). The Continuous director in Ptolemy II augments the standard ODE solvers with techniques that handle such discontinuities, but the signals must be piecewise continuous. Meeting this prerequisite requires some care, as we will discuss later in the chapter.

Recall that Ptolemy II uses a [superdense time](#) model. This means that a continuous-time signal is a function of the form

$$x: T \times \mathbb{N} \rightarrow V, \tag{9.10}$$

where  $T$  is the set of [model time](#) values (see Section [1.7.3](#)),  $\mathbb{N}$  is the non-negative integers representing the [microstep](#), and  $V$  is some set of values (the set  $V$  is the data [type](#) of the signal). This function specifies that, at each model time  $t \in T$ , the signal  $x$  can have several values, and these values occur in a defined order. For the square wave in Figure [9.5](#), at time  $t = 1.0$ , for example, the value of the square wave is first  $x(t, 0) = 2$  and then  $x(t, 1) = -2$ .

In order for time to progress past a model time  $t \in T$ , we need to ensure that every signal in the model has a finite number of values at  $t$ . Thus, we require that for all  $t \in T$ , there

exist an  $m \in \mathbb{N}$  such that

$$\forall n > m, \quad x(t, n) = x(t, m). \quad (9.11)$$

This constraint prevents **chattering Zeno** conditions, where a signal takes on infinitely many values at a particular time. Such conditions prevent an execution from progressing beyond that point in model time, assuming the execution is constrained to produce values in chronological order.

Assuming  $x$  has no chattering Zeno condition, then there is a least value of  $m$  satisfying (9.11). We call this value of  $m$  the **final microstep** and  $x(t, m)$  the **final value** of  $x$  at  $t$ . We call  $x(t, 0)$  the **initial value** at time  $t$ . If  $m = 0$ , then we say that  $x$  has only one value at time  $t$ .

Define the **initial value function**  $x_i: T \rightarrow V$  by

$$\forall t \in T, \quad x_i(t) = x(t, 0).$$

Define the **final value function**  $x_f: T \rightarrow V$  by

$$\forall t \in T, \quad x_f(t) = x(t, m_t),$$

where  $m_t$  is the final microstep at time  $t$ . Note that  $x_i$  and  $x_f$  are conventional continuous-time functions if we abstract model time as the real numbers  $T = \mathbb{R}$ .

A **piecewise continuous** signal is defined to be a function  $x$  of the form  $x: T \times \mathbb{N} \rightarrow V$  with no chattering Zeno conditions that satisfies three requirements:

1. the initial value function  $x_i$  is continuous on the left;
2. the final value function  $x_f$  is continuous on the right; and
3.  $x$  has only one value at all  $t \in T \setminus D$ , where  $D$  is a discrete subset of  $T$ .

The last requirement is a subtle one that deserves further discussion. First, the notation  $T \setminus D$  refers to a set that contains all elements of the set  $T$  except those in the set  $D$ .  $D$  is constrained to be a **discrete set**, described in the sidebar on page 334. Intuitively,  $D$  is a set of time values that can be counted in temporal order. It is easy to see that if  $D = \emptyset$  (the empty set), then  $x_i = x_f$ , and both  $x_i$  and  $x_f$  are continuous functions. Otherwise each of these functions is piecewise continuous.

A key constraint of the Continuous domain in Ptolemy II is that all signals are piecewise continuous in the above sense. Based on this definition, the square wave shown in Figure 9.5 is piecewise continuous. At each discontinuity, it has two values: an initial value that matches the values before the discontinuity, and a final value that matches the value after the discontinuity. It can be easy to create signals that are not piecewise continuous, however, as illustrated by the following example.

**Example 9.3:** The model in Figure 9.10 contains an [Expression](#) actor whose input is a continuous-time signal. The expression is shown below:

$$(in > 1.0) ? in + 1 : 0$$

If the input is greater than 1.0, then the output will be the input plus one; otherwise the output will be zero. This output signal is not [piecewise continuous](#) in the sense described above. Before or at time 1.0, the output value is zero. But at any time after 1.0, the value is not zero. The signal is not continuous from the right.

Figure 9.11 shows the resulting plot, where the output of the Expression actor is labeled “second.” The transition from zero to non-zero is not instantaneous, as

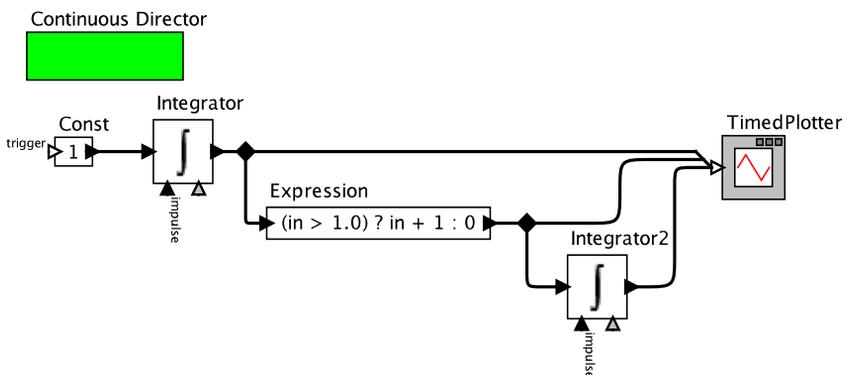


Figure 9.10: A model that produces a signal that is not piecewise continuous, and therefore can exhibit solver-dependent behavior. This problem is eliminated in the model in Figure 9.23. [\[online\]](#)

shown by the slanted dashed line in the middle plot. Worse, the width of the transition depends on seemingly irrelevant details of the model. The model shows the signal connected to a second integrator. If that second integrator is deleted from the

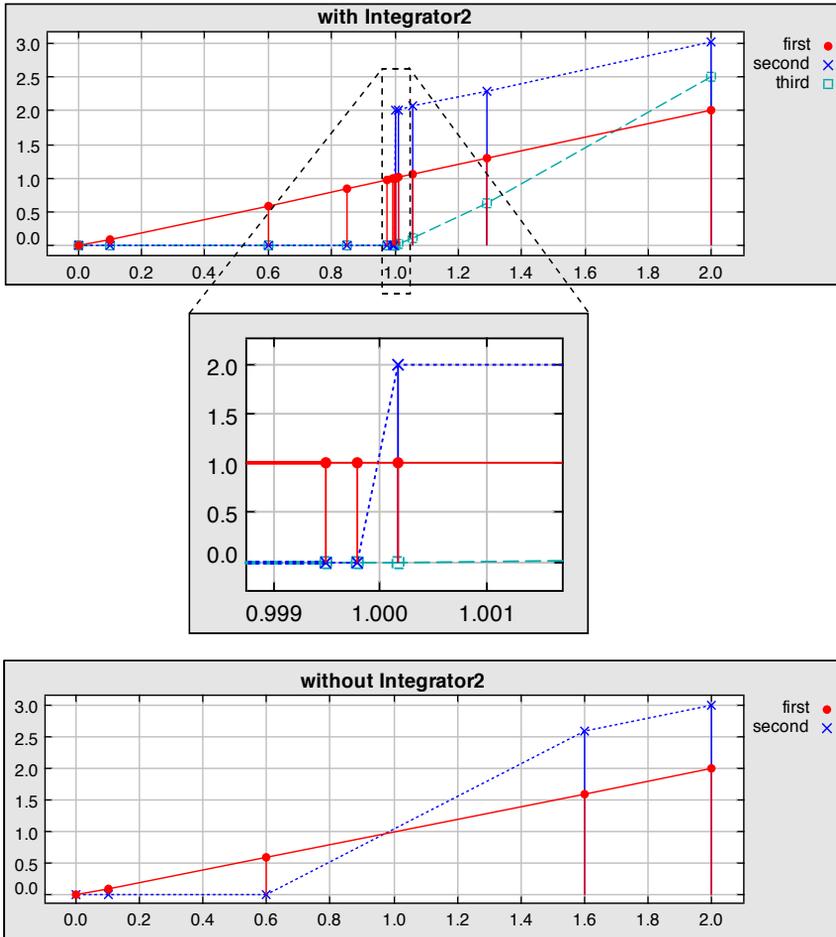


Figure 9.11: A plot of the signal produced by the model in Figure 9.10 with and without the second Integrator. Note that the output of the Expression actor seems to depend on whether the second Integrator is present. The signals labeled “first,” “second,” and “third” are the top-to-bottom inputs of the plotter, respectively.

model, then the width of the transition changes, as shown in the lower plot. This is because the second Integrator affects the step size taken by the solver. In order to achieve adequate integration accuracy, the second Integrator forces a smaller step size in the vicinity of time 1.0.

The problem is not solved by changing the expression to

```
(in >= 1.0) ? in + 1 : 0
```

Here, the transition from zero to non-zero occurs when the input is greater than *or equal to* 1.0. In this case, the resulting signal is not continuous from the left. The resulting plots are identical. The Expression actor simply calculates a specified function of its inputs when it fires. It has no mechanism for generating distinct values at distinct microsteps unless its input already has distinct values at distinct microsteps.

The previous example shows that using an actor whose output is a discontinuous function of the input can create problems if the input is a continuous-time signal. The next few sections describe various mechanisms for properly constructing discontinuous signals. The particular problem with Figure 9.10 is solved using [modal models](#) in Section 9.3.1.

### Sidebar: Probing Further: Discrete Sets

A set  $D$  is a **discrete set** if it is a totally ordered set (for any two elements  $d_1$  and  $d_2$ , either  $d_1 \leq d_2$  or  $d_1 > d_2$ ) where there exists a one-to-one function  $f: D \rightarrow \mathbb{N}$  that is **order preserving**. Order preserving simply means that for all  $d_1, d_2 \in D$  where  $d_1 \leq d_2$ , we have that  $f(d_1) \leq f(d_2)$ . The existence of such a one-to-one function ensures that we can arrange the elements of  $D$  in *temporal order*. Notice that  $D$  is a countable set, but not all countable sets are discrete. For example, the set  $\mathbb{Q}$  of rational numbers is countable but not discrete. There is no such one-to-one function.

## 9.2.2 Discrete-Event Signals in the Continuous Domain

As described earlier, the Continuous domain supports genuinely discrete signals, which are signals that are present only at particular instants. As a consequence, the `clock` actors that are used in the `DE` domain (see sidebar on page 241) can be used in the continuous domain.

**Example 9.4:** The `ContinuousClock` actor in Figure 9.4 is a composite actor that uses a `DiscreteClock` and a `ZeroOrderHold` (see box on page 338), as shown in Figure 9.12. The `DiscreteClock` produces a discrete-event signal and the `ZeroOrderHold` converts that signal to a continuous-time signal (see sidebar on page 338).

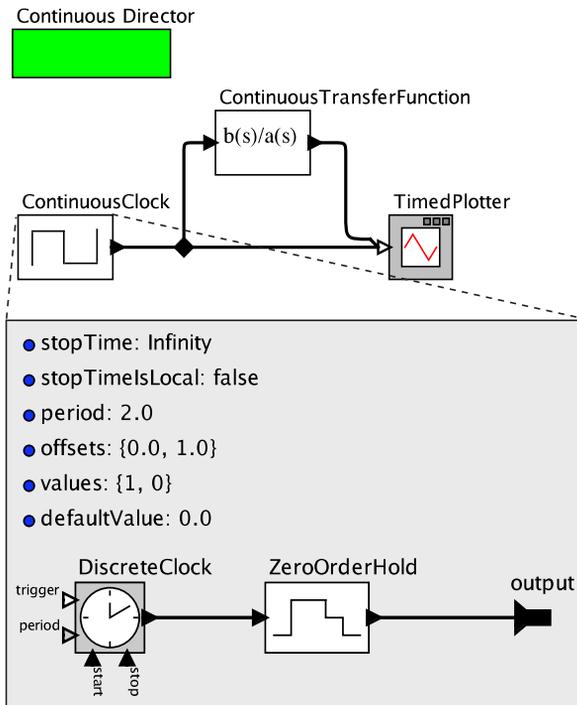


Figure 9.12: The `ContinuousClock` actor in this model is a composite actor that uses a `DiscreteClock` and a `ZeroOrderHold`.

Both signals are **piecewise continuous**. The output signal from `DiscreteClock` at the model time of each event is characterized as follows: it is absent at microstep zero (which matches its value at times just before the event); present at microstep one (which is the discrete event); and absent again at microsteps two and higher (which matches its value at larger times until the next discrete event). Therefore, the output of `DiscreteClock` is piecewise continuous, as required by the solver.

The **clock** actors described in the sidebar on page 241 all behave in a manner that is similar to `DiscreteClock` and hence they can all be used in the continuous domain.

Note that although many actors that operate on or produce discrete events have a *trigger* input port, there is rarely a reason to connect that port in the Continuous domain. In the DE domain, a *trigger* port is used to trigger execution of an actor at the time of the input event. But in the Continuous domain, every actor executes every time that there is an execution. Nevertheless, it is sometimes useful to use the *trigger* port, as illustrated by the example in Figure 9.13.

### 9.2.3 Resetting Integrators at Discrete Times

In addition to its signal input and output, the **Integrator** actor has two extra ports at the bottom of the icon. The one at the lower right is a **PortParameter** called *initialState*. When an input token is provided on that port, the state of the Integrator will be reset to the value of the token. The output of the Integrator will change instantaneously to the specified value.

**Example 9.5:** The model in Figure 9.14 uses a `DiscreteClock` actor to periodically reset an Integrator.

The input events on the *initialState* port are required to be purely discrete. This means that at all model times, the input signal must be absent at microstep 0. Any attempt to feed a continuous signal into this port results in an exception similar to the one below:

```
IllegalActionException: Signal at the initialState port is not purely discrete.
```

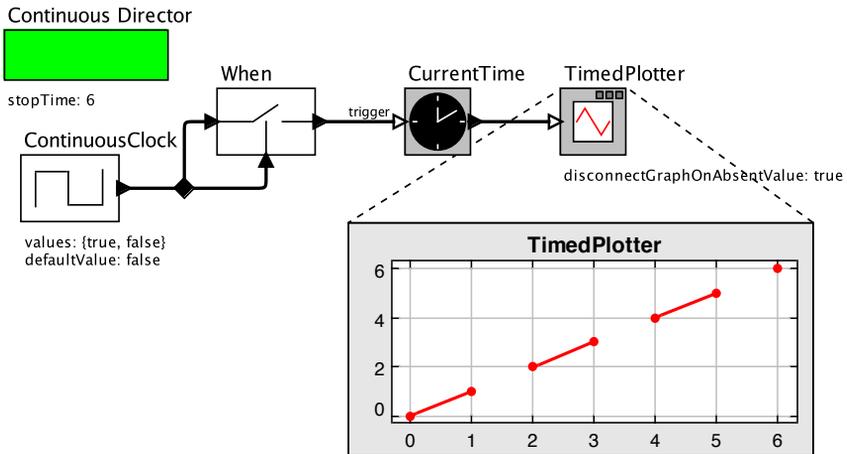


Figure 9.13: The *trigger* port of the CurrentTime actor in this model is used to turn on and off its output. During the time intervals where the output of the DiscreteClock actor is *false*, the CurrentTime actor is disabled, and hence its output will be absent. [\[online\]](#)

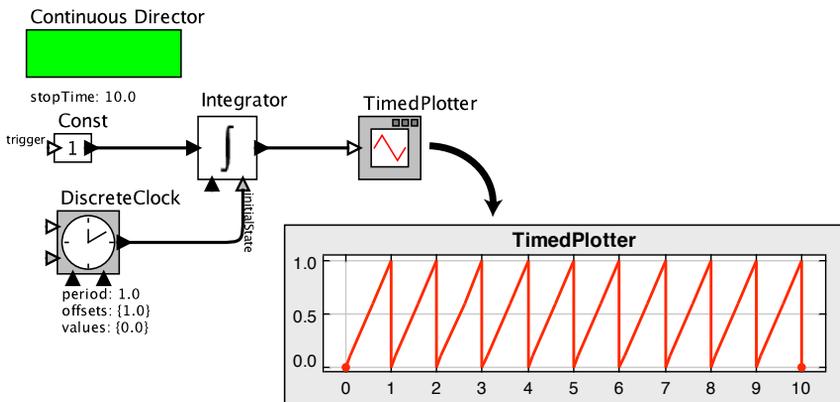


Figure 9.14: Illustration of the use of the *initialState* port of the Integrator actor. [\[online\]](#)

*in Integrator*

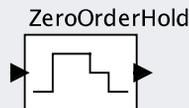
This check ensures that the output of the Integrator is [piecewise continuous](#). In Figure 9.14, for example, at the points of discontinuity the signal first takes the value of the Integrator state prior to applying the reset. In the subsequent microstep, it takes the value after the reset. In contrast, if the output of the Integrator had changed abruptly at microstep zero, then the output could have been a different value at a time infinitesimally earlier — thus violating the requirement for piecewise continuity.

## 9.2.4 Dirac Delta Functions

The other [Integrator](#) input port is called *impulse*. Like the *initialState* port discussed in the previous section, the signal at this port is required to be purely discrete. When an impulse event arrives, it causes an instantaneous increment or decrement of the state (and output)

### Sidebar: Continuous Signals from Discrete Events

The **ZeroOrderHold** actor, shown below, takes a [discrete-event signal](#) in and produces a [continuous-time signal](#) on its output:



This actor is in `DomainSpecific→Continuous→Discrete` to `Continuous`.

At times between input events, the value of the output is the value of the most recent event, so the output is [piecewise constant](#). At the time of each input event, the output at microstep zero is the value of the previous event, and at microstep one, it takes the value of the current event. Hence, the output signal is [piecewise continuous](#).

It may seem desirable to define an actor that interpolates between the values of the input events, as is done by the [Waveform](#) actor (see sidebar on page 326). However, in order to interpolate, the actor would have to know the value of a *future* event. Actors in the continuous domain are required to be **causal**, meaning that their outputs depend only on current and past inputs. The outputs cannot depend on future inputs. Hence, no such interpolation is possible. The [Waveform](#) actor is able to perform interpolation because the values that it is interpolating are specified as parameters, not as input events.

of the Integrator. That is, rather than resetting the state to a specified value, it adds to (or subtracts from) the current state.

Mathematically, such functionality is often represented as a **Dirac delta function** in signals and systems. A Dirac delta function is a function  $\delta: \mathbb{R} \rightarrow \mathbb{R}^+$  given by

$$\forall t \in \mathbb{R}, t \neq 0, \quad \delta(t) = 0$$

and

$$\int_{-\infty}^{\infty} \delta(\tau) d\tau = 1.$$

That is, the signal value is zero everywhere except at  $t = 0$ , but its integral is unity. At  $t = 0$ , therefore, its value cannot be finite. Any finite value would yield an integral of zero. This is indicated by  $\mathbb{R}^+$  in the form of the function,  $\delta: \mathbb{R} \rightarrow \mathbb{R}^+$ , where  $\mathbb{R}^+$  represents the **extended reals**, which includes infinity. Dirac delta functions are widely used in modeling continuous-time systems (see [Lee and Varaiya \(2011\)](#), for example), so it is important to be able to include them in simulations.

Suppose that a signal  $y$  has a Dirac delta function occurring at time  $t_1$  as follows,

$$y(t) = y_1(t) + K\delta(t - t_1),$$

where  $y_1$  is an ordinary continuous-time signal, and  $K$  is a scaling constant. Then

$$\int_{-\infty}^t y(\tau) d\tau = \begin{cases} \int_{-\infty}^t y_1(\tau) d\tau & t < t_1 \\ K + \int_{-\infty}^t y_1(\tau) d\tau & t \geq t_1 \end{cases}$$

The component  $K\delta(t - t_1)$  is a Dirac delta function at time  $t_1$  with weight  $K$ , and it causes an instantaneous increment in the integral by  $K$  at time  $t = t_1$ .

**Example 9.6:** LTI systems can be characterized by their impulse response, which is their response to a Dirac delta function. The model in [Figure 9.15](#) is an LTI system with [transfer function](#)

$$H(s) = \frac{1}{1 + as^{-1} + bs^{-2}}.$$

The model provides a Dirac delta function at time 0.2, producing the impulse response shown in the plot.

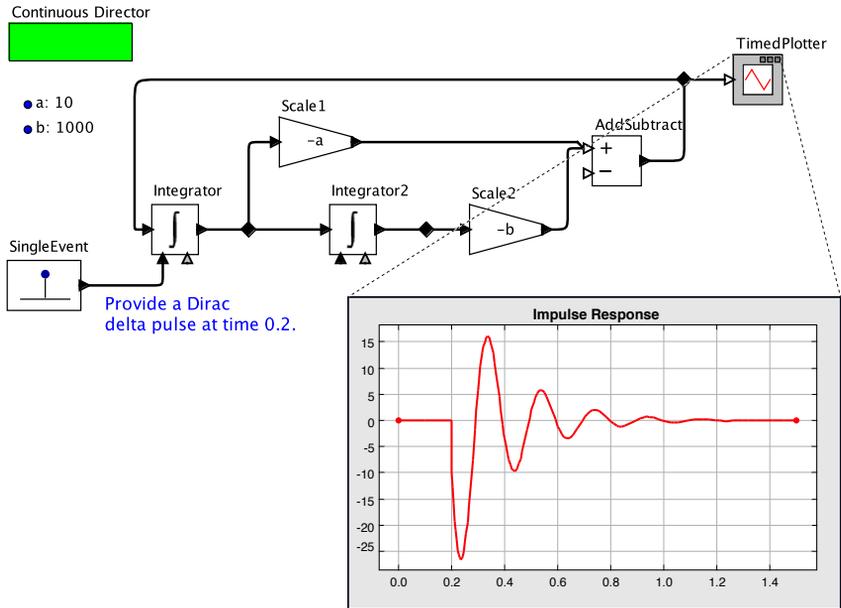
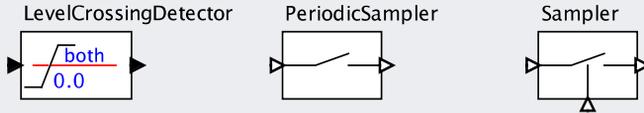


Figure 9.15: Response of an LTI system to a Dirac delta function. [[online](#)]

It is difficult to model Dirac delta functions in computing systems because of their instantaneous and infinite nature. The [superdense time](#) model of Ptolemy II coupled with the semantics of the Continuous domain provide a rigorous, unambiguous model that can support Dirac delta functions.

