Precise Modeling and Statistical Modeling: Toward Efficient and Modular Discrete Events

Thomas Huining Feng
CHESS, UC Berkeley
http://www.eecs.berkeley.edu/~tfeng
tfeng@eecs.berkeley.edu
Report for IEOR261 Class

Abstract

Discrete events have been widely accepted as a language to describe the behavior of timed event-driven systems. To model timed systems, designers usually benefit from visual modeling environments, such as Ptolemy II, that allow to compose and connect functional blocks to construct the systems from smaller components. The resulting designs, which precisely reflects the systems' desired behavior, can be directly simulated in the environments, or be deployed in the real applications.

Designers are usually interested in the systems' statistical data also. If only statistical data are of interest, then usually the systems can be re-modeled in such a way that the precise behavior is ignored, but a few variables that describe the systems' statistics remain. It turns out that this re-modeling often leads to more efficient execution due to the isolation of irrelevant facts that exist the precise execution.

This paper starts with a comparison between modeling precise execution and statical execution, and concludes with a future direction of bridging and taking advantage of both.

1 Introduction

Discrete events (DE) have been widely accepted as a language to describe the behavior of timed systems. Software environments have been developed for modeling these systems. Approaches to DE systems can be roughly categorized into two different but related kinds. The first kind of approaches, which we shall call *event-driven* ones here, focuses on the different tasks in the systems. Events are passed from one task to another. The tasks are event-driven in the sense that the operations in them are initiated by the receiving of input events. Tasks may also generate output events for other downstream tasks. Ptolemy II [1] (which we will call Ptolemy hereon for short), a heterogeneous modeling and simulation environment, is an example of these. [1] More examples include GPSS [10] and SIMAN (ARENA) [9], as well as many other simulation environments based on block languages. Timed state machines (see Harel's Statecharts [6] [7] and DCharts [5]) also fall into this category.

A second kind of software environments for DE focus on the events that occur in the systems, and the "causing" relationship between them. We shall characterize these systems as *event-centric* because the basic building blocks are events instead of tasks. SIGMA [12], which implements the event graphs model of computation [11], is an event-centric simulation environment. Nodes in the models represent events, while the connections between the nodes are the (directed) "causing" relations from one event to another.

The difference in the model representations favored by the two categories of environments has significant impact on many other aspects of the DE semantics. In the following sections, these aspects will be discussed. Advantages and disadvantages are identified. At the end of this paper, a direction is pointed out to bridge them and in this ways take advantage of both.

2 CarWash: An Example Model

In this section, we will examine a CarWash example. The block language implemented in Ptolemy and the event graphs implemented in SIGMA are used separately to model the same system. In this way, the difference between the two is shown in practice.

2.1 CarWash model in Ptolemy

In Ptolemy, to design a model with components (built-in and shipped with Ptolemy, or shipped in a 3rd-party

[1] Though Ptolemy supports heterogeneous models of computation, we will only focus on DE here.

library, or created by the users), the designer simply drags them into the canvas, and creates connections between their ports that designate data paths.

Figure 1 shows a CarWash model in Ptolemy, in which a single queue accepts incoming jobs of car wash, and three servers operate in parallel. The inter-arrival time between any two cars is randomly generated with "3+5*random()", where "random" is a built-in function that returns a random number in range $[0, 1)$. On a car's arrival, if one or more servers are available, then one of the available servers starts to wash the car, and becomes busy. The server remains busy until the car wash job is finished in "5+20*random()" time. If no server is available when a car arrives, the car stays in an infinite queue until a server finishes its job.

A node in this graph is a computation unit, which is called *actor* in Ptolemy terminology. A rich library of built-in actors is provided, and the users may also create their own actors. Actors may be created in two ways: by writing code in Java following a set of conventions and rules (see [2]), and by composing the existing actors and grouping them together. We call the actors designed in Java code *atomic actors*, which cannot be further divided. We call the actors created by composition *composite actors*.

In this example, a composite actor named ArrivalGenerator is created to generate car arrival events using existing actors in the Ptolemy library. (The internal design of this actor is discussed later in Figure 3.) The three servers are also modeled as composite actors, each of which can handle one car at a time. A register is assigned to each server, which records the current availability (as a boolean) of the server. When a car arrives, a test is first performed on the successive registers for the three servers. (This is by triggering the bottom ports of the registers.) If a register returns true to its output port on the left, the true-branch of the receiving BooleanSwitch is taken, which routes the car to the available server. If the register's output is false, then the false branch is taken, and the next server, if any, is then tested. If no more server exists for the test, the car goes to the queue and waits.

