Ptera: An Event-Oriented Model of Computation

Thomas Huining Feng
Edward A. Lee
Lee W. Schruben

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-40
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-40.html

April 10, 2010

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyc) and the following companies: Agilent, Bosch, National Instruments, Thales, and Toyota.

Ptera: An Event-Oriented Model of Computation

Thomas Huining Feng
Oracle Corporation
thomas.feng@oracle.com

Edward A. Lee
EECS, UC Berkeley
eal@eecs.berkeley.edu

Lee W. Schruben
IEOR, UC Berkeley
lees@berkeley.edu

April 10, 2010

Abstract

In event-oriented modeling, designers focus on the events that occur in time and on the causality relationship between events. This practice complements class-oriented, object-oriented, actor-oriented and state-oriented approaches. To facilitate event-oriented modeling, we have extended event graphs to create Ptera (Ptolemy event relationship actors), which we show to be appropriate for modeling complex discrete-event systems. A key capability is that Ptera models conform with an actor abstract semantics that permits hierarchical composition with other models of computation such as discrete-event actors, dataflow, process networks and finite state machines. This enables their use in complex system design, where not every aspect of the system is best described with event-oriented modeling.

1 Introduction

Event-oriented modeling allows designers to focus on *events* that occur in a process and the causality relationship between events. In the Unified Modeling Language (UML), activity diagrams can be considered event-oriented models. 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 shape is a special activity that tests a certain condition, and causes either the activity on its true branch or that on its false branch to take place next. However, it does not allow more than one branch to be taken at a time, and therefore the run-time state is entirely captured by the single current activity and the values of the global variables. Though activity diagrams are easy to understand, we realize that they are not always expressive enough for practical applications. Examples of its limitations are the lack of a notion of time, inability to schedule multiple future activities, the lack of support for model hierarchy, and the lack of concurrency.

In [1], one of us proposed event graphs as a visual formalism for event-oriented modeling. Blocks in an event graph are events with optional textual actions. Directed edges between events represent scheduling relations that can be guarded by Boolean expressions. Event graphs are timed, and model-time delays can be associated with scheduling relations. There is an event queue for each event graph that is not explicitly shown in the visual representation. [1] In each step of an execution, the execution engine removes the first imminent event from the event queue and processes it. The actions associated with that event are executed as a side effect, and events that it schedules with outgoing scheduling relations are inserted into the event queue. We find event graphs more expressive and more suitable for specifying timed systems than UML activity diagrams.

In this paper we introduce Ptera (Ptolemy event relationship actors) as a composable modal of computation that extends the existing event graphs. It has the following key extensions:

- Composition of Ptera models forms a hierarchical model, which can be flattened to obtain an equivalent model without hierarchy.
- Ptera models conform with an actor abstract semantics, which we will explain, that permits them to contain or be contained by other types of models, making it possible to create hierarchical heterogeneous designs.
- An interface consisting of parameters and input and output ports is revealed to the outside. Changes on the parameter values and arrival of data at input ports can trigger events that have been registered to react to them. Ptera models become actors.
- Actions of an event can be customized by the designer with programs in an imperative language (such as Java or C) conforming to a protocol.

We have implemented Ptera in Ptolemy II [13], and it is available in open source form.[2]

There have been various extensions to event graphs that aim to incorporate hierarchy into them. Two approaches are discussed in [2]. One is to associate submodels with scheduling relations. The only meaningful output of a submodel is a number used as the model-time delay for the scheduling relation. Another approach is to associate submodels with events instead of scheduling relations [3]. Processing such an event causes the unique start event in the submodel to be scheduled. That start event 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

[1] In multi-threaded simulation, multiple event queues may be used, which is out of the scope of this paper.

[2] The Ptera models discussed in this paper are all available for editing and execution at http://ptolemy.org/ptera. The PDF version of this paper provides hyperlinks to the appropriate model in every figure depicting the model. Just click on the figure to execute and edit the model. Ptolemy II can be obtained from http://ptolemy.org.

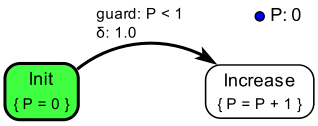


Figure 1: A simple model with two events that can be executed with a size-one event queue.

is considered processed. In [4], a third attempt is reported, in which a listener pattern is introduced as an extra gluing mechanism for composing event graphs. We believe that Ptera models are more flexible because they can be freely composed with heterogeneous models such as dataflow [5], finite state machines, and actor-oriented discrete-event models [6, 7].

The remaining sections are organized as follows. In Section 2, we discuss the syntax and semantics of flat models, where no hierarchical composition is involved. They are generalized into hierarchical models in Section 3. In Section 4, we demonstrate two types of heterogeneous hierarchical composition, and use an abstract semantics to explain their execution behavior. In Section 5 we provide three practical applications. Our work is compared to related work in Section 6. A conclusion is provided in Section 7.

2 Syntax and Semantics of Flat Models

A flat Ptera model is an attributed graph containing vertices connected with directed edges. Vertices may be associated with such attributes as *ID*, *actions*, *final*, *initial* and *parameters*. Edges may be associated with such attributes as *ID*, *arguments*, *canceling*, *delay*, *guard*, *initializing*, *priority* and *triggers*. All these attributes are included in the following discussion.

A hierarchical Ptera model is a generalization of flat models since the vertices in it may be additionally associated with a graph that represents a *submodel*. Submodels may themselves be Ptera models, and in general, they may also be models in other models of computation, such as FSMs (finite state machines), actor models, class diagrams and even Java code. The only requirement is that their behavior can be defined in an abstract semantics studied in the Ptolemy project, explained below.

Flat models are a simpler kind of model in which vertices are not associated with submodels.

2.1 Introductory Examples

A vertex in a model is called an *event*. A example model with two events is shown in Figure 1. Each event has a unique *identifier* (*ID*), which is displayed on its icon. In this case, the events' IDs are Init and Increase.

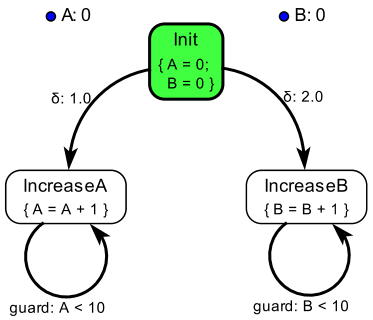


Figure 2: A model with multiple events in the event queue.

Variables in a model associate values with names. Each variable is visually represented as a name-value pair to the right of a dot. In the example, there is a single variable with name P and initial value 0. The initial value is provided by the model designer and is shown in the static model design. During execution, the variable may be updated with new values.

Init is an *initial event* (with the *initial* attribute set to true). This is indicated with a filled vertex having a thick border. At the beginning of an execution, all the initial events are scheduled to occur at model time 0. (As discussed later, even though they all conceptually occur at time 0, there is a well-defined order.) An *event queue* exists in the execution that can hold an unbounded number of scheduled events. An event is removed from the event queue and is processed when the model time reaches the time at which the event is scheduled to occur (called the *time stamp* of that event).

Events may be associated with *actions* that are specified with a list of assignments separated by semicolons. In Figure 1, Init has action "P = 0", which causes the side effect of setting variable P to 0 when Init is processed. The edge from Init to Increase is called a *scheduling relation*. It has Boolean expression "P < 1" as its *guard* and 1.0 as its *delay* represented by symbol δ. This means, after the Init event is processed, if P's value is less than 1 (which is true in this case), then Increase would be scheduled 1.0 unit of model time later than the current model time (which is 0). When Increase is processed at model time 1.0, its action "P = P + 1" is executed and P's value is increased to 1.

After processing Increase, the event queue becomes empty, and since no more event is scheduled, the execution terminates. In general, execution terminates when there is no event left in the event queue.

In the model in Figure 1, it is clear that there is at most one event in the event queue (either Init or Increase) at any time. In general, an unbounded number of events can be scheduled in the event queue.

As another example, execution of the model in Figure 2 requires an event queue of size greater than 1. The Init event schedules IncreaseA and IncreaseB to occur after 1.0 unit of model time and 2.0 units of model time, respectively.

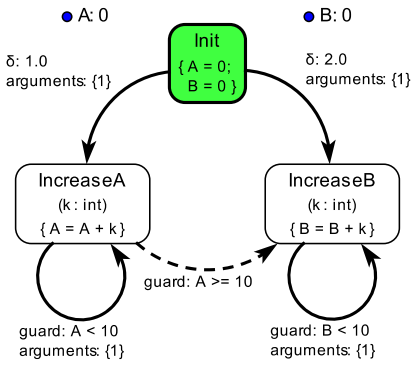


Figure 3: A model with arguments for the events and a canceling relation.

The guards of the two scheduling relations from Init take the default value "true," and are thus hidden in the visual representation. When IncreaseA is processed, it increases variable A by 1 and reschedules itself, 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. Similarly, IncreaseB repeatedly increases variable B at the current model time, until B's value reaches 10.

2.2 Arguments

Resembling a C function, an event may have a list of formal *arguments* (separated by commas). Each argument has a name and a type (separated by a colon). Figure 3 modifies Figure 2 by adding arguments k of type int to events IncreaseA and IncreaseB. These arguments are given 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 next.)

Each scheduling relation pointing to an event with arguments must specify a list of expressions in its *arguments* attribute. Those expressions are used to compute the actual values for the arguments when the event is processed. In the example, all scheduling relations pointing to IncreaseA and IncreaseB specify "{1}" in their arguments attributes, meaning that k should take value 1 when those events are processed. Values of the arguments declared by an event can be accessed in the event's actions and the guards and delays of the scheduling relations emanating from that event.