### Sidebar: Generating Discrete Events

Several actors convert **continuous-time signals** to **discrete-event signals** (these actors are located in DomainSpecific→Continuous→Continuous to Discrete):



- LevelCrossingDetector** converts continuous signals to discrete events when the input signal crosses a threshold specified by the *level* parameter. A *direction* parameter constrains the actor to detect only rising or falling transitions. This actor introduces a one-microstep delay before it produces an output. That is, when a level crossing is detected, the actor requests a refiring in the next microstep at the current time, and produces the output during that refiring. This ensures that the output satisfies the piecewise continuity constraint; it is always absent at microstep 0. The one-microstep delay enables the actor to be used in a feedback loop. An example is shown in Figure 9.16, where the feedback loop resets the Integrator each time it reaches a threshold (1.0 in the example).
- PeriodicSampler** generates discrete events by periodically sampling an input signal. The sampling rate is given by a parameter. By default, the actor reads the **initial value** of the input signal (the input value at microstep 0), but sends it to the output port one microstep later (at microstep 1). This ensures that the output at microstep 0 is always absent, thus ensuring that the output signal is **piecewise continuous**. (The input is absent prior to the sample time, so piecewise continuity requires that it be absent at microstep 0 at the sample time.) Because of the one-step delay, the PeriodicSampler can also be used in a feedback loop. For example, it can be used to periodically reset an Integrator, as shown in the example in Figure 9.17.
- Sampler** is a simple actor. Whenever the *trigger* signal (at the bottom port on the icon) is present, it copies the input from the left port to the output. There is no microstep delay. If the signal at the *trigger* port is a piecewise continuous **discrete-event signal**, then the output will also be a piecewise continuous discrete-event signal. Sampler will normally read its inputs at microstep 1 because the *trigger* input is discrete. (PeriodicSampler will behave in the same way if its *microstep* parameter is set to 1.)

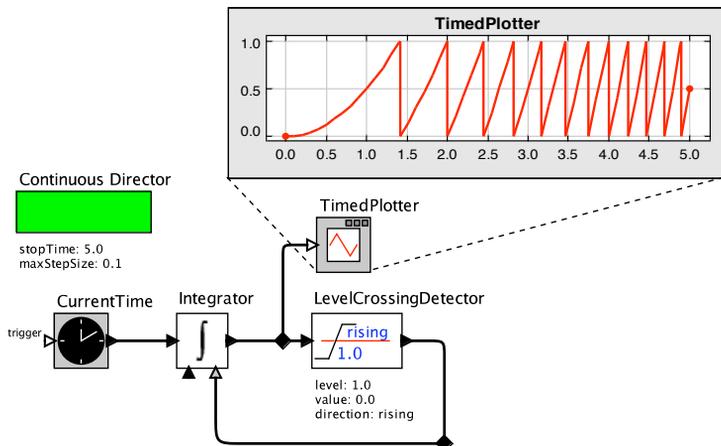


Figure 9.16: Illustration of the LevelCrossingDetector, which can be put in a feedback loop. In this case, whenever the output of the Integrator reaches 1.0, it is reset to zero. [\[online\]](#)

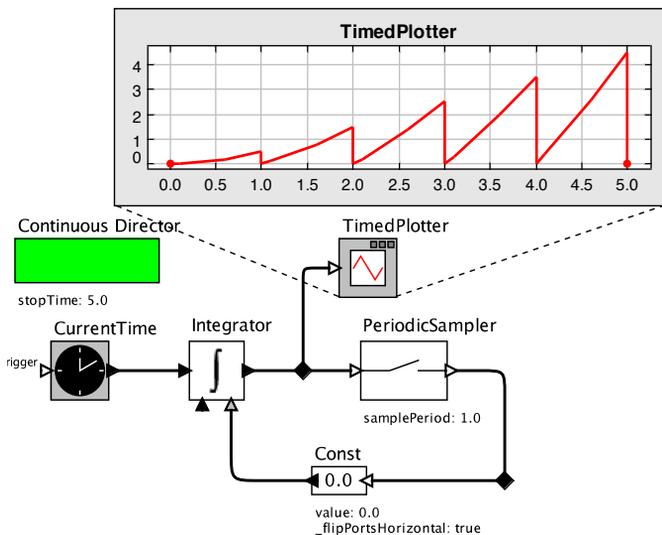


Figure 9.17: The PeriodicSampler actor, placed in a feedback loop. In this case, the Integrator will be reset to zero at intervals of one time unit regardless of its state. [\[online\]](#)

### 9.2.5 Interoperating with DE

The Continuous and DE domains both support **discrete-event signals**. There is a subtle but important difference between these domains, however. In the Continuous domain, at a time stamp selected by the **solver**, *all* actors are fired. In the DE domain, an actor is only fired if either it has an event at an input port or it has previously asked to be fired. As a consequence, DE models can be much more efficient, particularly when events are sparse.

It can be useful to build models that combine the two domains. Such combinations are suitable for many **cyber-physical systems**, for example, which combine continuous dynamics with software-based controllers. Constructing models with a mixture of Continuous and DE domains is easy, as illustrated by the following example.

**Example 9.7:** Consider the model in Figure 9.18. The top level of the model is implemented in the DE domain, and includes an **opaque** composite actor that is a Continuous model. This example models a “job shop,” where job arrivals are discrete events, the processing rate is given by an exponential random variable, and the job processing is modeled in continuous time.

The model assigns an integer number to each job. It then approaches that number with a slope given by the (random) rate. The higher the rate, the faster the job is completed. The job is complete when the blue (dashed) line in the plot reaches the red (solid) line in the lower plot. The upper plot shows the times at which each job is generated and completed. Note that this model has a feedback loop such that each time a job is finished, a new one is started with a new service time.

This example is somewhat contrived, however, in the sense that it does not actually require the use of the Continuous domain (see Exercise 4). In fact, models where continuous-time signals linearly increase or decrease can usually be realized within the DE domain alone, without the need for a solver. That said, there is still value in constructing the mixed-domain model because it can easily evolve to support more complex dynamics in the Continuous portion.

Continuous models can be placed within DE models, as shown in the previous example. Conversely, DE models can be placed within Continuous models. The choice of top level domain is often determined by emphasis. If the emphasis is on a discrete controller, then

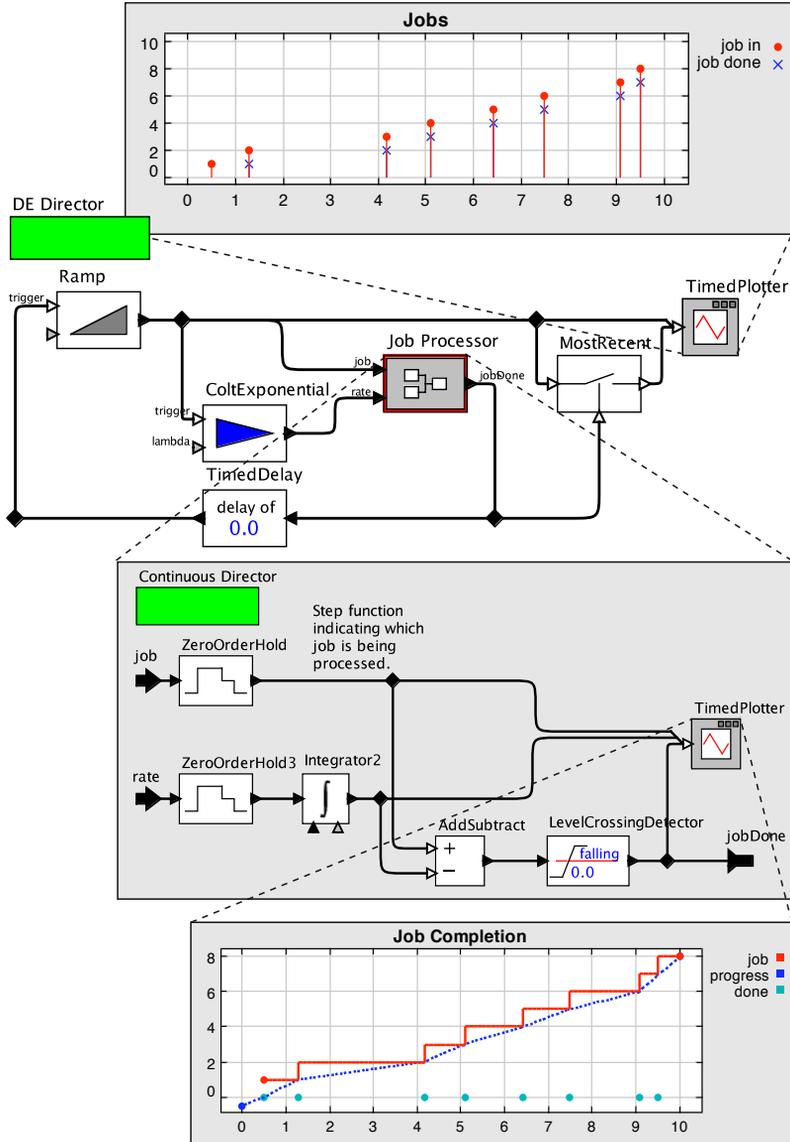


Figure 9.18: Illustration of a hierarchical combination of DE and Continuous models, as considered in Example 9.7. [online]

using DE at the top level often makes sense. If the emphasis is on the physical plant, then using Continuous at the top level may be better.

If multiple Continuous submodels are placed within a DE model, then the solvers in the submodels are decoupled. This can be useful for modeling systems with widely disparate time scales; one solver can use small step sizes without forcing the other submodel to use small step sizes.

The ability to combine DE and Continuous domains relies on a key property of DE, which is that events in DE normally occur at microstep one, not at microstep zero. When these events cross the boundary into a Continuous model, they preserve this microstep. Hence, a signal that passes from the DE domain to the Continuous domain will normally be absent at microstep zero, thus ensuring piecewise continuity. When a signal goes from the Continuous domain to the DE domain, however, it is important that the signal be discrete, as would be produced by a [Sampler](#) or [LevelCrossingDetector](#) (see page 341).

## 9.2.6 Fixed-Point Semantics

Recall from Section 7.3.4 that, as of this writing, the DE director in Ptolemy II implements an approximation of the fixed-point semantics described by [Lee and Zheng \(2007\)](#). In contrast, the Continuous director implements an exact fixed-point semantics, and can therefore execute some models that DE cannot.

**Example 9.8:** Consider the model shown in Figure 9.19. This model is identical to the model considered in Example 7.14, except that the Continuous director is used instead of the DE director. The Continuous director, unlike the DE director, is able to fire actors multiple times at a given time stamp. As a consequence, it does not need to know whether an event is present or absent at the input of the composite actor before it is fired. The director can fire the composite actor, obtain an event from the `DiscreteClock`, and then later fire the composite actor again once that event has been fed back.

## 9.3 Hybrid Systems and Modal Models

A **hybrid system** is a model that combines continuous dynamics with discrete mode changes. Such models are created in Ptolemy II using `ModalModel` actors, found in the `Utilities` library and explained in Chapter 8. This section starts by examining a pre-built hybrid system, and concludes by explaining the principles that make hybrid models work. Chapter 8 explains how to construct such models, and explains how time is handled in `mode refinements`.

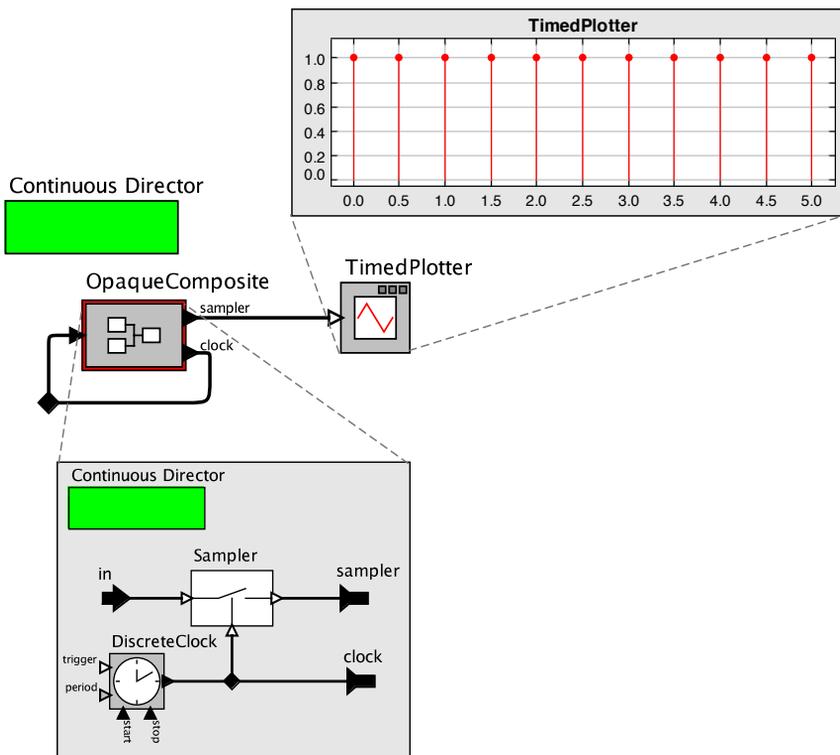


Figure 9.19: A discrete-event model that is executable using the Continuous director, but not using the DE director, as shown in Example 7.14. [online]

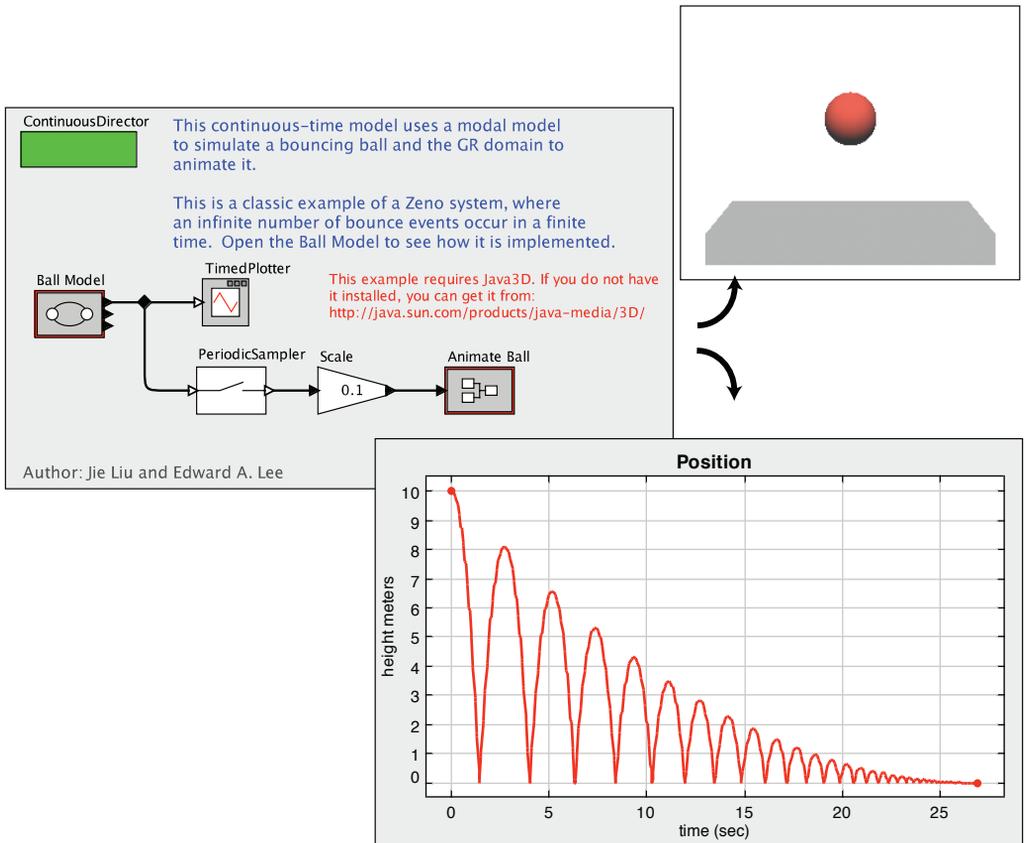


Figure 9.20: Top level of the bouncing ball hybrid system example. [\[online\]](#)

**Example 9.9:** A bouncing ball model is shown in Figure 9.20. It can be found under “Bouncing Ball” in the Tour of Ptolemy II (Figure 2.3, in the “Hybrid Systems” entry). The bouncing ball model uses a ModalModel component named Ball Model. Executing the model yields a plot like that in the figure (along with 3-D animation that is constructed using the GR (graphics) domain, which is not covered here). This model has continuous dynamics during times when the ball is in the air, and discrete events when the ball hits the surface and bounces.

Figure 9.21 shows the contents of Ball Model, which is a modal model with three states: init, free, and stop. During the time a modal model is in a state, its behavior is specified by the mode refinement. In this case, only the free state has a refinement,

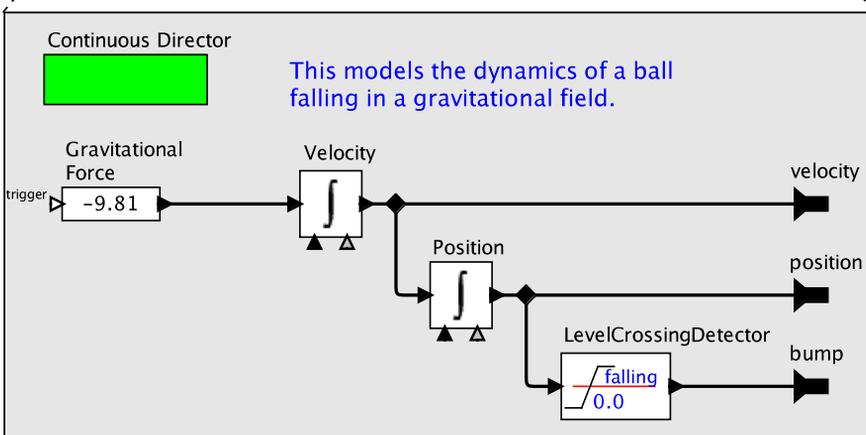
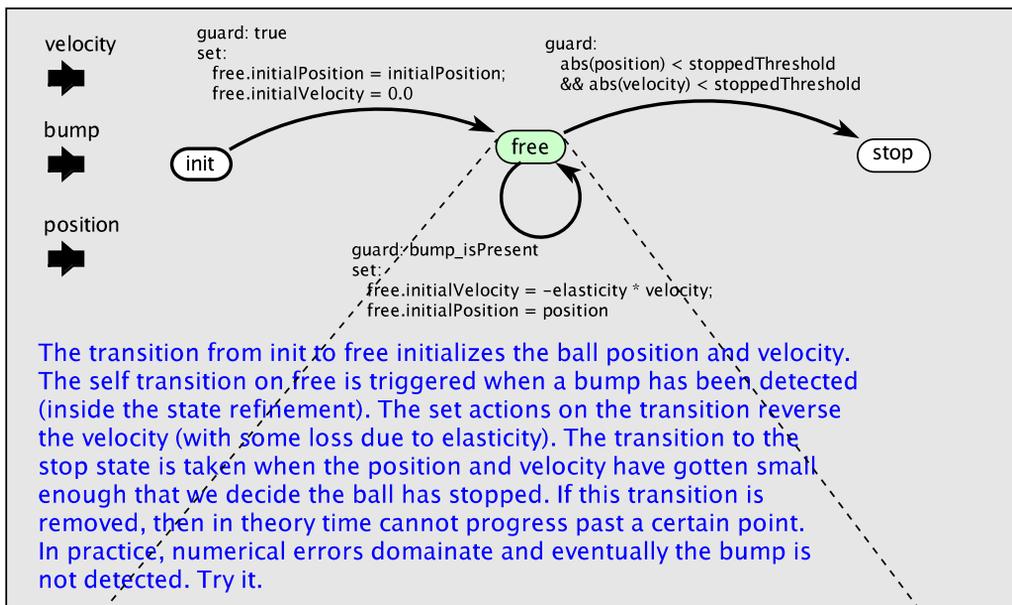


Figure 9.21: Inside the Ball Model of Figure 9.20.

shown at the bottom of Figure 9.21. The *init* state is the initial state, which is used only for its outgoing transition, and has **set actions** to initialize the ball model. Specifically, the transition is labeled as follows:

```
guard: true
set:
  free.initialPosition = initialPosition;
  free.initialVelocity = 0.0
```

The first line is a **guard**, which is a predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value `true`. Thus, the model immediately transitions to mode *free*. This transition occurs in microstep zero at the start of the execution. The “set:” line indicates that the successive lines define **set actions** (see Section 6.2). The third and fourth lines set the parameters of the destination mode *free*. The *free* state represents the mode of operation when the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

When the model begins executing, it is in the *init* state. Since the *init* state has no refinement, the outputs of the Ball Model will be absent while the modal model is in that state. The outgoing transition has a guard that is always enabled, so the Ball Model will be in that state for only one microstep.

Inside the *free* state, the refinement represents the law of gravity, which states that an object of any mass will have an acceleration of about  $9.81\text{meters/second}^2$ . The acceleration is integrated to find the velocity, which is, in turn, integrated to find the vertical position. In the refinement, a **LevelCrossingDetector** actor is used to detect when the vertical position of the ball is zero. Its output produces events on the (discrete) output port *bump*. Figure 9.21 shows that this event triggers a state transition back to the same *free* state, but now the *initialVelocity* parameter is changed to reverse the sign and attenuate its value by the *elasticity*. The ball loses energy when it bounces, as shown by the plot in Figure 9.20.

Figure 9.21 shows that when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop*, which has no refinement. At this point, the model produces no further outputs.

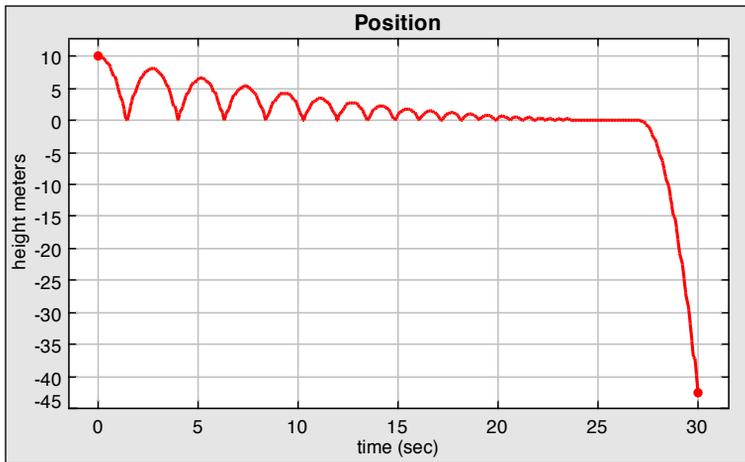


Figure 9.22: Result of running the bouncing ball model without the stop state.

The bouncing ball model illustrates an interesting property of hybrid system modeling. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. Removing the *stop* state from the FSM and re-running the model yields the result shown in Figure 9.22. In effect, the ball falls through the surface on which it is bouncing and then goes into a free-fall in the space below.

The error that occurs here illustrates a fundamental pitfall that can occur with hybrid system modeling. In this case, the event detected by the [LevelCrossingDetector](#) actor can be missed by the simulator. This actor works with the solver to attempt to identify the precise point in time when the event occurs. It ensures that the simulation includes a sample at that time. However, when the numbers become sufficiently small they are dominated by numerical errors, and the event is missed.

The bouncing ball is an example of a [Zeno](#) model (see Section 7.4). The time between bounces gets smaller as the simulation progresses, and it gets smaller fast enough that, with infinite precision, an infinite number of bounce events would occur in a finite amount of time.

### 9.3.1 Hybrid Systems and Discontinuous Signals

Recall from Example 9.3 that actors whose outputs are a discontinuous function of the input can create signals that are not *piecewise continuous*. This can result in solver-dependent behavior, in which arbitrary step-size decisions made by the solver strongly affect the execution of the model. These problems can be solved using modal models, as illustrated in the following example.

