Ptera: An Event-Oriented Model of Computation for Heterogeneous Systems*

Thomas Huining Feng
Oracle Corporation, CA, USA
thomas.feng@oracle.com

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

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

ABSTRACT

Many modeling techniques for embedded systems focus on events that occur in time and the causality relationships between them. Event-oriented modeling complements class-oriented, object-oriented, actor-oriented and state-oriented approaches. To facilitate event-oriented modeling, we have extended an older established model called event graphs to define new model of computation that we call Ptera (Ptolemy event relationship actors). Ptera is 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.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: *real-time and embedded systems*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*cyber-physical systems*; I.6.5 [**Simultation and Modeling**]: Model Development[models cyber-physical interactions]; J.7 [**Computers in Other Systems**][computer-based systems]

General Terms

Design, Theory

*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 #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-904-6/10/10 ...$10.00.

1. INTRODUCTION

Many modeling techniques for embedded systems focus on events that occur in time and the causality relationships between them. Some time ago, one of us proposed event graphs as a visual formalism for event-oriented modeling [22]. In this paper we introduce Ptera (Ptolemy event relationship actors), an extension of event graphs supporting hierarchy and heterogeneity. Ptera is much better suited to embedded system design, especially to cyber-physical systems, which are intrinsically heterogenous. In particular, Ptera models conform to an actor abstract semantics, which enables hierarchical composition with other models of computation (MoCs) [10], including for example dataflow models, finite state machines, discrete-event models, continuous-time models, and imperative programs in a conventional programming language. We have implemented Ptera in Ptolemy II [7], and it is available in open source form.[1]

The paper is organized as follows. Section 2 describes the syntax and semantics of flat models, where no hierarchical composition is involved. Hierarchical models are given in Section 3, including both hierarchical Ptera models and hierarchical compositions with other MoCs. In Section 4 we give an illustrative application. We compare Ptera to related work in Section 5. Additional details about this work are available in the expanded version at [9].

2. FLAT MODELS

We begin by explaining flat (non-hierarchical) Ptera 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*.

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

Variables in a model associate values with names. Each variable is visually represented as a name-value pair to the

[1]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. There is no need to pre-install anything. Ptolemy II can be obtained from `http://ptolemy.org`.

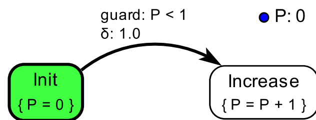


Figure 1: A simple model with two events.

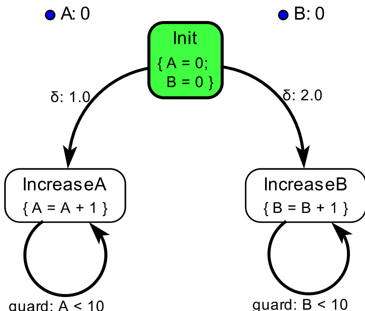


Figure 2: A model with multiple events.

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. 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. An *event queue* is used in the execution to 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. The guards of the two scheduling relations from Init take the

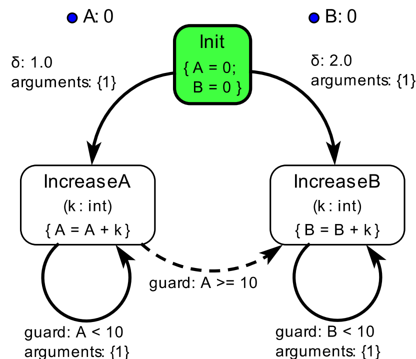


Figure 3: A model with arguments for the events and a canceling relation.

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.

2.1 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.

2.2 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.

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 [13]. Nonetheless, they can be convenient, making more compact and understandable models possible.

