This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

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.

11

Ptera: An Event-Oriented Model of Computation

Thomas Huining Feng and Edward A. Lee

Contents

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.

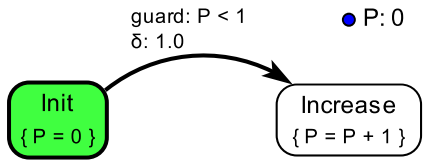


Figure 11.1: A simple Ptera model with two events. [online]

In a hierarchical Ptera model, vertices can also represent *submodel*s, 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`). 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 δ 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.

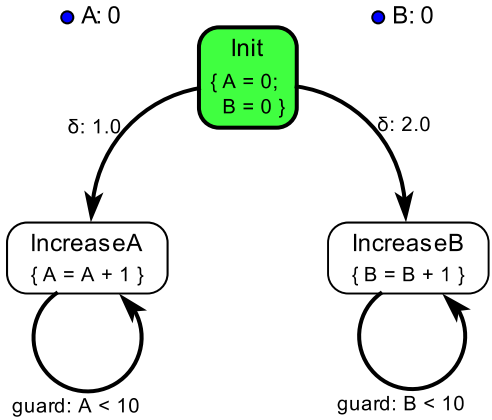


Figure 11.2: A model with multiple events in the event queue. [online]

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 δ 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.

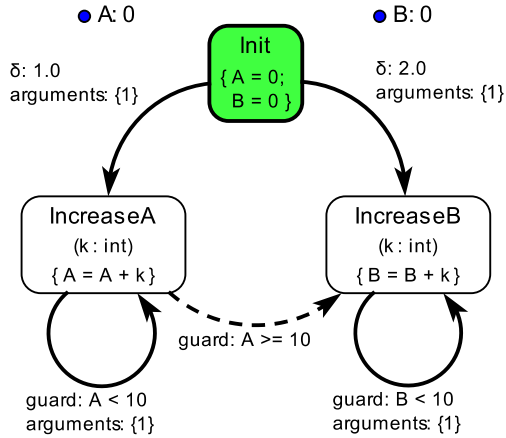


Figure 11.3: A model with arguments for the events and a canceling relation. [online]

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 δs 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.

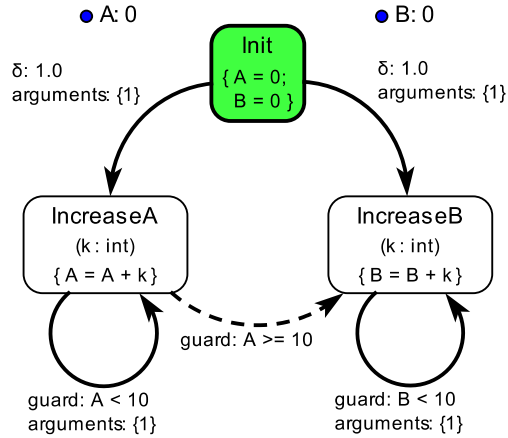


Figure 11.4: A model with simultaneous events. [online]

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	*Init*	*IncreaseA*	*IncreaseA*	...	*IncreaseA*
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	*Init*	*IncreaseB*	*IncreaseB*	...	*IncreaseB*	*IncreaseA*
A	0	0	0	...	0	1
B	0	1	2	...	10	10

Time	1.0	...	1.0
Event	*IncreaseA*	...	*IncreaseA*
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	*Init*	*IncreaseA*	*IncreaseB*	*IncreaseA*	*IncreaseB*	...
A	0	1	1	2	2	...
B	0	0	1	1	2	...

Time	1.0	1.0	1.0
Event	*IncreaseA*	*IncreaseB*	*IncreaseA*
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	*Init*	*IncreaseB*	*IncreaseA*	*IncreaseB*	*IncreaseA*	...
A	0	0	1	1	2	...
B	0	1	1	2	2	...

Time	1.0	1.0	1.0	1.0
Event	*IncreaseB*	*IncreaseA*	*IncreaseB*	*IncreaseA*
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.

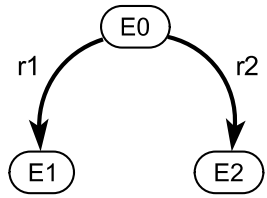


Figure 11.5: A scenario where event E0 schedules *E1* and *E2* after the same delay.

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 δ, **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 δ 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 δ 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.