**Example 9.10:** Figure 9.23 shows a variant of the model in Figure 9.10 that correctly produces a piecewise continuous signal. This variant uses a *ModalModel*, which specifies a transition at the discontinuity of the signal. The transitions of a modal model are instantaneous, in that *model time* does not advance. The *microstep*, however, does advance. In this model, the transition occurs within the *errorTolerance* (a director parameter) after time 1.0. At the time of the transition, the refinement of the *zero* state fires first, producing output 0 at microstep 0, and then the refinement of the *increment* state fires at microstep 1, producing output 2.0 (or within the *errorTolerance* of 2.0). Hence, the output signal is piecewise continuous.

The operation of *ModalModel* actors is explained in Chapter 8. When combined with the Continuous director, such operation translates naturally into an effective and useful semantics for hybrid systems. To fully understand the interoperation of modal models and the Continuous domain, it is useful to review the execution semantics of modal models, described in Section 6.2. Specifically, a firing of the modal model consists of firing of the refinement of the current state (if there is one), evaluating the guards, and taking a transition if a guard is true. It is also important to understand that while a mode is inactive, time does not advance in the refinement. Thus, the local notion of time within a refinement lags the global notion of time in its environment.

In modal models, transitions are allowed to have *output actions* (see Section 6.2). Such actions should be used with care because the transition may be taken in microstep 0, and the resulting output will not be piecewise continuous. If output actions are used to produce discrete events, the transition must be triggered by a discrete event from a piecewise continuous signal.

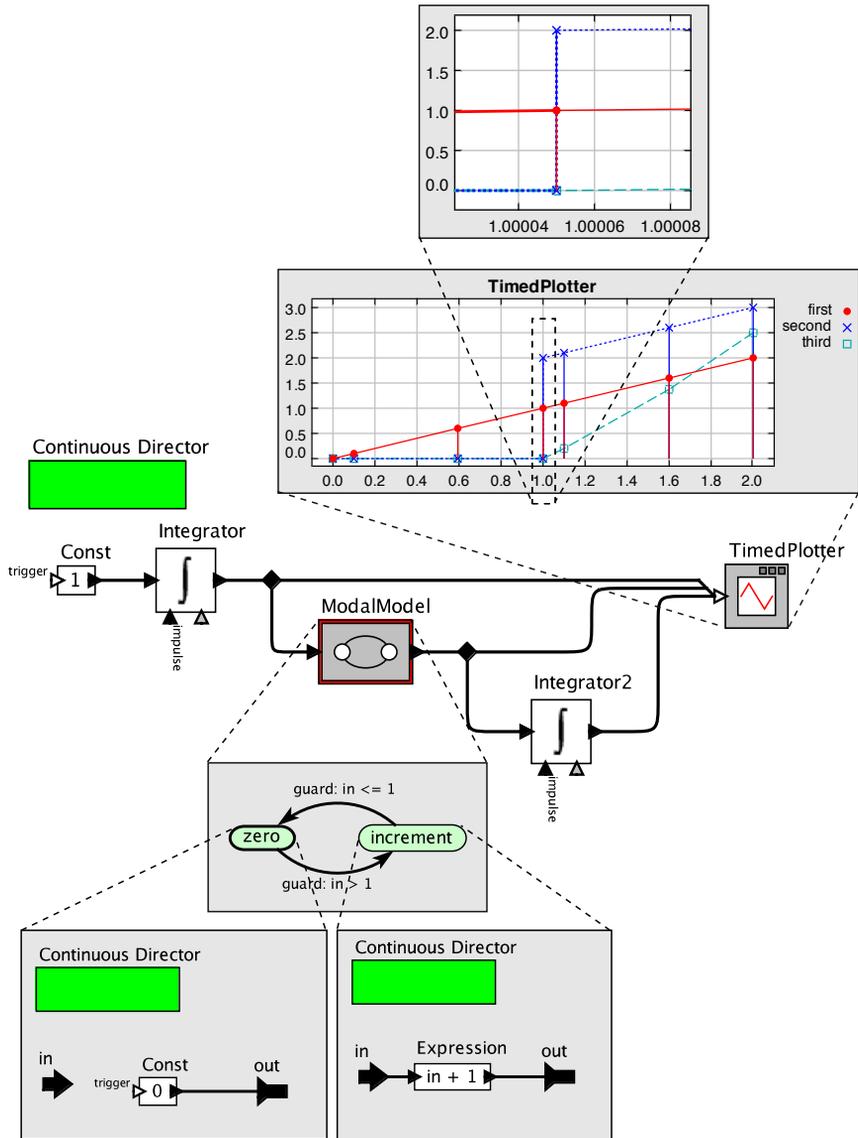


Figure 9.23: A variant of the model in Figure 9.10 that correctly produces a piecewise continuous signal. [\[online\]](#)

## 9.4 Summary

Modeling continuous-time systems and approximating their behavior on digital computers can be tricky. The [superdense time](#) model of Ptolemy II makes it easier to accurately model a large class of systems, and is particularly useful for systems that mix continuous and discrete behaviors. The Continuous domain, described in this chapter, exploits this model of time to deliver sophisticated modeling and simulation capabilities.

## Exercises

- Let  $x$  be a continuous-time signal where  $x(0) = 1$  and  $\ddot{x}(t) = -x(t)$ , where  $\ddot{x}$  is the second derivative of  $x$  with respect to time  $t$ . It is easy to verify that a solution to this equation is  $x(t) = \cos(t)$ .
  - Use [Integrator](#) actors to construct this signal  $x$  without using any actors or expressions involving trigonometric functions. Plot the execution over some reasonable time to verify that the solution matches what theory predicts.
  - Change the solver that the director uses from `ExplicitRK23Solver` to `ExplicitRK45Solver`. Describe qualitatively the difference in the results. Which solver gives a better solution? What criteria are you using for “better”? Give an explanation for the differences.
  - All numerical ODE solvers introduce errors. Although the theory predicts that the amplitude of the solution  $x(t) = \cos(t)$  remains constant for all time, a numerical solver will be unable to sustain this. Describe qualitatively how the `ExplicitRK23Solver` and `ExplicitRK45Solver` perform over the long run, leaving other parameters of the director at their default values. Which solver is better? By what criteria?
  - Experiment with some of the other director parameters. How does the *errorTolerance* parameter affect the solution? How about *maxStepSize*?
- Example 9.2 shows the use of [ContinuousTransferFunction](#) to specify a transfer function for a continuous-time system. Show that with the parameters given in the example, that the models in Figures 9.4 and 9.6 are equivalent. **Hint:** This problem is easy if you have taken a typical electrical engineering signals and systems class, but it is doable without that if you recognize the following fact: If a signal  $w$  has Laplace transform  $W$ , then the integral of that signal has Laplace transform  $W'/s$  where for all complex numbers  $s$ ,  $W'(s) = W(s)/s$ . That is, dividing by  $s$  in the Laplace domain is equivalent to integrating in the time domain.
- Consider the [Lorenz attractor](#) in Example 9.1. Implement the same system using the [DifferentialSystem higher-order actor](#). Give the parameter names and values for your `DifferentialSystem`.
- The model in Example 9.7 does not actually require the Continuous domain to achieve the same functionality. Construct an equivalent model that is purely a DE model.

# Modeling Timed Systems

Janette Cardoso, Patricia Derler, John C. Eidson, Edward A. Lee, Slobodan Matic,  
Yang Zhao, Jia Zou

## Contents

---

<b>10.1 Clocks</b> . . . . .	<b>357</b>
<b>10.2 Clock Synchronization</b> . . . . .	<b>361</b>
<b>10.3 Modeling Communication Delays</b> . . . . .	<b>365</b>
<i>Sidebar: Precision Time Protocols</i> . . . . .	367
10.3.1 Constant and Independent Communication Delays . . . . .	368
10.3.2 Modeling Contention for Shared Resources . . . . .	368
<i>Sidebar: Decorators</i> . . . . .	373
10.3.3 Composite Aspects . . . . .	375
<b>10.4 Modeling Execution Time</b> . . . . .	<b>378</b>
<b>10.5 Ptimes for Distributed Real-Time Systems</b> . . . . .	<b>380</b>
10.5.1 Structure of a Ptimes Model . . . . .	381
<i>Sidebar: Background of Ptimes</i> . . . . .	382
10.5.2 Ptimes Components . . . . .	389
<i>Sidebar: Safe-to-Process Analysis</i> . . . . .	390
<b>10.6 Summary</b> . . . . .	<b>393</b>
<b>10.7 Acknowledgements</b> . . . . .	<b>393</b>

---

---

This chapter is devoted to modeling timing in complex systems. We begin with a discussion of clocks, with particular emphasis on multiform time. We then illustrate how to use multiform time in three particular modeling problems. First, we consider clock synchronization, where network protocols are used to correct clocks in distributed systems to ensure that the clocks progress at approximately the same rates. Second, we consider the problem of assessing the effect of communication delays on the behavior of systems. And third, we consider the problem of assessing the effect of execution time on the behavior of systems. We then conclude the chapter with an introduction to a programming model called Ptides that makes possible systems whose behavior is unaffected by variations in the timing of computation and networking, up to a point of failure. The Ptides model of computation enables much more deterministic [cyber-physical systems](#).

As a preface to this chapter, we issue a warning to the reader. Discussing the modeling of timing in cyber-physical systems can be very confusing, because in such models, time is intrinsically [multiform](#). Several distinct views and measurements of time may simultaneously coexist, making the use of words like “when” and phrases such as “at the same time” treacherous.

The most obvious source of temporal diversity is in the distinction between [real time](#) and [model time](#). By “real time” we mean here the time that elapses while a model executes, or while the system that the model is supposed to model executes. If the execution of the model is a [simulation](#) of some physical system, then “real time” may refer to the time elapsing in the world where the simulation is executing (e.g. the time that your wristwatch measures while you watch a simulation run on your laptop). Model time, by contrast, exists within the simulation and advances at a rate that bears little relationship with real time.

But even this can be confusing, because the physical system being simulated may be a real-time system, in which case, model time is a simulation of real time. But not the same real time that your wristwatch is measuring. Worse, within a simulation of a cyber-physical system, there may be a multiplicity of time measuring devices. There is no single wristwatch. Instead, there are clocks on microcontroller boards and in networking infrastructure. These may or may not be synchronized, but even if they are synchronized, the synchronization is inevitably imperfect, and modeling the imperfections may be an important part of the model. As a consequence, a single model may have several distinct timelines against which the components of the system are making progress. Moreover, as discussed in Chapter 8, [modal models](#) lead to some timelines becoming frozen, while others progress. Keeping these multiple timelines straight can be a challenge. This is the primary topic of this chapter.

## 10.1 Clocks

As explained in Section 1.7, Ptolemy II provides a coherent notion of time across **domains**. Ptolemy II supports multiform time. Every **director** contains a **local clock** that keeps track of the **local time**. The local time is initialized with the *startTime* of the director and evolves at a given *clockRate*. The *clockRate* can change.

The parameter dialogue of a simple director is shown in figure 10.1 (most directors have more parameters than these, but every director has at least these). The *startTime* parameter, if given, specifies the time of the local clock when the model is initialized. If it is not given, then the time at initialization will be set to the time of the environment (the **enclosing director**, or the next director above in the **hierarchy**), or will be set to zero if the director is at the **top level** of the model. When the local time of the director reaches the value described by *stopTime*, the director will request to not be fired anymore (by returning *false* from its **postfire** method).

The parameter dialog of a director also contains a Configure button for configuring the local clock, as shown in Figure 10.1. This can be used to set the **time resolution**, which is explained in Section 1.7.3, and the **clock rate**. The *clockRate* parameter specifies how rapidly the local clock progresses relative to the clock of the enclosing director. A director refers to the time of the enclosing director as the **environment time**. If there is no enclosing director, then advancement of the clock is entirely controlled by the director.

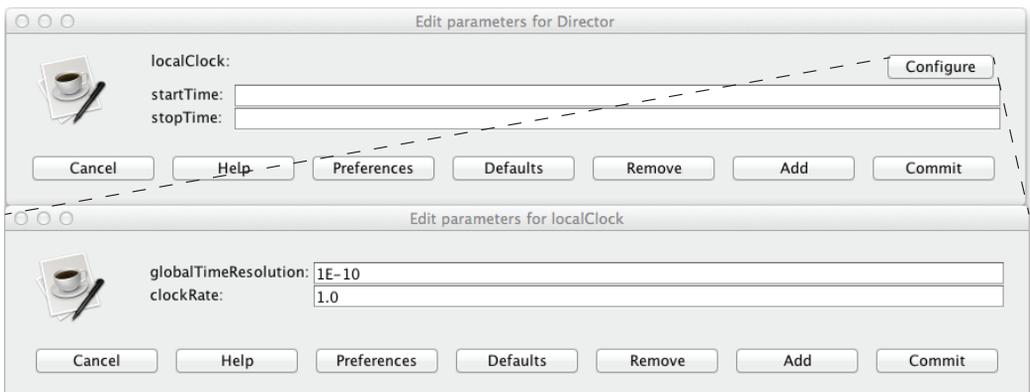


Figure 10.1: The director parameters for the local clock.

For example, if a **DE** director has an enclosing director, then the *clockRate* is used to translate the **time stamps** of input events from the environment into local time stamps. If it has no enclosing director, then all events are generated locally, and the director will always advance time to the least time stamp of unprocessed events.

Every director in Ptolemy has a local clock. If an untimed director such as **SDF** or **SR** has no enclosing director, then the clock value never changes (unless its *period* parameter is set to a non-zero value).

**Example 10.1:** Figure 10.2 shows four different time lines of clocks *c1* through *c4*. The clock *c1* (solid red line) represents a clock that evolves uniformly with the environment time. Clocks *c2* through *c4* have varying clock rates, clock values and offsets. During the first 5 time units, all clocks evolve with the same rate as the environment time. Clock *c3* starts with an offset of  $-5.0$ , i.e. it is 5 time units behind environment time. At environment time 5, the clock rates of *c2* and *c3* are modified; the clock rate of *c2* is increased and the clock rate of *c3* is decreased. Clock *c4* is suspended, so that its value does not change during the next 3 time units. At time 8, *c4* is resumed. At time 10, the value of *c3* is set to 10 to match the environment time. Because the clock rate of *c3* is still less than 1.0, the clock immediately starts lagging.

These different clock behaviors can be modeled in Ptolemy. We can perform the following actions on clocks: define an **offset**, change the **clock value**, **suspend** and **resume** the clock, and change the **clock rate**.

**Example 10.2:** The model that generates the plot shown in Figure 10.2 is presented in Figure 10.3. The clock rate is modified by changing the parameter *clockRate* of the parameter *localClock* in a director. In the Fast composite actor at the upper right, that parameter is set equal to the **port parameter** *rate*, so that each time a new rate is provided on that input port, the rate of the local clock changes. RegularToFast is a **DiscreteClock** actor that starts the clock with rate 1.0 at time 0.0, then changes the rate to 1.5 at time 5.0.

The clock value is modified by changing the value of the parameter *startTime* of a director. Modifying the parameter *startTime* any time during the simulation will

set the current value of the clock to the value in the *startTime* parameter. The *SlowWithOffset* composite actor at the middle right has a *port parameter* called *clockValue*, and its director's *startTime* is set to the expression `clockValue` to reference the port parameter. Whenever a new value arrives at that port parameter, the clock value gets set. The *Offset* actor at the left sets the clock to  $-5.0$  at environment time  $0.0$ , to  $2.5$  at environment time  $10.0$ , and to  $10.0$ , again at environment time  $10.0$ . The latter update leverages the *superdense time* model in Ptolemy II to instantaneously change the clock value from one value to another, as indicated by the vertical dashed line segment in Figure 10.2.

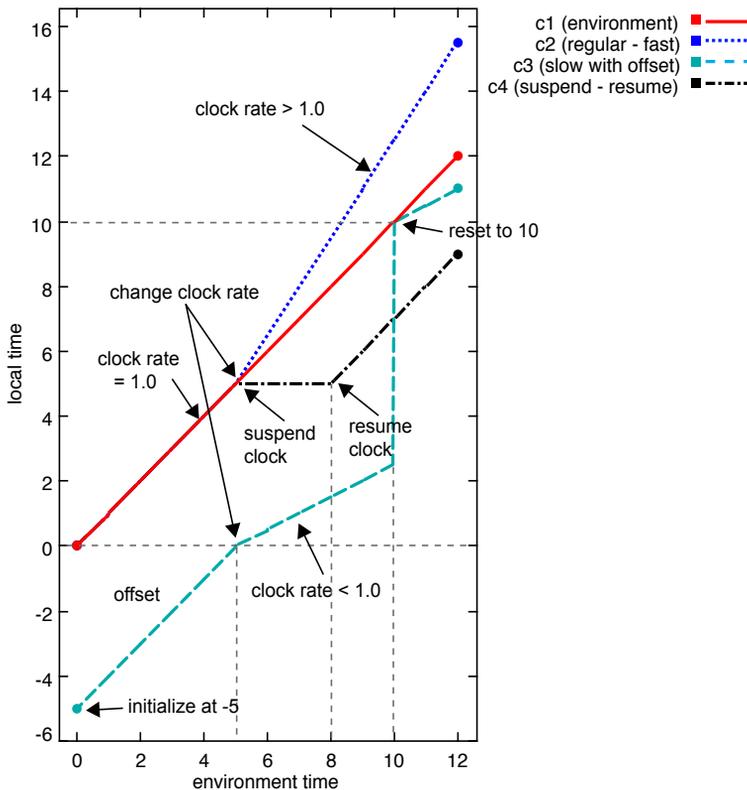


Figure 10.2: Clocks progressing at different rates relative to environment time.

The SuspendAndResume actor at the lower right is a **modal model** where the clock of the inside **Continuous** director is suspended when the state machine is not in the *active* state, resulting in the horizontal segment of the dash-dot line in Figure 10.2. Notice that the transition entering the *active* state is a **history transition** to prevent the local clock from being reset to its start time (if given) or the environment time (if a start time is not given).

The local time of a director can be plotted by using a **CurrentTime** actor with the *useLocalTime* parameter set to true (which is the default). If the *useLocalTime* parameter is set to false, then the output produced will be the environment time of the **top level** of the model. The time lines in Figure 10.2 are obtained by periodically triggering these CurrentTime actors.

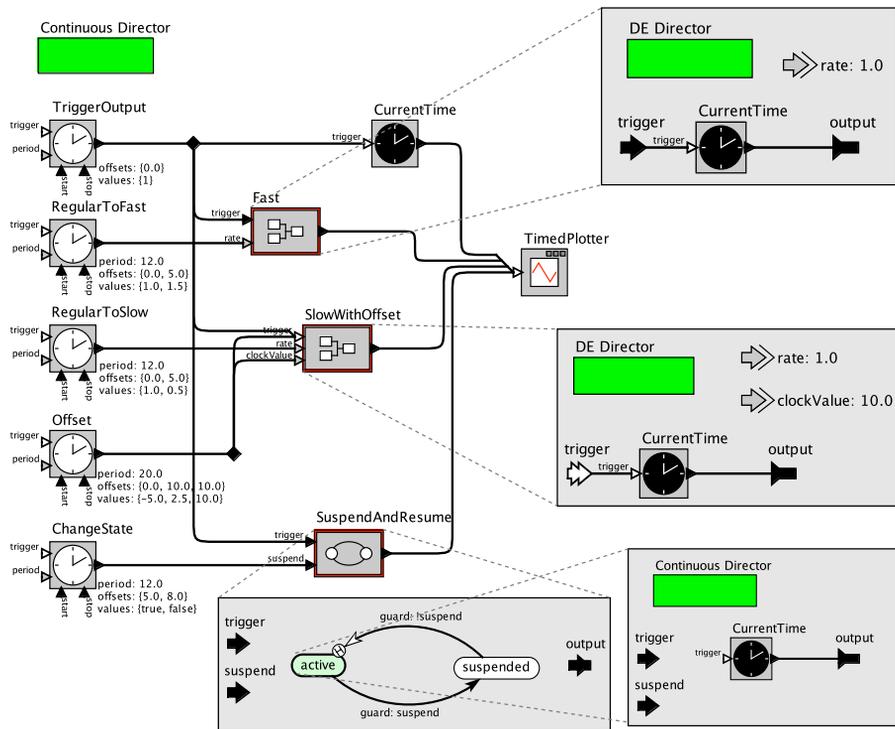


Figure 10.3: The model that generates the plot of Figure 10.2. [\[online\]](#)

## 10.2 Clock Synchronization

Many distributed systems rely on a common notion of time. A brute-force technique for providing a common notion of time is to broadcast a clock over the communication network. Whenever any component needs to know the time, it consults this broadcast clock. A well-implemented example of this is the **global positioning system (GPS)**, which with some care can be used to synchronize widely distributed clocks to within about 100 nanoseconds. This system relies on atomic clocks deployed on a network of satellites, and careful calculations that even take into account relativistic effects. GPS, however, is not always available to systems (particularly indoor systems), and it is vulnerable to spoofing and jamming. More direct and self-contained realizations of broadcast clocks may be expensive and difficult to implement, since lack of control over communication delays can render the resulting clocks quite inaccurate. In addition, avoiding brittleness in such systems, where the source of the clock becomes a single point of failure that can bring down the entire system, may be expensive and a significant engineering challenge (Kopetz, 1997; Kopetz and Bauer, 2003).

A more modern technique that improves robustness and precision is to use **precision time protocols (PTP)** to provide **clock synchronization**. Such protocols keep a network of loosely coupled clocks synchronized by exchanging time-stamped messages that each clock uses to make small corrections in its own rate of progress. This technique is more robust, because sporadic failures in communication have little effect, and even with permanent failures in communication, clocks can remain synchronized for a period of time that depends on the stability of the clock technology.

This technique is also usually more precise than what is achieved by a broadcast clock; for most such protocols, the achievable precision does not depend on the communication delays, but rather instead depends on the *asymmetry* of the communication delays. That is, if the latency of communication from point *A* to point *B* is exactly the same as the latency of the communication from point *B* to point *A*, then perfect clock synchronization is theoretically possible. In practice, such protocols can come quite close to this theoretical limit over practical networks. The White Rabbit project at CERN, for example, claims to be able to synchronize clocks on a network spanning several kilometers to under 100 picoseconds (Gaderer et al., 2009). This means that if you simultaneously ask two clocks separated by, say, 10 kilometers of networking cable, what time it is, their response will differ by less than 100 picoseconds. Over standard Ethernet-based local area networks, it is routine today to achieve precisions well under tens of nanoseconds using a PTP known

as IEEE 1588 (Eidson, 2006). Over the open Internet, it is common to use a PTP known as NTP (Mills, 2003) to achieve precisions on the order of tens of milliseconds.

Typically, one or several master clocks are elected (and reelected in the event of failure), and slaves synchronize their clocks to the master by messages sent over the network. This guarantees a common notion of time across all platforms, with a well-defined error margin. The following example simulates the effects of imperfect clock synchronization.