2.3 Canceling Relations

A *canceling relation* is represented as a dashed edge between events. It can be guarded by a Boolean expression. Its delay must be 0 and its arguments must be an empty list "{}," which are both hidden.

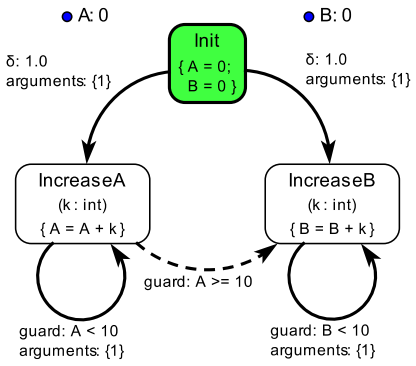


Figure 4: A model with simultaneous events.

When an event with an outgoing canceling relation is processed, if the guard is true and the event pointed to has been scheduled in the event queue, then that scheduled event would be removed from the event queue immediately without being processed. This yields the effect of canceling a previously scheduled event. If the event pointed to is scheduled multiple times, with multiple *instances* of it in the event queue (each of which belongs to the same event but may be associated with a different list of arguments), then the canceling relation causes only the first one in the event queue to be removed. If the event pointed to is not scheduled, the canceling relation has no effect.

Figure 3 provides an example of 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.

Canceling relations do not increase expressiveness. In fact, a model with canceling relations can always be converted into one without canceling relations, as is shown in [8]. Nonetheless, they can be convenient, making more compact and understandable models possible.

2.4 Simultaneous Events

Simultaneous events are events in a model that potentially have instances coexisting in the event queue and that are scheduled to occur at the same model time.

For example, consider setting both δ's in Figure 3 to 1.0, which yields the model in Figure 4. That makes IncreaseA and IncreaseB simultaneous events. Notice that instances of simultaneous events may not always occur at the same time. For example, if the δ's were set to $a + b$ and $a * b$ respectively, where a and b's values are not determined, then only some instances of IncreaseA and IncreaseB would occur at the same time. Moreover, even though multiple instances of IncreaseA occur at the same time, they do not coexist in the event queue, so IncreaseA is not simultaneous with itself. In general, it is a model

checking [9] problem to detect simultaneous events.

Table 1 contains four possible execution traces for the cases where IncreaseA always occurs before IncreaseB, where IncreaseB always occurs before IncreaseA, and where IncreaseA and IncreaseB are alternating in two different ways. There are many other possible execution traces as well. The state of the event queue is not shown in this representation of execution traces. The "Time" row shows the model time at which events are processed. The "Event" row shows the names of the events that are processed. Below the double lines are the states of variables A and B after events in the same columns are processed. The columns are arranged from left to right in the order of event processing.

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 the one that is cancelled can be any one of them, if a well-defined order is missing.

We provide a solution below to avoid nondeterministic execution results as demonstrated above.

2.4.1 LIFO and FIFO Policies

A LIFO (last in, first out) property or a FIFO (first in, first out) property can be associated with a model. If neither of them is explicitly specified, the model has the LIFO property by default.

Either the LIFO policy or the FIFO policy is used when multiple events are scheduled to occur at the same time by different instances of events. With LIFO, the event scheduled by a later instance of an event is processed earlier. The opposite occurs with FIFO.

As an example, consider using the LIFO policy to execute the model in Figure 4. Execution trace 3 and 4 in Table 1 would not be possible. Following the Init event, either IncreaseA or IncreaseB is processed first, depending on other mechanisms to untie simultaneous events to be discussed later.

- 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 goes on until A's value reaches 10, when IncreaseB is cancelled. That leads to execution trace 1.

- If IncreaseB is processed first, all 10 instances of IncreaseB are processed before IncreaseA. That yields execution trace 2.

With FIFO, however, instances of IncreaseA and IncreaseB interleave, resulting in execution traces 3 and 4 in the table.

In practice, LIFO is more commonly used because it atomically executes a chain of events, where one schedules the next with no delay. This achieves atomicity in the sense that no event that is not in the chain interferes with the processing of those events in the chain. This is convenient for specifying workflows where some tasks need to be finished sequentially without intervention.

Time	0.0	1.0	1.0	...	1.0
Event	Init	IncreaseA	IncreaseA	...	IncreaseA
A	0	1	2	...	10
B	0	0	0	...	0

1) IncreaseA is always scheduled before IncreaseB

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

2) IncreaseB is always scheduled before 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

3) IncreaseA and IncreaseB are alternating, starting with IncreaseA

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

4) IncreaseA and IncreaseB are alternating, starting with IncreaseB

Table 1: Four possible execution traces for the model in Figure 4.

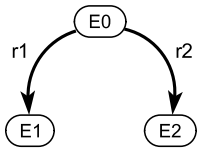


Figure 5: A scenario where event E0 schedules E1 and E2 after the same delay.

2.4.2 Priorities

For events that are scheduled by the same event with the same delay δ, such as E1 and E2 in Figure 5, *priority numbers* can be assigned to the scheduling relations r1 and r2 to determine the processing order. If r1 has a higher priority (i.e., a smaller priority number) than r2, then E1 is processed before E2, and vice versa. The default priority number for any scheduling relation is 0.

In Figure 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 1 would not be possible. On the contrary, 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.

2.4.3 Identifiers

Within a flat model, the events' identifiers (IDs) are unique. A scheduling relation also has a unique ID that is usually hidden in the visual representation.

In Figure 5, if r1 and r2 have the same delay δ and the same priority, then the order of E1 and E2 is determined as follows:

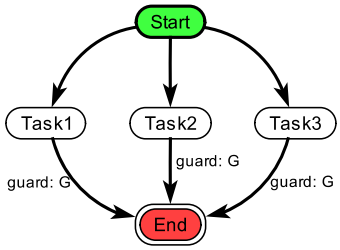
> *Assume total orders $\leq_e$ and $\leq_r$ exist for comparing events and relations based on their IDs, respectively. Let $e_1 =_e e_2$ be equivalent to $(e_1 \leq_e e_2) \wedge (e_2 \leq_e e_1)$ and $e_1 <_e e_2$ be equivalent to $(e_1 \leq_e e_2) \wedge \neg(e_2 \leq_e e_1)$. e_1 occurs before e_2 if and only if $(e_1 <_e e_2) \vee (e_1 =_e e_2 \wedge r_1 \leq_r r_2)$.*

Moreover, the total order $\leq_e$ is also used to sort the initial events in the model (which are scheduled implicitly at the beginning of an execution but not by scheduling relations).

2.4.4 Designs with Atomicity

An interesting research topic is to ensure atomicity for a sequence of events with the presence of other simultaneous events. Without requiring designers to explicitly control critical sections, as is the case for imperative programming languages, here we present two design patterns in Figure 6 that designers can reuse to obtain atomicity.

a) Sequentially perform all tasks



b) Sequentially perform tasks until G is satisfied

Figure 6: Two design patterns for controlling tasks.

Final events shown as filled vertices with double-line borders are used in the design patterns. They are special events that have the side effect of removing all events in the queue. They are used to force termination of the execution even though there may be events remaining in the event queue.

The design pattern in Figure 6a is used to sequentially and atomically perform a number of tasks, assuming the LIFO policy is chosen. Even if other events exist in the model (which are not shown in the figure), those events cannot interleave with the tasks. As a result, intermediate state between tasks is not infected by other events.

The design pattern in Figure 6b is used to perform tasks until the guard G is satisfied. This again assumes LIFO. After the Start event is processed, all tasks are scheduled. In this case the first one to be processed is Task1, because Task1 $\leq_e$ Task2 $\leq_e$ Task3. After Task1, if G is true, End is processed next, which terminates the execution. If G is not true, then Task2 would be processed. The processing of tasks continues until either G becomes true at some point, or all tasks are processed but G remains false.

2.5 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
   (a) Create an instance i_e
   (b) Set the time stamp of i_e to be 0
   (c) 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

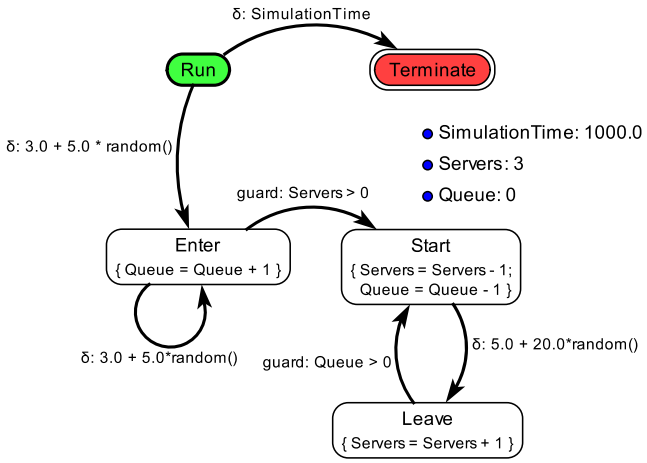


Figure 7: A model that simulates a car wash system.

i. Evaluate parameters for the event e' that r points to

ii. Create an instance $i_{e'}$ of e' and associate it with the parameters

iii. Set the time stamp of $i_{e'}$ to be greater than the current model time by r's delay

iv. 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''

2.6 Metaphorical Example: Car Wash Simulation

In this section, we describe a simple example that is a metaphor for many systems of interest. It is more familiar than many practical applications, which may be rooted in a highly technical application domain, and it is rich enough to admit elaborations that illustrate the expressiveness of Ptera. We begin with a simple multiple-server, single queue system, described as a car wash.