When a server accepts a car from its JobInput port, it serves the car until the wash is finished, at which time the car leaves via the JobOutput port. Another port, Availability, outputs the server's current availability information when it is changed (for cars' entering the server or leaving). This information is sent to the corresponding register.

2.2 CarWash Model in Event Graphs

Figure 2 shows a design of the CarWash model in event graph. The nodes in the graph represent events, each

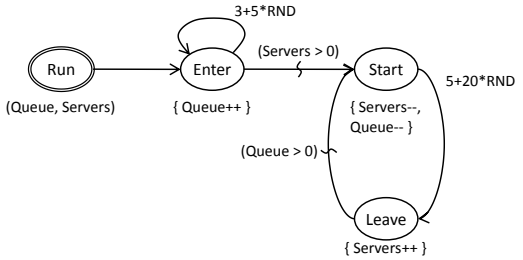

Figure 2: The CarWash example with a single queue and three parallel servers, modeled with event graph

with a unique name. The initial event, i.e., the event that is initially present, is shown with double-lined border. The text between parentheses below a node contains the names of the parameters that the event takes. The expression beside an arrow evaluates to a double number, representing the amount of time delay between the two events (0 if the expression is not omitted). The text between curly brackets contains the operation performed when an event is handled. The expression between parentheses above or beside a curly line is the boolean condition under which an event can be caused by another.

In this model, Queue is an integer variable that records the number of cars in the queue, and Servers is another integer variable that records the number of available servers. Initially, they are 0 and 3, respectively.

3 Aspects of the Discrete Events Implementations

By comparing the two CarWash models, we will discuss in this section aspects of the DE semantics as they are implemented in the two simulation environments, Ptolemy and SIGMA.

3.1 Event Time

The model in Ptolemy and the one in event graph share commonalities in their semantics.

They are both abstractions of the same system (though the levels of abstraction differ, which is to be discussed later). A number of facts in practice are ignored, e.g., how the cars are served. This is because the model designer is not interested in these facts.

The facts that the designer is interested in are modeled in detail. An important fact among these is the model time (or virtual time) of events. This model time (which we will call time hereon) is isolated from the

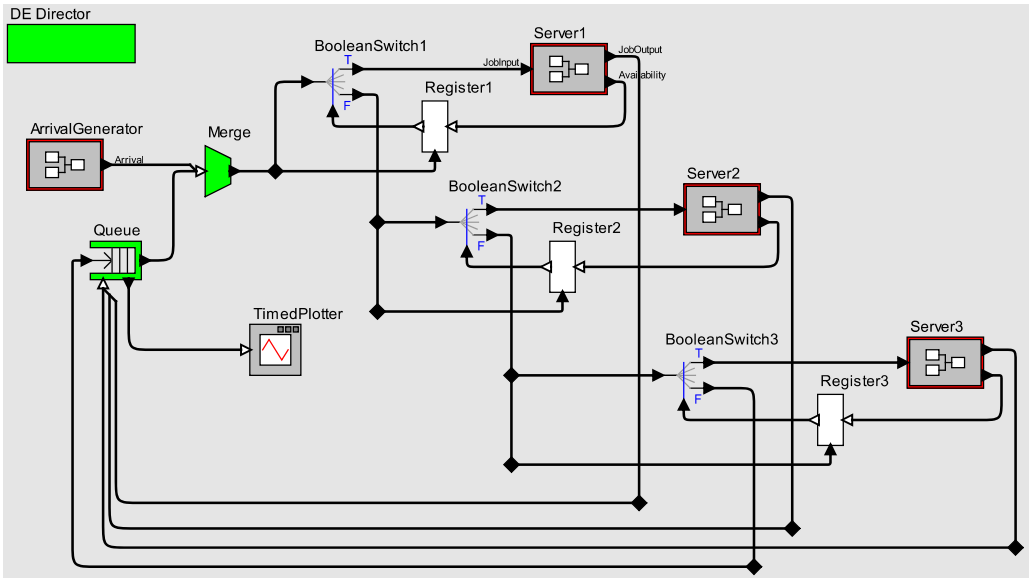

Figure 1: The CarWash example with a single queue and three parallel servers, modeled in Ptolemy

wall clock time. It is represented as a real number in many simulation environments, such as SIGMA. Each event is associated with a time that represents when it was scheduled. These times enforce a total order for all events that do not occur concurrently. (In fact, a partial order is sometimes sufficient.) An event that happens earlier may *schedule* events in the future by specifying positive delays for the events.

