Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems*

Edward A. Lee
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
eal@eecs.berkeley.edu

Haiyang Zheng
Department of EECS
University of California, Berkeley
Berkeley, CA 94720, USA
hyzheng@eecs.berkeley.edu

ABSTRACT

This paper gives a semantics for discrete-event (DE) models that generalizes that of synchronous/reactive (SR) languages, and a continuous-time (CT) semantics that generalizes the DE semantics. It shows that all three semantic models can be used in actor-oriented composition languages, and that despite the fact that CT is the most general, there are good reasons for using each of the more specialized semantics. Moreover, because of the generalization relationship between them, these three models of computation (MoCs) compose hierarchically in arbitrary order. We describe a design system that supports arbitrary combinations of these three MoCs, leveraging the actor abstract semantics of Ptolemy II.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Languages

Keywords

Composition, Operational Semantics, Model-based design, Synchronous Languages, Discrete Events, Simulation, Hybrid System

*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 award #CCR-0225610), the Air Force Research Lab (AFRL), the Army Research Office (ARO), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, General Motors, Hewlett Pack, Microsoft, National Instruments, and Toyota.

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

An embedded system mixes digital controllers realized in hardware and software with the continuous dynamics of physical systems [30]. Such systems are semantically heterogeneous, combining continuous dynamics, periodic timed actions, and asynchronous event reactions. Modeling and design of such heterogeneous systems is challenging. A number of researchers have defined concurrent **models of computation** (MoCs) that support modeling, specification, and design of such systems [11, 34, 22, 28, 26].

A variety of approaches have been tried for dealing with the intrinsic heterogeneity of embedded systems. This paper proposes a particularly useful combination of semantics, providing a disciplined and rigorous mixture of synchronous/reactive (SR) systems [4], discrete-event (DE) systems [29, 49, 19, 13], and continuous-time (CT) dynamics [42, 20, 46, 35]. Our approach embraces heterogeneity, in that subsystems can be modeled using any of the three semantics, and these subsystem models can be combined hierarchically to form a whole. We leverage the idea of an **actor abstract semantics** [33] to provide a coherent and rigorous meaning for the heterogeneous system. Our approach also provides improvements to conventional DE and CT semantics by leveraging the principles of synchronous/reactive languages. These improvements facilitate the heterogeneous combination of the three distinct modeling styles.

2. RELATED WORK

A number of authors advocate heterogeneous combinations of semantics. Ptolemy Classic [11] introduced the concept, showing useful combinations of asynchronous models based on variants of dataflow and timed discrete-event models. The concept was picked up for hardware design in SystemC (version 2.0 and higher) [45], on which some researchers have specifically built heterogeneous design frameworks [26]. Metropolis [22] introduced communication refinement as a mechanism for specializing a general MoC in domain-specific ways, and also introduced "quantity managers," which provide a unified approach to resource management in heterogeneous systems.

Our approach in this paper is closest in spirit to SML-Sys [41], which builds on Standard ML to provide for mixtures of MoCs. SML-Sys combines asynchronous models (dataflow

models) with synchronous models (which the authors call "timed"). Our approach, in contrast, combines only timed models, including both discrete-event and continuous-time dynamics.

A particular form of heterogeneous systems, hybrid systems provide for joint modeling of continuous and discrete dynamics. A few software tools have been built to provide simulation of hybrid systems, including Charon [2], Hysdel [47], HyVisual [10], Modelica [46], Scicos [16], Shift [15], and Simulink/Stateflow (from The MathWorks). An excellent analysis and comparison of these tools is given by Carloni, et al. [12]. We have previously extensively studied the semantics of hybrid systems as heterogenous combinations of finite state machines (FSM) and continuous dynamics [35]. Our approach in this paper extends this to include SR and DE models. We focus here on the interactions between SR, DE, and CT, because the interactions between FSM and SR, DE, and CT have already been extensively studied [21, 35].

Several authors advocate unified MoCs as a binding agent for heterogeneous models [3, 23, 9]. Heterogeneous designs are expressed in terms of a common semantics. Some software systems, such as Simulink from The MathWorks, take the approach of supporting a general MoC (continuous-time systems in the case of Simulink) within which more specialized behaviors (like periodic discrete-time) can be simulated. The specialized behaviors amount to a **design style** or **design pattern** within a single unified semantics. Conformance to design styles within this unified semantics can result in models from which effective embedded software can be synthesized, using for example Real-Time Workshop or DSpace.