**Example 10.3:** In electric power systems, a transmission line may span many kilometers. When a fault occurs, for example due to a lightning strike, finding the location of the fault may be very expensive. Hence, it is common to estimate the location of the fault based on the time that the fault is observed at each end of the transmission line. Assume a transmission line of length 60 kilometers between substation A and substation B. When a fault occurs, both substations will experience an observable event. Assuming that electricity travels through the transmission line

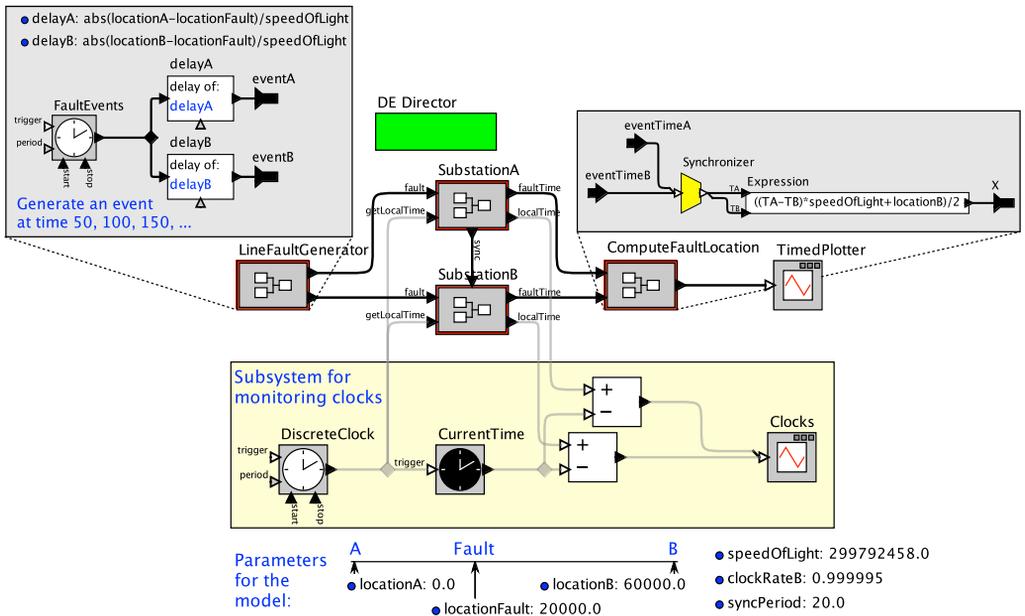
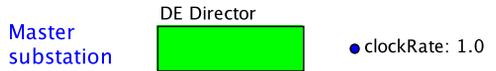
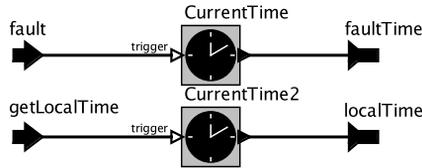


Figure 10.4: Line fault detection model. [online]



These provide a mechanism for the environment to query for the local time:



Periodically send the current time to slave substation(s):

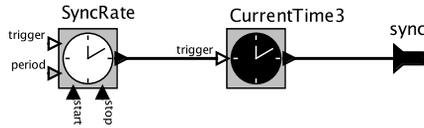


Figure 10.5: Line fault detection — Substation A, the clock master.

Controller to adjust the local clock rate to match that of the master.

● clockRate: 1.0000000016975

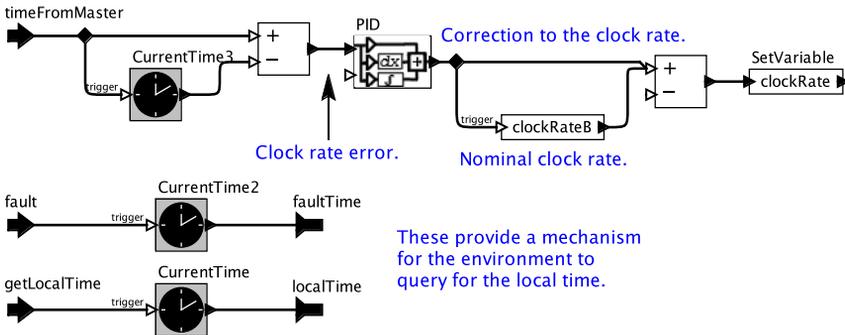


Figure 10.6: Line fault detection — Substation B, the clock slave.

at a known speed, then the time difference between when substation A observes the event and substation B observes the event can be used to calculate the location of the event.

Let  $X$  denote the location of the fault event along the transmission line (the distance from substation A). Then  $X$  satisfies the following equations,

$$\begin{aligned} s \times (T_A - T_0) &= X \\ s \times (T_B - T_0) &= D - X \end{aligned}$$

where  $T_0$  is time of fault (which is unknown),  $T_A$  is the time of fault detection at substation A,  $T_B$  is the time of fault detection at substation B,  $s$  is the speed of propagation along the transmission line (the speed of light),  $D$  is the distance from A to B, and  $D - X$  is the distance from B to the fault. Subtracting the above equations and solving for  $X$  yields

$$X = ((T_A - T_B) \times s + D)/2.$$

Of course, this calculation is only correct if the clocks at the two substations are perfectly synchronized. Suppose that the substations use a PTP to synchronize their clocks, and that substation A is the master. Then periodically, A and B will exchange messages that can be used to compute the discrepancy between their clocks. To see how this is done, see the sidebar on page 367 or [Eidson \(2006\)](#). Here, we will assume that this done perfectly (something that is only possible if the communication latency between A and B is perfectly symmetric). We focus in this model only on the effects of the control strategy that uses this information to adjust the clock of substation B. The model shown in Figure 10.4 shows SubstationA periodically sending its local time to SubstationB, where the period is given by the *syncPeriod* parameter, set to 20.0 seconds.

In that model, the LineFaultGenerator produces faults at times 50, 100, 150, etc., and the fault is assumed to occur 20 kilometers from substation A. The substation actors send the local times at which they observe the faults to a ComputeFaultLocation composite actor, whose task it is to determine the fault location using the above formulas. Since the measured fault times arrive at the ComputeFaultLocation actor at different times, a [Synchronizer](#) is used to wait until one data value from each substation has been received before it will do a calculation. Note that inputs will be misaligned if one of the substations fails to detect the event and provide an input, so a more realistic model needs to be more sophisticated.

The substation models are depicted in Figures 10.5 and 10.6. SubstationA is modeled very simply as, from top to bottom, responding to a *fault* input by sending the time of the fault, responding to a *getLocalTime* input with the local time, and periodically sending the local time to the *sync* output.

Substation *B* is a bit more complicated. When it receives a *sync* signal from the master, it calculates the discrepancy with its local clock; this calculation is not realistic, since there is an unknown time delay in receiving the *sync* signal, but PTP protocols take care of making this calculation, so this detail is not modeled here. Instead, the model focuses on what is done with the information, which is to use a **PID** controller to generate a correction to the local clock. The correction is added to the local clock rate and then stored in the *clockRate* parameter using a **SetVariable** actor. The director's clock uses this same parameter for the rate of its clock, so each time a correction is made, the rate of the local clock will change.

Figure 10.7 shows the simulation result. The upper plot shows the errors in the clocks. Since substation *A* is the master, it has no error, so its error is a constant zero. At the start of simulation, the clock of *B* is drifting linearly with respect to *A*. At 20 seconds, *B* receives the first *sync* input, and the **PID** controller provides a correction that reduces the rate of drift. At 40 seconds, another *sync* signal further reduces the rate of drift. The lower plot shows the estimated locations of the faults occurring at times 50, 100, 150, etc. The correct fault location is 20 km, so we can see that as the clocks get synchronized, the estimate converges to 20 km.

Figure 10.8 shows what happens if there is no clock synchronization (the *sync* signal never arrives at *B*). In this case, the clock of *B* drifts linearly with respect to *A*, and the error in the estimated fault location grows without bound.

## 10.3 Modeling Communication Delays

In design-space exploration, designers evaluate whether their designs work well on a given architecture. Part of the architecture is the communication network, which introduces delays that can affect the behavior of a system. A communication network introduces delay. It is straightforward to model constant communication delays that are independent of one another, but it is much more interesting (and realistic) to take into account shared resources, which result result in correlated and variable delays. We begin with the simpler models, and then progress to the more interesting ones.

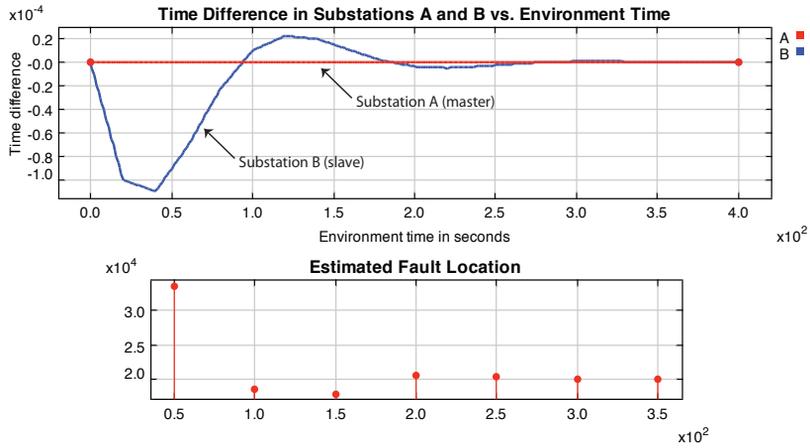


Figure 10.7: Line fault detection with clock synchronization.

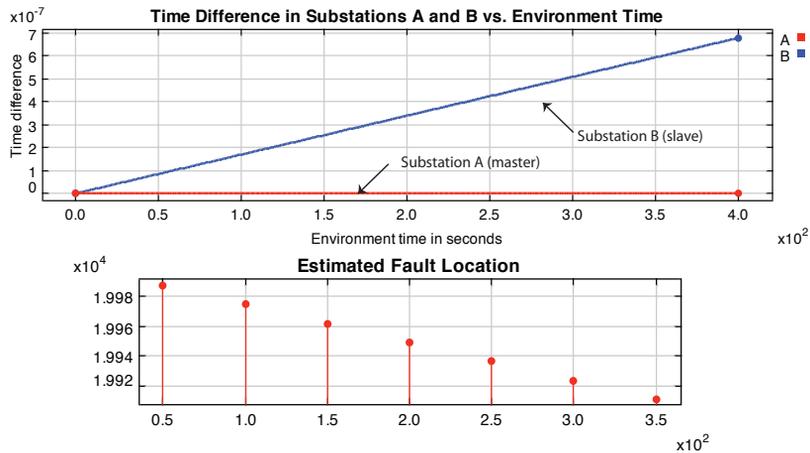


Figure 10.8: Line fault detection without clock synchronization.

### Sidebar: Precision Time Protocols

The figure at the right shows how a typical PTP works. The clock master  $A$  initiates an exchange of messages. The first message is sent at time  $t_1$  (by the master clock) and contains the value of  $t_1$ . That message is received by the slave  $B$  at time  $t_2$  (by the master clock), but since the slave does not have access to the master clock, the slave records the time  $t'_2$  that it receives the message according to its own clock. If its clock is off by  $e$  vs. the master clock, then

$$t'_2 = t_2 + e.$$

The slave responds by sending a message back to the master at time  $t'_3$  according to its clock, or  $t_3$  according to the master's clock, so

$$t'_3 = t_3 + e.$$

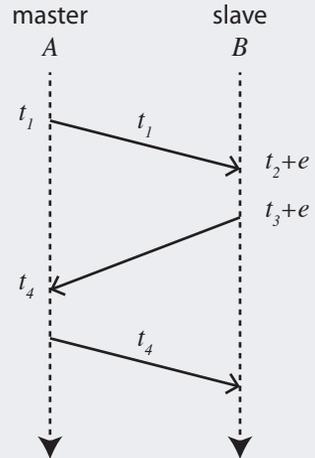
The master receives this second message at  $t_4$ , and replies with a third message containing the value of  $t_4$ . The slave now has  $t_1, t'_2, t'_3$ , and  $t_4$ . Now notice that the round trip communication latency (the time that a message from  $A$  to  $B$  and a reply message spend in transport) is

$$r = (t_2 - t_1) + (t_4 - t_3) = (t_4 - t_1) - (t_3 - t_2).$$

At  $B$ , this value can be calculated even though  $t_2$  and  $t_3$  are not known, because  $(t_3 - t_2) = (t'_3 - t'_2)$ , and slave  $B$  has  $t'_2$ , and  $t'_3$ . If the communication latencies are symmetric, then the one way latency is  $r/2$ . To correct the clock of  $B$ , we simply need to estimate  $e$ . This will tell us whether the clock is ahead or behind, so we can slow it down or speed it up, respectively. If the communication channel has symmetric delays (i.e.  $t_2 - t_1 = t_4 - t_3$ ), then a very good estimate is given by

$$\tilde{e} = t'_2 - t_1 - r/2.$$

In fact, if the communication latency is exactly symmetric, then  $\tilde{e} = e$ , the exact clock error.  $B$  can now adjust its local clock by  $\tilde{e}$ .



### 10.3.1 Constant and Independent Communication Delays

In the [DE](#) domain, network delays that are independent of one another can be easily modeled using the [TimeDelay](#) actor.

**Example 10.4:** The line fault detector of [Figure 10.4](#) idealizes the calculation of the clock error in substation *B*. In practice, calculating clock discrepancies is not trivial. A typical technique implemented in a PTP is described in the sidebar on [page 367](#) and implemented in the model in [Figure 10.9](#).

In [Figure 10.9](#), substation *A* periodically initiates a sequence of messages that are used to calculate the clock discrepancy. First, at master time  $t_1$ , it sends the value of this time to substation *B*. Substation *B* responds. Substation *A* responds to the response with the time  $t_4$  that it receives the response. When substation *B* has received this final message, it has enough information to estimate the discrepancy between its clock and that of the master. The [Synchronizer](#) actor ensures that this estimate is only calculated after all the requisite information has been received.

[Figure 10.9](#) has three [TimeDelay](#) actors that can model network latency in the communication of synchronization messages. Interestingly, if all three delays are set to the same value, even a rather large value such as 1.0 seconds, then the performance of the model in identifying the location of the fault is essentially identical to that of the idealized model. However, if, as shown, one of the delays is changed only slightly, to 1.0001, then the performance degrades considerably, as shown in [Figure 10.10](#). The slave clock settles into a substantial steady-state error, and the estimated fault location converges to approximately 13 kilometers, quite different from the actual fault location at 20 kilometers. Clearly, if the communication channel is expected to be asymmetric, then the designer has work to do to improve the control algorithm. A different choice of parameters for the [PID](#) controller would probably help, but perhaps at the expense of lengthening the convergence time.

### 10.3.2 Modeling Contention for Shared Resources

In the model in [Figure 10.9](#), each connection between actors has a fixed communication delay. This is not very realistic for practical communication channels, where the delay

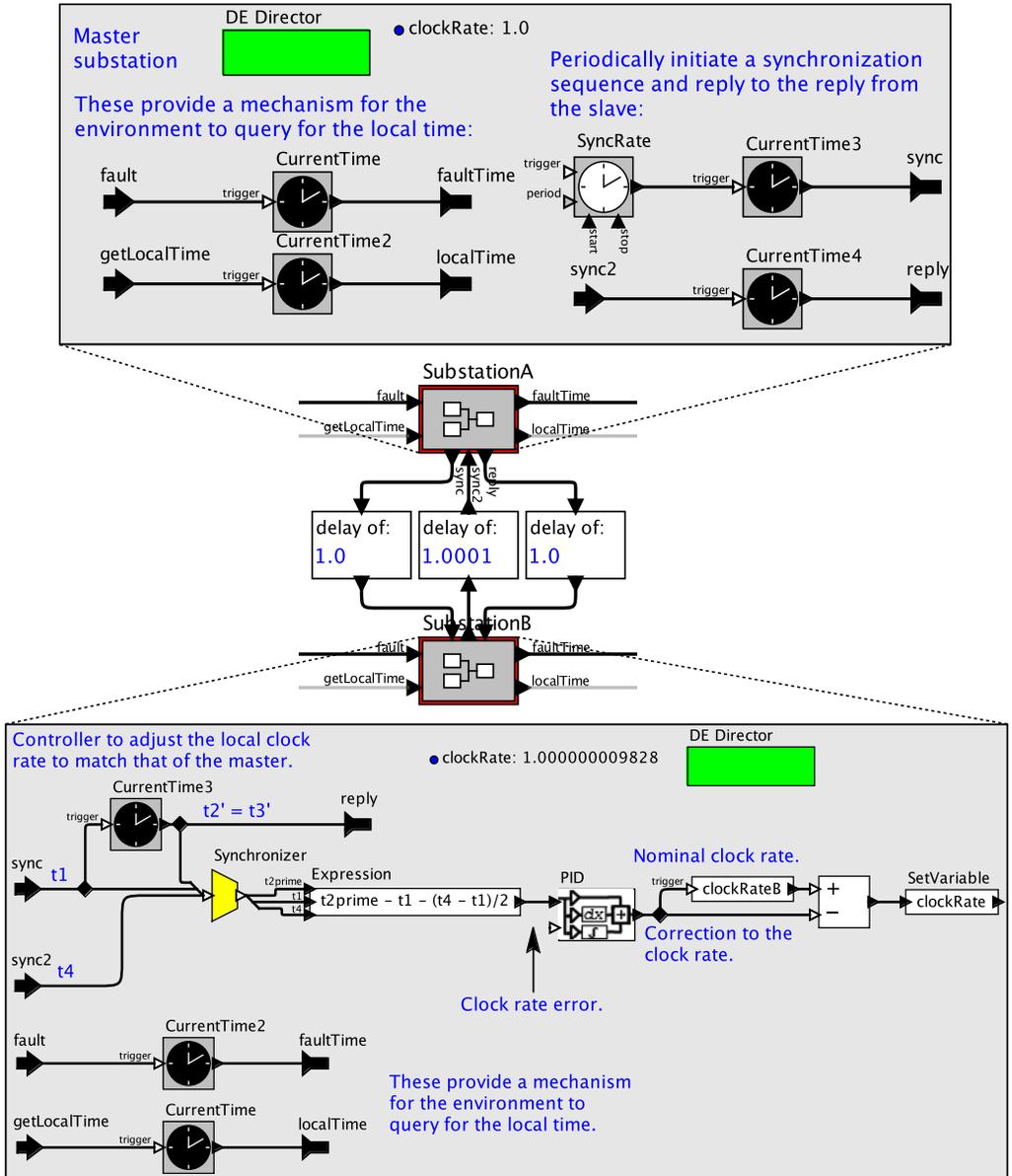


Figure 10.9: Line fault detection with communication delays in the PTP implementation. [online]

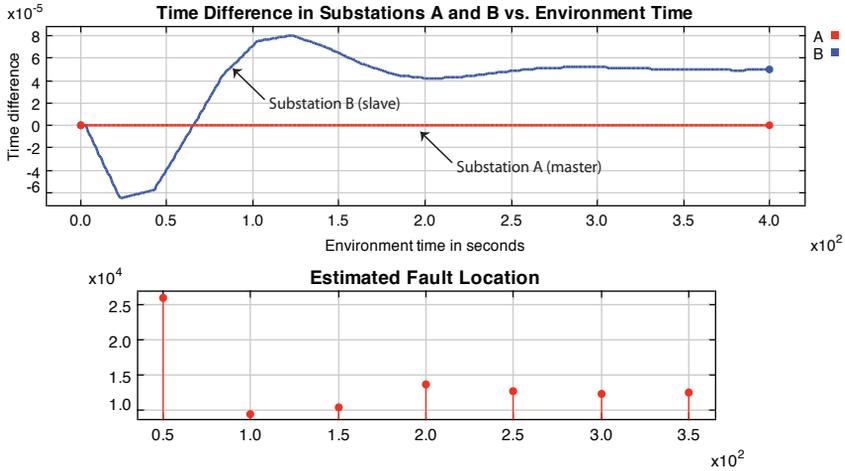


Figure 10.10: Line fault detection performs poorly with PTP when network latencies are asymmetric.

will depend on other uses of the channel. Most communication channels have shared resources (radio bandwidth, wires, buffer space in routers, etc.), and the latency through the network can vary significantly depending on other uses of these resources.

The model in Figure 10.9 could be modified to route all the messages through a single, more elaborate network model. However, this leads to considerable modeling complexity. Suppose for example that we wish to model the network using a single **Server** actor, perhaps the most basic model for a shared resource. Then all messages that traverse the channel would have to be merged into a single stream to feed to the Server actor. These streams would then have to be separated after emerging from the Server actor, so destination addresses would have to be encoded in the messages before the streams are merged. The model suddenly becomes very complicated.

Fortunately, Ptolemy II has a much cleaner mechanism for handling shared resources. We use **aspect-oriented modeling (AOM)**, which is based on aspect-oriented programming (Kiczales et al., 1997), to map functionality to implementation. This way of associating functional models with implementation models and schedulers was introduced in Metropolis (Balarin et al., 2003), where the mechanism was called a **quantity manager**. In Ptolemy II, an **aspect** is an actor that manages a resource; it is associated with the actors and ports that share the resource. In a simulation run, the aspect actor schedules the

use of the resource. The association between the resource and the users of the resource is done via parameters, not by direct connections through ports. As a consequence, aspects are added to an existing model without changing the interconnection topology of the existing model. The next example shows how **communication aspects** can be used to cleanly model shared communication resources.

**Example 10.5:** Figure 10.11 shows a variant of the line fault detector model where we have dragged into the model a communication aspect called **Bus**. In this example, the PTP communications between SubstationA and SubstationB use the shared bus. This is indicated in the figure by the annotation “Aspects: Bus” on the input ports, and by red fill in the port icon.

The Bus has a *serviceTime* parameter that specifies the amount of time that it takes a token to traverse the channel. During that time, the Bus is busy, so any further attempts to use the Bus will be delayed. The Bus therefore acts like a **Server** actor with an unbounded buffer, but since it is an aspect, there is no need for the model to explicitly show all the communication paths passing through a single instance of a **Server**.

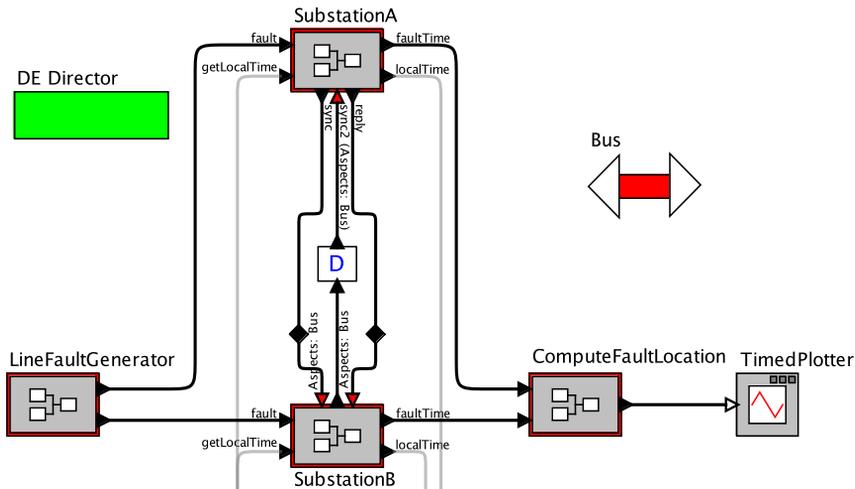


Figure 10.11: Line fault detection where communication uses a shared bus, modeled using a communication aspect. [\[online\]](#)

In this example, the communications latencies are symmetric, because there is no contention for the bus. Hence, the line fault detection algorithm performs well, behaving similarly to Figure 10.7. If, on the other hand, you enable use of the bus for other communication paths in the model, such as at the input ports of `ComputeFaultLocation`, then the performance will degrade considerably, because contention for the bus will introduce asymmetries in the communication latencies.