In a car wash system, a number of car wash machines share a single queue. When a car arrives, it is placed at the end of the queue to wait for service by any of the machines. The machines serve cars in the queue one at a time in a first-come-first-serve manner. The car arrival intervals and service times are produced with two stochastic processes.

The model to analyze the number of available servers and the number of waiting cars over time is provided in Figure 7. The Servers variable is initialized

to 3, which is the total number of servers. The Queue variable starts with 0, since no car is 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 event schedules itself to occur again. It also schedules the Start event if there is any available server. The LIFO policy guarantees both Enter and Start to be processed atomically, so it is not possible for the value of the Servers variable 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.

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 finish of service for that car. Whenever a car leaves, the number of available servers must be greater than 0, so the Leave event immediately schedules Start if there is at least one car in the queue. Again, due to atomicity provided by the LIFO policy, testing for the queue size and changing it in the following Start event would not be interfered with by any other event in the event queue.

Without the Terminate event prescheduled at the beginning, an execution of the model would not terminate because the event queue would never be empty.

3 Hierarchical Models

Hierarchical and component-based design can mitigate complexity and improve reusability. Examples of existing hierarchical modeling languages include Statecharts [10], DEVS (Discrete Event System Specification) [11] and hierarchical Petri nets [12].

Hierarchical heterogeneous composition is studied in research projects such as Ptolemy II [13], ForSyDe [14], SPEX [15] and ModHel'X [16]. In Ptolemy II, models of computation that can be composed in the model hierarchy include DE (discrete events) [6], CT (continuous-time models) [17], FSM (finite state machine), SDF (synchronous dataflow) [5], DDF (dynamic dataflow) [18], HDF (heterochronous dataflow) [19], SR (synchronous reactive) [20], PN (process networks) [21, 22, 23] and CSP (communicating sequential processes) [24]. They form a rich set of formal languages that can be mixed to achieve high flexibility and reusability [25].

An abstract semantics has been defined in Ptolemy II for specifying the execution behavior of hierarchical heterogeneous models [26]. The semantics enables hierarchical composition of distinct models of computation. It consists of 1) an execution policy with designated extension points, and 2) a protocol containing high-level specifications of the expected behavior at those extension points.

After an overview of the abstract semantics, we provide algorithms for use at the extension points, which, in combination with the execution algorithm, define the operational semantics of hierarchical Ptera models.

3.1 An Abstract Semantics for Model Execution

Under the actor abstract semantics, a model component M is executed according to the following policy. M may be an atomic component within a model, or may itself be a composition of other components (in which case, it is called a composite actor). The policy is extensible in that the methods invoked may perform arbitrary (finite) computation. Some of the methods have Boolean return values.

- *Execute M:*

```
Preinitialize M
Initialize M
while true
    if Prefire M then
        Fire M one or more times
        if not Postfire M then
            break
Finalize M
```

For model M to be executed, Preinitialize, Initialize, Prefire, Fire, Postfire and Finalize in the algorithm must be defined. Given M as the parameter, those methods delegate to the corresponding methods defined for M's director. If M is hierarchical, which means M is a composite actor containing actors inside, then its director also invokes those methods for the contained actors. The way the director invokes the methods for the contained actors depends on the model of computation that the director implements. In general, for each contained actor, invocation of the methods should follow the sequence in the execution algorithm, though the sequences for different contained actors may be interleaved.

Expected behavior of the methods of component M is specified as follows.

- *Preinitialize.* Set up the structure of M for execution. This method performs any and all actions within the component that may affect static analysis, such as type inference.

- *Initialize.* Initialize M for execution. If M is contained in another model and its model of computation is timed, it may issue an initial firing request to explicitly request its container to prefire, fire and postfire it at the current model time or at a model time in the future.

- *Prefire.* Return a Boolean value that specifies whether a firing of M can be performed given the current conditions of the inputs.

- *Fire*. Fire M. The fire method should react to inputs and (possibly) produce outputs.
- *Postfire*. Update the state of M as a side effect of the last firing, and return a Boolean value that tells whether future firings are permitted. A timed model may now a issue firing request to its container, if any, or it may advance model time if itself is at the top level of the model hierarchy.
- *Finalize*. Terminate the execution and release the resources allocated for the given model. Once finalized, the model should not be prefired, fired or postfired, unless it is initialized again.

An additional *fireAt* method is defined as a callback that can be invoked in the Initialize and Postfire methods to request firing from the container. It takes as the first parameter the model that issues the request and as the second parameter a model time when firing of that model should occur. That model time should be equal to or greater than the current model time. For example, when Postfire is invoked with model M contained in M', fireAt may be issued with parameters M and t. The director of M' receives that request and schedules to fire M when the model time reaches t.

3.2 Ptera Semantics in the Actor Abstract Semantics

Compared to the execution algorithm for flat Ptera models in 2.5, an alternative way to define the semantics is by defining all the methods in the actor abstract semantics. This alternative way yields an equivalent semantics for flat models, but it additionally supports hierarchical models.

In the following discussion, M denotes a Ptera model. M' is the model that contains M in the model hierarchy, if it exists. m represents a submodel contained in M, if any. Submodels are contained inside Ptera events.

The event queue of M is denoted by Q. It is a priority queue that stores scheduled events and fireAt requests from the submodels. The events are sorted in the order discussed previously. The fireAt requests are sorted by their time stamps. If a fireAt request and an event have the same time stamp, then the fireAt request is sorted before the event. If two fireAt requests have the same time stamp, they are sorted with the LIFO or FIFO policy, depending on when those fireAt requests were received.

An additional *initializing* attribute is defined for each scheduling relation. It takes a Boolean value that determines whether the submodel of the scheduled event, if there is any, should be initialized *every time* that event is processed. If the attribute is false, the submodel would be initialized only if it has not been initialized or its postfire has returned false last time (meaning that its previous execution has finished).

An additional variable S denotes a set of references to the submodels of M that have been initialized.

- *Preinitialize M:*
  Initialize Q and S to be empty

For each event e in M in the $\leq_e$ order
If e is associated with a submodel m
Preinitialize m

- *Initialize M:*

For each initial event e in M in the $\leq_e$ order
Create an instance i_e and append it to Q
Set the time stamp of i_e to be the current model time
If container M' exists and Q is not empty
Issue fireAt to M' with the current model time

- *Prefire M returns Boolean:*

If Q is not empty
Peek the first item i in Q
Let t be i's time stamp
If $t <$ current time
Report error and return false
else if $t >$ current time
Return false
Return true

- *Fire M:*

If Q is not empty
Peek the first item i in Q
Let t be i's time stamp
If $t <$ current time
Report error and return
else if $t >$ current time
Return
Remove i from Q (step 3a in 2.5)
If i is an instance of event e
Execute the actions of e (step 3b in 2.5)
If e is a final event
Clear Q (step 3c in 2.5)
else if e has submodel m
If the scheduling relation that scheduled i has initializing attribute set to true, or m is not in S
Initialize m
Add m to S
If m need not be initialized or no fireAt request was received when it was initialized
RunOnce m // defined next
else // e does not have submodel
Evaluate scheduling relations and canceling relations from e (steps 3d through 3j in 2.5)
else // i is a fireAt request

Let m be the submodel that issued i
RunOnce m

- *RunOnce m:*
  If Prefire m
    Fire m
    If not Postfire m
      Finalize m
      Remove m from S
      Let e be the event that m is associated with
      Evaluate scheduling relations and canceling relations from e (steps 3d through 3j in 2.5)

- *Postfire M returns Boolean:*
  If Q is not empty
    Peek the first item i in Q
    Let t be i's time stamp
    If M has container M'
      Issue fireAt to M' with time stamp t
    else
      Advance model time to t
    Return true
  Return false

- *Finalize M:*
  For each event e in M in the $\leq_e$ order
    If e is associated with a submodel m and m is in S
      Finalize m
  Clear Q and S

As part of the protocol established in the actor abstract semantics, in a hierarchical model, each composite actor with a discrete-event director maintains its event queue locally. Model M reports only the next firing time t to its container M' with a fireAt request in postfire. If t is greater than or equal to the current model time, M' ensures to fire M no later than t. M' does not need to know the events that are scheduled in the event queue of M. This effectively encapsulates the internal behavior of M. Another benefit is that by adjusting the *time advance rate* of M (a factor to be multiplied with the time stamps of events occurring in M, which could be different from 1), M may conceptually run at a different speed. This is particularly interesting for simulation and performance analysis.

3.3 Semantic Equivalence for Flat Models

Operationally, it is straightforward to prove that the above implementation in the abstract semantics defines an equivalent semantics for flat models compared to the execution algorithm in 2.5, which does not use the abstract semantics.

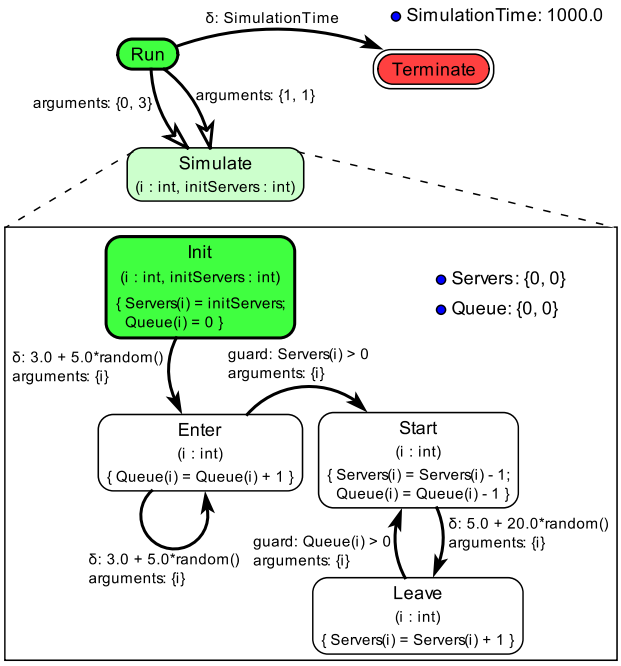


Figure 8: A hierarchical model that simulates a car wash system with two settings.