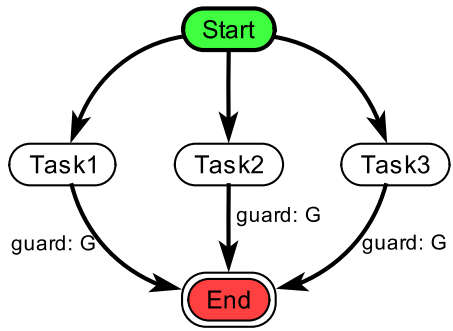
11.1.9 Designs with Atomicity

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 * random()," where 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 * 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

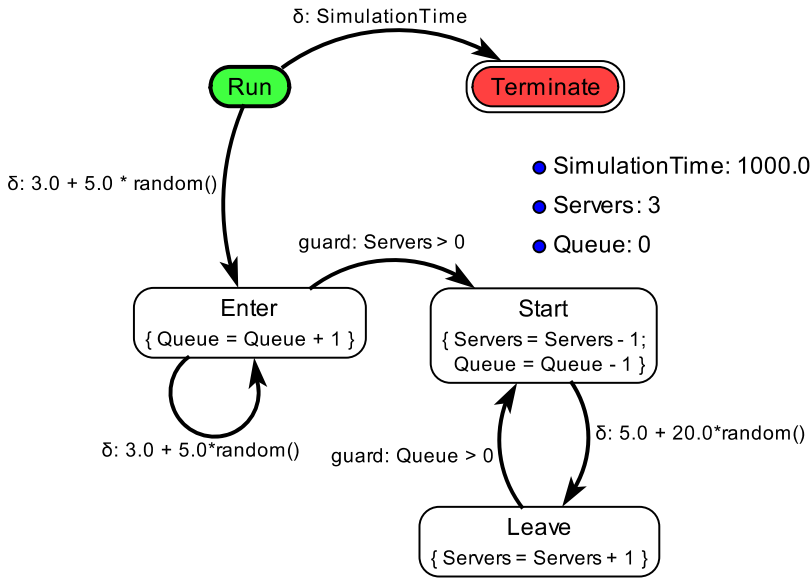


Figure 11.7: A model that simulates a car wash system. [online]

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 serves, 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,

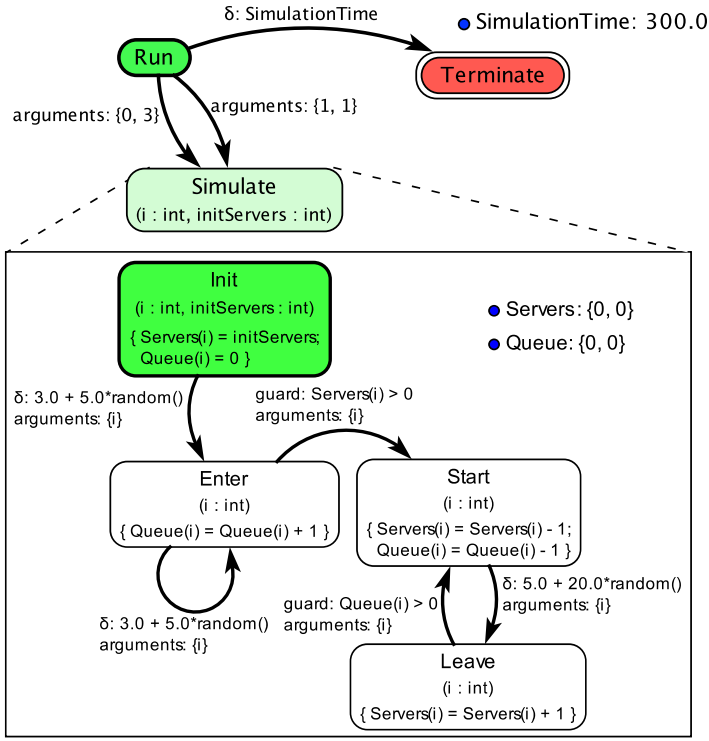


Figure 11.8: A hierarchical model that simulates a car wash system with two settings. [online]

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 is 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.