To use an aspect for modeling communication, simply drag one into the model from the library and assign it a meaningful name. The `Bus`, along with several others that are (as of this writing) still rather experimental, can be found in the `MoreLibraries`→`Aspects` library.

An aspect is a *decorator*, which means that it endows elements of the model with parameters (see sidebar on page 373). In the case of the `Bus`, it decorates ports with an *enable* and *messageLength* parameter, as shown in Figure 10.12. When an input port has the `Bus` enabled, then messages sent to that input port will be delayed by at least the product of the *messageLength* parameter of the port and the *serviceTimeMultiplicationFactor* parameter of the `Bus`. The delay is *at least* this, because if the bus is busy when the message is sent, then the message has to wait until the bus becomes free.

You can add any number of aspects to a model. Each is a decorator, and each can be independently enabled. If an input port enables multiple communication aspects, then

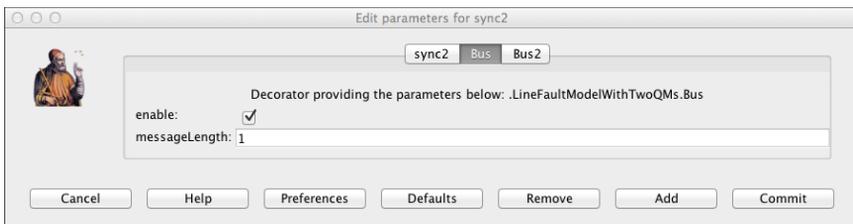


Figure 10.12: The `Bus` aspect decorates ports with an *enable* and *messageLength* parameter. This figure shows a parameter editor for a port in model in Figure 10.14, which has two busses.

those aspects mediate the communication in the order in which the aspects are enabled. Hence, aspects may be composed.

**Example 10.6:** In Figure 10.14, a second bus has been added to the model, and the communication from SubstationB to SubstationA traverses Bus and Bus2, in that order, as you can see from the annotation on the *sync2* input port to SubstationA. As a consequence, the communication latencies become asymmetric, and the line fault detection algorithm performs poorly, yielding results similar to those in Figure

### Sidebar: Decorators

A **decorator** in Ptolemy II is an object that adds to other objects in the model parameters, and then uses those parameter values to provide some service. The simplest decorator provided in the standard library is the **ConstraintMonitor**, which can be found in the `Utilities→Analysis` library. The ConstraintMonitor is an attribute that, when inserted in model, adds a parameter called *value* to actors in the model. The ConstraintMonitor keeps track of the sum of all the values that are set for actors in the model, displays that sum in its icon, and compares that sum against a *threshold*.

An example use of ConstraintMonitor is shown in Figure 10.13, where a ConstraintMonitor has been dragged into a model with three actors and renamed “Cost.” Once that ConstraintMonitor is in the model, then the parameter editing window for each actor acquires a new tab, as shown at the top of the figure, where the label on the tab matches the name of the ConstraintMonitor. The user can enter a cost for each actor in the model, and the ConstraintMonitor will display the total cost in its icon.

The ConstraintMonitor has a parameter *threshold*, which specifies a limit on the sum of the values. When the total approaches the limit, the color of the ConstraintMonitor icon changes to yellow. When the total hits or exceeds the limit, if the *warningEnabled* parameter is true, then the user is warned. The default value for *threshold* is `Infinity`, which means no limit.

The ConstraintMonitor has two other parameters, as shown at the bottom of Figure 10.13. If *includeOpaqueContents* is true, then actors inside **opaque composite actors** will also be decorated. Otherwise, they will not be decorated. If *includeTransparents* is true, then **transparent composite actors** will be decorated. Otherwise, they will not be.

There are many other uses for decorators. A director can be a decorator. The **aspects** described in this chapter are decorators.

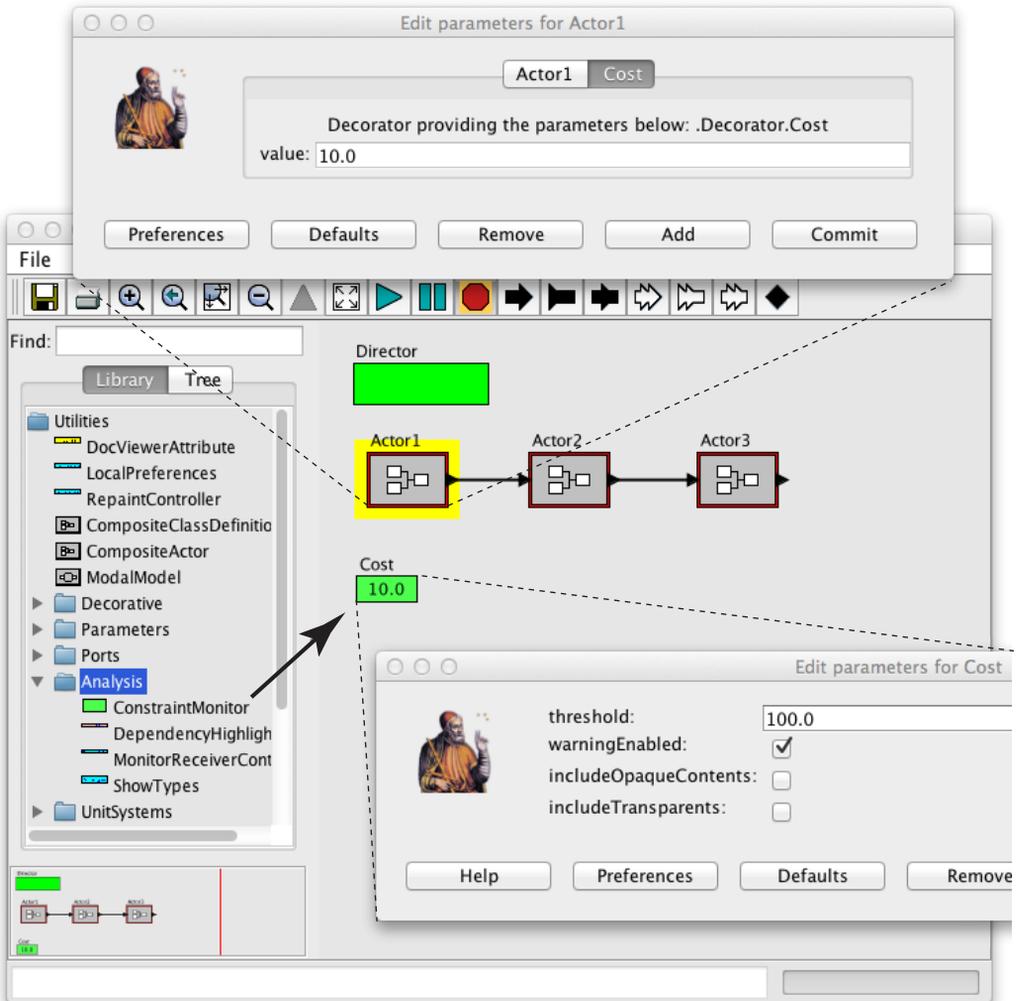


Figure 10.13: A decorator in Ptolemy II is an object that adds to other objects in the model parameters, and then uses those parameter values to provide some service. In this example, a ConstraintMonitor (which has been renamed Cost) is monitoring the total cost of components in the model, checking them against a threshold of 100.0.

10.10. The *decoratorHighlightColor* parameter of Bus2 has been changed from red to green, resulting in green highlighting of both the port and the bus icon.

### 10.3.3 Composite Aspects

The aspects discussed in the previous section are like atomic actors; their logic is defined in a Java class. For a more flexible way of describing communication aspects, we use the **CompositeCommunicationAspect**. This actor can be found in Ptolemy under `MoreLibraries→Aspects`.

**Example 10.7:** Figure 10.15 shows the bus example implemented using a **CompositeQuantityManager**. In this case, the bus behavior is modeled using a discrete-event subsystem with a **Server** actor. Requests to use the bus queue up at the input

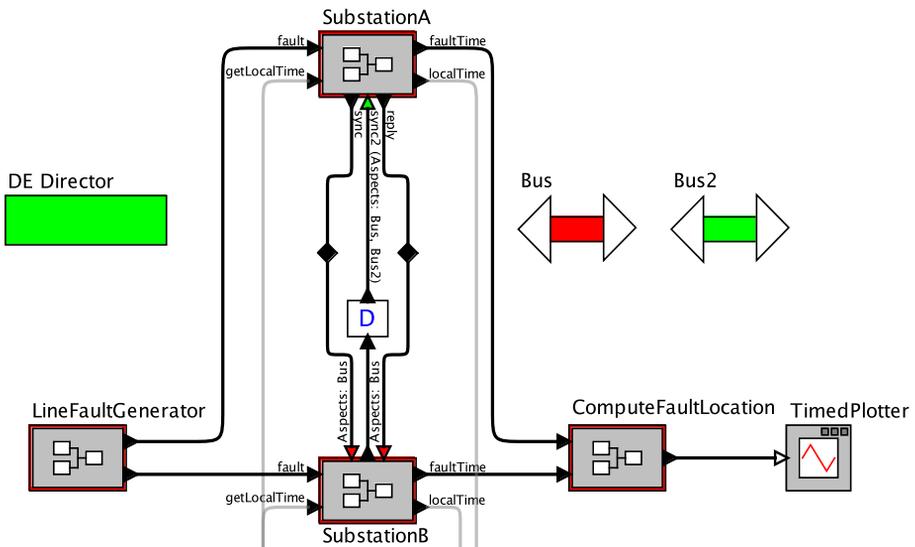


Figure 10.14: Line fault detection where one of the communications traverses two buses, yielding asymmetric communication. [\[online\]](#)

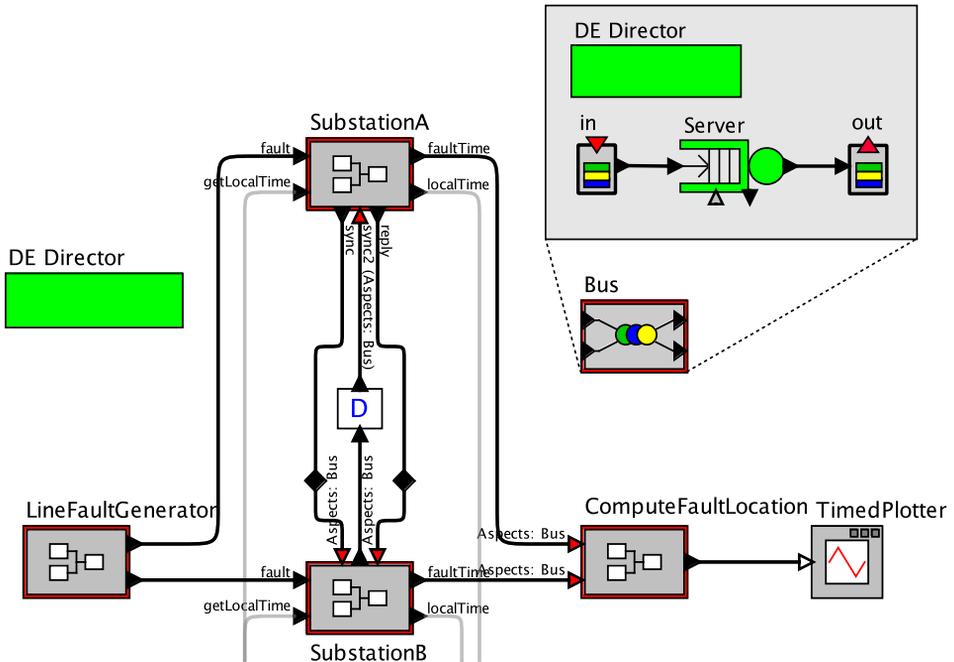


Figure 10.15: A bus implemented as a composite communication aspect. [online]

to the server. When the server becomes free, the first queued input is delayed by the *serviceTime*. The behavior is identical to that of the atomic Bus aspect.

Since a composite communication aspect is simply a Ptolemy II model, we have a great deal of freedom in its design.

**Example 10.8:** In the example of Figure 10.15, the bus is being used not only for the communication between SubstationA and SubstationB, but also in the communication to the ComputeFaultLocation actor. Contention for the bus makes the communication latencies asymmetric, degrading the performance of the clock syn-

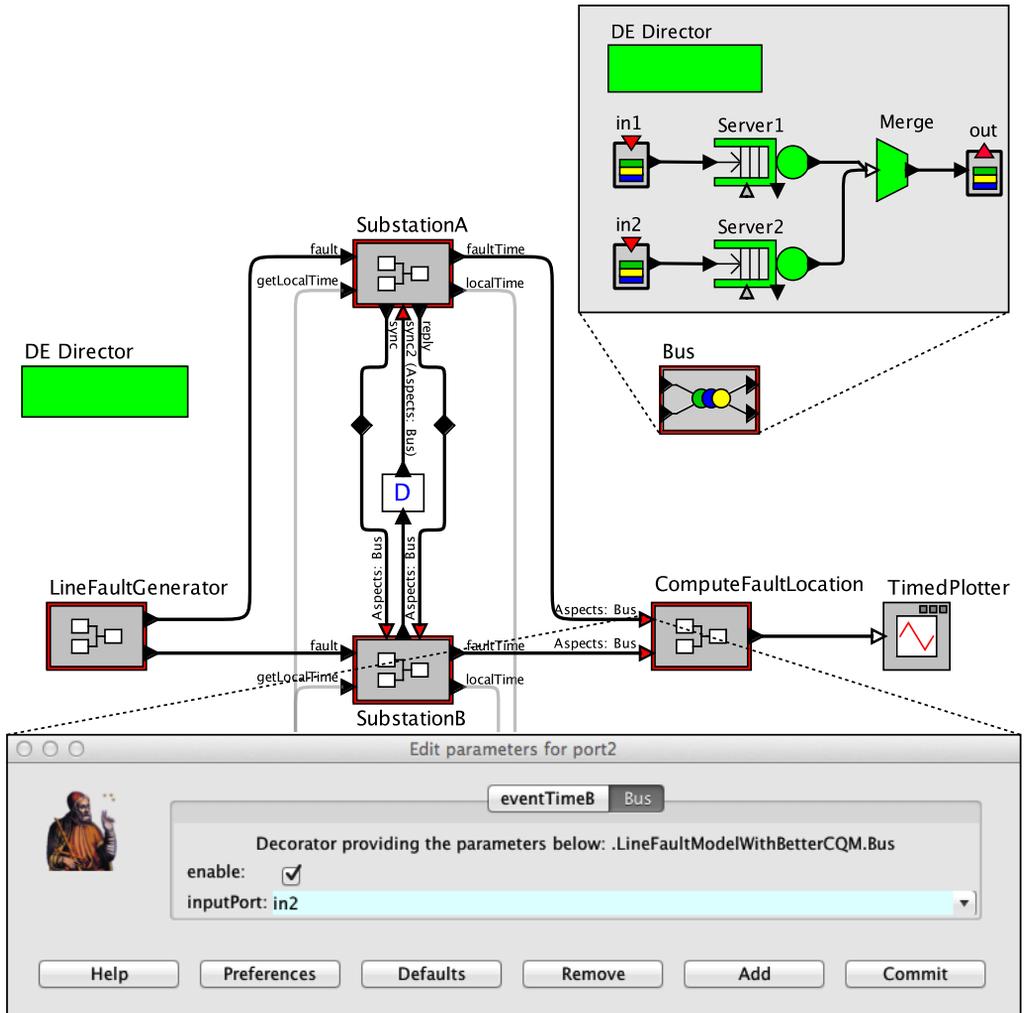


Figure 10.16: A network that reduces contention implemented as a composite aspect. At the bottom is shown how the decorator parameters of a port are used to select the port of the aspect that handles the communication. [online]

chronization, and resulting in very poor performance in computing the fault location.

We can improve the performance with a better network, as shown in Figure 10.16. In that figure, we have modified the Bus so that it is now a more sophisticated network with two input ports and two distinct servers. By routing communication to the two servers, contention can be reduced. Each input port involved in a communication specifies which input port, *in1* or *in2*, of the aspect should handle the communication. At the bottom of the figure is shown how the decorator parameters of a port are used to select the port of the aspect that handles the communication. In the figure, the top input port of `ComputeFaultLocation` is using *in2*. If the bottom port also uses *in2*, and the ports handling the communication between `SubstationA` and `SubstationB` use *in1*, then contention is reduced enough to deliver excellent performance, similar to that in Figure 10.7.

## 10.4 Modeling Execution Time

In addition to modeling network characteristics such as communication delays, one might also want to model **execution time**, the time it takes an actor to perform its function on a particular implementation platform. The joint modeling of an application's functionality and its performance on a model of the implementation platform is a very powerful tool for **design-space exploration**. It makes it much easier to understand the impact of choices in networking infrastructure and processor architecture.

In a discrete-event model, execution times can be simulated using a `Server` actor for each execution resource (such as a processor), where the service time is the execution time. Example 7.5 and Figure 7.8 illustrate this for a simple storage system. However, such models are difficult to combine with models of complex functionality.

Fortunately, Ptolemy provides **execution aspects**, which, like **communication aspects**, provide a form of **aspect-oriented modeling**. Execution aspects can be used to model contention for resources that are required to execute an application model. The mechanisms are similar to those of the communication aspects, as illustrated in the next example.

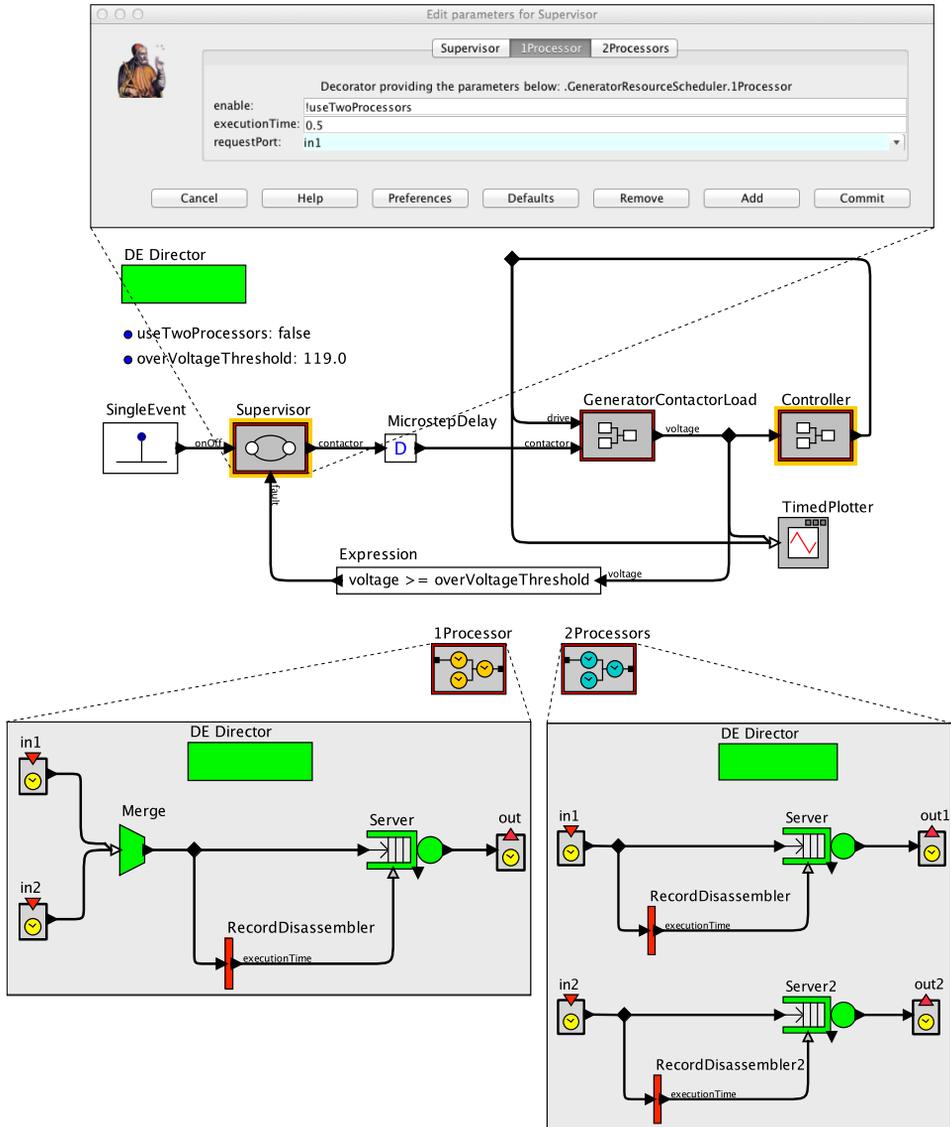


Figure 10.17: A model with two alternative execution aspects, one that models a one-processor execution platform and one that models a two-processor execution platform. [online]

**Example 10.9:** A variant of the generator model considered in Section 1.9 is shown in Figure 10.17. This model includes two possible implementation platforms, one with one processor, one with two processors, shown at the bottom of the figure. These two implementation platforms are modeled using the **CompositeResourceScheduler** actor, found in `MoreLibraries→ResourceScheduler`.

In this model, the Supervisor and Controller actors execute on one of the two processor architectures. Which one is determined by the value of the *useTwoProcessors* parameter in the model. If the value of this parameter is true, then the `2Processor` aspect will be used to execute Supervisor and Controller. Otherwise, `1Processor` will be used.

When this model is executed, the behavior changes with the value of *useTwoProcessors*. When two processors are used, there is no contention for resources, since Supervisor and Controller can execute simultaneously, as modeled by the two **Server** actors at the lower right in the figure. However, when only one processor is used, the Supervisor and Controller compete for the use a single processor, as modeled by the single server at the lower left. In that case, there is more delay in one of the two feedback loops, which changes the dynamics of the model. In particular, with certain choices of parameters and test conditions, the choice of processor architecture could affect whether the over voltage protection conditions shown in the plot in Figure 1.11 occurs.

Notice the use of **RecordDisassembler** actors in the composite execution aspects. The input to the submodel in the composite aspect is a record that contains the values of the decorator parameter *executionTime* of the actor requesting an execution resource. This execution time is extracted from the record and becomes the service time of the Server.

## 10.5 Ptides for Distributed Real-Time Systems

So far, this chapter has focused on modeling and simulating timing behavior in system implementations. Another role for timed models, however, is to *specify* timing behavior. That is, a timed model may give the *required* behavior of an implementation without

completely describing the implementation. Towards this end, we focus for the remainder of this chapter on a programming model for distributed real-time systems called **Ptides**.\*

Ptides models are designed to solve the problem identified in Example 10.9 above, where the behavior of a system depends on the details of the hardware and software platform that executes the system. A key goal in Ptides is to ensure that every *correct* execution of a system delivers exactly the same dynamic behavior.