Our approach in this paper is different in that the binding agent is an **abstract semantics**. By itself, it is not sufficiently complete to specify system designs. Its role is exclusively as a binding agent between diverse concrete MoCs, each of which is expressive enough to define system behavior (each in a different way).

We are heavily inspired here by the fixed-point semantics of synchronous languages [4], particularly Lustre [25], Esterel [8], and Signal [24]. SCADE [7] (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on the synchronous language Lustre [25], providing a graphical programming framework with Lustre semantics. All the synchronous languages have strong formal properties that yield quite effectively to formal verification techniques. Our approach, however, is to use the principles of synchronous languages in the style of a **coordination language** rather than a programming language, as done in Ptolemy [17] and ForSyDe [44]. This allows for "primitives" in a synchronous system to be complex components rather than built-in language primitives. This approach will allow for heterogeneous combinations of MoCs, since the complex components may themselves be given as compositions of further subcomponents under some other MoC.

A number of researchers have combined synchronous languages with asynchronous interactions using a principle called **globally asynchronous, locally synchronous** or GALS (see for example [5]). In our case, all the MoCs we consider are timed.

3. ACTOR-ORIENTED MODELS

Our approach here closely follows the principles of **actor-oriented design** [33], a component methodology where components called *actors* execute and communicate with other actors in a *model*, as illustrated in figure 1. Actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes *ports* that represent points of communication for an actor, and *parameters* which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed. The configuration of a model also contains explicit communication *channels* that pass data from one port to another. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

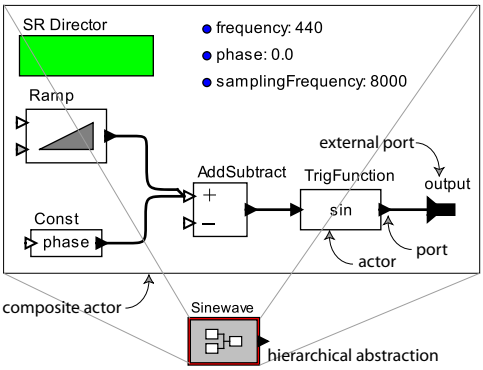

Figure 1: Illustration of a composite actor (above) and its hierarchical abstraction (below).

Like actors, which have a well-defined external interface, a composition of actors may also define an external interface, which we call its *hierarchical abstraction*. This interface consists of *external ports* and *external parameters*, which are distinct from the ports and parameters of the individual actors in the composite. The external ports of a composite can be connected (on the inside) by channels to other external ports of the composite or to the ports of actors within the composite. External parameters of a composite can be used to determine the values of the parameters of actors inside the composite. Actors that are not composite actors are called **atomic actors**. We assume that the behavior of atomic actors is given in a **host language** (in Ptolemy II, Java or C).

Taken together, the concepts of actors, ports, parameters and channels describe the **abstract syntax** of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in figure 1, in XML [32], or in a program designed to a specific API (as in SystemC). Ptolemy II [18] offers all three alternatives. In some systems, such as SML-Sys, the syntax of the host language specifies the interconnection of actors.

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by an MoC. The model of computation might give operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication.

The model of computation also defines the nature of communication between components (buffered, rendezvous, etc.).

Our notion of actor-oriented modeling is related to the work of Gul Agha and others. The term *actor* was introduced in the 1970's by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [27]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. We are further developing the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Moreover, although communication is still through some form of message passing, it need not be strictly asynchronous.

In this paper, we will consider three MoCs, namely SR, DE, and CT. We carefully define the semantics of DE and CT so that the three MoCs are related in an interesting way. Specifically, SR is a special case of DE, and DE is a special case of CT. This does not mean that we should automatically use the most general MoC, CT, because execution efficiency, modeling convenience, and synthesizability all may be compromised. In fact, there are good reasons to use all three MoCs.

Most interestingly, we will show that these three MoCs can be combined hierarchically in arbitrary order. That is, in a hierarchical model like that figure 1, the higher level of the hierarchy and the lower level need not use the same MoC. In fact, all combinations of SR, DE, and CT are supported by our framework. We describe a prototype of this framework constructed in Ptolemy II.

4. ACTOR ABSTRACT SEMANTICS

In order to preserve the specialization of models of computation while also building general models overall, we concentrate on the hierarchical composition of heterogenous models of computation. The composition of arbitrary models of computation is made tractable by an *abstract semantics*, which abstracts how communication and flow of control work. The abstract semantics is (loosely speaking) not the union of interesting semantics, but rather the intersection. It is abstract in the sense that it represents the common features of models of computation as opposed to their collection of features.