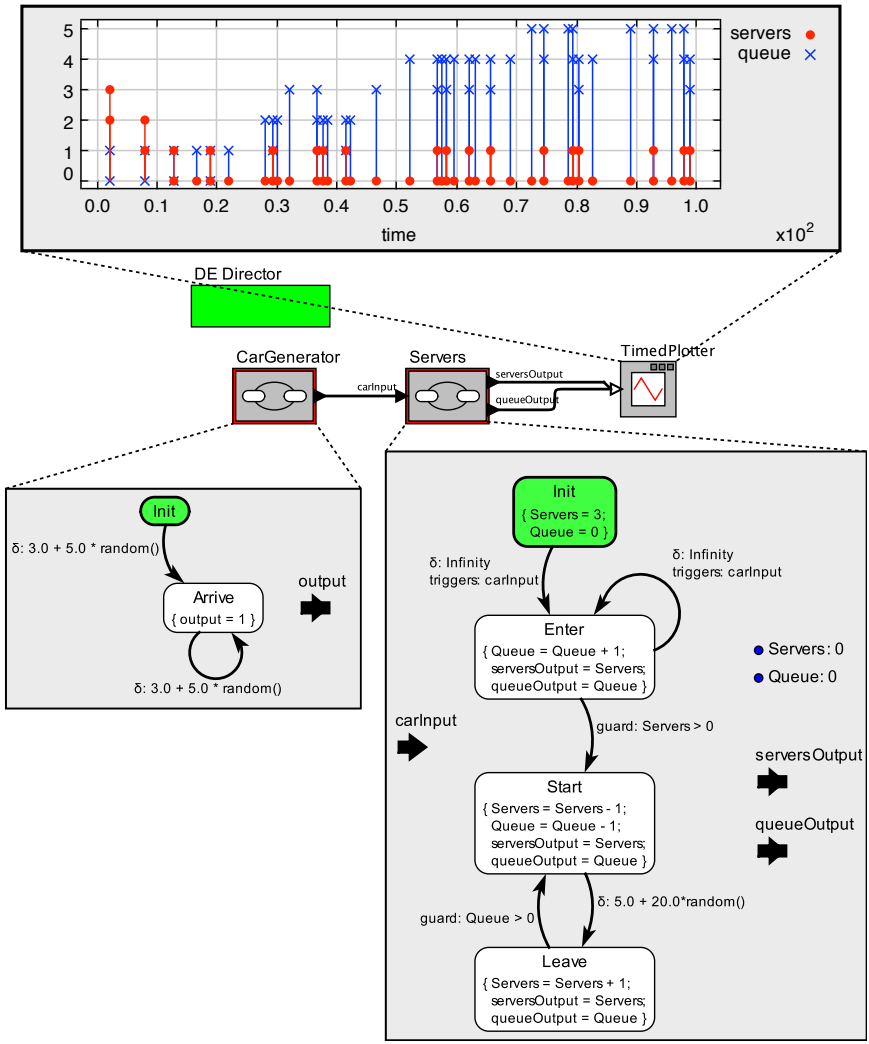


Figure 11.9: A car wash model using DE and Ptera in a hierarchical composition. [online]

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 δ 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 `δ: Infinity`, which means the timeout will never expire. It is also labeled with `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 δ to determine when the event is processed. Suppose that in a model the *triggers* parameter is "$p_1, p_2, \cdots, p_n$." The event is processed when the model time is δ-greater than the time at which the scheduling relation is evaluated *or* one or more DE events are received at *any* of $p_1, p_2, \cdots, p_n$. To schedule an event that indefinitely waits for input, `Infinity` may be used as the value of δ.

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).

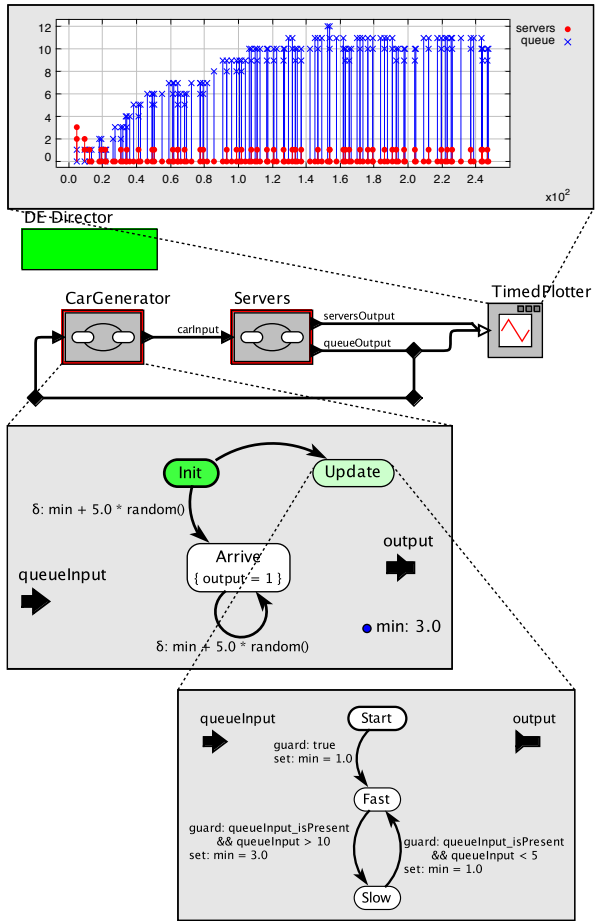


Figure 11.10: A car wash model using DE, Ptera and FSM in a hierarchical composition. [online]