When M is flat, its container M' does not exist and no submodel is associated with any event. As specified by the actor abstract semantics, an execution starts by preinitializing and initializing M. The set S is always empty. The event queue Q begins with only the initial events sorted in their previously defined total order.

After initialization, M is executed in a loop. In each iteration, it is first prefired. Prefire returns false only when the nearest event in Q has a time stamp greater than the current model time, which should not happen for a flat model, because either initialization or the last postfire should have set the model time to that time stamp already. In each firing, if there are imminent events in Q, the first one is processed following the steps 3a through 3j in 2.5. When Q is empty, M is still fired but nothing is done in the firing. Its postfire returns false to terminate the execution.

Because of the semantic equivalence for flat models, the execution algorithm specified in the actor abstract semantics can be considered as an extension to that in 2.5, with the difference that the former also supports hierarchical models.

3.4 Hierarchical Car Wash Model

As a demonstration of hierarchical models, Figure 8 is a modification from Figure 7. 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 have hollow arrow heads, which reveal the fact that their initializing attributes are set to true. For that reason, they are called *initializing scheduling relations*. They make the submodel initialized each time the Simulate event is processed. In this example, the Simulate event is processed twice, causing two instances of the Init event to be scheduled in the submodel's local event queue. They are the start of two concurrent simulations, one with 3 servers and the other with 1 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 for the Simulate event distinguishes the two simulations. Compared to the model in Figure 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 enhanced in the same way. Each event in the submodel also takes a parameter i and supplies the value of i that it receives 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.

Table 2 shows a possible execution trace. Numbers in superscript represent the i values for the events.

Let M and m be the top-level model and the submodel, respectively. The execution starts by running the execution algorithm with M. At the beginning, M is preinitialized, causing m to be preinitialized as well. After that, M is initialized, so its initial event Run is placed in its event queue denoted by Q_M.

In the first iteration of the loop in the execution algorithm, M is prefired and the prefire returns true because the Run event is scheduled at the current model time, which is 0. M is then fired. Its Run event is processed and two instances of the Simulate event and a Terminate event are scheduled in Q_M. Postfire of M returns true because there are events remaining in Q_M.

In the second iteration, M is prefired, fired and postfired again. In the firing, assume the instance of Simulate event to be processed is Simulate^0. (As mentioned above, an opposite assumption leads to the same result.) m is initialized for the first time and is added to set S. In its initialization, m schedules the Init event in its event queue Q_m. It also issues a fireAt request to M, requesting M to fire it at the current model time. After initializing m, the firing of M finishes. M is then postfired with true return result.

In the third iteration, the submodel m is prefired, fired and postfired. (Simulate^1

Time	0.0	0.0	0.0	0.0	0.0	3.654	3.654
Event	Start^1	Enter^0	Start^0	Enter^1	Run	Simulate^0	Init^0
Servers(0)	2	2	1	1	0	0	3
Queue(0)	0	1	0	0	0	0	0
Servers(1)	0	0	0	0	0	0	0
Queue(1)	0	0	0	1	0	0	0

Time	4.028	4.028	8.158	8.158	8.406	11.007	12.384
Event	Simulate^1	Init^1	Enter^0	Start^0	Enter^1	Leave^0	Leave^1
Servers(0)	3	3	3	2	2	2	2
Queue(0)	0	0	1	0	0	0	0
Servers(1)	0	1	1	1	1	0	1
Queue(1)	0	0	0	0	1	1	1

Time	12.384	12.997	14.613	14.613	...	1000
Event	Start^1	Enter^1	Enter^0	Start^0	...	Terminate
Servers(0)	2	2	2	1	...	2
Queue(0)	0	0	1	0	...	0
Servers(1)	0	0	0	0	...	0
Queue(1)	0	1	1	1	...	105

Table 2: An example execution trace for the model in Figure 8.

remains in Q_M, because fireAt requests are always ordered before event instances with the same time stamp.) In m's firing, the Init event is processed, which schedules an Enter event in Q_m after a random delay. In m's postfire, another fireAt request is issued to M.

In the fourth iteration, Simulate^1 is processed. This causes m to be again initialized (due to the initializing scheduling relation). Another instance of the Init event is scheduled in Q_m. That instance is processed in the fifth iteration.

The execution continues until the model time is eventually advanced to 1000 and the Terminate event originally scheduled in Q_M is processed.

As a remark, one can conceptually execute multiple instances of a submodel by initializing it multiple times. However, the event queue and variables are not copied. Therefore, the variables need to be enhanced into arrays and an extra index parameter (i in this case) needs to be provided to every event.

Actor-oriented subclassing [27, 28] provides an alternative approach to enhancing a submodel into multiple executable instances. A class can be defined for the design of the submodel, with which instances can be created for execution.

Yet another approach is higher-order model composition [29], which allows to specify the model with a parametrizable higher-order model description. For example, using the Ptalon language, specification of a complex and large-scale model can be greatly simplified with a compact description [30]. Growth in

model size does not require larger descriptions. Another example is the higher-order model composition language offered by the VIATRA2 model transformation tool [31].

4 Composition with Heterogeneous Models of Computation

In a hierarchical heterogeneous composition, Ptera models can be composed with actor-oriented discrete-event (DE) models and finite state machines (FSMs), as well as models using other models of computation implementable in the actor abstract semantics.

We demonstrate the concept with two examples. One is to embed Ptera in DE and the other is to embed Ptera in DE and FSM in Ptera. A discussion that follows lists other types of composition that are currently known to be compatible. The general idea behind is extremely powerful because it allows designers to choose a convenient and expressive model of computation to model each part of the their systems, and to obtain a well-defined semantics for the overall composition.

4.1 Composition with DE

Compared to Ptera models, DE models are a different kind of discrete-event model. Their visual representation uses the actor-oriented modeling language. That is, actors are represented with boxes, ports of actors are triangles, and the lines between ports are communication channels.

Actors perform computation on the data received at their input ports, and produce data via their output ports. Those data are wrapped in events that also carry time stamps representing the model time at which they are produced. The events in DE, which we call *DE events* in the following discussion, are not visible in the model design, and are different from events in Ptera shown as vertices.

There are two kinds of actors. An atomic actor is implemented in an imperative programming language, such as Java and C. A composite actor encapsulates interconnected actors and acts as a single actor. For a composite actor to be executable, a director, which is a special attribute, must be associated with it. The director implements a model of computation. In particular, a DE model is a composite actor with a DE director.

4.1.1 Overview of the DE Director

The DE director implements DE semantics [6] for the composite actor that it is associated with. In an execution, the DE director maintains an event queue, which temporarily stores DE events received at the input ports of actors in that composite actor, as well as the fireAt requests issued by those actors. When the model time becomes equal to the time stamp of a DE event (or a fireAt

request), the DE director fires the corresponding actor and provides it with the DE event.

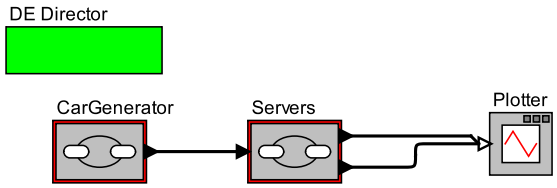
The following is a brief description of the methods defined for the DE director to be used in the actor abstract semantics.

- *Preinitialize.* Clear the event queue used for executing the composite actor. Preinitialize all the actors in the composite actor.
- *Initialize.* Analyze data dependency between actors. Initialize all the actors in the composite actor. Some actors may issue fireAt requests, which are stored in this director's event queue.
- *Prefire.* Check whether there is any DE event or any fireAt request in the event queue to be processed at the current model time, or there is any DE event available at an input port of the composite actor, or the composite actor is at the top level. If any of those conditions is true, return true; otherwise, return false.
- *Fire.* Move the events at the input ports of the composite actor into the event queue for the actors in the composite actor to process. Remove the DE events and fireAt requests scheduled at the current model time from the event queue, one at a time, in an order preserving data dependency. For each DE event or fireAt request, prefire the responsible actor and fire it if prefire returns true. The fired actor may generate DE events for the actors connected to its output ports. Those DE events are stored in this director's event queue until the receiving actors are fired at the model time equal to the time stamps of those DE events. After that, postfire the actor to determine whether it needs to be fired again. The actor may issue a fireAt request to be added to this director's event queue.
- *Postfire.* Check whether there are DE events or fireAt requests remaining in the event queue. If so, either advance model time to the smallest time stamp in case the associated composite actor is at the top level of model hierarchy, or otherwise issue a fireAt request to the containing model.
- *Finalize.* Clear the event queue and finalize all actors in the composite actor.