For those events that are scheduled at exactly the same time, different mechanisms have been developed for a deterministic ordering. SIGMA, which employs a single global queue to arrange scheduled events. The events scheduled at the same time are ordered either by their pre-defined priorities (in the form of integers) or by the sequence in which they are scheduled (either FIFO, short for first-in-first-out, or LIFO, last-in-first-out). A similar mechanism has been taken in STATEMATE, Harel's implementation of Statecharts. DCharts goes one step further by allowing to specify in a hierarchy of states whether the transitions from the outer states have higher priority or those from the inner states have higher priority. Transitions from outer states can be considered "big steps," while transitions from inner states are "small steps." With DCharts' priority scheme, it is very easy to choose whether big steps or small steps are desired when events happen concurrently. Only when there is no containment relationship between their start states does the simulator fall back on the traditional mechanism of conflict resolution by making use of priority numbers.

Ptolemy has a quite different mechanism for event ordering compared to the previous approaches. The real-numbered time is enhanced with a sequence number. These two components together form a tuple, which is called *tag* to distinguish with the real-numbered time in other implementations. Let $\tau_1 = (t_1, s_1)$ and $\tau_2 = (t_2, s_2)$ be two tags. $\tau_1 \leq \tau_2$ if any only if $(t_1 < t_2) \vee (t_1 == t_2 \wedge s_1 \leq s_2)$. This partial order between tags is called *dictionary order*. In Ptolemy, events with smaller tags are always handled earlier. (We are not considering distributed systems in which it may be beneficial to handle events out of order.) An actor's output events are assumed to have the same tag as the tag of the event that it most recently handles, unless the actor is a delay actor. As an example, in Figure 1, the Merge actor accepts events from either channels of its input port, and outputs the events to its output port. The output event is exactly the same as the last input event. Their tags are also the equal.

In Figure 3, which shows the inside of CarWash model's ArrivalGenerator (recall that it is a composite actor), a VariableDelay is used to delay each arrival by a random amount of time. The delay amount is defined in the Expression actor, which generates a new random every time a trigger is received on its input port on the left. Interestingly, a feed back loop is created to repeatedly generate arrivals. The initial arrival with ID 1 is generated by the SingleEvent actor (which executes autonomously exactly once). The signal is then increased by 1 and fed back to the Merge as the trigger of the next arrival. The Arrival output port corresponds to the ArrivalGenerator's Arrival output port in Figure 1,

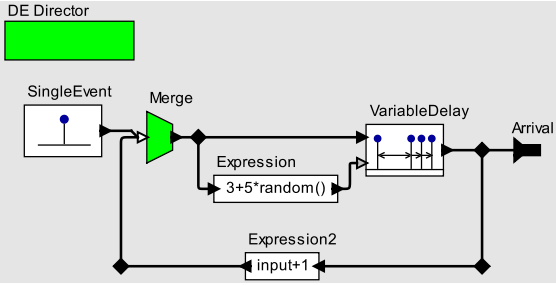

Figure 3: Inside the ArrivalGenerator in the CarWash model

through which the arrivals are sent out.

A lot of events may occur at the same tag in Ptolemy. For example, at the top of Figure 1 Server1's availability is tested. An event is sent to its register as well as the BooleanSwitch connected to it. The choice of BooleanSwitch1 (the bottom port) comes from the output of the register. These all happen at the same tag conceptually. What actually happens is that the arrival event first triggers the register's output, then the register sends its recorded information to the switch, then since the switch is now supplied with both an arrival event and the availability information from the register, it is able to choose a branch and route the input to it.