**Example 10.10:** In Example 10.9, choosing to execute Supervisor and Controller on a single processor yields different dynamic behavior than choosing two processors. If these were Ptides models, the two behaviors would be identical, as long as the processor resources were sufficient to deliver a *correct* execution. Moreover, the execution times of the Supervisor and Controller will also not affect the dynamics until they get so large that a correct execution is no longer possible. Hence, Ptides has the potential to reduce the sensitivity that a system has to implementation details. Behavior is exactly the same over a range of implementations.

A Ptides model is a DE model with certain constraints on **time stamps**. Ptides is used to design event-triggered distributed real-time systems, where events may be occurring regularly (as in sampled-data systems) or irregularly. A key idea in Ptides is that, unlike DE, time stamps have a relationship with **real time** at sensors and actuators (which are the devices that bridge the cyber and the physical parts of **cyber-physical systems**). A second key idea in Ptides is that it leverages network time synchronization (Johannessen, 2004; Eidson, 2006) to provide a coherent global meaning to time stamps in distributed systems. The most interesting, subtle, and potentially confusing part about Ptides is the relationship between multiple time lines. But herein also lies its power.

### 10.5.1 Structure of a Ptides Model

A Ptides model consists of one or more **Ptides platforms**, each of which models a computer on a network. A Ptides platform is a **composite actor** that contains actors representing sensors, actuators, and network ports, and actors that perform computation and/or

---

\*The name comes from the somewhat tortured acronym for “programming temporally integrated distributed embedded systems.” The initial “P” is silent, as in Ptolemy, so the name is pronounced “tides.”

### Sidebar: Background of Ptides

Ptides leverages network **time synchronization** (Johannessen, 2004; Eidson, 2006) to provide a coherent global temporal semantics in distributed systems. The Ptides programming model was originally developed by Yang Zhao as part of her Ph.D. research (Zhao et al., 2007; Zhao, 2009). Zhao showed that, subject to assumed bounds on network latency, Ptides models are deterministic. The case for a time-centric approach like Ptides is elaborated by Lee et al. (2009b), and an overview of Ptides and an application to power-plant control is given by Eidson et al. (2012),

A number of implementations followed the initial work. A simulator is described by Derler et al. (2008), and an execution policy suitable for implementation in embedded software systems by Feng et al. (2008) and Zou et al. (2009b). Zou (2011) developed **PtidyOS**, a lightweight microkernel implementing Ptides on embedded computers, and a code generator producing embedded C programs from models. Matic et al. (2011) adapted PtidyOS and the code generator to demonstrate their use in smart grid technologies.

Feng and Lee (2008) extended Ptides with incremental checkpointing to provide a measure of fault tolerance. They showed conditions under which rollback can recover from errors, observing that the key constraint in Ptides is that actuator actions cannot be rolled back. Ptides has also been used to coordinate real-time components written in Java (Zou et al., 2009a).

A technique similar to Ptides was independently developed at Google for managing distributed databases (Corbett et al., 2012). In this work, clocks are synchronized across data centers, and messages sent between data centers are time stamped. The technique provides a measure of determinacy and consistency in database accesses and updates.

Assuming that the network latency bounds are met, a correct implementation of Ptides is deterministic in that a sequence of time-stamped events from sensors always results in a unique and well-defined sequence of time-stamped events delivered to actuators. However, this determinism does not provide any guarantee that events are delivered to actuators *on time* (prior to the deadline given by the time stamp). The problem of determining whether events can be delivered on time to actuators is called the **schedulability** problem. The question is, given a Ptides model, does there exist a schedule of the firing of actors such that deadlines are met. Zhao (2009) solved this problem for a limited class of models. The problem is further discussed by Zou et al. (2009b), and largely solved by Matsikoudis et al. (2013).

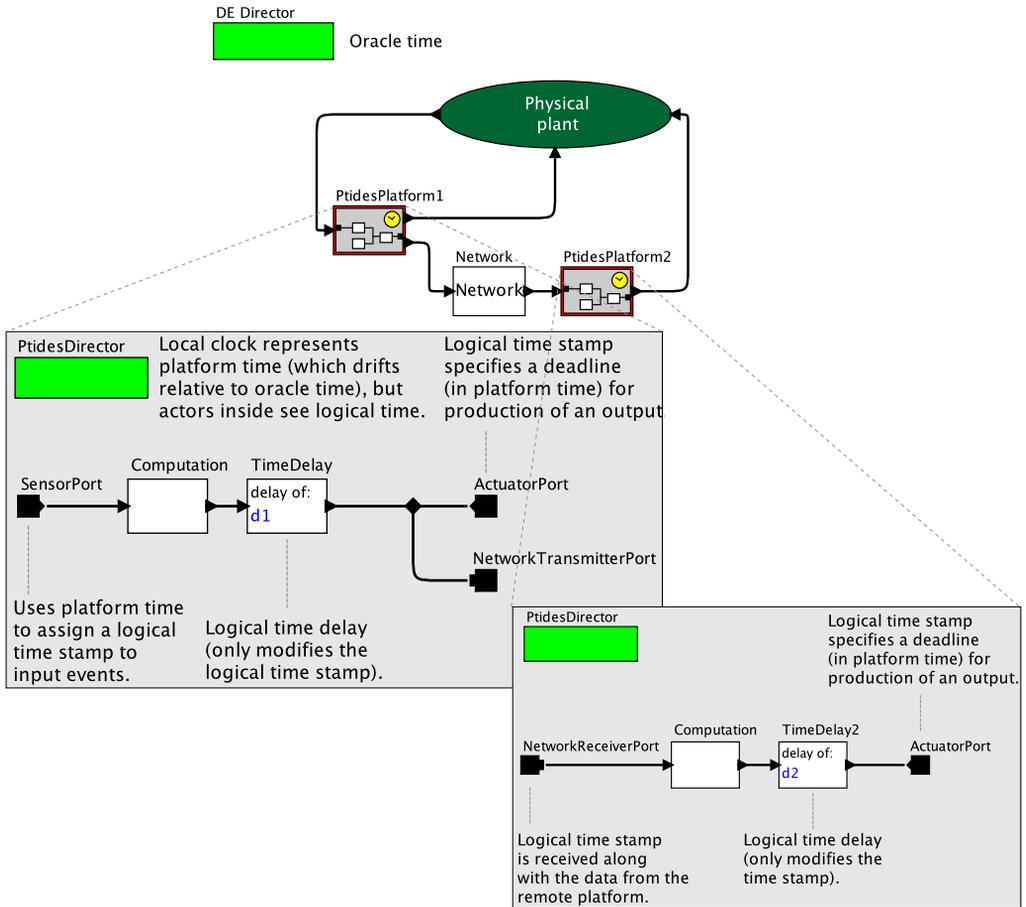


Figure 10.18: Ptdes model with two Ptdes platforms, sensor, actuator and network ports. [\[online\]](#)

modify time stamps. A Ptdes platform contains a PtdesDirector and represents a single device in a distributed cyber-physical system, such as a circuit board containing a microcontroller and some set of sensor and actuator devices. For simulation purposes, a Ptdes platform is placed within a DE model that models the physical environment of the platform.

**Example 10.11:** A simple Ptimes model is shown in Figure 10.18. This model has two platforms connected to a physical plant (via sensors and actuators) and to a network. The top-level director is a DE director, whereas the platform directors are Ptimes directors. The physical plant may internally be a [Continuous](#) model.

Ptimes models leverage Ptolemy's [multiform time](#) mechanism. A common pattern in such models assumes the time line at the top-level of the model hierarchy represents an idealized physical time line that advances uniformly throughout the system. This time line cannot be directly observed by computational devices in the network, which must instead use [clocks](#) to approximately measure it. We refer to such an idealized time at the top level as the **oracle time**. In the [MARTE](#) time library, the same idealized concept of physical time is referred to simply as **ideal time** ([André et al., 2007](#)).

Within a platform, a local clock maintains a time line called **platform time**, which approximates oracle time. Platform time is **chronometric**, an imperfect measurement of oracle time. The builder of a Ptimes model may choose to assume that platform time perfectly tracks oracle time or, more interestingly, to model imperfections in tracking and discrepancies across the network, as illustrated in Section 10.2 above.

**Example 10.12:** In the Ptimes model of Figure 10.18, the top-level director's clock represents oracle time. The clocks of the Ptimes directors represent platform time. These can be parameterized to drift with respect to each other and platform time and to have offsets.

A key innovation in Ptimes, however, is that a second time line called **logical time** plays a key role in a platform. The notion of logical time in distributed systems was introduced by [Lamport et al. \(1978\)](#), and is applied in Ptimes to achieve determinism in distributed real-time systems. Any actor that requests the current time from the PtimesDirector will be told about logical time, not about platform time. The only actors that have access to platform time are sensors, actuators, and network interfaces, i.e. the actors at the cyber-physical boundary. Specifically, when a sensor produces an event in a Ptimes model, the time stamp of the event is a logical time value equal to the platform time at which the sensor takes its

measurement. That is, Ptides binds logical time to physical time (as measured by platform time) at sensors.

**Example 10.13:** In the Ptides model of Figure 10.18, the sensor uses the platform time of PtidesPlatform1 to construct a time stamp for each event that it produces. Such an event represents a measurement made on the physical plant, and its (logical) time stamp is equal to the local measurement of time.

Let  $t_s$  be the platform time at which a measurement is made by a sensor. The event produced by that sensor actor will have logical time stamp  $t_s$ .

A **TimeDelay** actor, however, operates in logical time. It simply manipulates the logical time stamp. Platform time is not visible to it. If an input to a TimeDelay actor whose delay value is  $d_1$  has (logical) time stamp  $t$ , then its output will have time stamp  $t + d_1$ .

Once an event from a sensor has been produced, it is processed by the Ptides model like any other discrete event in a DE system. That is, events with logical time stamps are processed by the PtidesDirector in time-stamp order, without particular concern for platform time or oracle time. Actors are fired as they would be in simulation. A key property of Ptides models is that this time-stamp-ordered processing of events is preserved despite the distributed architecture and imperfect clocks. This key property delivers determinism.

An actuator port inside a Ptides platform acts as an output from the platform. When it receives an event from the platform Ptides model, that event has a logical time stamp  $t$ . The actuator interprets  $t$  as a **deadline** relative to platform time. That is, an event with time stamp  $t$  sent to an actuator is a command to perform some physical action no later than the (platform) time equal to  $t$ . Hence, actuators, like sensors, also bridge logical and physical times.

**Example 10.14:** In the Ptides model of Figure 10.18, in PtidesPlatform1, assume the SensorPort produces an event with time stamp  $t_s$ . This represents the platform time at which a sensor measurement is made. Assume further that Computation is a **zero-delay actor**, and that it reacts to the event from SensorPort by producing an output event with the same time stamp  $t_s$ . The output of the TimeDelay actor, therefore, will have time stamp  $t_s + d_1$ , where  $d_1$  is the delay of the TimeDelay

actor. That event goes to the ActuatorPort, which interprets the time stamp  $t_s + d_1$  as a deadline. That is, the actuator should produce its actuation at platform time no later than  $t_s + d_1$ .

By default, when executing a Ptides model, actors are assumed to be instantaneous (in platform time). Hence, the deadline at ActuatorPort in Figure 10.18 will never be violated. In fact, in the simulation, the ActuatorPort will be able to perform its actuation as early as platform time  $t_s$ . This is not very realistic, because any physical realization of this platform will incur some latency. It cannot react instantaneously to sensor events. More realistic simulation models can be constructed by combining the [execution aspects](#) of Section 10.4 with Ptides, but we will not do that here. Instead, here, we will assume that there is some variability in the latency introduced by the physical realization of the platform, but that the deadline will nevertheless be met. Verifying this is a [schedulability](#) problem.

The actuation of ActuatorPort in Figure 10.18 affects the physical plant, which in turn affects the SensorPort. There is a feedback loop, and the closed-loop behavior will be affected by the latency of the platform. If that latency is unknown or variable, then the overall closed-loop behavior of the system will be unknown or variable, yielding a [non-deterministic](#) model. To regain determinism, Ptides actuators can be configured to perform their actuation *at the deadline* rather than *by the deadline*. As long as events arrive at or before the deadline, the actuator will be able to produce its actuation deterministically, independent of the actual arrival time of the events, and hence independent of execution time variability. The response of PtidesPlatform1 to a sensor event will be a *deterministic* actuator event (in platform time and oracle time). This makes the behavior of the entire closed-loop system independent of variability in execution times (and, as we will show below, network delays). To configure a Ptides actuator to provide this determinism, set the *actuateAtEventTimestamp* parameter of the *ActuatorPort* to `true`. Ptides, therefore, provides a mechanism to hide underlying uncertainty and variability (up to a failure threshold, when deadlines are not met), yielding deterministic closed-loop behavior.

A natural question arises now about what to do if the failure threshold is crossed. By default, an actuator port in Ptides will throw an exception if it receives an event with time stamp  $t$  and platform time has already exceeded  $t$ . Such an exception is an indication that assumptions about the ability of the platform to meet the deadline have been violated. A well-designed model will catch such exceptions, using for example [error transitions](#) in a [modal model](#) (see Section 8.2.3). How to handle such exceptions, of course, is application

dependent. It might be necessary, for example, to switch to a safe but degraded mode of operation. Or it might be necessary to restart some portion of the system, or to switch to a backup system.

Multiple Ptides platforms in a model may communicate via a network. When such communication occurs, logical time stamps are conveyed along with the data. Unlike an actuator port, a network transmitter port always produces its output immediately when it becomes available, rather than waiting for platform time to match the time stamp. The logical time stamp of the event will be carried along with the event to the network receiver port, which will then produce on its output an event with that same time stamp.

Like an actuator port, a network transmitter port treats the time stamp as a deadline and will throw an exception if the platform time exceeds the time stamp value when the event arrives.<sup>†</sup>

**Example 10.15:** In the Ptides model of Figure 10.18, in PtidesPlatform1, assume that the sensor makes a measurement at platform time  $t_s$ , and that consequently the network transmitter port receives an event with time stamp  $t_s + d_1$ . Assume further that it receives this event at platform time  $t_s$ , because the execution time of actors is (by default) assumed to be zero. Hence, the Network actor in Figure 10.18 will received an event containing as its payload both a value (the value of the event delivered to the NetworkTransmitterPort) and a logical time stamp  $t_s + d_1$ .

The NetworkTransmitterPort will launch this payload into the network at platform time  $t_s$ . The network will incur some delay, simulated by the Network actor in the figure, and will arrive at PtidesPlatform2 at some time  $t_2$ , a local platform time at PtidesPlatform2. The NetworkReceiverPort on PtidesPlatform2 will produce an output event with (logical) time stamp  $t_s + d_1$ , extracted from the payload. In Figure 10.18, this event will pass through another Computation actor and another TimeDelay actor. Assuming the TimeDelay actor increments the time stamp by  $d_2$ , the ActuatorPort on PtidesPlatform2 will receive an event with time stamp  $t_s + d_1 + d_2$ . This deadline will be met if  $t_s + d_1 + d_2 \geq t_2$ .

If the *actuateAtEventTimestamp* parameter of the ActuatorPort is `true`, and all deadlines are met, then the overall latency from the sensor in platform 1 to the actuator in platform 2 is deterministic and independent of the actual network delay

<sup>†</sup>This deadline may be modified to be earlier or later by changing the *platformDelayBound* parameter of the network transmitter port, as explained below.

and actual computation times. This ability to have a fixed latency in a distributed system is central to the power of the Ptdes model.

As with the actuator on platform 1, if the deadline is not met at the actuator on platform 2, the `ActuatorPort` will throw an exception. This exception is an indication that some timing assumption about the implementation has been violated; for example, an assumed bound on the network latency has not been actually met by the network. This should be handled by the model as an error condition, which could, for example, cause the model to switch into a safe but degraded mode of operation.

Although the end-to-end latency from the sensor on platform 1 to the actuator on platform 2 is deterministic, it is not exactly clear from this model what that latency is. Nominally, the latency is the logical time delay,  $d_1 + d_2$ . However, the time at which the actuation occurs,  $t_s + d_1 + d_2$ , is relative to the local platform clock at platform 2. This time, however, also depends on the clock on platform 1, since  $t_s$  is the time on platform 1 when the sensor measurement is taken. Hence, to be useful, a distributed Ptdes system requires that clocks be synchronized (see Section 10.2). They need not be perfectly synchronized, but if the error between them is not bounded, then there is no bound on the end-to-end latency (in oracle time).

If these two platform clocks are perfectly synchronized, then the actual latency will be exactly  $d_1 + d_2$ , relative to these platform clocks. The latency in oracle time, of course, depends on the drift of these clocks relative to oracle time (see Figure 10.2). If these two clocks progress at exactly the rate of oracle time, then the actual latency will be exactly  $d_1 + d_2$  in oracle time. Hence, with sufficiently good clocks and sufficiently good clock synchronization, Ptdes gives an overall timing behavior that is precise and deterministic up to the precision of these clocks.

To help ensure that a realization meets the requirements of a specification, a network receiver port also imposes a constraint on timing. As mentioned above, the network transmitter port will throw an exception if it receives an event at a platform time greater than the time stamp of the event.<sup>‡</sup> So if a network receiver port receives a message, it knows that the message was transmitted at a platform time no later than the time stamp on the message it receives. The receiver has a parameter *networkDelayBound*, which is an upper bound on the network delay that it assumes. When the network receiver receives a

<sup>‡</sup>This deadline may be modified to be earlier or later by changing the *platformDelayBound* parameter of the network transmitter port, as explained below.

message, it checks that the platform time does not exceed the time stamp on the message plus the *networkDelayBound* plus a fudge factor to account for clock discrepancies and device delays, which also have assumed bounds specified by parameters, described below. If the platform time is too large, then the network receiver knows that one of these assumptions was violated (though it cannot know which one), and it throws an exception. Although this constraint is very subtle, the consequences on models are relatively easy to understand.

**Example 10.16:** In the Ptides model of Figure 10.18, along the path from the sensor to the network receiver port, there cannot be a physical delay greater than the logical delay along the same path. The logical delay along this path is simply  $d_1$ , the parameter of the *TimeDelay* actor. The physical delay is the sum of the executions times of the actors along the path (which in simulation is assumed to be zero by default) and the network delay. Hence, if the network imposes a delay greater than  $d_1$ , the model in this figure will fail with an exception (by default, though other error handling strategies are also possible).

Notice that if we were to replace the Ptides directors in the platforms with DE directors, then the behavior would be significantly different. In this case, the latency from the sensor in platform 1 to the actuator in platform 2 would include the actual network delay. A key property of Ptides is that network delays and computation times are segregated from the logical timing of a model. The logical timing becomes a *specification* of timing behavior, whereas network delays and computation times are part of the *realization* of the system. Ptides models enable us to determine conditions under which realizations will meet the requirements of the specification. And the simulator enables evaluation of behavior under elaborate conditions that would be very difficult to validate analytically, for example taking into account the complicated dynamics of *PTP* clock synchronization protocols.

### 10.5.2 Ptides Components

**Ptides ports.** Ports in a Ptides platform represent devices that communicate with the environment or the network. Ptides ports can model device delays, although by default these delays are zero. Every Ptides port has a *deviceDelay* and a *deviceDelayBound* parameter. The *deviceDelay*  $d$  models delay of the device. For example, if a sensor makes a mea-

### Sidebar: Safe-to-Process Analysis

The execution of actors inside a Ptides platform follows DE semantics. Actors must process events in time-stamp order (unless they are memoryless). In simulation, it is straightforward to ensure that events are processed in time-stamp order, but when Ptides models are deployed, things get more complicated. In particular, a deployed system cannot easily coordinate the scheduling of actor firings across platforms. Each platform must be able to make its own scheduling decisions.

Consider the platform model shown in Figure 10.19. This example has a sensor port and a network receiver port. Suppose that the sensor produces an event with time stamp  $t_s$ . If we assume that every sensor produces events in time-stamp order, then the scheduler can immediately fire Computation1. Suppose that firing produces another event with time stamp  $t_s$ , which then results in an event with time stamp  $t_s + d_1$  available at the top input of Computation3. When can Computation3 be fired to react to that event? The scheduler has to be sure that no event with time stamp less than or equal to  $t_s + d_1$  will later become available at the bottom input of Computation3.

A simple approach, developed by [Chandy and Misra \(1979\)](#) for distributed DE simulation, is to wait until there is an event available on the bottom input of Computation3 with time stamp greater than or equal to  $t_s + d_1$ . But this could result in quite a wait, particularly if a fault occurs and the source of events on this path fails.

An alternative approach due to [Jefferson \(1985\)](#) fires Computation3 speculatively, assuming no problematic event will later arrive, and if it does, reverses the computation by restoring the state of the actor. This approach is fundamentally limited by the inability to backtrack actuators.

The Ptides approach ensures that events are processed in order *as long as all deadlines are met*. In our example, an event at the top input of Computation3 with time stamp  $t_s + d_1$  can be safely processed when the local platform time meets or exceeds  $t_s + d_1$ . This is because [schedulability](#) requires that an event with time stamp  $t_s + d_1$  or earlier is required to arrive at the network receiver port at platform time  $t_s + d_1$  or earlier.

Conversely, suppose an event with time stamp  $t_n$  is at the bottom input of Computation3. That event is **safe to process** when platform time meets  $t_n - d_1 + s$ , where  $s$  is a bound on the **sensor delay**, the time between a sensor event time stamp and the event becoming visible to the scheduler. See page 382 for citations that explain safe-to-process analysis in more detail.

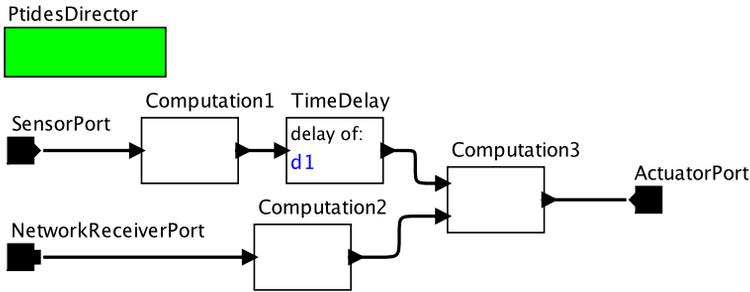
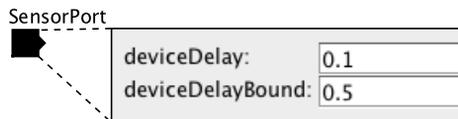


Figure 10.19: Simple Ptides example used to illustrate safe-to-process analysis.

surement at platform time  $t_s$ , it will produce an event with time stamp  $t_s$ . But that event will not appear until platform time  $t_s + d$ . For example,  $d$  might represent the amount of platform time that it takes for the sensor device to raise an interrupt request, and for the processor to respond to the interrupt request.

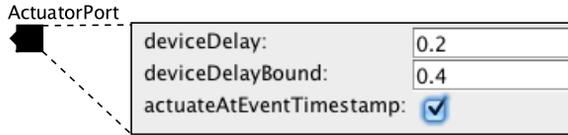
The *deviceDelayBound*  $d_B$  gives an upper bound on the *deviceDelay*  $d$ . The Ptides framework assumes that *deviceDelay* can vary during execution but will never exceed *deviceDelayBound*, which does not vary. This bound is used in [safe to process](#) analysis (see sidebar on page 390), which ensures that events are processed in time-stamp order.

