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.

7

Discrete-Event Models

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

Contents

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

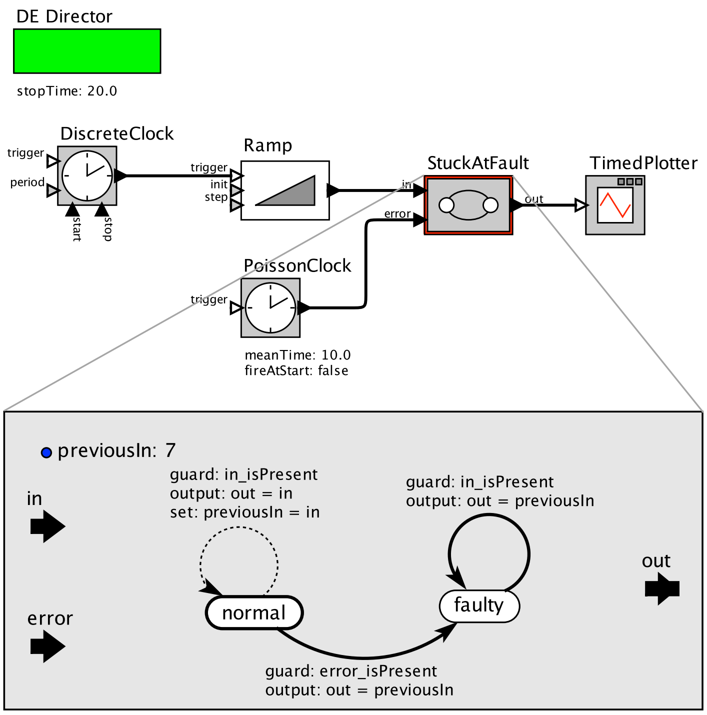


Figure 7.1: Simple example of a DE model. [online]

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.

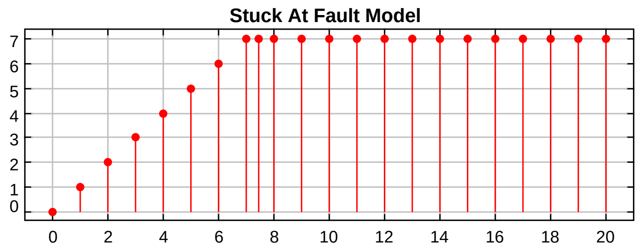


Figure 7.2: Sample output from the model in Figure 7.1.

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 and model time are indeed quite different; in this case, 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

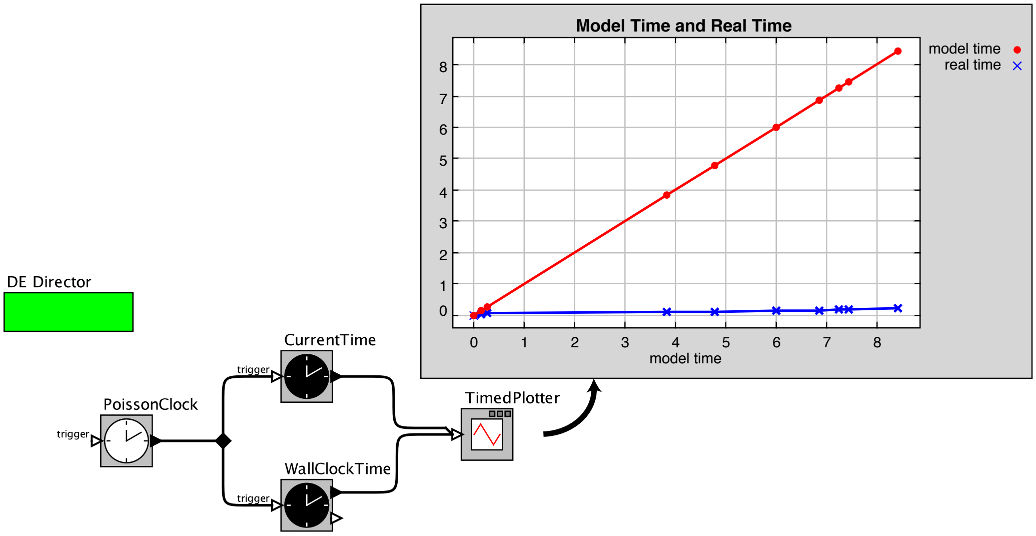


Figure 7.3: Model time vs. real time (wall clock time). [online]