For the software to generate a valid order to execute the actors, which we call *firing sequence* (since the performance of one operation in an actor is called a *firing*), the fixpoint theory is leveraged (refer to [4] for an introduction). The firings are performed in iterations. In each iteration, each connection in the graph either has a unique imminent event being handled by the actor at its end, or no event needs to be handled at the current tag. We call this event to be handled or the absence of such an event a *signal*. A lattice is defined for signals, with the "absent" element as its bottom. At the beginning of an iteration, all the signals are absent, except those with scheduled imminent events. In the above example, the connection between the ArrivalGenerator and the Merge first has an event present. The software then tries to find a fixpoint for all the signals at the current tag by repeatedly applying this rule: If all required inputs of an actor are present, then its outputs are also present. Due to this rule, the Merge's output is made present, then one input for BooleanSwitch1 is present, as well as Register1's trigger. Because Register1 only requires a trigger to output, its output is also present, providing another input to BooleanSwitch1. At this point, BooleanSwitch1 is provided with all required inputs, so its output to either of the channels is also present. This process repeats until all signals are fixed with a unique value, or "absent." If a signal cannot be computed (e.g., a signal on a connection that is part of an unstable and 0-delay feed back loop), or it has different possible values other than "absent," then a runtime error is returned. The above process is one iteration in the model execution. The current tag is increased to the next most imminent event before the next iteration starts.

The handling of time in Ptolemy is significantly more complex than that in many other simulations environments. It is arguable that this scheme facilitates the design of many models seen in block language. In SIGMA, concurrent events are usually handled in either FIFO order or in LIFO order. This is operationally easy to implement and to understand, but it lacks a mathematic foundation on which the result can be justified. That is, it is very arbitrary and sometimes application-dependent to choose FIFO or LIFO for the concurrent events. No theory exists so far that guides the designer to make the right decision. There are also cases in which neither is desired if, as in the current implementation, the choice has to be made at design time and may not be changed at dynamically. Additional priority numbers may solve this problem in small-scale models, but they cause other problems in large models. They are not easy to use, and there may be unexpected effect on the actual ordering if the designer fails to consider every possible combination of concurrent events. Therefore, it appears that the fixpoint mechanism is more mathematically justified and eventually easy to use correctly.

3.2 Modularity Consideration

Ptolemy models are modular in three senses. First, each atomic actor hides its Java implementation but only reveals an interface with a set of typed and attributed ports. Users usually do not modify the implementation (unless they design their own actors, or subclass existing actors, which in effect creates new actors). This black-box abstraction helps to prevent errors caused by modifying the actors based on false knowledge.

Second, events are the only means of interaction between actors. This interaction can be very easily identified in the model design because events traverse via connections, and all connections are shown as edges in the graph. Actors do not affect each other in any other way (if we do not consider such side effects as disk operations and arbitrariness of the threading in the operating system). Therefore, an actor is confident of the validity of its current state.

Third, a model can have hierarchy. Actors (atomic or composite) can be grouped together to form a larger actor, revealing only an interface to the outside as an atomic actor does. Well tested components can be supplied to the designers for use in their models. This, on

the one hand, saves debugging time, and on the other hand, captures domain-specific knowledge in a black box without requiring the designers to understand it.

The modularity characteristic of Ptolemy helps to significantly reduce the possibility of design errors. Event graphs, however, are not equipped with enough modularity. All the variables are named in the same namespace, which may lead to misuse when the number of variables get large. The number of potentially concurrent events also grows exponentially as more events are created. These facts make it hard to manage large models.

3.3 Transient Entities and Resident Entities

In event graphs, variables usually record the numbers of certain kinds of objects. In Figure 2, the Queue variable records the number of cars in the queue, and the Servers variable records the number of available servers. Operations update those variables by increasing or decreasing them, as the consequence of event handling. In the whole CarWash system there are only two variables in total, which are called *resident entities*. Individual objects, such as cars and servers as distinct objects, are deliberately omitted from the design.

Contrary to this, in Ptolemy, events are transient. In the example, events represent distinct car objects. When a car is served, the corresponding event is delayed by a proper amount of time. After that, it leaves the server and disappears from the system. (In the model, a car that leaves the server actually serves as a trigger for the queue output if any car is waiting in the queue.) These car objects are *transient entities* that get generated and destroyed repeatedly at run-time, resulting in poor efficiency if the number of objects gets large.

If the model allows, it is almost always more efficient to use resident entities than to use transient entities. However, this may or may not be practical, depending on the level of abstraction that is interesting to the designer.

Another argument for resident entities is that they make it very easy to change the number of objects in the system. In our CarWash example, because the Servers variable keeps the number of available servers, adding one more server to the system is as easy as increasing this variable by one. In Ptolemy, this would require copy-and-paste of a server and a register, and creating new connections. Some existing connections also need to be reroute, e.g., the connection to the Queue's input port. Errors are easily made in this modification. To solve this problem so that the users can easily add or remove parts of a model (either at design time or at run-time), higher-order model composition and model transformation are under active research. [3]

3.4 Level of Abstraction

Model is an abstraction of the reality. From the same reality, different models can be built at different levels of abstraction. A model at a higher level of abstraction generally ignores more facts than one at a lower level. In the previous example, the event graph model is at a higher level of abstraction than the Ptolemy model, because the former does not keep track of individual cars and servers, but only the current numbers of them.