A familiar example of an abstract semantics is represented by the Simulink S-function interface. Although not formally described as such, it in fact functions as such. In fact, Simulink works with Stateflow to accomplish a limited form of hierarchical heterogeneity through this S-function interface. We will describe an abstract semantics that is similar to that of Simulink, but simpler. It is the one realized in the Ptolemy II framework for actor-oriented design.

In Ptolemy II models, a **director** realizes the model of computation. A director is placed in a model by the model builder to indicate the model of computation for the model. For example, an SR director is shown visually as the uppermost icon in figure 1. The director manages the execution of the model, defining the flow of control, and also defines the communication semantics.

When a director is placed in a composite actor, as in figure 1, the composite actor becomes an **opaque composite actor**. To the outside environment, it appears to be an atomic actor. But inside, it is a composite, executing under the semantics defined by the local director. Obviously, there has to be some coordination between the execution on the outside and the execution on the inside. That coordination is defined by the abstract semantics.

setup	Initialize the actor.
prefire	Test preconditions for firing.
fire	Read inputs and produce outputs.
postfire	Update the state.
wrapup	End execution of the actor.

Figure 2: The key flow of control operations in the Ptolemy II abstract semantics. These are methods of the Executable interface.

The flow of control and communication semantics are abstracted in Ptolemy II by the *Executable* and *Receiver* interfaces, respectively. These interfaces define a suite of methods, the semantics of which are the actor abstract semantics of Ptolemy II. A receiver is supplied for each channel in a model by the director; this ensures that the communication semantics and flow of control work in concert to implement the model of computation.

In the Ptolemy II abstract semantics, actors execute in three phases, *setup*, a sequence of *iterations*, and *wrapup*. An **iteration** is a sequence of operations that read input data, produce output data, and update the state, but in a particular, structured way. The operations of an iteration consist of one or more invocations of the following pseudocode:

```
if (prefire()) {
   fire();
}
```

If *fire* is invoked at least once in the iteration, then the iteration concludes with exactly one invocation of *postfire*.

These operations and their significance constitute the Executable interface and are summarized in figure 2. The first part of an iteration is the invocation of *prefire*, which tests preconditions for firing. The actor thus determines whether its conditions for firing are satisfied. If it indicates that they are (by a return value of true), then the iteration proceeds by invoking *fire*. This may be repeated an arbitrary number of times. The contract with the actor is that *prefire* and *fire* do not change the state of the actor. Hence, multiple invocations with the same input values in a given iteration will produce the same results. This contract is essential to guarantee convergence to a fixed point.

If *prefire* indicates that preconditions are satisfied, then most actors guarantee that invocations of *fire* and *postfire* will complete in a finite amount of time. Such actors are said to realize a *precise reaction* [37]. A director that tests these preconditions prior to invoking the actor, and fires the actor only if the preconditions are satisfied, is said to realize a *responsible framework* [37]. Responsible frameworks coupled with precise reactions are key to hierarchical heterogeneity.

The abstract semantics also provides the set of primitive communication operations shown in figure 3. These operations allow an actor to query the state of communication channels, and subsequently retrieve information from the channels or send information to the channels. These oper-

get	Retrieve a data token via the port.
put	Produce a data token via the port.
hasToken(k)	Test whether *get* will succeed k times.
hasRoom(k)	Test whether *put* will succeed k times.

Figure 3: Communication operations in Ptolemy II. These are methods of the Receiver interface.

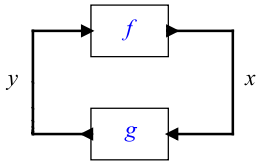

Figure 4: A simple feedback system.

ations are invoked in *prefire* and *fire*. Actors are also permitted to read inputs in *postfire*, but they are not permitted to produce outputs (by invoking *put*). Violations of this contract can lead to nondeterminism.

These operations are abstract, in the sense that the mechanics of the communication channel is not defined. It is determined by the model of computation. A domain-polymorphic actor is not concerned with how these operations are implemented. The actor is designed assuming only the abstract semantics, not the specific realization.

A hierarchically heterogeneous model is supported by this abstract semantics as follows. Figure 1 shows an opaque composite actor. It is opaque because it contains a director. That director gives the composite a behavior like that of an atomic actor viewed from the outside. A director implements the Executable interface, and thus provides the operations of figure 2.