2.3 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 changing both δ's in Figure 3 to 1.0. That makes IncreaseA and IncreaseB simultaneous events. In Ptera, events are processed sequentially. Hence, there are two questions that we need to address. The first one is which of IncreaseA or IncreaseB should be executed first, after Init. Next, once we have chosen one of these to execute, since there is a self loop on both that schedules another instance of the same event without delay, we again face the question of which event to process next.

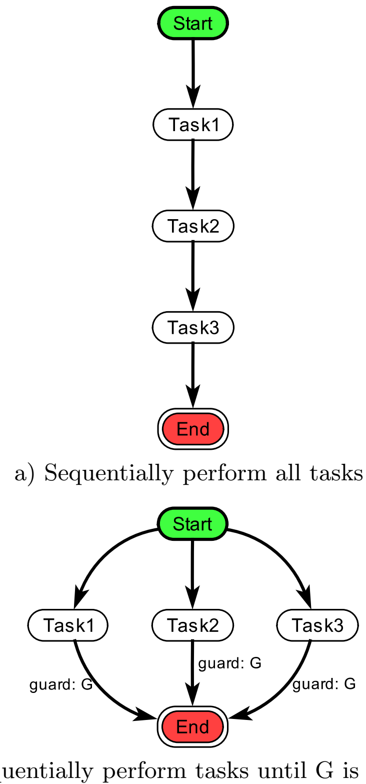
The first question is resolved with priorities. Init schedules both IncreaseA and IncreaseB with the same delay $\delta = 1.0$. Each scheduling relation can be assigned a *priority number*. If the left relation has a lower priority number, then IncreaseA will be processed first.

By default, a scheduling relation has a priority number 0. In the example, no priority numbers are shown, which implies that the scheduling relations have the same priority (the default), so we still need a way to resolve the question of which event to process first. Ptera does this with a second ordering relation denoted $\leq_e$, defined as the lexical order of events based on their IDs. Event IDs are unique, so this always determines an order. That is, IncreaseA $\leq_e$ IncreaseB, because its name would appear first in a dictionary. Thus, IncreaseA will be processed before IncreaseB.

Once IncreaseA is processed, because of the self loop, it will post another instance of IncreaseA on the event queue. This situation is significantly different from the previous one because this new instance came into existence distinctly after the instance of IncreaseB that is already on the event queue. In such situations, Ptera does not use the lexical ordering of events, but rather follows either a LIFO policy (last in, first out) property or a FIFO policy (first in, first out). The choice between these is determined by a user-settable property of the model. The default policy is LIFO, for reasons that we will see.

With LIFO, the event scheduled by a later instance of an event is processed earlier. The opposite occurs with FIFO. Hence, in our example, with the LIFO policy, 10 instances of IncreaseA will be processed before the first instance of IncreaseB is processed. After the 10-th IncreaseA, the canceling relation becomes enabled, so the first instance of IncreaseB will never be processed. At this point, there are no more events in the event queue and the execution terminates with final values A=10 and B=0.

With the FIFO policy, the first instance of IncreaseB will be processed before the second instance of IncreaseA. Under



a) Sequentially perform all tasks

b) Sequentially perform tasks until G is satisfied

Figure 4: Two design patterns for controlling tasks.

this policy, the execution will terminate with values A=10 and B=9.

In practice, LIFO is more useful 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 ensures atomicity without requiring designers to explicitly control critical sections, as is the case for imperative programming languages and modeling formalisms like UML Message Sequence charts.

To illustrate this notion of atomicity, consider two design patterns shown in Figure 4. The design pattern in Figure 4a 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 figure also shows *final events*, depicted with filled vertices with double-line borders. These are special events that have the side effect of removing all events in the event queue, thus forcing termination of the execution of the model.