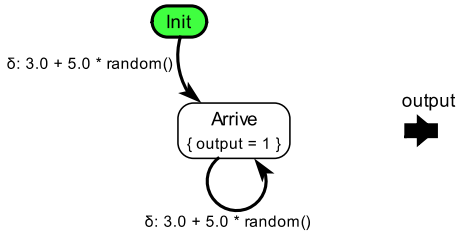
4.1.2 Example

Figure 9 shows a model that uses DE at its top level and contains Ptera submodels. In Figure 9a, boxes are actors except that the filled box with caption "DE Director" represents a DE director, which is essentially an attribute that defines the DE semantics for the diagram. CarGenerator and Servers are two composite actors associated with Ptera submodels. Plotter is an atomic actor defined in Java.

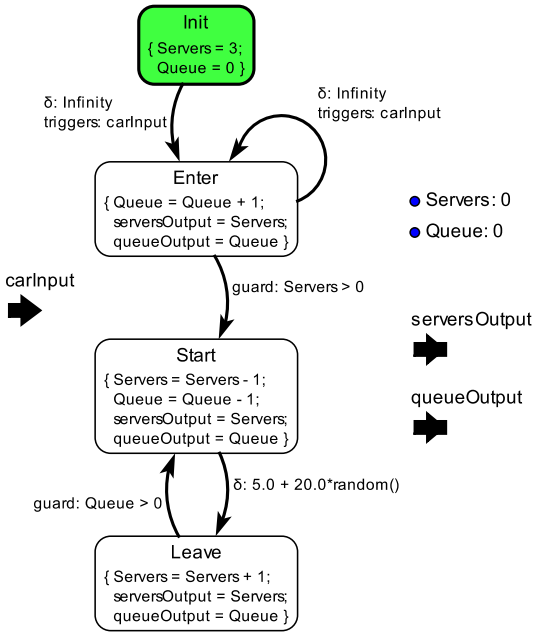
Figure 9b shows the internal design of CarGenerator. The Init event schedules the first Arrive event after a random delay. Each Arrive event schedules



a) Top level

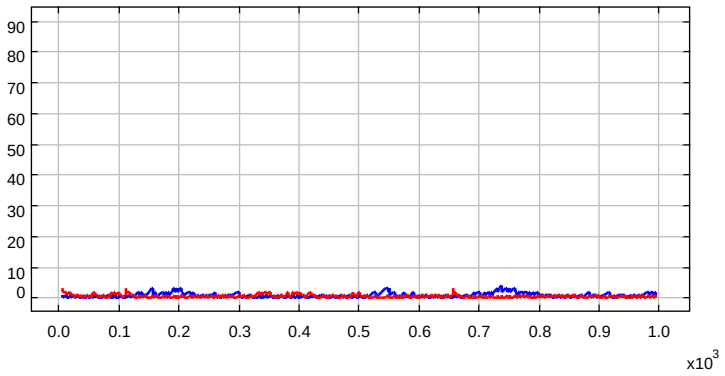


b) Internal design of CarGenerator

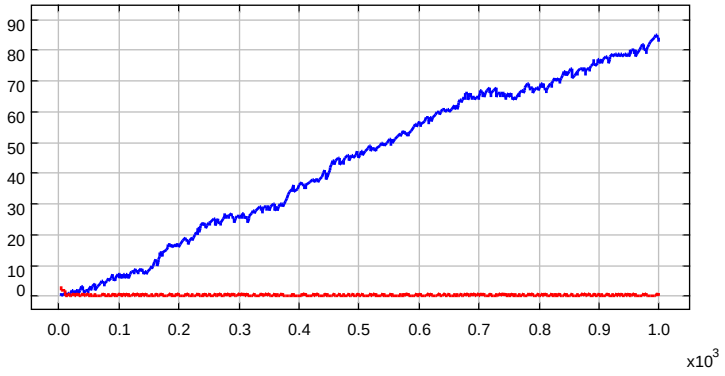


c) Internal design of Servers

Figure 9: A car wash model using DE and Ptera in a hierarchical composition.



a) InterarrivalTime = 3.0 + 5.0 * random()



b) InterarrivalTime = 1.0 + 5.0 * random()

Figure 10: Plotter output for two configurations of Figure 9.

the next one. Whenever it is processed, the Arrive event generates a car arrival signal and sends it via the output port using assignment "output = 1" with the left hand side being the port name and the right hand side being an expression that computes the output value. In this case, the value 1 is unimportant and only the presence of a value at the output port is interesting.

Figure 9c 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 external 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 arrival signals. At the top level, the connection from CarGenerator's output port to Servers' input port makes explicit the producer-consumer relationship. This separation of concerns leads to a more modular and reusable design.

Plotter at the top level plots the outputs from Servers in a separate window. An example plot obtained by executing the model to time 1000 is provided in

Figure 10a, where the upper (blue) curve represents the number of waiting cars over time, and the lower (red) curve represents the number of available servers. If the car interarrival time in Figure 9b is changed from "3.0 + 5.0 * random()" to "1.0 + 5.0 * random()," one may observe a different plot as in Figure 10b.

4.1.3 Processing of DE Events

In initialization, the DE director computes the causality dependency between the interconnected actors. This information can be used in the Fire method to determine an unambiguous actor firing order [32]. Each actor is also initialized in this phase. For a Ptera submodel (such as CarGenerator and Servers in Figure 10), initializing it causes initial events to be scheduled in its event queue at time 0 and a fireAt request to be issued to the DE director. The request is stored in the DE director's event queue.

DE events and fireAt requests in the event queue of a DE director need not be totally ordered. The director only needs to ensures that if DE event e_2 is causally dependent on e_1, then e_1 must be processed before e_2. For DE events that do not have causality dependency between them, the order in which they are processed does not affect the observable output, and can be arbitrary [33, 34].

Each time the DE director is fired, it retrieves imminent DE events and fireAt requests from its event queue. For a DE event, the director fires the receiving actor. The actor can read the DE event when it is fired. For a fireAt request, the actor is fired with no available input.

In the example, when fired the first time at model time 0, the two submodels process their Init events. In CarGenerator, an Arrive event is scheduled in its event queue. A new fireAt request is issued to the DE director in postfire that requests to fire again in the future when the model time reaches the time of the Arrive event. In Servers, processing the Init event causes the Servers and Queue variables to be reset to their initial values. The Enter event is scheduled with the delay "Infinity" and is registered to receive inputs at carInput port (discussed next). Postfire of this submodel does not issue fireAt request but simply returns true. This is because, even though the Servers component has events remaining in its event queue, it cannot decide the time when it should be fired again. The DE director at the top level should fire it when a DE event is received at its input port.

In postfire, the DE director advances model time to the time of the fireAt request from CarGenerator. In the next firing, the DE director fires CarGenerator, which processes the Arrive event and generates a DE event to the output port. That DE event is tagged with a time stamp equal to the current model time, and is stored temporarily in the DE director's event queue. When CarGenerator postfires, it schedules the Arrive again and issues another fireAt request. The DE director also fires the Servers submodel since it has a DE event at its input port. The latter processes the Enter event, which is triggered by the input as specified by the triggers attribute of the scheduling relation that scheduled it.

Plotter passively waits for DE events from Servers. Every time it is fired, it is provided with two DE events, one at each channel of the input port, because

Servers always generates DE events at both output ports at the same time. In reaction, Plotter extracts the values from those DE events and plots them in a separate window.

4.1.4 External Inputs and Outputs

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.

Let the triggers be "$p_1, p_2, \cdots, p_n$." 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. To refer to the input value available at a port, the port name may be used in an expression.

For example, the Enter event in Figure 9c is scheduled to indefinitely wait for DE events at the carInput port. When one 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 actions with port names on the left hand side and expressions that compute the values on the right hand side. The time stamps of the outputs are always equal to the model time at which the actions are executed.

4.2 Composition with FSMs

Ptera models can also be composed with untimed models such as FSMs (finite state machines). 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.

The opposite composition, having Ptera submodels be refinements of states in an FSM, is also interesting because by changing states, the submodels may be disabled and enabled, and the execution can switch between modes. That style of composition is addressed by the Ptolemy II modal models [35], which can interact well with discrete-event models [36].

4.2.1 Overview of the FSM Director

Here we only concern FSMs without the modal model extension embedded in Ptera as submodels.

In the actor abstract semantics, the FSM director can be defined as follows.

- *Preinitialize.* Nothing needs to be done.

- *Initialize*. Set the current state to be the initial state.
- *Prefire*. Return true, because an FSM is always willing to be fired unless its final state has been reached, in which case its Postfire should have returned false and Prefire would not be invoked again.
- *Fire*. Evaluate the guards of all transitions from the current state. Among those that are enabled (if any), pick one according to a predefined or user-specified scheme. Execute the actions of the chosen transition that produce output data at the FSM's output ports. Record the chosen transition to be used in Postfire.
- *Postfire*. If a transition is chosen in Fire, execute the actions of that transition that assign new values to variables. After that, set the current state to be the destination state. Return true if the new state is not a final state. Return false otherwise.
- *Finalize*. Nothing needs to be done.

Because an FSM is untimed, if it is contained in a discrete-event model, the data that it outputs in Fire are automatically associated with the current model time as their time stamps.

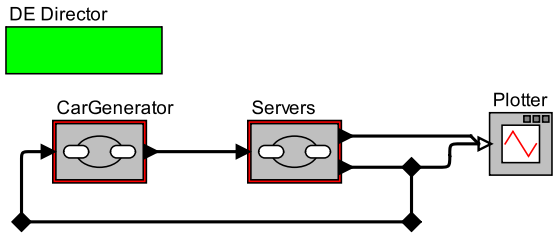
4.2.2 Example

To demonstrate composition of Ptera and FSM, consider the case where drivers can perceive the number of cars waiting in the queue and may avoid entering the queue if there have already been too many waiting cars. That leads to a lower arrival rate (or equivalently, longer interarrival time in average). Conversely, if there are only few or no waiting cars, the drivers would always enter the queue, resulting in a higher arrival rate.