Resulting from the difference in the levels of abstraction, the event graph model potentially executes more efficiently since it has fewer variables. If all the designer wants is the statistical result from the system (e.g., the number of cars waiting in the queue as a function of time), then this appears to be a model at the appropriate level of abstraction.

The Ptolemy model, however, allows more different analyses to be perform by providing extra facts. Imagine that after building the model, the designer realizes that another aspect of the system also needs to be analyzed, e.g., the average of car waiting times. The original design in event graph cannot be used any more, because it does not keep track of the waiting times for individual cars. The designer then has to create a separate event graph model for this new analysis. If the requirement keeps changing, the work of redesigning statistical models for the same system may be overwhelming.

With the Ptolemy model, if the waiting time is to be analyzed, one only needs to add a plot that shows time difference between each event's entering the queue and leaving the queue.

This comparison shows that models at a higher level of abstraction tend to execute more efficiently, but those at a lower level of abstraction tend to be easier to adapt to the changing requirements.

3.5 Matching Analysis with the Implementation

Successful stories have been seen in the industry about using SIGMA to construct event graphs models and to analyze them. These models are constructed in such a way that their behavior of interest should be similar to the actual system. This similarity can be numerically analyzed, using output analysis [8].

The models build in SIGMA are for statistical purpose only. They cannot be actually deployed in the hardware or the software of the target system. Ptolemy, however, allows to generate software in C that is directly deployed to embedded devices. This helps to tightly

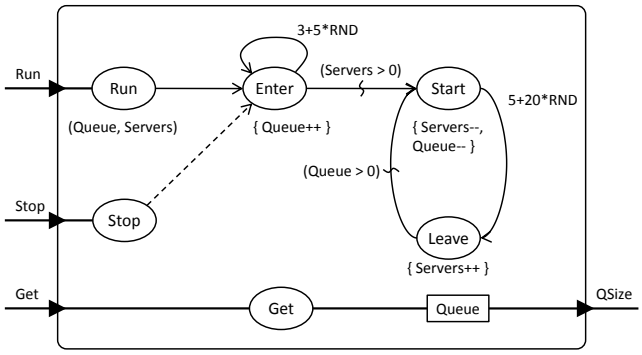

Figure 4: The hierarchical CarWash example as a component with ports in its interface

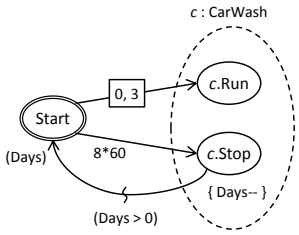

Figure 5: The higher level model that uses the CarWash component to perform simulation for a certain number of days

connect the implementation with the analysis. A modification on the model affects both of the two uses. The analysis is always up-to-date with the implementation, unless the designer decides to manually modify the implementation and at the same time to sacrifice this benefit.

4 Hierarchical Composition with Event Graphs

In the previous section we have seen that event graphs and Ptolemy DE models both have pros and cons. We will now discuss the improvements that can be made on event graphs by introducing advantageous the Ptolemy hierarchy element. We will also discuss improving the efficiency of Ptolemy simulation by adding event graphs to it as a new supported model of computation.

4.1 Compositional Event Graphs with Variable Scopes

An idea of hierarchical event graphs for better modularity was probably first discussed in [13]. In this section, we will try to make this idea more concrete, and to find a direction in which this idea can be implemented.

Figure 4 exemplifies hierarchical event graphs by creating a separate component for the CarWash system. This component has 4 ports. The Run port accepts start of execution events from the external world. These events carry two attributes as the initial values of Queue and Servers. The Stop port accepts stop events that ends a day's car wash service. When a Stop event is received, no more cars are allowed to enter, reflected by the fact that the future car arrival event is cancelled. (The dash line in the graph represents a cancelling edge). The cars already waiting in the queue will still be served, until the queue becomes empty. The external environment can also retrieve the current queue size by sending a Get event to this component, and the current queue size will be immediately sent out with a QSize event. This immediacy is obtained with the LIFO priority scheme, which guarantees that the queue size not being changed after receiving Get but before sending out QSize.