**Sensors.** A **sensor port** is a particular kind of Ptides port that looks like this:



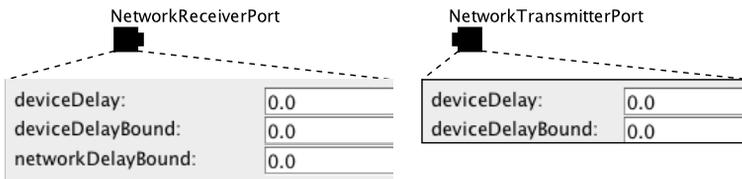
It receives inputs from the environment and creates new events with the time stamp equal to the current platform time (which is the current local time of the PtidesDirector) and posts this event on the event queue. An event that is received by a sensor at platform time  $t_s$  is produced with logical time stamp  $t_s$  at platform time  $t_s + d$ .

**Actuators.** An **actuator port** is a particular kind of Ptypes port that looks like this:



By default, an actuator port produces events on the output of the platform when the time stamp of the event equals the current platform time. If you change *actuateAtEventTimestamp* to false, then the event may be produced earlier if it is available earlier. The *deviceDelayBound* parameter specifies a **setup time** for the device. Specifically, the deadline for delivery of an event with time stamp  $t$  to an actuator is platform time  $t - d_B$ , where  $d_B$  is the value of *deviceDelayBound*. An exception is thrown if this deadline is not met.

**Network transmitters and receivers.** **Network transmitter ports** and **network receiver ports** are also particular kinds of Ptypes port that look like this:



The *NetworkTransmitterPort* takes an event from the inside of the Ptypes platform and sends to the outside a record that encodes the time stamp of the event and its value (the **payload**). The *NetworkReceiverPort* extracts the time stamp and the payload and produces at the inside of the destination Ptypes platform an event with the specified value and time stamp.

The parameter *networkDelayBound* ( $d_N$ ) specifies the assumed maximum amount of time an incoming token spends in the network before it arrives at the receiver. It is used to determine whether an event can be processed safely, or whether another event with a smaller time stamp may still be in the network. If the actual network delay exceeds this bound and the delay causes the message to be received too late, then the network receiver port will throw an exception.

## 10.6 Summary

Modeling of complex timed systems is not easy. We all harbor a naive notion of a uniform fabric of time, shared by all participants in the physical world. But such a notion is a fiction, and real systems are strongly affected by errors in time measurement and discrepancies between logical and physical notions of time. A major focus of recent work in the Ptolemy Project has been to provide a solid modeling foundation for the far-from-solid realities of time.

## 10.7 Acknowledgements

The authors would like to thank Yishai Feldman and Stavros Tripakis for very helpful suggestions for this chapter.

# Ptera: An Event-Oriented Model of Computation

*Thomas Huining Feng and Edward A. Lee*

## Contents

---

<b>11.1 Syntax and Semantics of Flat Models</b> . . . . .	<b>395</b>
<i>Sidebar: Notations, Languages for Event-Oriented Models</i> . . . . .	396
<i>Sidebar: Background of Ptera</i> . . . . .	397
11.1.1 Introductory Examples . . . . .	398
11.1.2 Event Arguments . . . . .	399
11.1.3 Canceling Relations . . . . .	401
11.1.4 Simultaneous Events . . . . .	401
11.1.5 Potential Nondeterminism . . . . .	402
11.1.6 LIFO and FIFO Policies . . . . .	404
11.1.7 Priorities . . . . .	405
11.1.8 Names of Events and Scheduling Relations . . . . .	406
11.1.9 Designs with Atomicity . . . . .	406
11.1.10 Application-Oriented Examples . . . . .	408
<i>Sidebar: Model Execution Algorithm</i> . . . . .	409
<b>11.2 Hierarchical Models</b> . . . . .	<b>411</b>
<b>11.3 Heterogeneous Composition</b> . . . . .	<b>413</b>
11.3.1 Composition with DE . . . . .	413
11.3.2 Composition with FSMs . . . . .	416
<b>11.4 Summary</b> . . . . .	<b>417</b>

---

FSMs and DE models, covered in Chapters 6 and 7, focus on **events** and the causal relationships between those events. An event is **atomic**, conceptually occurring at an instant in time. An **event-oriented model** defines a collection of events in time. Specifically, it generates events, typically in chronological order, and defines how other events are triggered by those events. If there are externally provided events, the process also defines how those events may trigger additional events. In the DE domain, the timing of events is controlled by timed sources and delay actors (see sidebars on pages 241 and 243). Models in the FSM domain primarily react to externally provided events, but may also generate timed events internally using the *timeout* function in a guard (see Table 6.2 on page 196).

There are many ways other ways to specify event-oriented models (see sidebar on page 396). This chapter describes a novel one called **Ptera** (for Ptolemy event relationship actors), first given by Feng et al. (2010). Ptera is designed to interoperate well with other Ptolemy II models of computation, to provide model hierarchy, and to handle concurrency in a deterministic way.

## 11.1 Syntax and Semantics of Flat Models

A flat (i.e., non-hierarchical) Ptera model is a graph containing vertices connected with directed edges, such as shown in Figure 11.1, which contains two vertices and one edge. A vertex contains an **event**, and a directed edge represents the conditions under which one event will cause another event to occur. Vertices and edges can be assigned a range of attributes and parameters, as described later in this chapter.

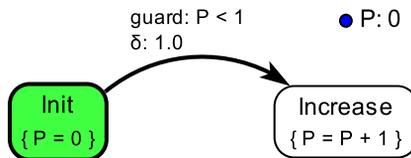


Figure 11.1: A simple Ptera model with two events. [[online](#)]

In a hierarchical Ptera model, vertices can also represent *submodels*, which may be other Ptera models, FSMs, actor models using some other Ptolemy II director, or even custom Java code. The only requirement is that their behavior must be defined to conform with the [actor abstract semantics](#), as explained below.

### Sidebar: Notations, Languages for Event-Oriented Models

Many notations and languages have been developed for describing [event-oriented models](#). A popular one today is the [UML activity diagram](#), a derivative of the classical **flowcharts** that date back to the 1960s (see [http://en.wikipedia.org/wiki/Activity\\_diagram](http://en.wikipedia.org/wiki/Activity_diagram)). In an activity diagram, a block represents an **activity**, and an arrow from one activity to another designates the causality relationship between the two. A diamond-shaped activity tests for a condition, and causes the activity on one of its outgoing branches to occur. Activity diagrams include **split** and **join** activities, which spawn multiple concurrent activities, and wait for their completion. As is common with UML notations, the semantics of concurrency (and even of activities) is not clear. Activities may be interpreted as events, in which case they are [atomic](#), or they may take time, in which case the triggering of an activity and its completion are events. The meaning of an activity diagram also becomes unclear when connections are made into and out of concurrent activities, and the model of time is not well defined. Nevertheless, activity diagrams are often easy to understand intuitively, and hence prove useful as a way to communicate event-oriented models.

Another relevant notation is the **business process model and notation (BPNL)**, which, like UML, is now maintained and developed by the [OMG](#). BPNL makes a distinction between events and activities, allowing both in a diagram. It also offers a form of hierarchy and concurrency, with rather complicated interaction and synchronization mechanisms.

A third related notation is **control flow graphs**, introduced by [Allen \(1970\)](#), which today are widely used in compiler optimizations, program analysis tools, and electronic design automation. In a CFG, the nodes in a graph represent **basic blocks** in a program, which are sequences of instructions with no branches or flow control structures. These basic blocks are treated as atomic, and hence can be considered events. Connections between basic blocks represent the possible flow control sequences that a program may follow.

### Sidebar: Background of Ptera

Ptera is derived from **event graphs**, given by [Schruben \(1983\)](#). Blocks in an event graph (which he called vertices) contain events, which can include actions to be performed when that event is processed. Connections between events (called “directed edges”) represent scheduling relations that can be guarded by Boolean and temporal expressions. Event graphs are timed, and time delays can be associated with scheduling relations. Each event graph has an event queue, although it is not explicitly shown in the visual representation. In multi-threaded execution, multiple event queues may be used, in principle. In each step of an execution, the execution engine removes the next event from the event queue and processes it. The event’s associated actions are executed, and additional events specified by its scheduling relations are inserted into the event queue.

The original event graphs do not support hierarchy. [Schruben \(1995\)](#) gives two approaches for supporting hierarchy. One is to associate submodels with scheduling relations, in which the output of a submodel is a number used as the delay for the scheduling relation. Another approach is to associate submodels with events instead of scheduling relations ([Som and Sargent, 1989](#)). Processing such an event causes the unique start event in the submodel to be scheduled, which in turn may schedule further events in the submodel. When a predetermined end event is processed, the execution of the submodel terminates, and the event that the submodel is associated with is considered processed. [Buss and Sanchez \(2002\)](#) report a third attempt to support hierarchy, in which a listener pattern is introduced as an extra gluing mechanism for composing event graphs.

Ptera is based on event graphs, but extends them to support heterogeneous, hierarchical modeling. Composition of Ptera models forms a hierarchical model, which can be flattened to obtain an equivalent model without hierarchy. Ptera models conform with the **actor abstract semantics**, which permits them to contain or be contained by other types of models, thus enabling hierarchical heterogeneous designs. Ptera models can be freely composed with other models of computation in Ptolemy II.

Ptera models include an externally visible interface with parameters, input and output ports. Changes to parameters and the arrival of data at input ports can potentially trigger events within the Ptera model, in which case it becomes an actor. In addition, event actions can be customized by the designer with programs in an imperative language (such as Java or C) conforming to a protocol.

### 11.1.1 Introductory Examples

**Example 11.1:** An example Ptera model is shown in Figure 11.1. This model includes two vertices (events), *Init* and *Increase*, and one **variable**, *P*, with an initial value of 0. When the model is executed, this variable may be updated with new values. *Init* is an **initial event** (its *initial* parameter is set to true), as indicated by a filled rounded rectangle with a thick border.

At the start of execution, all initial events are scheduled to occur at model time 0. (As discussed later, even when events occur at the same time conceptually, there is still a well-defined order of execution.) The model's *event queue* holds a list of scheduled **event instances**. An event instance is removed from the event queue and processed when the model time reaches the time at which the event is scheduled to occur (i.e., at the **time stamp** of that event).

Ptera events may be associated with **actions**, which are shown inside brackets in the vertex.

**Example 11.2:** In Figure 11.1, for example, *Init* specifies the action " $P = 0$ ", which sets *P* to 0 when *Init* is processed. The edge (connection) from the *Init* event to the *Increase* event is called a **scheduling relation**. It is guarded by the Boolean expression " $P < 1$ " (meaning that the transition will only be taken when this condition is met) and has a **delay** of 1.0 units of time (represented by the  $\delta$  symbol). After the *Init* event is processed, if *P*'s value is less than 1 (which is true in this case, since *P* is initially set to 0), then *Increase* will be executed at time 1.0. When *Increase* is processed at time 1.0, its action " $P = P + 1$ " is executed and *P*'s value is increased to 1.

After processing the *Increase* event, the event queue is empty. Since no more events are scheduled, the execution terminates.

In this simple example, there is at most one event in the event queue (either *Init* or *Increase*) at any time. In general, however, an unbounded number of events can be scheduled in the event queue.

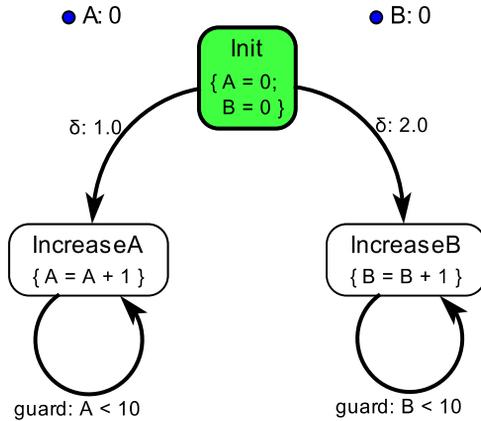


Figure 11.2: A model with multiple events in the event queue. [\[online\]](#)

**Example 11.3:** Figure 11.2 shows a slightly more complex Ptera model that requires an event queue of size greater than 1. In this model, the *Init* event schedules *IncreaseA* to occur after a 1.0-unit time delay, and *IncreaseB* to occur after a 2.0-unit delay. The guards of the two scheduling relations from *Init* have the default value “true,” and are thus not shown in the visual representation. When *IncreaseA* is processed, it increases variable A by 1 and reschedules itself, creating another event instance on the event queue, looping until A’s value reaches 10. (The model-time delay  $\delta$  on the scheduling relation from A to itself is also hidden, because it takes the default value “0.0,” which means the event is scheduled at the current model time, but the next [microstep](#).) Similarly, *IncreaseB* repeatedly increases variable B at the current model time until B’s value reaches 10.

### 11.1.2 Event Arguments

Like a C function, an event may include a list of formal *arguments*, where each argument is assigned a name and type.

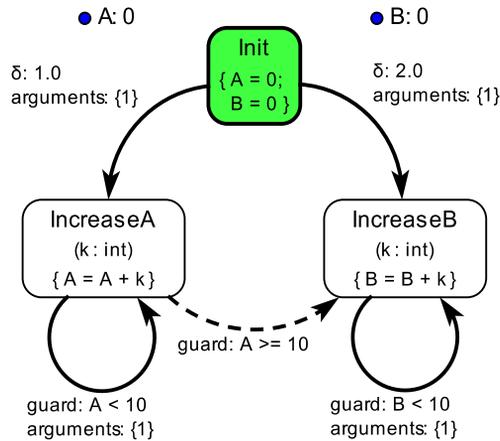


Figure 11.3: A model with arguments for the events and a canceling relation. [\[online\]](#)

**Example 11.4:** Figure 11.3 modifies Figure 11.2 by adding arguments  $k$  of type `int` to events *IncreaseA* and *IncreaseB*. These arguments are assigned values by the incoming relations and specify the increments to variables *A* and *B*. (The dashed edge in the figure is a canceling relation, which will be discussed in the next subsection.)

Each scheduling relation pointing to an event with arguments must specify a list of expressions in its *arguments* attribute. Those expressions are used to specify the actual values of the arguments when the event instance is processed. In this example, all scheduling relations pointing to *IncreaseA* and *IncreaseB* specify “{1}” in their argument attributes, meaning that  $k$  should take value 1 when those events are processed. Argument values can be used by event actions, guards, and delays.

### 11.1.3 Canceling Relations

A **canceling relation** is represented as a dashed line (edge) between events, and can be guarded by a Boolean expression. It cannot have any delays or arguments. When an event with an outgoing canceling relation is processed, if the guard is true and the target event has been scheduled in the event queue, the target event instance is removed from the event queue without being processed. In other words, a canceling relation cancels a previously scheduled event. If the target event is scheduled multiple times, i.e. multiple event instances are in the event queue, then the canceling relation causes only the first instance to be removed. If the target event is not scheduled, the canceling relation has no effect.

**Example 11.5:** Figure 11.3 provides an example of a canceling relation. Processing the last *IncreaseA* event (at time 1.0) causes *IncreaseB* (scheduled to occur at time 2.0 by the *Init* event) to be cancelled. As a result, variable B is never increased.

It should be noted that canceling relations do not increase expressiveness. In fact, a model with canceling relations can always be converted into a model without canceling relations, as is shown by Ingalls et al. (1996). Nonetheless, they can yield more compact and understandable models.

### 11.1.4 Simultaneous Events

**Simultaneous events** are defined as multiple **event instances** in an event queue that are scheduled to occur at the same model time.

**Example 11.6:** For example, in Figure 11.3, if both  $\delta$ s are set to 1.0, the model is as shown in Figure 11.4. Instances of *IncreaseA* and *IncreaseB* scheduled by *Init* become simultaneous events. Moreover, although multiple instances of *IncreaseA* occur at the same **model time**, they occur at different **microsteps** in **superdense time**, and they do not coexist in the event queue, so instances of *IncreaseA* are not simultaneous.

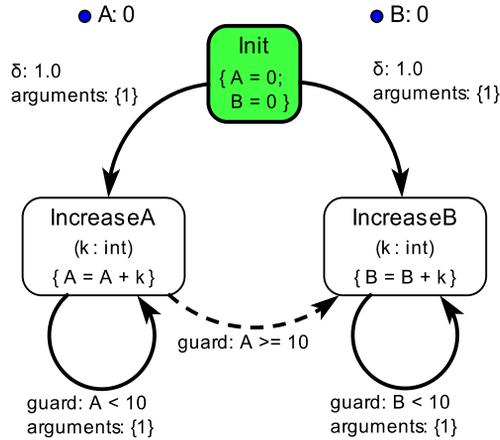


Figure 11.4: A model with simultaneous events. [\[online\]](#)

In general, it is a model checking (Clarke et al., 2000) problem to detect simultaneous events.

### 11.1.5 Potential Nondeterminism

When there are simultaneous events, there is potential **nondeterminism** introduced by the ambiguous order of event processing.

**Example 11.7:** For example, in Figure 11.3, what is the final value of the variables *A* and *B*? Suppose that all instances of *IncreaseA* are processed before any instance of *IncreaseB*. In that case, the final value of *B* will be 0. If instead, all instances of *IncreaseB* are processed before any instance of *IncreaseA*, then the final value of *B* will be 10.

Table 11.1 shows four possible execution traces that seem consistent with the model definition. The columns are arranged from left to right in the order of event processing. These traces include the case where *IncreaseA* always occurs before *IncreaseB*, where *IncreaseB* always occurs before *IncreaseA*, and where *IncreaseA*

**1) IncreaseA is always scheduled before IncreaseB:**

Time	0.0	1.0	1.0	...	1.0
Event	<i>Init</i>	<i>IncreaseA</i>	<i>IncreaseA</i>	...	<i>IncreaseA</i>
A	0	1	2	...	10
B	0	0	0	...	0

**2) IncreaseB is always scheduled before IncreaseA:**

Time	0.0	1.0	1.0	...	1.0	1.0
Event	<i>Init</i>	<i>IncreaseB</i>	<i>IncreaseB</i>	...	<i>IncreaseB</i>	<i>IncreaseA</i>
A	0	0	0	...	0	1
B	0	1	2	...	10	10
Time	1.0	...	1.0			
Event	<i>IncreaseA</i>	...	<i>IncreaseA</i>			
A	2	...	10			
B	10	...	10			

**3) IncreaseA and IncreaseB are alternating, starting with IncreaseA:**

Time	0.0	1.0	1.0	1.0	1.0	...
Event	<i>Init</i>	<i>IncreaseA</i>	<i>IncreaseB</i>	<i>IncreaseA</i>	<i>IncreaseB</i>	...
A	0	1	1	2	2	...
B	0	0	1	1	2	...
Time	1.0	1.0	1.0			
Event	<i>IncreaseA</i>	<i>IncreaseB</i>	<i>IncreaseA</i>			
A	9	9	10			
B	8	9	9			

**4) IncreaseA and IncreaseB are alternating, starting with IncreaseB:**

Time	0.0	1.0	1.0	1.0	1.0	...
Event	<i>Init</i>	<i>IncreaseB</i>	<i>IncreaseA</i>	<i>IncreaseB</i>	<i>IncreaseA</i>	...
A	0	0	1	1	2	...
B	0	1	1	2	2	...
Time	1.0	1.0	1.0	1.0		
Event	<i>IncreaseB</i>	<i>IncreaseA</i>	<i>IncreaseB</i>	<i>IncreaseA</i>		
A	8	9	9	10		
B	9	9	10	10		

Table 11.1: Four possible execution traces for the model in Figure 11.4.

and *IncreaseB* are alternating in two different ways. There are many other possible execution traces.

The traces end with different final values of A and B. The last instance of *IncreaseA*, which increases A to 10, always cancels the next *IncreaseB* in the event queue, if any. There are 10 instances of *IncreaseB* in total, and without a well-defined order, the one that is cancelled can be any one of them.

To avoid these nondeterministic execution results, we use the strategies discussed in the next sections.

### 11.1.6 LIFO and FIFO Policies

Ptera models can specify a **LIFO** (last in, first out) or **FIFO** (first in, first out) policy to control how event instances are accessed in the event queue and to help ensure deterministic outcomes. With LIFO (the default), the event scheduled later is processed sooner. The opposite occurs with FIFO. The choice between LIFO and FIFO is specified by a parameter *LIFO* in the Ptera model that defaults to value true.

If we use a LIFO policy to execute the model in Figure 11.4, then execution traces 3 and 4 in Table 11.1 are eliminated as possible outcomes. This leaves two possible execution traces, depending on whether *IncreaseA* or *IncreaseB* is processed first (that choice will depend on scheduling rules discussed later).

**Example 11.8:** Suppose *IncreaseA* is processed first. According to the LIFO policy, the second instance of *IncreaseA* scheduled by the first one should be processed before *IncreaseB*, which is scheduled by *Init*. The second instance again schedules the next one. In this way, processing of instances of *IncreaseA* continues until A's value reaches 10, when *IncreaseB* is cancelled. That leads to execution trace 1.

If instead *IncreaseB* is processed first, all 10 instances of *IncreaseB* are processed before *IncreaseA*. That yields execution trace 2.

With a FIFO policy, however, instances of *IncreaseA* and *IncreaseB* are interleaved, resulting in execution traces 3 and 4 in the table.

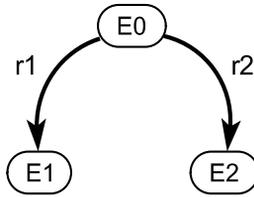


Figure 11.5: A scenario where event  $E0$  schedules  $E1$  and  $E2$  after the same delay.

In practice, LIFO is more commonly used because it executes a chain of events; one event schedules the next without delay. This approach is **atomic** in the sense that no event that is in the chain interferes with the processing of events in the chain. This is convenient for specifying workflows where some tasks need to be finished sequentially without intervention.

### 11.1.7 Priorities

For events that are scheduled by the same event with the same delay  $\delta$ , **priority** numbers can be assigned to the scheduling relations to force an ordering of event instances. A priority is an integer (which can be negative) that defaults to 0.

**Example 11.9:** Simultaneous instances of  $E1$  and  $E2$  in Figure 11.5 are scheduled by the scheduling relations  $r1$  and  $r2$  (with delay 0.0, since  $\delta$  is not shown). If  $r1$  has a higher priority (i.e., a smaller priority number) than  $r2$ , then  $E1$  is processed before  $E2$ , and vice versa.

**Example 11.10:** In Figure 11.4, if the priority of the scheduling relation from *Init* to *IncreaseA* is -1, and the priority of that from *Init* to *IncreaseB* is 0, then the first instance of *IncreaseA* is processed before *IncreaseB*. Execution traces 2 and 4 in Table 11.1 would not be possible. On the other hand, if the priority of the

scheduling relation from *Init* to *IncreaseA* is 1, then the first instance of *IncreaseB* is processed earlier, making execution traces 1 and 3 impossible.

### 11.1.8 Names of Events and Scheduling Relations

Every event and every scheduling relation in a Ptolemy II Ptera model has a name. In Figure 11.4, for example, *Init*, *IncreaseA*, and *IncreaseB* are the names of the events. These names may be assigned by the builder of the model (see Figure 2.15). Vergil will assign a default name, and at each level of a hierarchical model, the names are unique. Ptera uses these names to assign an execution order for simultaneous events when the priorities of their scheduling relations are the same.