The design pattern in Figure 4b is used to perform tasks until the guard G is satisfied. This again assumes a LIFO policy. After the Start event is processed, all tasks are scheduled. In this case the first one to be processed is Task1, because 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

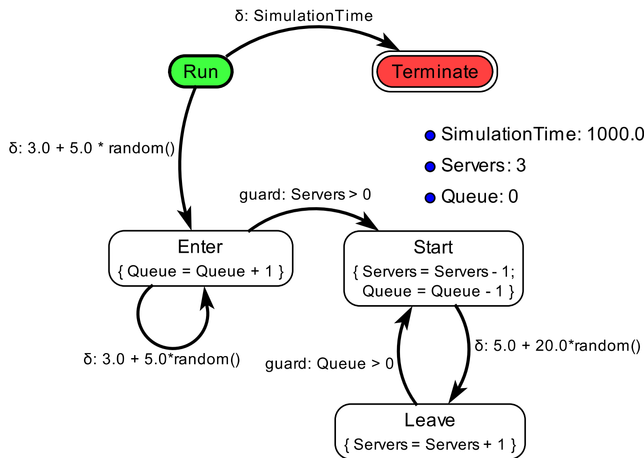


Figure 5: A model that simulates a car wash system.

of tasks continues until either G becomes true at some point, or all tasks are processed but G remains false.

To recap, priorities on scheduling relations or lexical ordering of event names is used when two simultaneous events are asserted by a single event. The LIFO or FIFO policy is used when simultaneous events are asserted by different instances of events. The LIFO policy enables simple specification of atomic processing of a sequence of events.

2.4 Illustrative Example: A Car Wash

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 yet 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 5. 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

Hierarchy can mitigate complexity and improve reusability in models. An abstract semantics has been defined in Ptolemy II for specifying the execution behavior of hierarchical heterogeneous models [7]. The semantics enables hierarchical composition of distinct models of computation, as described in detail in [10]. The abstract semantics 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.

Under the actor abstract semantics, a model component M is executed according to the following (simplified) 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:*
  - Initialize M
  - while model is running:
    - Fire M zero or more times
    - if M was fired at least once, Postfire M
  - Finalize M

For model M to be executed, Initialize, Fire, Postfire and Finalize must be defined. These methods are part of the definition of the component.

If the hierarchical component contains an annotation called a *director*, then it is said to be *opaque*, and it executes exactly as if it were an atomic component. In this case, the Initialize, Fire, Postfire, and Finalize methods are provided by the director, which will typically also invoke those methods on 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 above execution algorithm, though the sequences for different contained actors may be interleaved. We have implemented a Ptolemy II director that realizes the Ptera semantics.

The strictest form of the abstract semantics imposes a key constraint on these methods, which is that the Fire method not change the state of an actor [10]. Our Ptera director realizes the loose abstract semantics; the Fire method updates the event queue, and therefore changes the state of the system. This results in some constraints on how Ptera

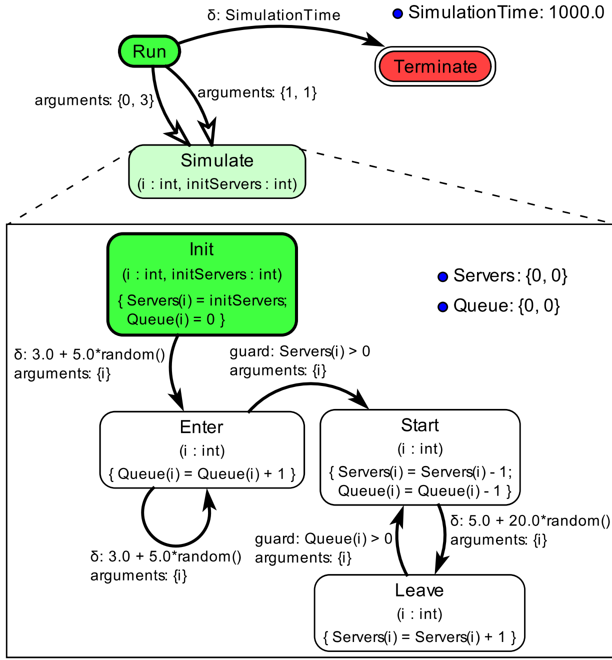


Figure 6: A hierarchical model that simulates a car wash system with two settings.

models can be used within models of computation that iterate to a fixed point, such as synchronous reactive (SR) and continuous-time. See [10] for details.