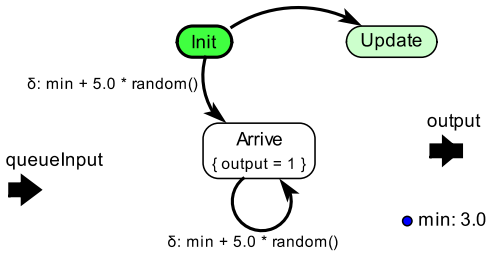
The model in Figure 9 is modified for this scenario and the revised model is shown in Figure 11. Figure 12 shows the result of executing the new model. At the top level, the queueOutput port of Servers (whose internal design is the same as Figure 9c) is fed back to the queueInput port of CarGenerator. The FSM submodel in Figure 11c is associated with 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 initialized to be in 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. Since then, the interarrival time is generated with expression "1.0 + 5.0 * random()." Notice that the min variable is defined in CarGenerator, and a scoping rule enables the contained FSM to read from and write to that variable.

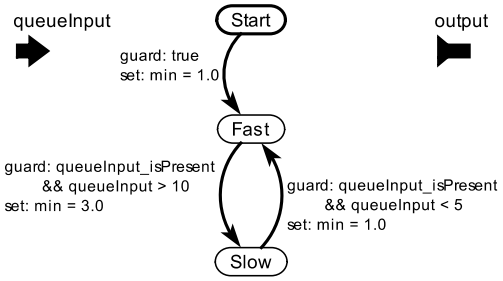
Postfire of the FSM always returns true, because there is no final state. The FSM would be fired again when either the Update event is processed again (which does not happen in this example) or when input is received at any input



a) Top level



b) Internal design of CarGenerator



c) Internal design of Update

Figure 11: A car wash model using DE, Ptera and FSM in a hierarchical composition.

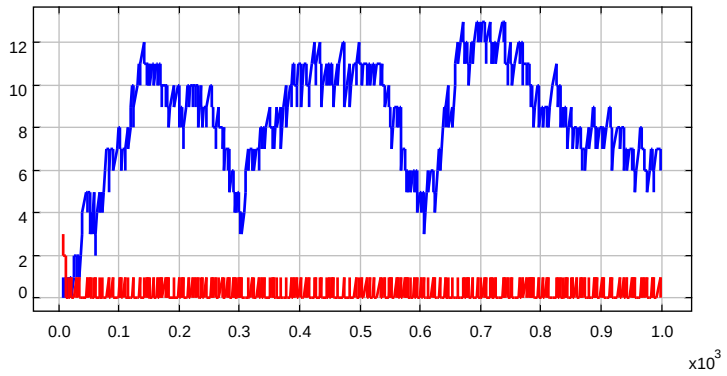


Figure 12: Plotter output for the model in Figure 11.

port. When the Servers composite actor sends out the number of waiting cars via its queueOutput port, the number is transferred to the queueInput port of CarGenerator by the top-level DE model and is made available to the FSM submodel. The FSM submodel is fired at that time. It may or may not change state depending on whether that received number exceeds the bounds.

In general, when a Ptera model receives input at a port, all the initialized submodels are fired, regardless of the models of computation that those submodels use.

4.3 Hierarchical Heterogeneous Model Design

Models of computation implementable in the actor abstract semantics can be composed with Ptera to create executable models, but some types of composition may not be as common as others.

We identify types of compositions involving Ptera models, and perform experiments in Ptolemy II to back the theoretical results.

These are the currently studied models of computation that can be used to create models containing Ptera submodels.

- DE. As is demonstrated in the examples in Figure 9 and Figure 11, DE models can contain Ptera submodels and provide well-defined discrete-event semantics. The Ptera submodels can communicate with actors in the DE model using DE events via ports.

- Ptera. We have shown the composition of Ptera and itself in Figure 8. The containing and the contained Ptera models have separate event queues, and the event queue of the outer model only needs to store the nearest fireAt request from the submodel. It fires the submodel when the model time reaches the requested time, or an input is received at any input port, or the event that the submodel is associated with is processed.

- TM (timed multitasking). This model of computation emulates the behavior of a real-time operating system running on a single-core CPU. An actor is a task that can be triggered by timed events received at its input ports. The actor may also issue fireAt requests which the director considers as interrupts occurring at the time equal to their time stamps.

These are the models of computation that can be contained in Ptera.

- DE. DE can be embedded in Ptera as well, though the examples do not show this type of composition. A DE submodel reports to the Ptera model that contains it with fireAt requests. It would also be fired when the event it is associated with is processed and when inputs are received at the input ports.
- FSM. As an untimed model of computation, FSM cannot issue fireAt requests, so its firing can only be triggered by processing the event that it is associated with in the Ptera model or by receiving an input.
- Ptera, as is discussed above.
- Dataflow. SDF (synchronous dataflow) [5], DDF (dynamic dataflow) [18] and HDF (heterochronous dataflow) are different flavors of dataflow and are all untimed. Their semantics can be defined in the actor abstract semantics. Similar to FSMs, dataflow submodels are fired when the containing Ptera model processes the events they are associated with or when inputs are received.
- SR (synchronous reactive) [20]. An SR model is timed and the time advances in fixed step sizes called periods (which may be 0). In each period, the data values on the communication channels are computed as a fixpoint. For an SR submodel contained in a Ptera model, at the end of each period, it requests firing for the next period, and the Ptera model stores the fireAt request in its event queue.

5 Example Applications

As a general-purpose model of computation, Ptera can be applied to a wide range of applications. It is especially suitable for modeling timed sequential processes. Hierarchical composition with other models of computation, such as dataflow and DE, allows for concurrency when desired.

We present three practical applications, each highlighting a strength that Ptera is equipped with.

5.1 Complex Control System

Complex system designs require selecting from hugely many possible realizations of a specification. Code generation from models and compilation from higher-level languages help, but these techniques are focused on selecting from relatively

few possible realizations. Most of the hard work has already been done in the construction of the models and the software. The NAOMI project [37] is a collaboration of LMATL (Lockheed Martin Advanced Technology Lab), UC Berkeley, UIUC, and Vanderbilt University that addresses the problem of designing such systems. It focuses on the problem of multimodeling, which is the combination of several distinct models in the construction and analysis of systems. NAOMI supports design of complex systems by enabling composition of multiple models and providing techniques for ensuring consistency across models, correctness of the composition, and synthesis of compositions via model transformation. Central to this is effective characterization of the interfaces of the components that make up the whole.

The NAOMI project has also leverages a metaphorical example application, a traffic light system, which provides an arbitrarily extensible test case for the *multimodeling*. Models can range from a single pedestrian crossing, useful for illustration, to entire cities with superimposed and conflicting constraints. The approach in NAOMI is to show how multimodeling helps make such systems tractable.

In [38], two types of multimodeling are identified, namely, *hierarchical multi-modeling* and *multi-view modeling*. The NAOMI project explores both types of multimodeling. It incorporates a variety of modeling tools into the framework of a multimodeling manager. For the metaphorical example, the Ptolemy II modeling environment [13] is used to create a Traffic Light model that exhibits the behavior of the car lights and pedestrian lights at a 4-way intersection.

The Traffic Light model contains separate components for the 4 car lights, which turn red, green and yellow after every preconfigured time duration. Each Car Light component is connected to a Pedestrian Light via a wireless channel. The wireless signals are modeled with the Wireless director in Ptolemy II, a variant of DE director that transfers data on implicit wireless channels instead of explicit connections between ports.

Here we focus on the part of Car Light component shown in Fig. 13. It demonstrates how Ptera can be used to model complex control systems. The Init event is the only initial event to be fired at model time 0. Depending on the initial state of the car light, it schedules Red, Green or Yellow to occur immediately, which set the car light to the corresponding color. The PedRequest port receives pedestrian requests. According to the specification, when the first pedestrian arrives at the car light and if the car light is green at the moment, the pedestrian presses a button to request crossing. Handling of the requests is according to the design policy. When a request is handled, a data value is sent to the PedReqeust port. The scheduling relation from Green to Yellow is set to have delay "GreenDuration" (a constant defined at a higher level of the model hierarchy) and its "triggers" attribute is set to "PedRequest." This means the light turns yellow either the GreenDuration expires, or a pedestrian request needs to be handled.