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

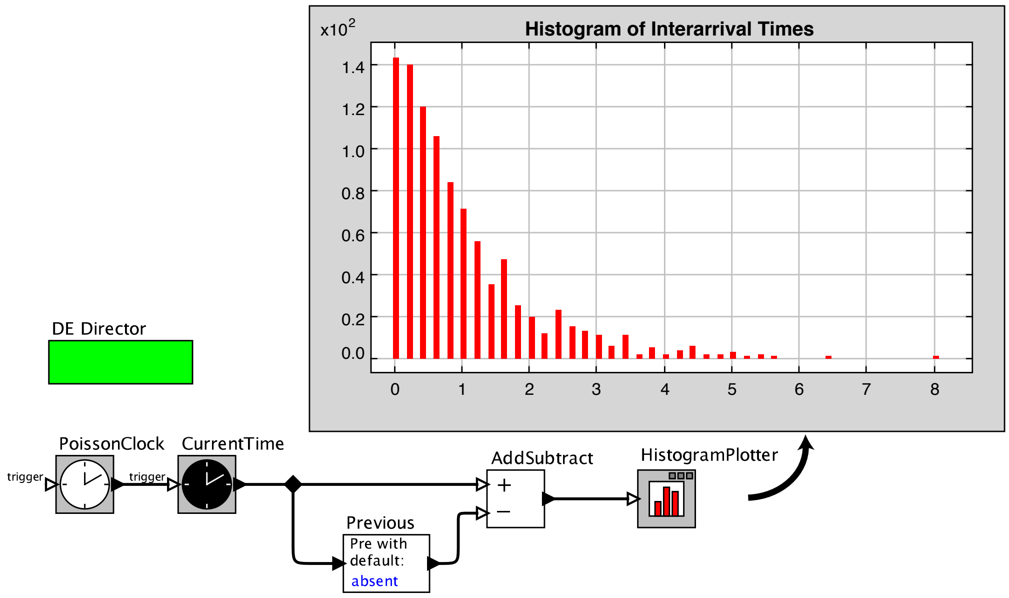


Figure 7.4: Histogram of interarrival times, illustrating handling of simultaneous events. There is a subtle bug in this model, corrected in Figures 7.5 and 7.6. [online]

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

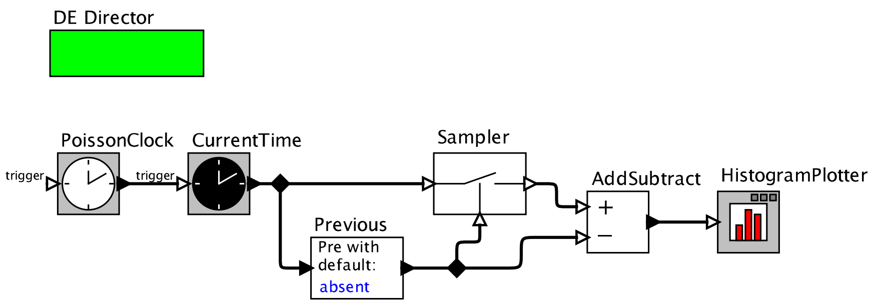


Figure 7.5: Histogram of interarrival times, correcting the subtle bug in Figure 7.4 using the Sampler actor. [online]

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.

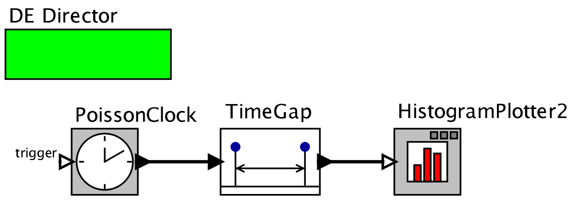
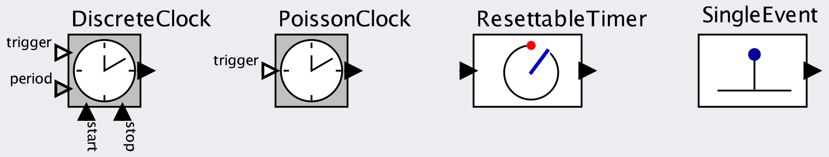


Figure 7.6: Histogram of interarrival times using the TimeGap actor. [online]

Sidebar: Clock Actors

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

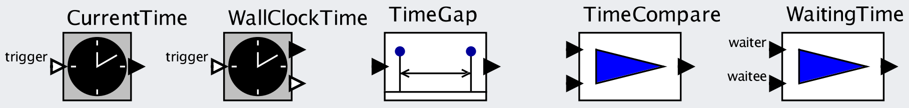


These actors are found in the `Sources→TimedSources` library, except for Resettable-Timer, which is in `DomainSpecific→DiscreteEvent`.

- **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:



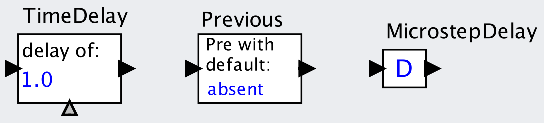
The **CurrentTime** actor, which is found in `Sources→TimedSources`, observes the model time of events. It produces an output event whenever an input event arrives, where the value of the output event is the model time of the input event. (The time value on the output of the CurrentTime actor is represented 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.

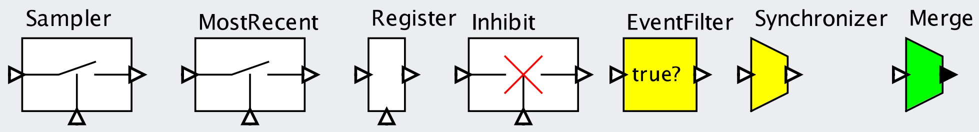
The amount of the delay can optionally be given on the bottom input port. If an event is provided on the bottom input port, then the value of that event specifies the delay for input events that arrive at the same time or later (i.e., that have a time stamp greater than or equal to that of the event 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

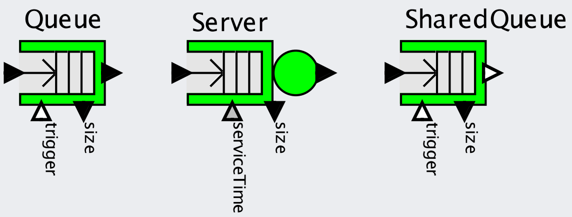
Discrete-event modeling based on time stamps has been around for a long time. One of the earliest complete discrete-event formalisms is called DEVS, for Discrete Event System Specification (Zeigler, 1976). Many subtly different variants have appeared over time (see for example Ramadge and Wonham (1989), 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.