Suppose that in figure 1 the hierarchical abstraction of the Sinewave component is used in a model of computation different from SR. Then from the outside, this model will appear to be a domain-polymorphic actor. When its *prefire* method is invoked, for example, the inside director must determine whether the preconditions are satisfied for the model to execute (in the SR case, they always are), and return true or false accordingly. When *fire* is invoked, the director must manage the execution of the inside model so that input data (if any) are read, and output data are produced. When *postfire* is invoked, the director must update the state of the inside actors by invoking their *postfire* methods. Obviously, directors must be carefully designed to obey the actor abstract semantics contract. By obeying it, they gain the ability to be nested arbitrarily with other directors that also obey the contract.

The communication across the hierarchical boundary will likely end up heterogeneous. In figure 1, the connection between the TrigFunction actor and the external port will be a channel obeying SR semantics (that is, it will be realized as a simple buffer with length one, support for *unknown* state, and enforcement of monotonicity constraints). The connection between the external port and some other port on the outside will obey the semantics of whatever director is provided on the outside. This need not be the same as the SR semantics.

In this paper, we will focus on the use of three directors in Ptolemy II implementing synchronous/reactive, discrete-event, and continuous-time MoCs.

5. SYNCHRONOUS/REACTIVE MODELS

We begin with the principle of synchronous languages, but used in the style of a coordination language rather than a programming language, as done in Ptolemy [17] and ForSyDe [44]. We will show that by adding time to this, we get a clean semantics for DE systems, and that by adding continuous dynamics, we get a clean semantics for continuous-time, hybrid and mixed signal systems (CT models).

The principle behind synchronous languages is simple, although the consequences are profound [4]. Execution follows "ticks" of a global "clock." At each tick, each variable (represented visually by the wires that connect the blocks) may have a value (it can also be absent, having no value). Its value (or absence of value) is defined by functions associated with each block. That is, at each tick, each block is a function from input values to output values (the function can vary from tick to tick). In figure 4, the variables x and y at a particular tick are related by

$$x = f(y), \text{ and } y = g(x).$$

The task of the compiler is to synthesize a program that, at each tick, solves these equations. We assume a Scott order on the values that x and y can take on at each tick, with $\perp$ being the bottom of the order, representing "unknown." It is well known that if the functions f and g are monotonic in this order, then a unique least fixed point solution can be found in finite time. See [17] for a readable exposition of this semantics.

In most synchronous languages, there is no metric associated with the time between ticks (ForSyDe is an exception, as it defines a fixed constant "distance" between events of a signal). This means that programs can be designed to simply react to events, whenever they occur. This contrasts with Simulink, which has temporal semantics.

In our case, SR models by default have no notion of the passage of time, and only a notion of sequences of ticks. The semantics can be easily extended by associating a fixed interval of time between ticks, or even by associating a variable interval (somehow determined by the environment). These extensions have no effect on the SR semantics, and hence just amount to interpretations of an execution.

Unlike most work with synchronous languages, we assume that the SR model is being used to coordinate components of arbitrary complexity, rather than to coordinate known language primitives. For example, a Lustre compiler "knows" that *pre* breaks data precedences, whereas in our framework, we may not be able to know whether a component breaks data precedences. We can give an execution model that is very simple, but rather inefficient. In particular, at each tick of the clock, we can start with all signal values unknown, $\perp$, and invoke *prefire* and *fire* on each actor, in arbitrary order, until signal values stabilize at a fixed point. Here, *prefire* will specify whether there is sufficient information about the inputs for *fire* to execute. Once a fixed point is reached, then *postfire* is invoked on each actor exactly once, allowing the actor to update its state.

This execution strategy requires no knowledge of the internals of the components, except that they conform with the actor abstract semantics and implement monotonic func-

tions. But it may result in rather inefficient execution. Following [17], we rely on causality interface information for actors (see [36]) to develop more efficient execution strategies. A key property of SR models is that the results of execution do not depend on the execution strategy. Only the efficiency of execution does.