An additional part of the abstract semantics enables control of the passage of time by components in the model. In particular, a *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 at model time t.

An immediate consequence of this architecture is that Ptera models may hierarchically contain Ptera submodels, as we will discuss next.

3.1 Hierarchical Ptera Models

As a demonstration of hierarchical models, Figure 6 is a modification from Figure 5. 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 5, 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.

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 initialized, so its initial event Run is placed in its event queue denoted by Q_M. When M is fired the first time, 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 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 does not change the final result.) m is initialized for the first time. 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.

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

3.2 Heterogeneous Composition

Since the Ptera director conforms with the actor abstract semantics, Ptera models can be composed hierarchically with other models of computation such as discrete-event (DE) models and finite state machines (FSMs). We now 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. 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.

Compared to Ptera models, DE models are a different kind of discrete-event models. Actors are visually represented with boxes, ports of actors are triangles, and the lines between ports are communication channels. Syntactically, this is quite different from Ptera models, where the boxes are events and the relations interconnecting them represent scheduling actions.

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*, are not visible in the model design, and are different from events in Ptera shown as vertices.

A DE model is a composite actor with a DE director, which implements a rigorous DE semantics [15]. The DE director invokes the Fire method of actors to have them react to input events in time-stamp order. It accomplishes this, like Ptera, with 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.

Figure 7 shows a model that uses DE at its top level and contains Ptera submodels. In Figure 7a, 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 7b shows the internal design of CarGenerator. The Init event schedules the first Arrive event after a random delay. Each Arrive event schedules the next one. Whenever it is processed, the Arrive event generates a car arrival signal and sends it via the output port using 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 7c 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.

The 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 8, where the upper (blue) curve represents the number of waiting cars over time, and the lower (red) curve represents the number of available servers. For this plot, the car interarrival time is set to "1.0 + 5.0 * random()." The system is unstable with an unbounded queue.