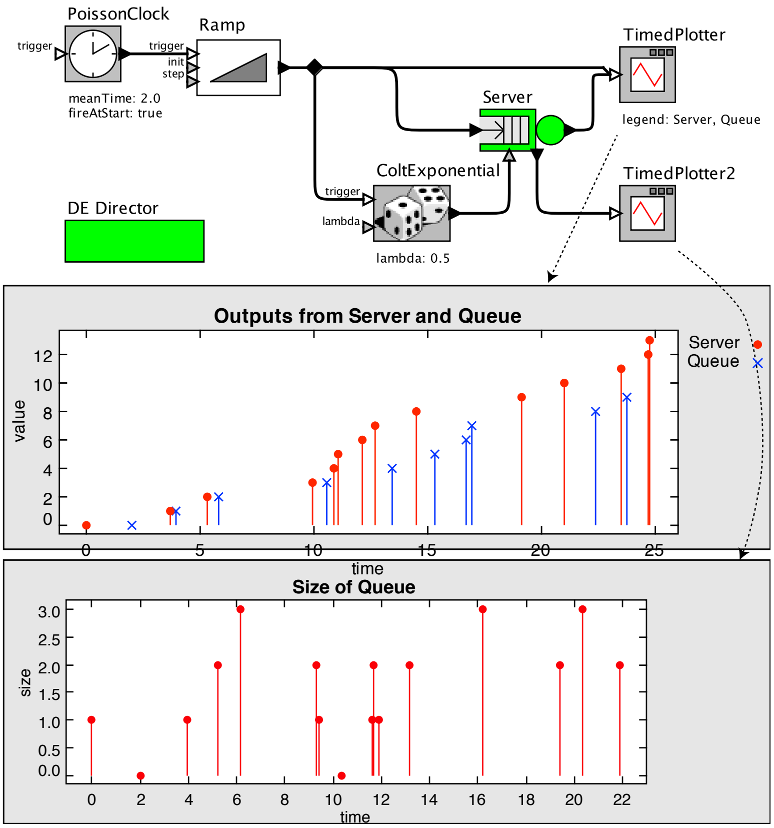


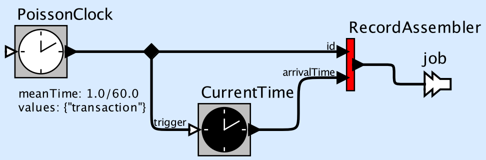
Figure 7.7: A model of an M/M/1 queue. [online]

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:



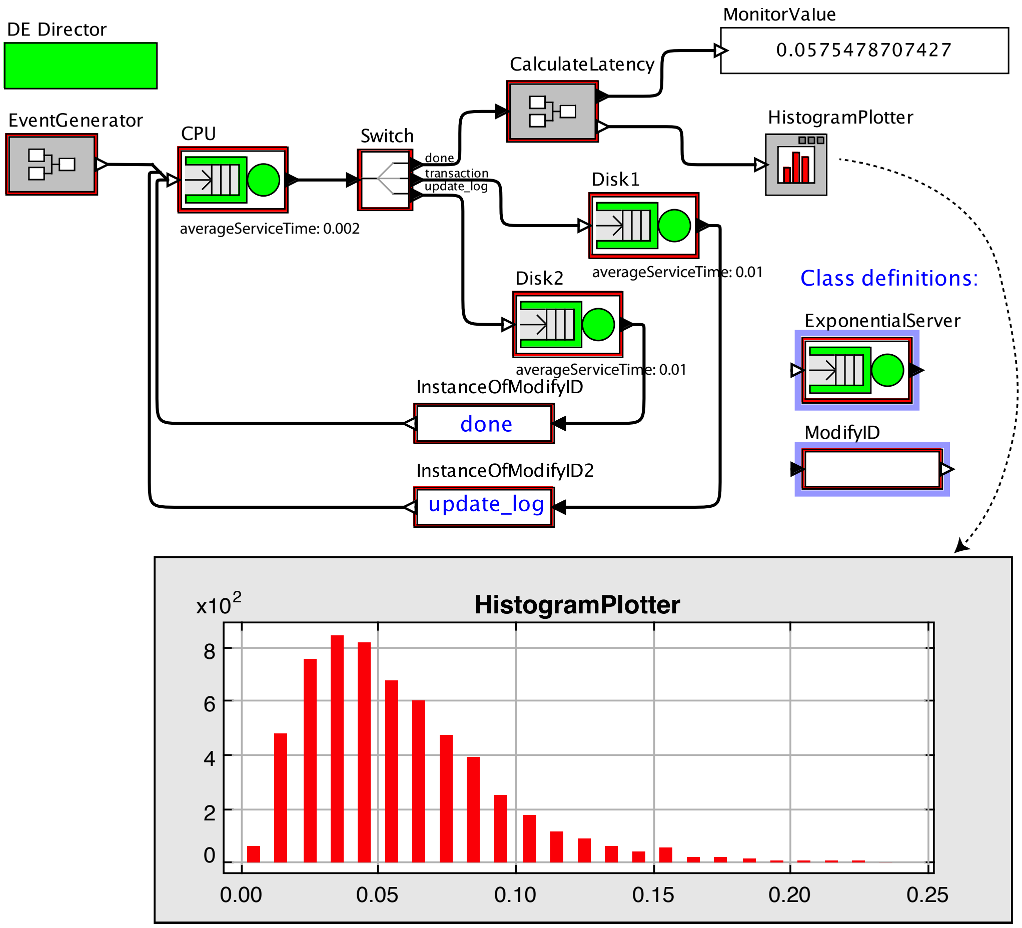
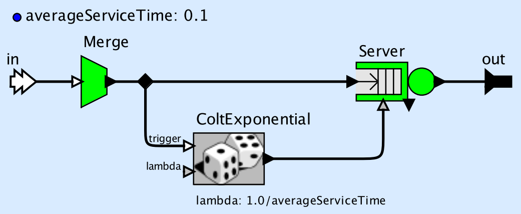