To give a denotational semantics for such systems, we follow the approach in [38], which specializes the tagged signal model [34] to timed systems. Denotationally, a **signal** is the entire history of a communication between an output port and an input port. Specifically, let $T = \mathbb{N}$ be the tag set, where $\mathbb{N}$ is the set of natural numbers with the usual numerical order. Let V be an arbitrary family of values (a data type, or alphabet). Let

$$V_\varepsilon = V \cup \{\varepsilon\}$$

be the set of values plus "absent." Then a signal s is a partial function:

$$s : T \rightharpoonup V_\varepsilon$$

defined on an initial segment of T. Execution of an SR model begins with all signals being empty (defined on the empty initial segment $\emptyset \subset \mathbb{N}$). The first step of execution extends the signals so that they are all defined on the initial segment $\{0\} \subset \mathbb{N}$. If this is not possible (some signals remain undefined, or equivalently, unknown ($\perp$), at tag 0), then we declare the model to be flawed (it has a "causality loop").[1] The second step of execution extends the signals so that they are defined on the initial segment $\{0, 1\} \subset \mathbb{N}$. The third step defines them on $\{0, 1, 2\} \subset \mathbb{N}$, and so on.

For hierarchical models, like those in figure 1, we can exploit the actor abstract semantics to get multiclock SR models. Specifically, consider the example in figure 5. This shows a classic example from the early literature on synchronous languages, the "guarded count." The interesting part of the model is that the composite actor in the middle of the figure has its own SR Director, and hence is an opaque composite actor. The outside director cannot distinguish between it and an atomic actor, like the When actor at the bottom. This particular composite actor has a specialized behavior. When its *fire* method is invoked, it examines the *enable* input port. If the signal value at the current tick is known and present at this port, then it fires the inside actors until it reaches a fixed point. This fixed point may define output values. When the *postfire* method of the composite actor is invoked, then it invokes *postfire* on all the contained actors that were fired in the iteration. Thus, the clock of the inside model is a subclock of the outside model. The composite actor with a director itself behaves just like an atomic actor, conforming to the actor abstract semantics. It is precisely this compositional property that will enable us to put SR models within DE or CT models and vice versa.

Note that the composite actor at the bottom of figure 5 has no director, and hence is said to be **transparent**. The actors within it are controlled by the top-level director as if there were no hierarchy. Readers familiar with synchronous language operators like "when" and "default" should be able to read this model and understand how it implements the guarded count.

[1]This conforms to the *constructive semantics* of Berry [6].

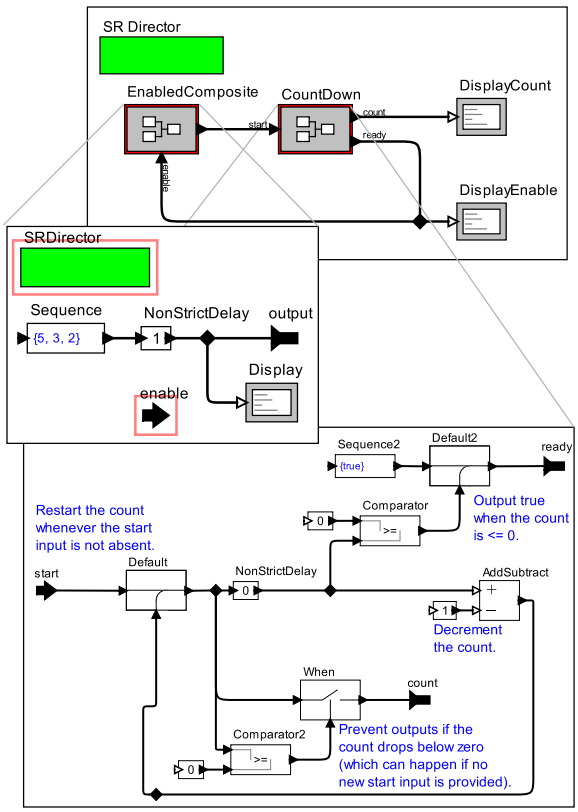

Figure 5: Example of a multiclock SR model.

6. DISCRETE-EVENT MODELS

There is a long history of DE modeling techniques, and a number of widely used simulation systems (e.g., Opnet, NS-2) and programming languages (e.g., VHDL, Verilog) with DE semantics. The approach we take, however, is a bit different from all the DE systems we know of. In particular, we define DE to be an extension of SR where there is a measurable time between ticks of the clock. Specifically, a DE model becomes an SR model if we ignore the passage of time and focus instead only on the sequence of times at which events occur (each such time is a "tick" of the SR clock). The SR model is an abstraction of the DE model.