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

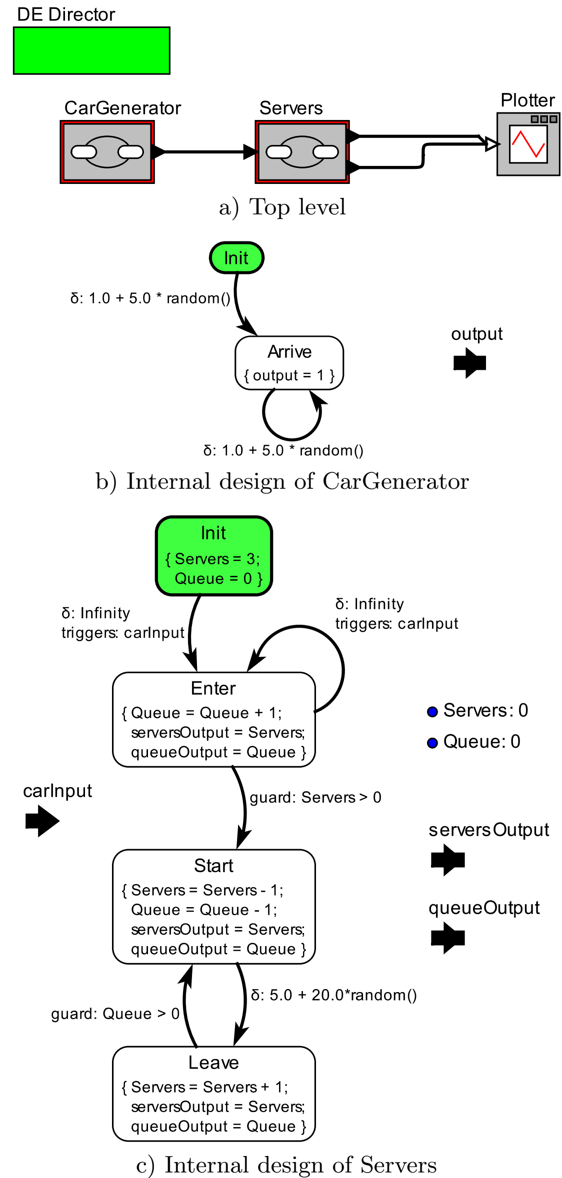


Figure 7: A car wash model using DE and Ptera in a hierarchical composition.

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 with the *triggers* attribute of the scheduling relation. 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

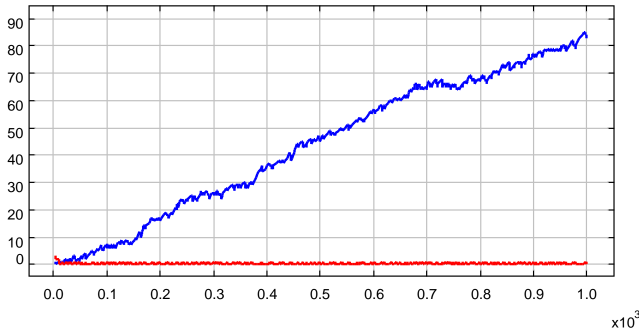


Figure 8: Plotter output for Figure 7, where InterarrivalTime = 1.0 + 5.0 * random().

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.

The 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.

In general, multiple port names may be specified in a triggers attribute, separated by commas. This can be used to schedule an event to react to different external inputs. The triggers attribute is used in conjunction with the delay δ to determine when the event is processed. Let the triggers attribute 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 7c 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.

3.3 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

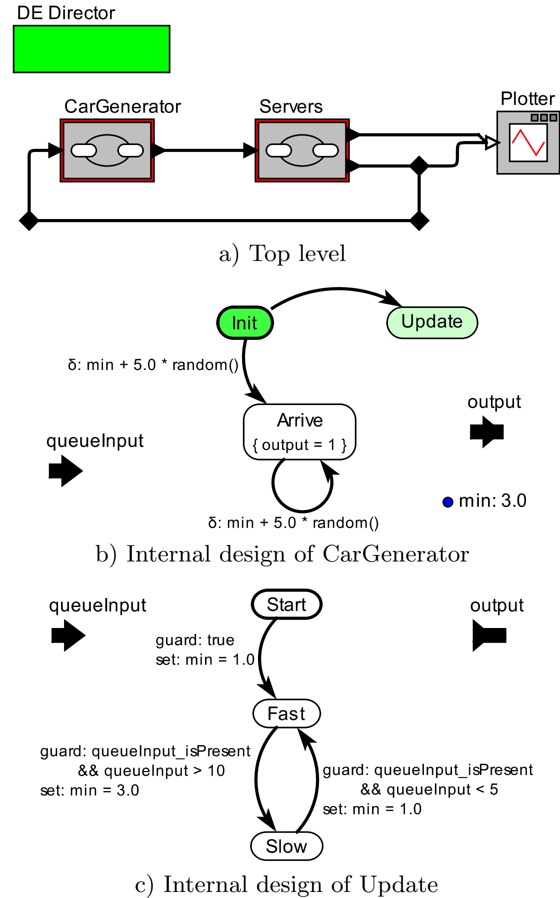


Figure 9: A car wash model using DE, Ptera and FSM in a hierarchical composition.

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 [17], which interact well with DE models [20].