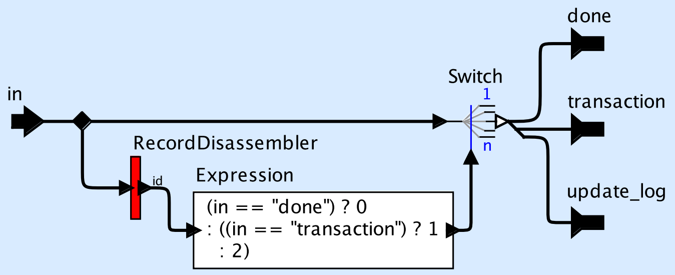
Figure 7.8: A queueing model of a transaction processor that writes to two disks. [online]

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:



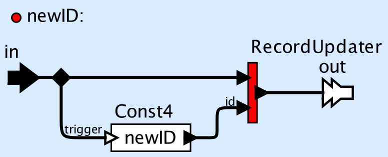
This composite actor has a single parameter, *averageServiceTime*. It implements a server with a random service time. Upon arrival of an event on any input channel, the ColtExponential actor generates a new random number from an exponential distribution. This random number specifies the service time 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 serviced 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:

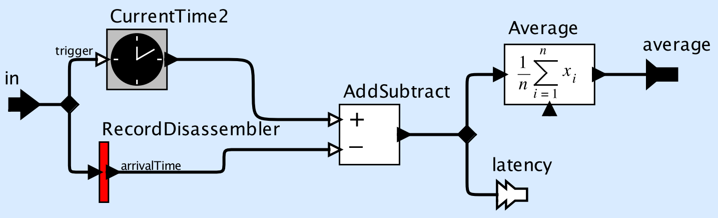


This actor extracts the *id* field from the incoming record, and, using an Expression and a Switch actor, routes the input event to one of three output ports depending on the *id*.

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

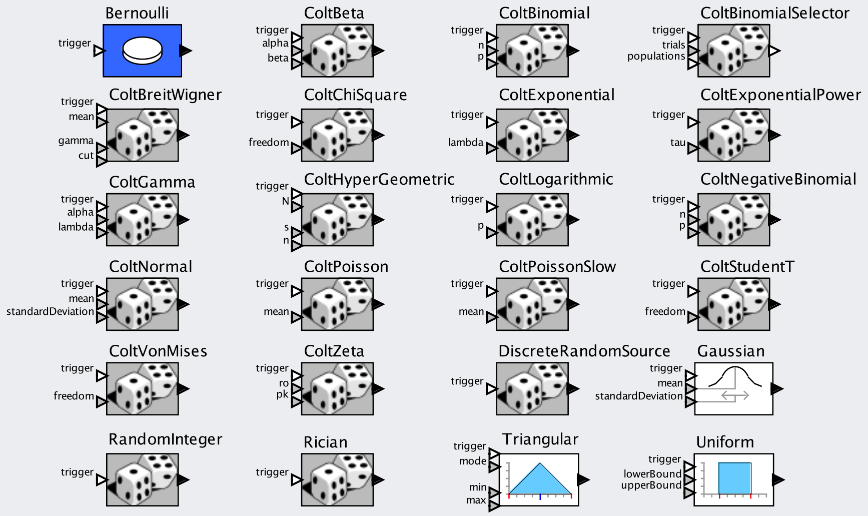
The main task of the DE director is to fire actors in time-stamp order. The DE director maintains an event queue that is sorted by time stamp, first by model time and then by microstep. The DE director controls the execution order by selecting the earliest event in the event queue and making its time 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.[†]

[†]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:



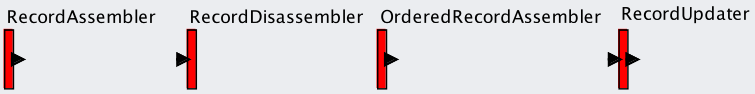
The actors all produce a random number on their output port on each firing. Most of them have parameters that can also be set on input ports, so that parameters can vary on each firing. See the documentation of each actor for details.

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

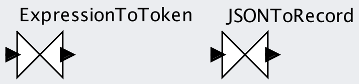
The record type in Ptolemy II is similar to a `struct` in C. It can contain any number of named fields, each of which can have an arbitrary data type (it can even be a record). There are several actors that manipulate records, found in [`Actors→FlowControl→Aggregators`] and shown below.



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.

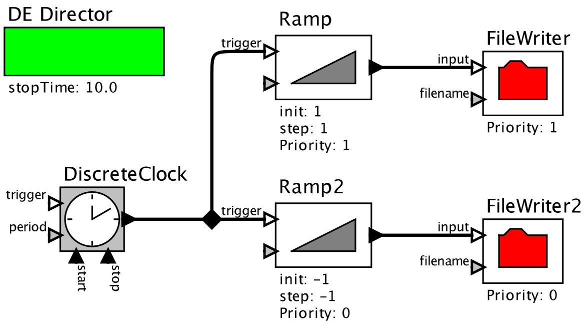


Figure 7.9: Although it is rarely necessary, it may sometimes be useful to set the priorities of actors in DE. This only has an effect when the firing order is otherwise not determined by time stamps or by data precedences (levels). [online]

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.