In Figure 11.5, if *r1* and *r2* have the same delay  $\delta$  and the same priority, then we use names to determine the order of event processing.\* The order of *E1* and *E2* is determined by first comparing the names of the events. In a flat model, these names are guaranteed to be different, so they have a well-defined alphabetical order. The earlier one in this order will be scheduled first.

In a hierarchical model (discussed below), it is possible for simultaneous events to have the same name. In that case, the names of the scheduling relations are used instead. These are not usually shown in the visual representation, but can be determined by hovering over a scheduling relation with the cursor.

### 11.1.9 Designs with Atomicity

In some applications, designers need to ensure **atomic** execution of a sequence of events in the presence of other simultaneous events. That is, the entire sequence should be processed before any other simultaneous event is processed. There are two design patterns shown in Figure 11.6 that can be used to ensure atomicity without requiring designers to explicitly control critical sections (as would be the case for imperative programming languages).

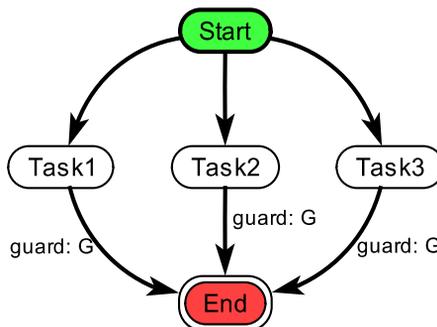
---

\*It would be valid in a variant of Ptera to either choose nondeterministically or process the two events concurrently, but this could lead to nondeterminism, so in our current implementation, we have chosen to define the order.

The design pattern in Figure 11.6(a) is used to sequentially and atomically perform a number of tasks, assuming the LIFO policy is chosen. Even if other events exist in the



a) Sequentially perform all tasks



b) Sequentially perform tasks until G is satisfied

Figure 11.6: Two design patterns for controlling tasks.

model (which are not shown in the figure), those events cannot interleave with the tasks. As a result, intermediate state between tasks is not affected by other events.

The **final event**, End, is a special class of event that removes all events in the queue. Final events are used to force termination even when there are events remaining in the event queue. They are shown as filled vertices with double-line borders.

The design pattern in Figure 11.6(b) is used to perform tasks until the guard  $G$  is satisfied. This pattern again assumes a LIFO policy. After the Start event is processed, all tasks are scheduled. In this case, the first one to be processed is Task1, because of the alphabetical ordering. After Task1, if  $G$  is true, End is processed next, which terminates the execution. If  $G$  is not true, then Task2 is processed. The processing of tasks continues until either  $G$  becomes true at some point, or all tasks are processed but  $G$  remains false.

### 11.1.10 Application-Oriented Examples

In this section, we describe two simple Ptera models that have properties that are similar to many systems of interest. We begin with a simple multiple-server, single-queue system: a car wash.

**Example 11.11:** In this example, multiple car wash machines share a single queue. When a car arrives, it is placed at the end of the queue to wait for service. The machines serve cars in the queue one at a time in a first-come-first-served manner. The car arrival intervals and service times are generated by stochastic processes assigned to the edges.

The model shown in Figure 11.7 is designed to analyze the number of available servers and the number of cars waiting over an elapsed period of time. The *Servers* variable is initialized to 3, which is the total number of servers. The *Queue* variable starts with 0 to indicate that there are no cars waiting in the queue at the beginning. *Run* is an initial event. It schedules the *Terminate* final event to occur after the amount of time defined by a third variable, *SimulationTime*.

The *Run* event also schedules the first instance of the *Enter* event, causing the first car arrival to occur after delay “ $3.0 + 5.0 * \text{random}()$ ,” where  $\text{random}()$  is a function that returns a random number in  $[0, 1)$  with a uniform distribution. When *Enter* occurs, its action increases the queue size in the *Queue* variable by 1. The *Enter*

### Sidebar: Model Execution Algorithm

We operationally define the semantics of a flat model with an execution algorithm. In the algorithm, symbol  $Q$  refers to the event queue. The algorithm terminates when  $Q$  becomes empty.

1. Initialize  $Q$  to be empty
2. For each initial event  $e$  in the  $\leq_e$  order
  1. Create an instance  $i_e$
  2. Set the time stamp of  $i_e$  to be 0
  3. Append  $i_e$  to  $Q$
3. While  $Q$  is not empty
  - (a) Remove the first  $i_e$  from  $Q$ , which is an instance of some event  $e$
  - (b) Execute the actions of  $e$
  - (c) Terminate if  $e$  is a final event
  - (d) For each canceling relation  $c$  from  $e$ 

From  $Q$ , remove the first instance of the event that  $c$  points to, if any
  - (e) Let  $R$  be the list of scheduling relations from  $e$
  - (f) Sort  $R$  by delays, priorities, target event IDs, and IDs of the scheduling relations in the order of significance
  - (g) Create an empty queue  $Q'$
  - (h) For each scheduling relation  $r$  in  $R$  whose guard is true
    1. Evaluate parameters for the event  $e'$  that  $r$  points to
    2. Create an instance  $i_{e'}$  of  $e'$  and associate it with the parameters
    3. Set the time stamp of  $i_{e'}$  to be greater than the current model time by  $r$ 's delay
    4. Append  $i_{e'}$  to  $Q'$
  - (i) Create  $Q''$  by merging  $Q'$  with  $Q$  and preserving the order of events originally in  $Q'$  and  $Q$ . For any  $i' \in Q'$  and  $i \in Q$ ,  $i'$  appears before  $i$  in  $Q''$  if and only if the LIFO policy is used and the time stamp of  $i'$  is less than or equal to that of  $i$ , or the FIFO policy is used and the time stamp of  $i'$  is strictly less than that of  $i$ .
  - (j) Let  $Q$  be  $Q''$

event schedules itself to occur again. It also schedules the *Start* event if there are any available servers. The LIFO policy guarantees that both *Enter* and *Start* will be processed atomically, so it is not possible for the number of servers available to be changed by any other event in the queue after that value is tested by the guard of the scheduling relation from *Enter* to *Start*. In other words, once a car has entered the queue and has started being washed, its washing machine cannot be taken by another car.

The *Start* event simulates car washing by decreasing the number of available servers and the number of cars in the queue. The service time is “ $5.0 + 20.0 * \text{random}()$ .” After that amount of time, the *Leave* event occurs, which represents the end of service for that car. Whenever a car leaves, the number of available servers must be greater than 0 (since at least one machine will become available at that point), so the *Leave* event immediately schedules *Start* if there is at least one car in the queue. Due to atomicity provided by the LIFO policy, the model will test the queue

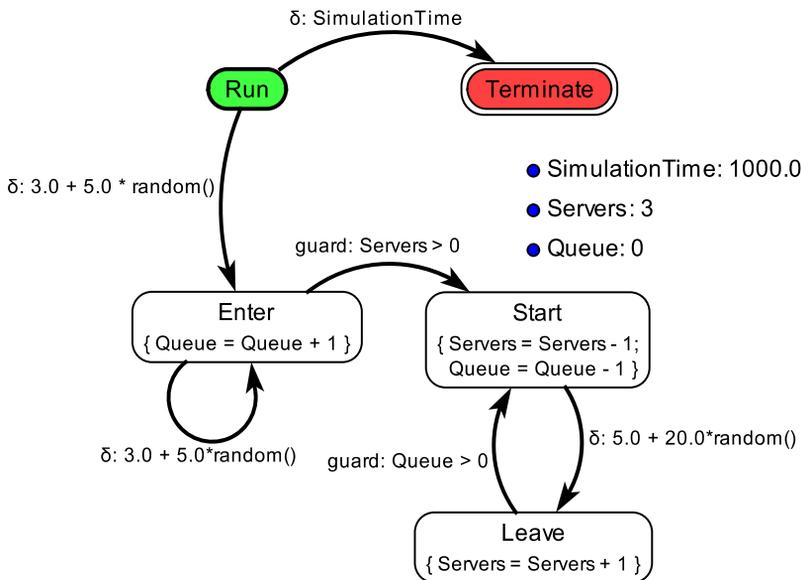


Figure 11.7: A model that simulates a car wash system. [online]

size and decrement its value during the subsequent *Start* event without allowing interruption by any other event in the queue.

The *Terminate* event, which halts execution, is prescheduled at the beginning of execution. Without this event, the model's execution would not terminate because the event queue would never be empty.

## 11.2 Hierarchical Models

**Hierarchy** can mitigate model complexity and improve reusability. **Hierarchical multi-modeling** enables combining multiple models of computation, each at its own level of the hierarchy. Here, we show how to construct hierarchical Ptera models. In the next section, we will show how to hierarchically combine Ptera models with other models of computation, such as those described in preceding chapters.

**Example 11.12:** Consider a car wash with two stations, one of which has one server, and one of which has three servers, where each station has its own queue. Figure 11.8 shows a hierarchical modification of Figure 11.7 with two such stations. Its top level simulates an execution environment, which has a *Run* event as the only initial event, a *Terminate* event as a final event, and a *Simulate* event associated with a submodel. The submodel simulates the car wash system with the given number of servers.

The two scheduling relations pointing to the *Simulate* event cause the submodel to trigger two instances of the *Init* event in the submodel's local event queue. These represent the start of two concurrent simulations, one with three servers (as indicated by the second argument on the left scheduling relation) and the other with one server. The priorities of the initializing scheduling relations are not explicitly specified. Because the two simulations are independent, the order in which they start has no observable effect. In fact, the two simulations may even occur concurrently.

Parameter  $i$  (the first argument on the two scheduling relations into *Init*) distinguishes the two simulations. Compared to the model in Figure 11.7, the *Servers* variable in the submodel has been extended into an array with two elements. *Servers*(0) refers to the number of servers in simulation  $i = 0$ , while *Servers*(1)

is used in simulation  $i = 1$ . The *Queue* variable is extended in the same way. Each event in the submodel also takes a parameter  $i$  (which specifies the simulation number) and sends it to the next events that it schedules. This ensures that the events and variables in one simulation are not affected by those in the other simulation, even though they share the same model structure.

It is conceptually possible to execute multiple instances of a submodel by initializing it multiple times. However, the event queue and variables would not be copied. Therefore,

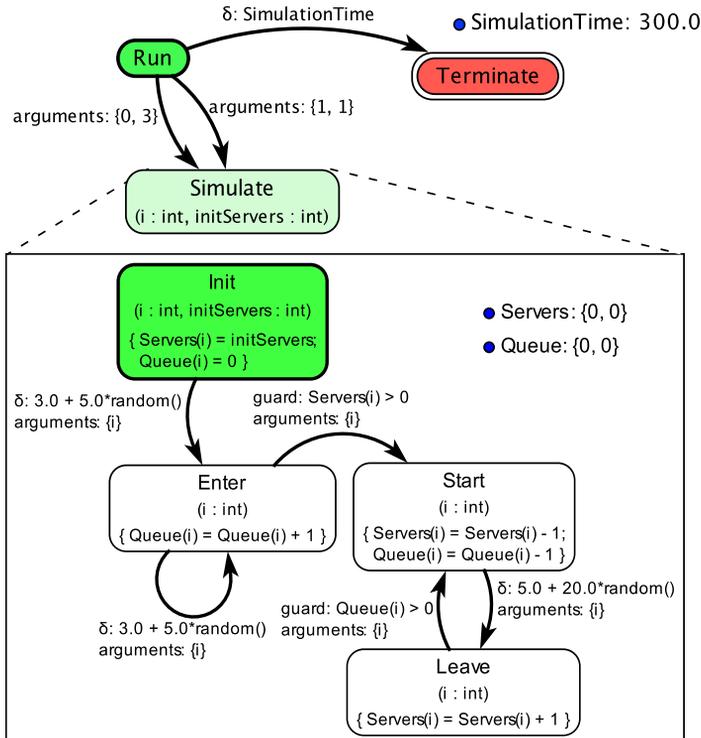


Figure 11.8: A hierarchical model that simulates a car wash system with two settings. [\[online\]](#)

the variables must be defined as arrays and an extra index parameter ( $i$  in this case) must be provided to every event.

[Actor-oriented classes](#) (see Section 2.6) provide an alternative approach to creating multiple executable instances of a submodel. The submodel can be defined as a class, with multiple instances executing in parallel.

## 11.3 Heterogeneous Composition

Ptera models can be composed with models built using other models of computation. Examples of such compositions are described in this section.

### 11.3.1 Composition with DE

Like Ptera models, discrete event (DE) models (discussed in Chapter 7) are based on events. But the notation in DE models is quite different. In DE, the components in the model, actors, consume input events and produce output events. In Ptera, an entire Ptera model may react to input events by producing output events, so Ptera submodels make natural actors in DE models. In fact, combinations of DE and Ptera can give nicely architected models with good separation of concerns.

**Example 11.13:** In Figure 11.8, the modeling of arrivals of cars is intertwined with the model of the servicing of cars. It is not easy, looking at the model, to separate these two. What if we wanted to, say, change the model so that cars arriving according to a [Poisson process](#)? Figure 11.9 shows a model that uses a DE director at its top level and separates the model of car arrivals from the servicing of the cars. This model has identical behavior to that in Figure 11.8, but it would be easy to replace the CarGenerator with a [PoissonClock](#) actor.

In Figure 11.9, in the CarGenerator, the *Init* event schedules the first *Arrive* event after a random delay. Each *Arrive* event schedules the next one. Whenever it is processed, the *Arrive* event generates a car arrival signal and sends it via the output port using the assignment “output = 1,” where “output” is the port name. In this case, the value 1 assigned to the output is unimportant, since only the timing of the output event is of interest.

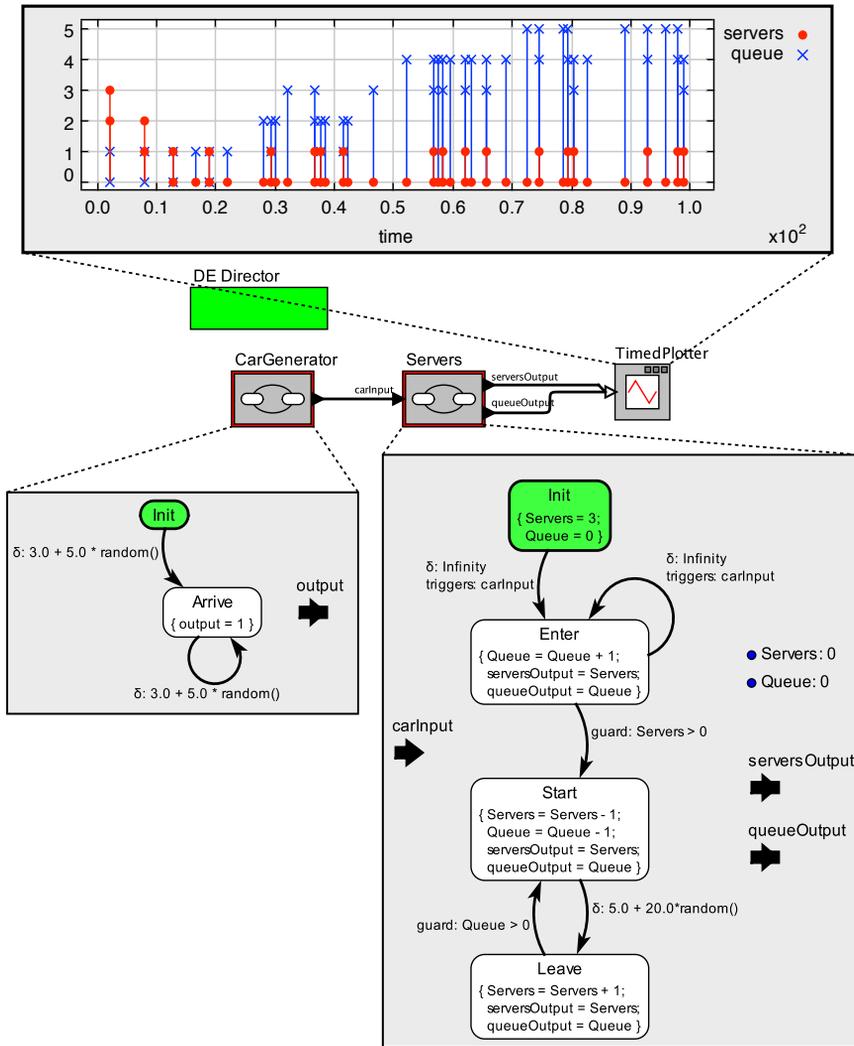


Figure 11.9: A car wash model using DE and Ptera in a hierarchical composition. [\[online\]](#)

Figure 11.9 also shows the internal design of Servers. It is similar to the previous car wash models, except that there is an extra *carInput* port to receive DE events

representing car arrival signals from the outside and the *Enter* event is scheduled to handle inputs via that port. No assumption is made in the Servers component about the source of the car arrivals. At the top level, the connection from CarGenerator's output port to Servers' input port makes explicit the producer-consumer relationship, and leads to a more modular and reusable design.

The *TimedPlotter* shows the number of servers available and the number of waiting cars waiting in the queue over time. In the particular trace shown in the figure, the queue builds up to five cars over time.

A Ptera model within a DE model will execute either when an input event arrives from the DE model or when a timeout  $\delta$  expires on a scheduling relation.

**Example 11.14:** In the Servers model in Figure 11.9, the relation from *Init* to *Enter* is labeled with  $\delta: \text{Infinity}$ , which means the timeout will never expire. It is also labeled with  $\text{triggers: carInput}$ , which means that the *Enter* event will be scheduled to occur when an event arrives from the DE model on the *carInput* port.

A scheduling relation may be tagged with a *triggers* attribute that specifies port names separated by commas. This can be used to schedule an event in a Ptera submodel to react to external inputs. The attribute is used in conjunction with the delay  $\delta$  to determine when the event is processed. Suppose that in a model the *triggers* parameter is " $p_1, p_2, \dots, p_n$ ." The event is processed when the model time is  $\delta$ -greater than the time at which the scheduling relation is evaluated *or* one or more DE events are received at *any* of  $p_1, p_2, \dots, p_n$ . To schedule an event that indefinitely waits for input, *Infinity* may be used as the value of  $\delta$ .

To test whether a port actually has an input, a special Boolean variable whose name is the port name followed by string "*\_isPresent*' " can be accessed, similarly to *FSMs*. To refer to the input value available at a port, the port name may be used in an expression.

**Example 11.15:** The *Enter* event in Figure 11.9 is scheduled to indefinitely wait for DE events at the *carInput* port. When an event is received, the *Enter* event is processed ahead of its scheduled time and its action increases the queue size by 1. In that particular case, the value of the input is ignored.

To send DE events via output ports, assignments can be written in the [action](#) of an event with port names on the left hand side and expressions that specify the values on the right hand side. The time stamps of the outputs are always equal to the model time at which the event is processed.

#### 11.3.2 Composition with FSMs

Ptera models can also be composed with untimed models such as [FSMs](#). When a Ptera model contains an FSM submodel associated with an event, it can fire the FSM when that event is processed and when inputs are received at its input ports. FMSs are described in Chapter 6.

**Example 11.16:** To demonstrate composition of Ptera and FSM, consider the case where drivers may avoid entering a queue if there are too many waiting cars. This can lead to a lower arrival rate (or equivalently, longer average interarrival times). Conversely, if there are relatively few cars in the queue, the driver would always enter the queue, resulting in a higher arrival rate.

The model is modified for this scenario and shown in Figure 11.10. At the top level, the *queueOutput* port of Servers (whose internal design is the same as Figure 11.9) is fed back to the *queueInput* port of CarGenerator. The FSM submodel in Figure 11.10 refines the *Update* event in CarGenerator. It inherits the ports from its container, allowing the guards of its transitions to test the inputs received at the *queueInput* port. In general, actions in an FSM submodel can also produce data via the output ports.

At the time when the *Update* event of CarGenerator is processed, the FSM submodel is set to its initial state. When fired the first time, the FSM moves into the

*Fast* state and sets the minimum interarrival time to be 1.0. Subsequently, the interarrival time is generated using the expression “1.0 + 5.0 \* random()”. Notice that the *min* variable is defined in CarGenerator, and the scoping rules enable the contained FSM to read from and write to that variable.

When a Ptera model receives input at a port, all the initialized submodels are fired, regardless of the models of computation those submodels use.

The converse composition, in which Ptera submodels are refinements of states in an FSM, is also interesting. By changing states, the submodels may be disabled and enabled, and execution can switch between modes. That style of composition is provided by modal models, described in Chapter 8.

## 11.4 Summary

Ptera provides an alternative to FSMs and DE models, offering a complementary approach to modeling event-based systems. Ptera models are stylistically different from either. Components in the model are events, vs. states in FSMs and actors in DE models. The scheduling relations that connect events represent causality, where one event causes another under specified conditions (guard expressions, timeouts, and input events).

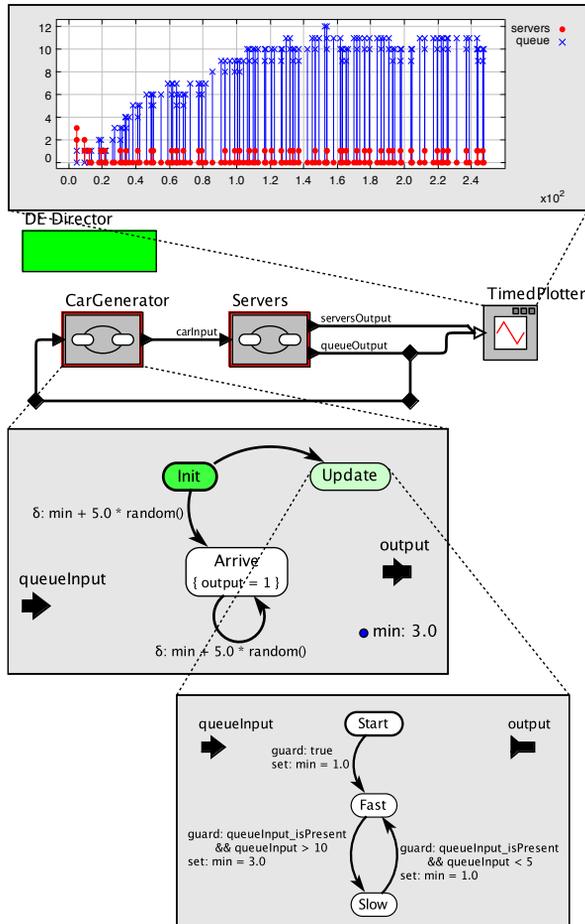


Figure 11.10: A car wash model using DE, Ptera and FSM in a hierarchical composition. [\[online\]](#)

## Part III

# Modeling Infrastructure

This part of this book focuses on the modeling infrastructure provided by Ptolemy II. Chapter 12 provides an overview of the software architecture with the goal of providing the reader with a good starting point for creating extensions of Ptolemy II. Chapter 13 describes the expression language used to set parameter values and perform computation in the [Expression](#) actor. Chapter 17 gives an overview of signal plotting capabilities provided in the actor library. Chapter 14 describes the Ptolemy II type system. Chapter 15 describes the ontology system. and Chapter 16 describes interfaces to the web, including mechanisms for exporting Ptolemy models to websites and mechanisms for creating custom web servers.