Since for DE we want to measure the passage of time, the tag set $T = \mathbb{N}$ that we used for SR becomes inadequate. A first attempt at a denotational semantics would simply use real numbers for time increments between ticks of the clock. Specifically, we would replace the tag set T with $T = \mathbb{R}_+$, the non-negative real numbers, instead of $T = \mathbb{N}$. A signal s becomes a partial function,

$$s : \mathbb{R}_+ \rightharpoonup V_\varepsilon$$

defined on an initial segment of $\mathbb{R}_+$. If we let $I \subseteq \mathbb{R}_+$ be the initial segment where the signal is defined, then for s to be a discrete-event signal, we require that $s(\tau) = \varepsilon$ for all $\tau \in I$ except for a discrete subset $D \subset \mathbb{R}_+$.[2]

This semantic model, however, is not sufficiently rich. In particular, signals cannot have multiple values at the same

[2]A discrete subset is one that is order isomorphic to a subset of the natural numbers. This property ensures that every DE model has an SR abstraction.

value of time. This makes the model awkward for models of software sequences that are abstracted as instantaneous (the perfect synchrony hypothesis [4]), transient states in modal models [35], and batch arrivals in network systems, to name a few examples.

Following [40, 35, 39], we solve these problems by using **super-dense time**. Let $T = \mathbb{R}_+ \times \mathbb{N}$ be a set of tags, and give a signal s as a partial function:

$$s \colon T \rightharpoonup V_\varepsilon$$

defined on an initial segment of T, assuming a lexical ordering on T:

$$(t_1, n_1) \leq (t_2, n_2) \iff t_1 < t_2, \text{ or } t_1 = t_2 \text{ and } n_1 \leq n_2 \,.$$

For a particular tag $t = (\tau, n) \in T$, $\tau \in \mathbb{R}_+$ represents physical time, whereas $n \in \mathbb{N}$ represents the ordering of events that occur at that physical time. We again require $s(\tau) = \varepsilon$ for all tags τ in the initial segment on which s is defined except a discrete subset.

Execution again starts with all signals being empty (i.e. nowhere defined, or unknown at all tags), and finds a fixed point at the first tag $(0, 0) \in T$, just as in SR. To proceed to the next step, however, it would not be sufficient to simply go to the second tag $(0, 1) \in T$. With such an execution policy, we would have an infinite sequence of steps to resolve signal values at all tags of the form $(0, n)$, where $n \in \mathbb{N}$. Physical time would not advance. We thus need to augment the actor abstract semantics.

The augmentation of the actor abstract semantics that is realized in Ptolemy II reflects a preference for absent values, ε over present values.[3] Specifically, we assume that if at a tag $t \in T$ an actor is presented with all input signals that have value ε at that tag, then all its outputs will be ε unless it has specifically declared otherwise. Operationally, each time the *setup* phase or *postfire* method of an actor is invoked, it has the option of requesting a firing at some future tag t, in which case it will be fired at that tag even if all its inputs are ε. In Ptolemy II, the actor does this by calling a Director method called *fireAt*, passing it a future time value at which the actor wishes to be fired. Otherwise, it has no expectation of being fired at any tag when its inputs are all absent.

Thus, when a fixed point is reached at any tag $t = (\tau, n)$ (e.g., on the first step, a fixed point is reached at $t = (0, 0)$), then when *postfire* is invoked on all the actors, some of them may call *fireAt*. If no actor called *fireAt*, then implicitly all actors have declared that they will not produce non-absent events given absent inputs any time in the future, and we can infer that all signals are therefore absent for all future tags. Now all signals are completely defined on the tag set T and the execution is complete. More commonly, some actors will have called *fireAt* specifying a time value. We find the minimum such time value τ_{min}. If $\tau_{min} = \tau$, the current time value, then we proceed with the next iteration at tag $t = (\tau, n+1)$. Otherwise, we proceed with the next iteration at tag $t = (\tau_{min}, 0)$.

In our operational semantics *postfire* is called exactly once at each tag for any actor that was fired. Thus, τ_{min} is uniquely determined for each tag τ at which actors are fired. The first τ at which to fire actors is uniquely determined by giving each actor an opportunity to call *fireAt* in its setup phase. Thus, composition of actors remains associative, as it is with SR.

Of course, this execution strategy does not guarantee that time will advance forever. In particular, by invoking *fireAt* with sufficiently small time increments, any actor may block the progress of time. Such a situation is known as a Zeno condition. Zeno conditions can be prevented with appropriate constraints on the behavior of actors (see for example [14, 38, 39].

When we proceed from tag to tag, we must keep track of all previous *fireAt* requests that have not yet been satisfied. This task is, essentially, what a typical DE simulator uses an **event queue** for. In our model, an event queue can be as simple as a set of tags in the future. A more efficient implementation (but semantically equivalent) would keep the set ordered (using for example a priority queue). A still more efficient implementation would keep track, for each event, of which actor(s) requested a firing at the tag. Then the execution engine would not need to fire actors that have already (implicitly) declared that their outputs will be absent.