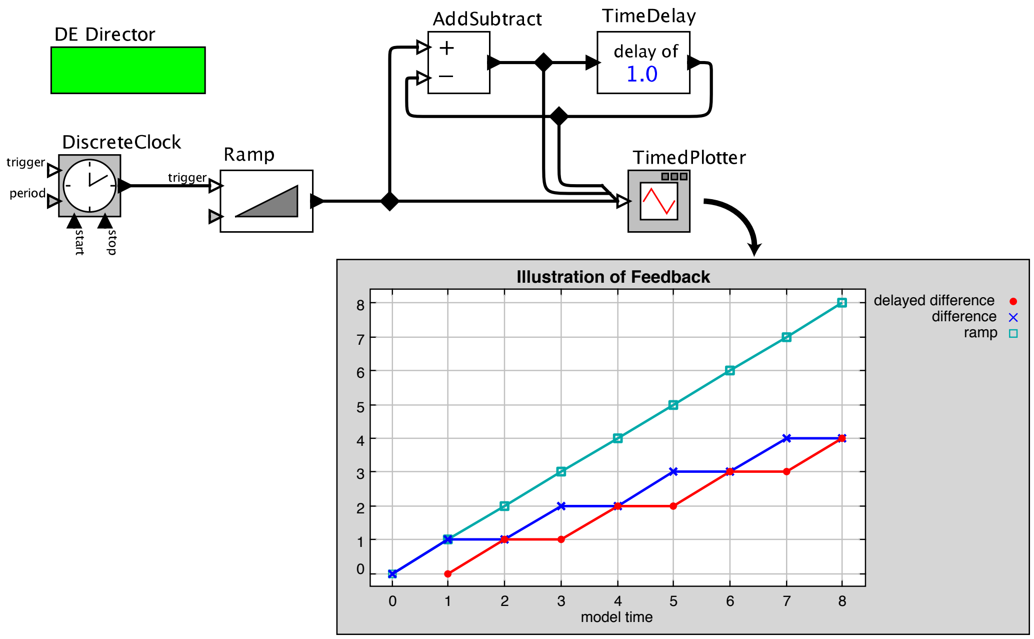


Figure 7.10: Discrete-event model with feedback, which requires a delay actor such as TimeDelay. [online]

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.

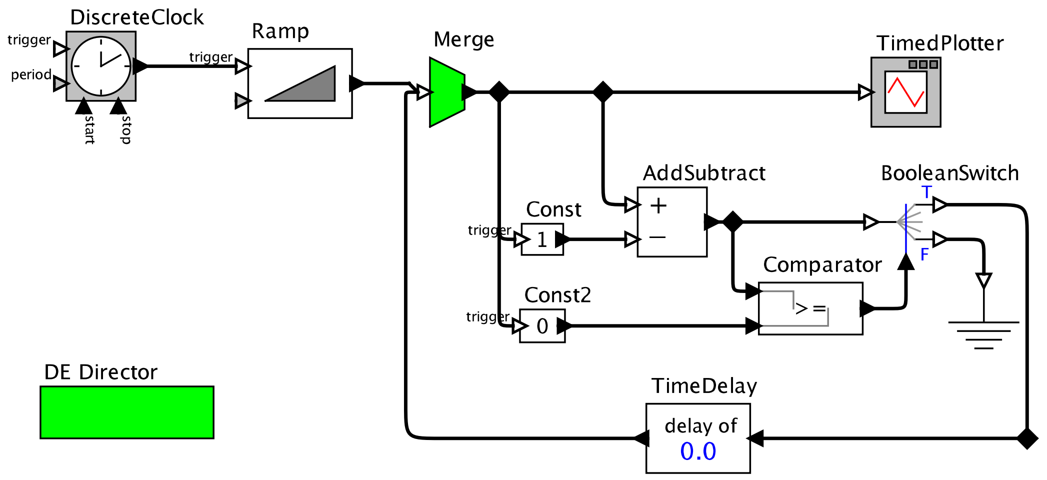


Figure 7.11: Illustration of TimeDelay with delay value of zero. [online]

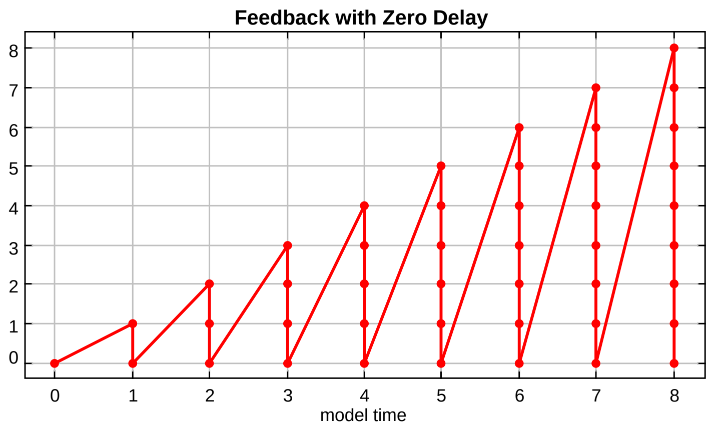


Figure 7.12: Result of executing the model in Figure 7.11.

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.