A similar design is found between the Red and Green events. When the car light is red, it may receive an external event at the OrthogonalRequest port, signifying that a pedestrian request is handled by a car light in the orthogonal

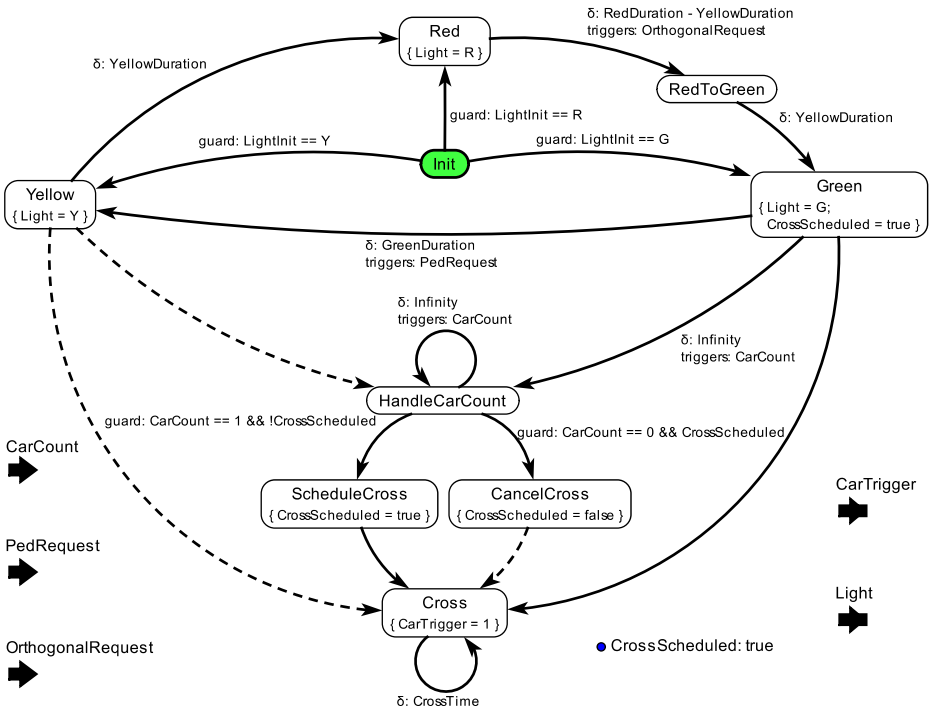


Figure 13: A Car Light component in the Traffic Light model

direction. In that case, the orthogonal car light turns yellow immediately, and it informs this car light. After the yellow duration, the orthogonal car light turns red, and this car light should turn green. Here we assume RedDuration to be greater than YellowDuration, as it is in all the cases we have encountered. The scheduling relation from Red to an additional event RedToGreen takes place when the amount of time "RedDuration - YellowDuration" expires, or a car light in the orthogonal direction handles a pedestrian request. When either happens, this car light turns green after YellowDuration.

A third input port, CarCount, receives the numbers of cars waiting for this car light. It receives increasing consecutive numbers when the light is red according to a Poisson arrival process, and decreasing consecutive numbers when the light is green. We assume it takes constant time for each car to cross the intersection, whereas all waiting pedestrians cross in negligible time. To reflect this, when the car light turns green, the HandleCarCount event is scheduled to listen to inputs at the CarCount port. The Cross event is also scheduled to occur immediately to allow the first waiting car to cross, if any. Cross repeatedly schedules itself after every CrossTime period. If no car is waiting any more, the next Cross event is cancelled. If a car arrives when the light is still green, a new Cross event is scheduled if none has already been scheduled (which is detected with the CrossScheduled variable being false).

The two output ports send out controlling signals. CarTrigger informs the car counting component that a car can leave the waiting queue, if any. It has no effect if the queue is already empty. The Light port outputs signals that controls the car light hardware device.

Despite the complexity of the control logic as described in the specification, the car light component remains in manageable size. An equivalent FSM would have much more states than the events here. From time to time in an execution, multiple events are scheduled in the event queue, and using FSM, it would be required to explicitly model each possible state of the event queue as an explicit state.

It would be possible to automatically and precisely model-check this Ptera component. It has only finite states for its event queue, and by exploiting all those states, the exact behavior can be checked. In future work, we will further explore model checking techniques [39] for bounded and unbounded Ptera models.

5.2 Sequential Workflow: Model Optimization

Sequential workflows can be specified with Ptera models. A practical application has been developed by UC Berkeley and Bosch to transform and optimize large-scale models of automotive engines. A simplified model of the application is shown in Fig. 14. It is a Ptera model at the top level. (The rectangles in the figure identify the surrounded objects to be derived from an actor-oriented class [27]. They have no significance in the model's semantics.) This model has a Model parameter, which has a special "actor" icon to make it clear that it contains a model as its value, rather than a traditional primitive value.

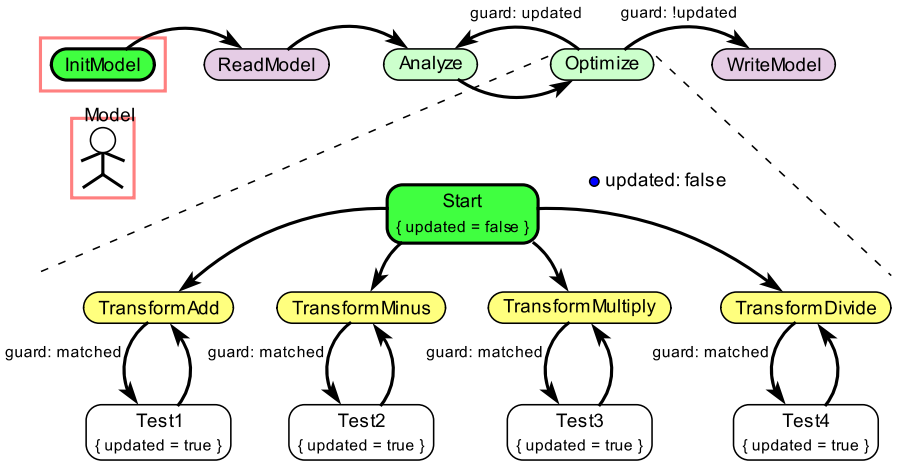


Figure 14: A hierarchical workflow to optimize a model

A few special events are used in this case, which have actions implemented in Java. InitModel is an initial event that resets the Model parameter so that it contains an empty model. ReadModel reads the contents of a file on the disk, parses it into a model, and stores that model in the Model parameter. Analyze is an event associated with an SDF (synchronous dataflow) submodel that analyzes the model in the Model parameter to detect constant signals. We ignore the submodel in the figure because its internal design is proprietary and irrelevant to our discussion.

Optimize is an event associated with another submodel to eliminate computation that returns constants from the model if possible. It has 4 Transform events started by the single Start initial event. Each Transform event represents a model transformation specified with a graph transformation rule [40, 41]. For example, TransformAdd transforms an Add actor within the model in the Model parameter with at least one constant input into either a simpler Add actor, or a constant actor if all its inputs are constant. Each Transform event has an internal "matched" parameter that tells whether the model was updated in the last transformation. An "updated" variable in the submodel determines whether the model was updated in the last execution of the submodel. At the end of the top-level workflow, a WriteModel event is used to write the model into a file.

This example highlights a hierarchical model optimization workflow that can easily grow to a much bigger size in practice. The complexity at each level remains low, however, due to the restricted and cleanly defined behavior of the submodels.

5.3 Timed System with Unbounded States: Car Wash

In the previous sections, we have discussed various versions of car wash simulations. The model in Fig. 11 is derived from an imaginary but yet very realistic application introduced in [42, 43]. We take that model as a concrete example to show that designers can use Ptera to conveniently model timed systems with unbounded states.

In car wash, because the number of servers is an unbounded input from the external environment, the number of scheduled car leaving events at any time in a simulation is unbounded as well. Using Ptera, this unboundedness can be very easily modeled with the model's event queue. An Enter event or a Leave event may or may not schedule a Start event, which in turn always schedules a Leave event. The designer need not worry about the number of Leave events that are potentially in the event queue at any time. Furthermore, though it is not shown here, the readers may imagine an enhancement to this component such that each Start event takes a parameter that identifies the served car's plate number or make and year. The same parameter can be passed to the Leave event that it schedules, which can be used for record or statistics at the time when the car leaves. This ability of associating parameters to an event and retrieving their values when the event is processed comes in handy in this sort of applications.

Ptera provides an unambiguous notion of model time, even for hierarchical models. Events are processed in their time stamp order. For events occurring at the same model time, a deterministic order is guaranteed. This makes it easy to model timed applications, such as the car wash system above.

6 Related Work

6.1 Modeling Views

The Unified Modeling Language (UML) encompasses a series of diagrams that facilitate different modeling habits. Among those, class diagrams are widely used to create designs in class-oriented views, where classes are the basic building blocks, and designers focus on the functionality provided by the classes as well as the relationship between classes. In object diagrams and communication diagrams, however, designers model dynamic objects that come into existence in an execution. The important relations in those diagrams are messages and method invocations between objects. Statecharts, which are an extended form of finite state machines, promote state-oriented modeling, emphasizing run-time states of the system. Sequence diagrams consider a parallel system to be a set of sequential processes that interact with each other at specified execution points. This is related to actor-oriented modeling studied in a number of research projects [13, 44, 45], with the difference that in the former approach, the model of computation for the interaction between processes is predefined, whereas in the actor-oriented modeling approach, designers can freely choose their preferred models of computation.

In this paper we discuss the event-oriented view of systems in the context of Ptera. This view complements other views by allowing designers to focus on events and their causality relationship. We also show how event-oriented models can be composed with actor-oriented models in the practice of heterogeneous multimodeling [38] with the Traffic Light application.

6.2 Hierarchical Composition

Ptera is an extension of event graphs in [1] with additional features, among which is the support for hierarchical heterogeneous composition.

By defining an actor abstract semantics for model execution that invokes methods to be defined in the implementation of each model of computation, a rigorous execution semantics is obtained. This work surpasses the three existing approaches to hierarchically composing event graphs [2, 3, 4], because heterogeneous models of computation can be composed with Ptera, including but not limited to DE [6], FSM, SDF (synchronous dataflow) [5], DDF (dynamic dataflow) [18] and SR (synchronous reactive) [20].