Although these implementations are more efficient, they are semantically equivalent to a very simple execution strategy. Start at tag $t = (0, 0)$ and fire actors to find a fixed point. If a fixed point results in some signals being undefined ($\perp$) at $t = (0, 0)$, then declare the model to be flawed (causality loop). Otherwise, postfire all actors that were fired. Then select the smallest tag in the event queue, increment t, and repeat. Semantically, this is exactly SR, except that now there is a measurable time between ticks of the clock, and the measure of the time increment is determined by the actor invocations of *fireAt*.

Consider the example in figure 6, which represents a fairly typical scenario. The top-level model is a DE model of a system that is clocked at a regular rate (by the Clock actor), but is also affected by irregular events generated by the PoissonClock actor. This example produces a sinusoidal signal, generating one sample for each event from the Clock actor. When the PoissonClock actor produces an event, however, the model switches from generating a clean sinusoid to a noisy sinusoid, or vice versa. The switching is modeled one level down in the hierarchy by a **modal model** [21] with two modes, labeled "clean" and "noisy." The sinusoids themselves are generated one level further down in the hierarchy by a SR opaque composite actors, as shown. The SR composites themselves have no timed semantics. They simply produce the next sample of the sinusoid on each tick. When these ticks occur is controlled above these composites in the hierarchy.

In this example, the ability to put SR composites within a modal model within a DE model is a direct consequence of the fact that each of these opaque composite actors conforms to the actor abstract semantics.

The converse containment is also possible, where DE is put within SR. However, if SR is the top level, then the top-level SR Director needs to regulate the passage of time, effectively acquiring DE semantics. That is, it cannot abstract away the passage of time if its submodels have to be concrete about the passage of time. For this reason, the SR Director in Ptolemy II includes a *period* parameter, that if set, defines a fixed time increment between ticks of the SR model. If used, this turns an SR model into a simple DE model with a fixed time increment between ticks.

[3]This observation is due to Eleftherios Matsikoudis.

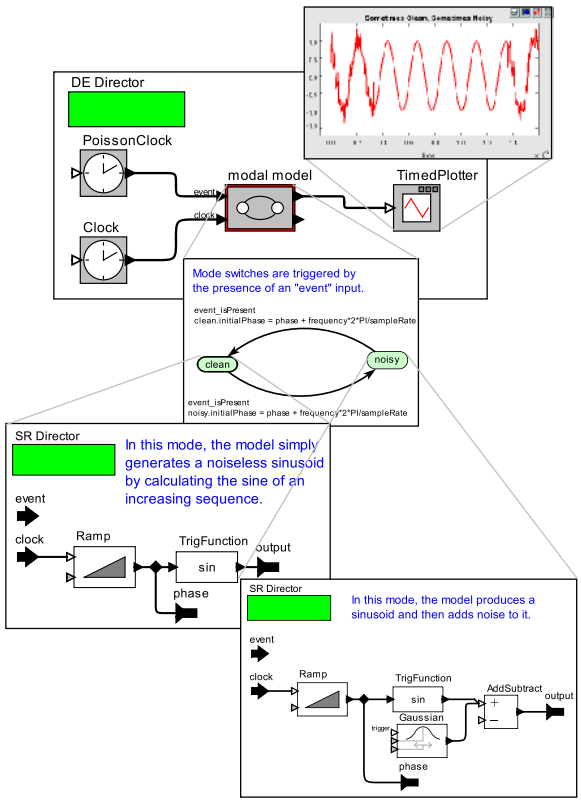

Figure 6: Example of mixed DE and SR model.

7. CONTINUOUS-TIME MODELS

CT models include continuous dynamics, typically given as ordinary differential equations (ODEs). Figure 7 shows a block diagram representation representing a simple third-order nonlinear differential equation. An ODE can be represented by a set of first-order differential equations on a vector-valued state,

$$\begin{aligned} \dot{x}(t) &= g(x(t), u(t), t), \\ y(t) &= f(x(t), u(t), t), \end{aligned}$$

where $x : \mathbb{R} \to \mathbb{R}^n$, $y : \mathbb{R} \to \mathbb{R}^m$, and $u : \mathbb{R} \to \mathbb{R}^l$ are state, output, and input signals. The functions $g : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^m$ are state functions and output functions respectively. The state function g is represented collectively by the actors in the grey area (the grey box is purely decorative and devoid of semantics) on the left in figure 7.