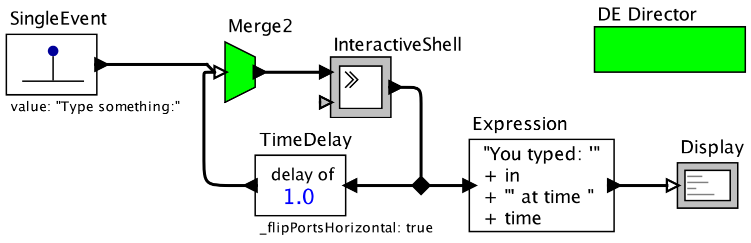
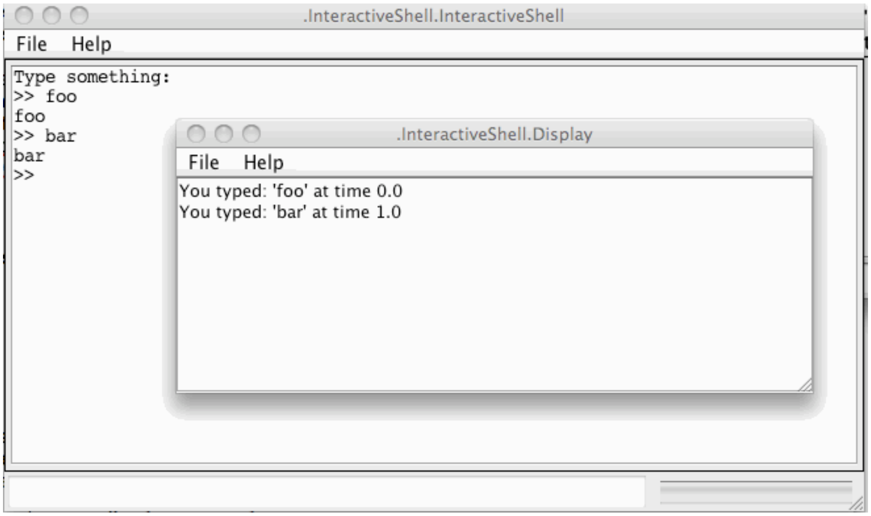


Figure 7.13: A DE model using the InteractiveShell actor, whose execution is stalled until a user types something. [online]



```
Type something:
>> foo
foo
>> bar
bar
>>
```

```
You typed: 'foo' at time 0.0
You typed: 'bar' at time 1.0
```

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

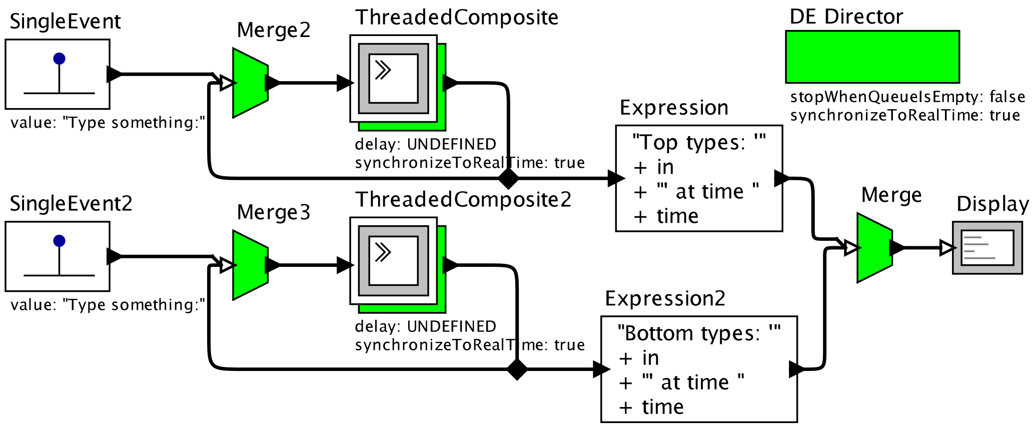
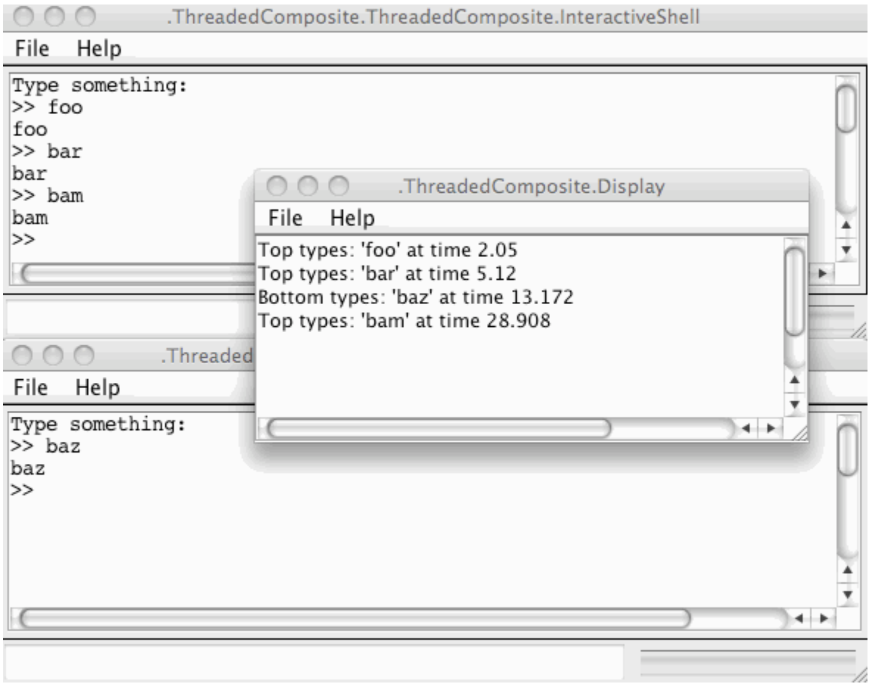


Figure 7.15: A model where two instances of InteractiveShell are executed in separate threads by the ThreadedComposite actor. [online]


.ThreadedComposite.ThreadedComposite.InteractiveShell
File Help
.ThreadedComposite.Display
File Help
.Threaded
File Help

```
Type something:
>> foo
foo
>> bar
bar
>> bam
bam
>>
```

```
Top types: 'foo' at time 2.05
Top types: 'bar' at time 5.12
Bottom types: 'baz' at time 13.172
Top types: 'bam' at time 28.908
```

```
Type something:
>> baz
baz
>>
```