6.3 FSMs and UML Statecharts

In an FSM (finite state machine) model, all states that the system can move in must be explicitly represented (unless extra variables are introduced, which sometimes compromise the benefits of using FSMs). In Ptera, the state is implicit because the event queue is not visible at design time. Model checking is much easier in the former case due to the bounded reachable state space. The latter provides Turing-complete expressiveness, because it has been proved that 1) Petri nets with inhibitor arcs are Turing-complete [46] and 2) any such Petri net can be modeled with an equivalent event graph [47]. The enhanced expressiveness usually makes model checking a much harder problem. One way to formally check Ptera models is by exploring all the possible states of its event queue. This, however, is not an efficient solution in general. Future work would be to apply recent research results on checking models with infinite states [48].

For a reactive FSM model, at any state all possible inputs need to be accepted by a transition. This concept is exploited in work such as interface automata in [49]. This requirement, however, dramatically complicates the design and may lead to design errors for not being able to anticipate potential inputs. The problem is eliminated in Ptera because multiple events can be scheduled to handle different inputs. When designing outgoing scheduling relations from an event, the designer only needs to concern about the causality relationship between events when certain inputs occur, while the rest of possible inputs are already taken care of by other events in the event queue. In our practice with the model optimization application, we find this amendment greatly alleviates the designer's work.

UML Statecharts, originally introduced by Harel [10], are an extended form of FSMs, which support hierarchical composition and concurrency (by means of orthogonal states). This removes some limitations of FSMs when they are used

to design software in practice. However, Statecharts are limited by the finite number of states and the lack of temporal semantics.

6.4 UML Activity Diagrams

UML activity diagrams can be used to specify workflows. This makes them a rival of Ptera in the same domain of applications, such as the model optimization application. In a brief comparison between the two, we find that Ptera models can be considered as a descendant of activity diagrams, which have extra event queues that allow to schedule multiple tasks in the workflow for future processing.

As an example, the top-level model in Fig. 14 is similar to an activity diagram with the only difference being the conditions are tested on the scheduling relations instead of in dedicated test blocks. Each event corresponds to a task, with the initial event corresponding to a starting task. There is no explicit ending task but we could mark the WriteModel event to be final in that case. The submodel associated with the Optimize event looks less like an activity diagram because in the latter, never can multiple tasks be scheduled at the same time. In our case, the multiple Transform events are scheduled to immediately occur after Start because they independently transform different parts of the model in the Model parameter. The designer need not explicitly specify an order (though the discussion in Section 2.4.3 provides a unique execution order). Furthermore, each Test event in the submodel schedules back the corresponding Transform event only if the last transformation in that Transform event was successful. This demonstrates how a more complicated workflow can be broken down into independent workflows that are designed separately.

6.5 Models of Computation in Ptolemy II

In the Ptolemy project [13, 50] we study heterogeneous models of computation and their composition. As part of this work, we implemented Ptera in the actor abstract semantics that permits composition of Ptera with other models of computation. We show that, as as member in the family, Ptera interacts with other existing models of computation out of the box. In this paper we discussed using Ptera within DE and FSM within Ptera. There are many other possible types of composition, such as SDF within Ptera and SR within Ptera. In our future work, we intend to explore these types of composition in practical applications.

Some models of computation in Ptolemy II intrinsically support concurrency. For example, the DE model of computation requires only a partial order between events in an execution. Any execution that satisfies the partial order, including parallel execution featuring concurrent firing of actors, is allowed. More importantly, the designer need not explicitly create threads or manage the intercommunication between threads. This significantly simplifies the designer's work and reduces the chances of design errors [51]. By composing DE with

Ptera, we envision disciplined concurrency to be exercised over sequential Ptera components, yielding flexible, extensible, robust and efficient designs.

7 Conclusion

We introduce the Ptera model of computation, an extension of event graphs, and we provide algorithms for executing models. We show that Ptera models can be composed hierarchically with other models of computation, and well-defined semantics can be obtained by employing an actor abstract semantics for model execution. Simple, suggestive examples are given to demonstrate the behavior of certain types of composition.

In practice we have encountered various applications. In this paper we emphasize the strengths of Ptera in modeling complex control systems, sequential workflows and timed systems with unbounded states. We provide our assessment based on a thorough comparison with existing modeling languages including FSMs (finite state machines), UML Statecharts and activity diagrams.

8 Acknowledgements

The authors would like to acknowledge very helpful comments and suggestions from Elizabeth Latronico (Bosch). We also thank Christopher Brooks for packaging the models for on-line access.

References

[1] Lee W. Schruben. Simulation modeling with event graphs. *Communications of the ACM*, 26(11):957–963, 1983.

[2] Lee W. Schruben. Building reusable simulators using hierarchical event graphs. In *Winter Simulation Conference (WSC 95)*, pages 472–475, Los Alamitos, CA, USA, December 1995. IEEE Computer Society.

[3] Som T. K. and R. G. Sargent. A formal development of event graph models as an aid to structured and efficient simulation programs. *ORSA Journal on Computing*, 1(2):107–125, 1989.

[4] Arnold H. Buss and Paul J. Sanchez. Building complex models with LEGOs (listener event graph objects). *Winter Simulation Conference (WSC 02)*, 1:732–737, December 2002.

[5] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[6] Edward A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1–4):25–45, 1999.

[7] Xiaojun Liu and Edward A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, December 2008.

[8] Ricki G. Ingalls, Douglas J. Morrice, and Andrew B. Whinston. Eliminating canceling edges from the simulation graph model methodology. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 825–832, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, January 2000.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] Arturo I. Concepcion and Bernard P. Zeigler. DEVS formalism: A framework for hierarchical model development. *IEEE Transactions on Software Engineering (TSE)*, 14(2):228–241, February 1988.

[12] Rainer Fehling. A concept of hierarchical Petri nets with building blocks. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 148–168, London, UK, 1993. Springer-Verlag.

[13] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[14] Axel Jantsch and Ingo Sander. Models of computation and languages for embedded system design. *IEEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, 2005.

[15] Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid, and Krisztian Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando, November 2006.

[16] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Model Driven Engineering Languages and Systems (MoDELS)*, pages 247–258, Nashville, TN, USA, September 2007.

[17] Jie Liu and Edward A. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. In *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design*, pages 95–100, Anchorage, Alaska, USA, September 2000.

[18] Gang Zhou. Dynamic dataflow modeling in Ptolemy II. Technical Report UCB/ERL M05/2, EECS Department, University of California, Berkeley, December 2004.

[19] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[20] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[21] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland Publishing Company.

[22] Gilles Kahn and David B. Macqueen. Coroutines and networks of parallel processes. *Information Processing*, pages 993–998, 1977.

[23] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[24] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[25] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole A. Goble. Composing different models of computation in Kepler and Ptolemy II. In *International Conference on Computational Science (ICCS)*, pages 182–190, Beijing, China, May 2007.

[26] Edward A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, 2003.

[27] Edward A. Lee and Stephen Neuendorffer. Classes and subclasses in actor-oriented design. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 161–168, San Diego, California, USA, June 2004.

[28] Edward A. Lee, Xiaojun Liu, and Stephen Andrew Neuendorffer. Classes and inheritance in actor-oriented design. Technical Report UCB/EECS-2006-154, EECS Department, University of California, Berkeley, November 2006.

[29] Adam Cataldo, Elaine Cheong, Thomas Huining Feng, Edward A. Lee, and Andrew Christopher Mihal. A formalism for higher-order composition languages that satisfies the Church-Rosser property. Technical Report UCB/EECS-2006-48, EECS Department, University of California, Berkeley, May 2006.

[30] James Adam Cataldo. *The power of higher-order composition languages in system design*. PhD thesis, EECS Department, University of California, Berkeley, December 2006.

[31] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1280–1287, Esslingen, Germany, October 2006.

[32] Ye Zhou and Edward A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–35, 2008.

[33] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A programming model for distributed real-time embedded systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[34] Patricia Derler, Edward A. Lee, and Slobodan Matic. Simulation and implementation of the PTIDES programming model. In *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, Canada, October 2008.

[35] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, November 1 2009.

[36] Jie Liu and Edward A. Lee. A component-based approach to modeling and simulating mixed-signal and hybrid systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):343–368, October 2002.

[37] Trip Denton, Edward Jones, Srini Srinivasan, Ken Owens, and Richard W. Buskens. NAOMI – an experimental platform for multi-modeling. In *MoDELS '08: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, pages 143–157, Toulouse, France, 2008.

[38] Christopher Brooks, Chihhong Cheng, Thomas Huining Feng, Edward A. Lee, and Reinhard von Hanxleden. Model engineering using multimodeling. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM 2008)*, Toulouse, France, September 2008.

[39] Armin Biere, Alessandro Cimatti, Edmund E. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[40] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer-Verlag, 1994.

[41] Thomas Huining Feng and Edward A. Lee. Scalable models using model transformation. In *1st International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB 2008)*, Toulouse, France, September 2008.

[42] A.R. van der Valk. A conversion from SIGMA event graphs to the SMS C++ class library, August 1995.

[43] SigmaWiki. `http://sigmawiki.com/`.

[44] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[45] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management (SSDBM)*, pages 423–424, Santorini Island, Greece, June 2004.

[46] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[47] Lee Schruben and Enver Yücesan. Transforming Petri nets into event graph models. In *Winter Simulation Conference (WSC 94)*, pages 560–565, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[48] Edmund M. Clarke, Himanshu Jain, and Nishant Sinha. Grand challenge: Model check software. In *Verification of Infinite-State Systems with Applications to Security (VISSAS)*, pages 55–68, Timisoara, Romania, March 2005.

[49] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/FSE: Proceedings of the 8th European Software Engineering Conference / 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, Vienna, Austria, 2001. ACM.

[50] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, April 2008.

[51] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.