According to [13], the higher level model may be another event graph that contains this component. The containing model may send an event to the component and use the result returned from the component as the delay or in a condition. This is one possible kind of composition.

Another way of composition, which appears to be more structural, may be to consider each port of a component as a separate event in the higher level model, as Figure 5 demonstrates. This higher level model performs simulation for a given number of days (specified in the Days parameter), assuming that on each day the car wash place opens for 8 consecutive hours. In this way, this higher level model need not be aware of the internal implementation of the CarWash component, as long as it conforms to the established interface.

Another interesting point is that the higher level model can potentially contain several components of the same type. In this example, "CarWash" is the type name, and c on the left is the name of an instance, so it has all the behavior defined in that type. Another imaginary model may use two distinct instances of CarWash to simulate two car wash places that open next to each other. One can simulate the cars' choosing one place over the other based on the number of waiting cars and the speed of the servers. Also, by changing the initial parameter, one can simulate the shop that continues to serve unfinished cars the next day: It remembers how many cars are there left in the queue when the shop closes (assuming all service stop right at the close time), and those car owners get coupons for coming back the next day.

An important effect of encapsulation that should be pursued is the isolation of variables in different components. They may be made *private*, and only the components that own them have the access privilege.

4.2 Composing Event Graphs with Other Models of Computation

By encapsulating the implementation of components, one no longer need to be aware of the components' implementation, not event the models of computation in which they are implemented. Because of this, it should be possible to compose event graphs components with models in other domains.

The similar idea of heterogeneous model composition has been successfully exercised by Ptolemy. It allows the designers to create components in any supported model of computation as they see fit, and compose them as a whole, provided that the composition between the two models of computation is meaningful.

This triggers the idea of composing event graphs with other models of computation once the event graphs are properly encapsulated. All that it needs is to create an instance of the component, feed in necessary events, and maybe accept its output events. Currently, the DE model of computation in Ptolemy seems to be compatible with event graphs. Other models of computation with a time concept, such as continuous time (CT), should also work.

For untimed models of computation, such as data flow and process network (PN), composition is still possible. A convention needs to be made between the timed parts and the untimed parts. One example is that the untimed parts repeatedly executes the timed parts (as in Figure 5), but before every execution, the time of the component is reset to 0. The times of the output events are unimportant, as long as the events are correctly ordered in a sequence.

With event graphs as a component of Ptolemy models, the designer may obtain faster execution because the transient entities in the component are not explicitly modeled. This is practical in some cases. For example, for a Ptolemy model that implements part of an embedded system, one may be interested in how well this system works if the power level drops monotonically as a function of time (assuming that the embedded device is uncharging). The designer is not interested in precisely modeling the power supply. In this case, an event graph with only resident entities can be used in place of the power supply that generates the current power voltage according to a statistical distribution.

5 Conclusion

In this paper, we compared the two approaches to discrete events, namely, the block language approach implemented in Ptolemy and the event graphs approach. An example is created in both approaches to illustrate the difference. The direction of combining the advantages of the two and making event graphs compositional with other models of computation is pointed out.

References

[1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng (eds.). Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy II). Memorandum UCB/ERL M05/21, EECS, University of California, Berkeley, CA, USA, Jul 2005.

[2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng (eds.). Heterogeneous concurrent modeling and design in java (volume 2: Ptolemy ii software architecture). Memorandum UCB/ERL M05/22, EECS, University of California, Berkeley, CA, USA, Jul 2005.

[3] 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 9 2006.

[4] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2nd Edition)*. Cambridge University Press, April 2002.

[5] Thomas Huining Feng. DCharts, a formalism for modeling and simulation based design of reactive software systems. Master's thesis, School of Computer Science, McGill University, Montréal, Canada, May 2004.

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

[7] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[8] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, second edition, 1991.

[9] C. Dennis Pegden, Randall P. Sadowski, and Robert E. Shannon. *Introduction to Simulation Using SIMAN*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[10] Thomas J. Schriber. *Simulation Using GPSS*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1990.

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

[12] Lee W. Schruben. Simulation graphical modeling and analysis (SIGMA) tutorial. In *Winter Simulation Conference*, pages 158–161, 1990.

[13] Lee W. Schruben. Building reusable simulators using hierarchical event graphs. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 472–475, 1995.