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 τ, 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 τ, 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
```

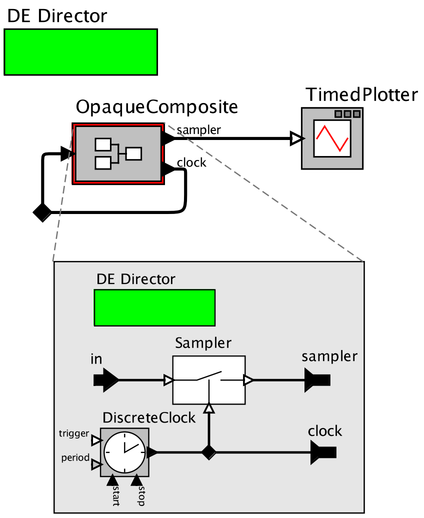


Figure 7.17: A model that should, in theory, be executable, but is not executable in the current implementation of DE.

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.

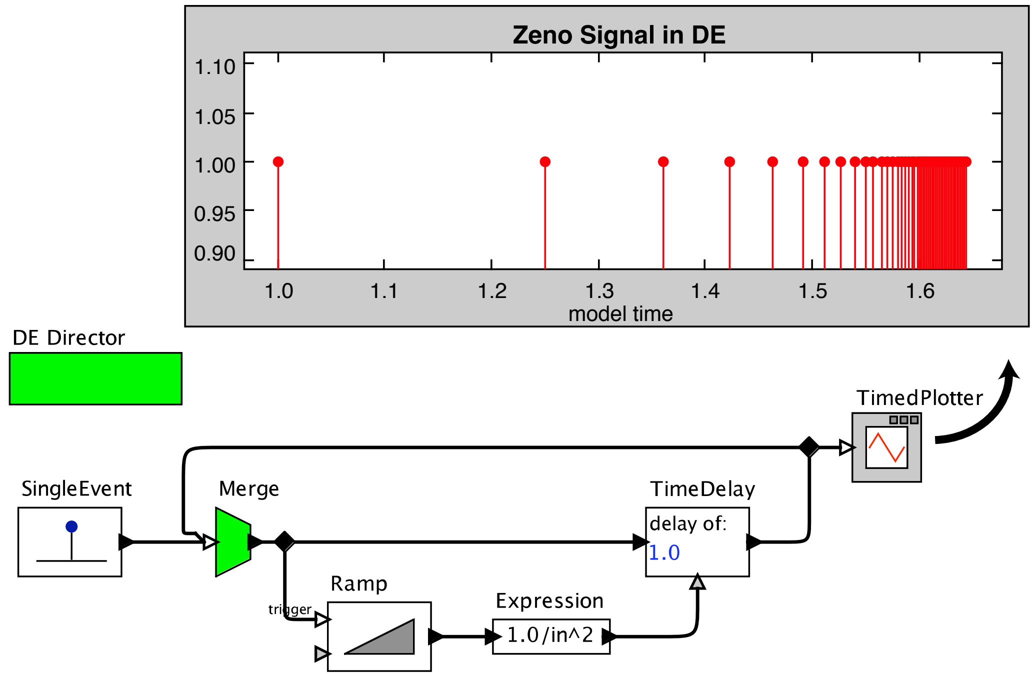


Figure 7.18: Example of a Zeno model, where the model time fails to advance. [online]

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 FSMActor. 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.

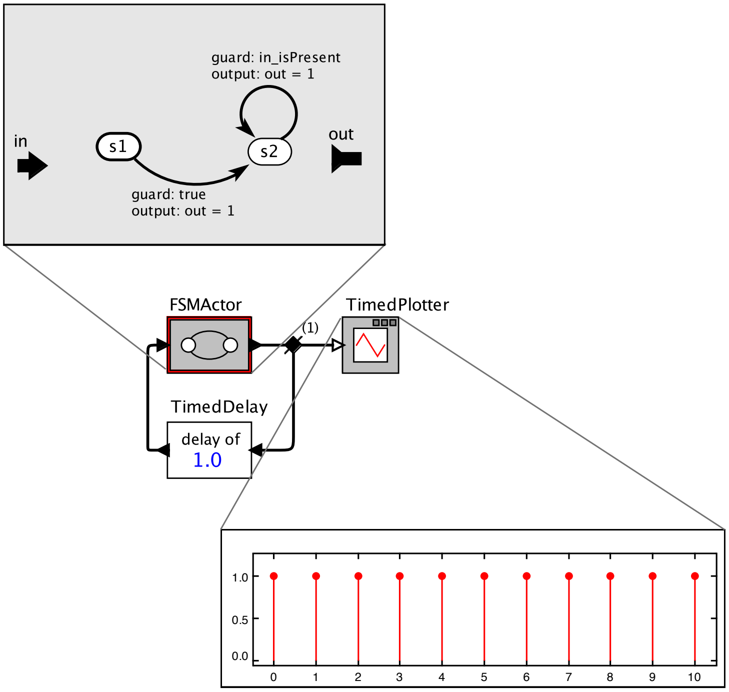


Figure 7.19: A simple DE model containing an FSM. [online]

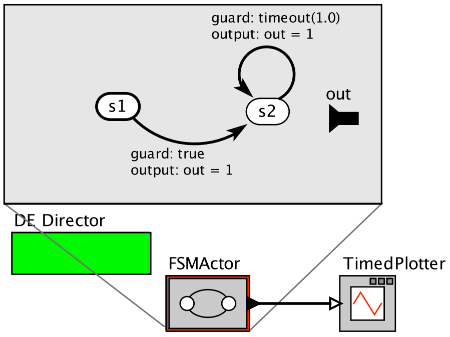


Figure 7.20: A DE model with identical behavior to that in Figure 7.19, but containing an FSM that controls timing with the timeout function. [online]