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 7 is modified for this scenario and the revised model is shown in Figure 9. Figure 10 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 7c) is fed back to the queueInput port of CarGenerator. The FSM submodel in Figure 9c 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,

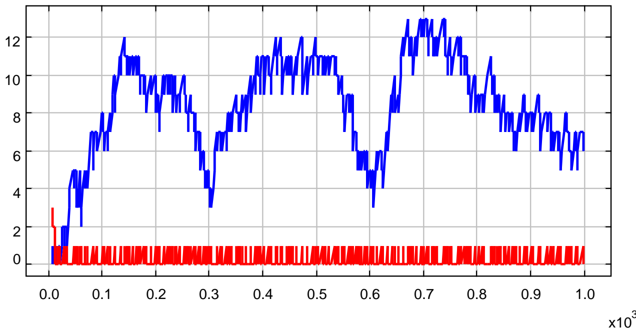


Figure 10: Plotter output for the model in Figure 9.

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 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. EXAMPLE APPLICATION

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. In this section, we give one illustrative application drawn from [3], simplified enough to be fully described here. The goal of Ptera, however, is to represent much more complex systems in an understandable and analyzable way.

This example models a traffic light and pedestrian cross 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. In this model, 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 over models of wireless channels that can reflect latencies and losses that are realistic in such channels.

Here we focus on the part of Car Light component shown in Fig. 11. It illustrates 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. Those events 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 used 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 direction. In that case, the orthogonal car light turns yellow immediately and 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. In either case, 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, 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 many more states than the events here. From time to time in an execution, multiple events are scheduled in the event queue, and using an FSM, it would be required to explicitly model each possible state of the event queue.

5. RELATED WORK

A classic event-oriented modeling formalism is finite state machines (FSMs). Ptera has some of the flavor of FSMs,

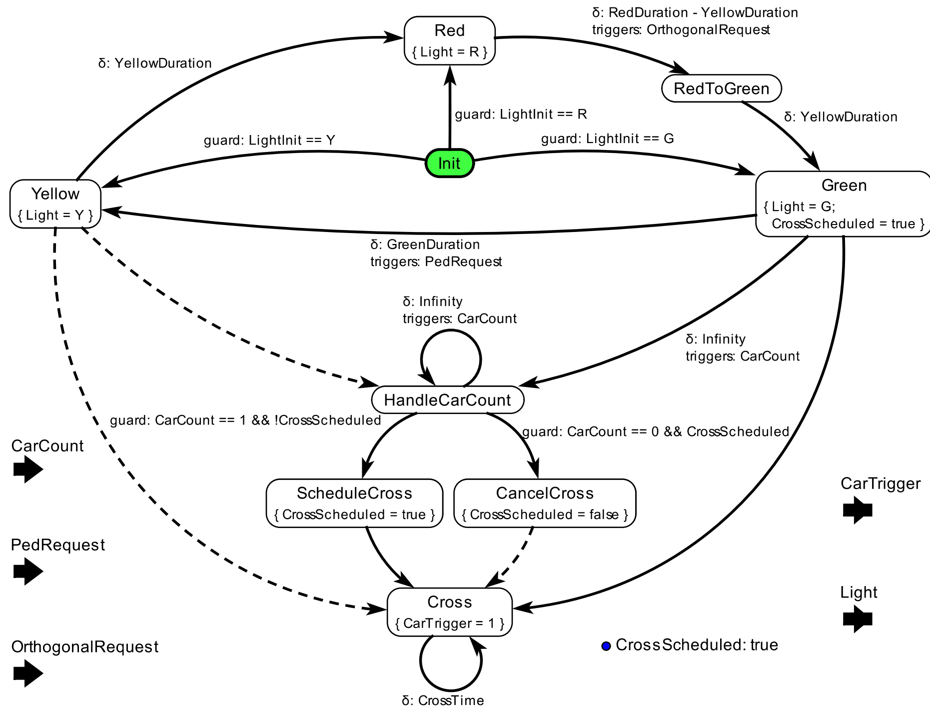


Figure 11: A Car Light component in the Traffic Light model

but is more expressive. In Ptera, the state is implicit in the event queue, and the state space is not necessarily finite. In fact, since Ptera extends event graphs, and event graphs are Turing complete [21], Ptera is a Turing complete modeling language. An interesting angle for future inquiry would determine how to apply recent research on checking models with infinite states [5] to Ptera models.