After the first firing, the actor is in state *s2*, 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.

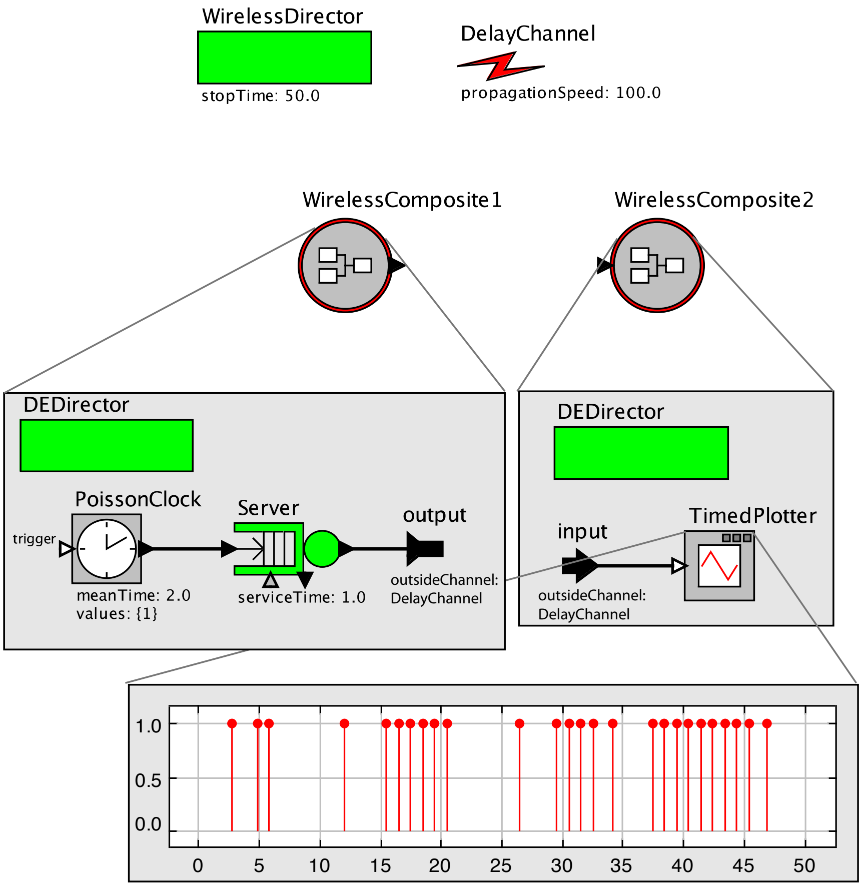


Figure 7.21: A model of a wireless system. [online]

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?

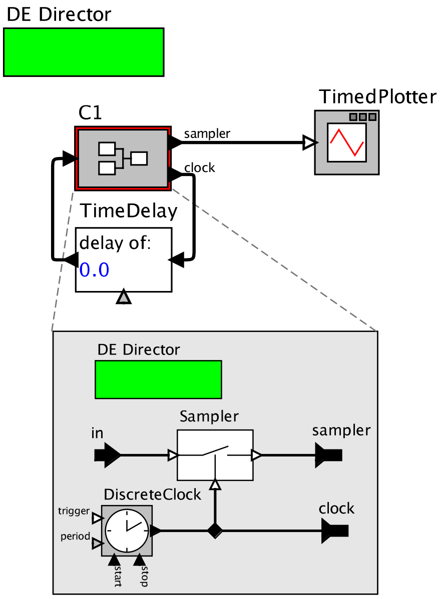


Figure 7.22: A model that is executable but is not equivalent to the model in Figure 7.17. [online]

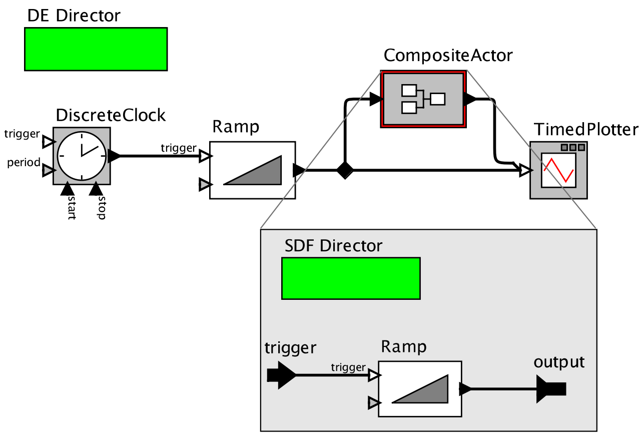


Figure 7.23: Simple example of an SDF submodel within a DE model. [online]

(b) Find values of the *period* parameter of the SDF director and of the DiscreteClock 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

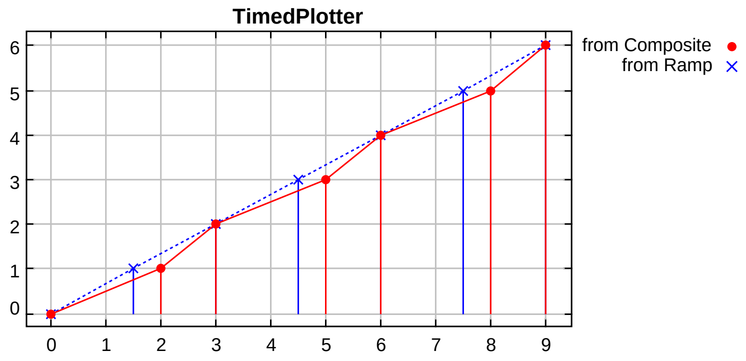


Figure 7.24: A plot that can be generated by the model in Figure 7.23.

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.