Ptera also addresses an awkwardness that sometimes arises in FSM models. In particular, for applications where the FSM must be receptive (i.e., there are no constraints on the inputs), the FSM model must specify in every state the reactions to all possible inputs. Ptera provides a more convenient and (we believe) less error-prone notation because multiple events can be scheduled to handle different inputs.

UML activity diagrams are more recent than the event graphs that Ptera is based on. Ptera models have some commonality with activity diagrams, and with relatively minor syntactic changes, activity diagrams can be translated into Ptera models. Ptera models are more expressive, however, because they support extended event queues.

For event graphs, there have been extensions to incorporate hierarchy. Two approaches are discussed in [23]. They both require adjusted semantics for submodels. Therefore, an event graph originally designed to be at the top level cannot be easily reused as a submodel in a bigger environment. In [4], a third attempt is reported, in which a listener pattern is introduced as an extra gluing mechanism for composing event graphs. Only two levels are allowed in a composition hierarchy – a specicial gluing model at the top level, and event graphs contained in it. We believe that Ptera models are more flexible, because they can be freely composed with any other Ptera models as well as other types of models, such as dataflow, FSM, and actor-oriented discrete-event models [15].

Statecharts, introduced by Harel [12] and subsequently realized in UML, SyncCharts [1], and many other variants, are an extended form of FSMs that support hierarchical composition and concurrency (by means of synchronous composition). Ptera models define a total ordering of event handling, and therefore do not have the concurrency evident in Statecharts or other synchronous reactive formalisms. However, because Ptera models conform with the actor abstract semantics, they can be composed within concurrent models of computation, including synchronous reactive, discrete-events, and continuous-time models using the techniques given in [18]. Ptera separates concurrency concerns from event handling.

Other hierarchical modeling languages are DEVS (Discrete Event System Specification) [6] and hierarchical Petri nets [8], as well as many familiar modeling languages like Simulink and LabVIEW. Hierarchical heterogeneous composition is studied in research projects such as ForSyDe [14], SPEX [19] and ModHel'X [11]. Ptera could presumably become one of the available modeling formalisms in these frameworks.

UML sequence diagrams also have an event-oriented flavor, but they are deeply rooted in the notion of composing sequential processes and threads. Ptera provides a more natural, understandable, and scalable way of modeling and reasoning about event interactions because it is less vulnerable to the conceptual complexities of threads [16].

BIP, like Ptera, is event oriented and supports heterogeneous models [2]. However, its approach to heterogeneity is less hierarchical. The principle behind BIP is to compose state machines using rendezvous interactions with priorities

that have global significance in the model. We believe that Ptera will yield more intuitive and understandable models, though such a claim is extremely difficult to substantiate.

6. 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.

7. 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.

8. REFERENCES

[1] C. André. SyncCharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), University of Sophia-Antipolis, April 1996.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, Pune, 2006.

[3] C. Brooks, C. Cheng, T. H. Feng, E. A. Lee, and R. v. Hanxleden. Model engineering using multimodeling. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM 2008)*, Toulouse, France, September 2008.

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

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

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

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[8] R. 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.

[9] T. H. Feng, E. A. Lee, and L. W. Schruben. Ptera: An event-oriented model of computation. Technical Report UCB/EECS-2010-40, EECS Department, University of California, Berkeley, Apr 2010.

[10] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, 2009.

[11] C. Hardebolle and F. 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.

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

[13] R. G. Ingalls, D. J. Morrice, and A. 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.

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

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

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

[17] E. 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.

[18] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, 2007. ACM.

[19] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando, November 2006.

[20] J. Liu and E. 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.

[21] L. Schruben and E. 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.

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

[23] L. 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.