In [35], we have shown how to extend the semantics of such differential equation systems to superdense time, allowing us to use the same model of time that we used with DE. This is useful for modeling hybrid systems, which combine continuous dynamics like that in figure 7 with discrete mode transitions like that in figure 6. In fact, for the tags at which these discrete mode transitions occur, the model functions exactly like a DE model. The interesting extension with CT is that between these tags, instead of all signals being absent, some (or all) signals have continuously evolving values, as governed by an ODE.

Of course, in a digital computer, values do not evolve continuously. The ODE must be approximated by a **solver** [43].

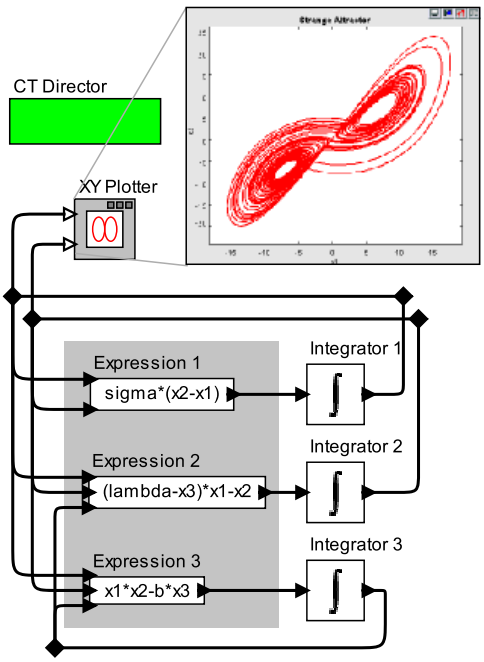

Figure 7: Continuous-time model.

The solver will provide samples of the continuous evolution, and typically, to maintain adequate accuracy, must control the time steps between such samples. The actor abstract semantics, it turns out, supports the inclusion of such solvers as part of the execution of models.

One family of solvers use *Runge-Kutta* (RK) methods, which perform interpolation at each integration step to approximate the derivative at a discrete subset of time points. An explicit k stage RK method has the form

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i, \tag{1}$$

where

$$\begin{aligned} K_0 &= h_n g(x(t_{n-1}), u(t_{n-1}), t_{n-1}), \\ K_i &= h_n g(x(t_{n-1}) + \sum_{j=0}^{i-1} A_{i,j} K_j, u(t_{n-1} + h b_i), \\ &\qquad t_{n-1} + h b_i), \quad i \in \{1, \cdots, k-1\} \end{aligned}$$

and $A_{i,j}$, b_i and c_i are algorithm parameters calculated by comparing the form of a Taylor expansion of x with (1).

The first order RK method, also called the *forward Euler* method, has the (much simpler) form

$$x(t_n) = x(t_{n-1}) + h_n \dot{x}(t_{n-1}).$$

This method is conceptually important but not recommended for practical usage on real applications. More practical RK methods have $k = 3$ or 4, and also control the step size for each integration step. An RK method implemented in Matlab ODE suite is a $k = 3$ stage method and given by

$$x(t_n) = x(t_{n-1}) + \frac{2}{9} K_0 + \frac{3}{9} K_1 + \frac{4}{9} K_2, \tag{2}$$

where

$$K_0 = h_n g(x(t_{n-1}), t_{n-1}), \quad (3)$$

$$K_1 = h_n g(x(t_{n-1}) + 0.5K_0, u(t_{n-1} + 0.5h_n), t_{n-1} + 0.5h_n), \quad (4)$$

$$K_2 = h_n g(x(t_{n-1}) + 0.75K_1, u(t_{n-1} + 0.75h_n), t_{n-1} + 0.75h_n). \quad (5)$$

Notice that in order to complete one integration step, this method requires evaluation of the function g at intermediate times $t_{n-1}+0.5h_n$ and $t_{n-1}+0.75h_n$, in addition to the times t_{n-1}, where h_n is the step size. This fact has significant consequences for compositionality of this method. In fact, any method that requires intermediate evaluations of the state functions g, such as the classical fourth-order RK method, the linear multi-step methods (LMS) and Burlirsch-Store methods, will have to face the same issue during composition. An additional complication is that validity of a step size h_n is not known until the full integration step has been completed.

To show how solving these problems is facilitated by the actor abstract semantics, consider the model shown in figure 7. In the grey area on the left is a collection of actors that collectively implement the state function g. We assume these actors are black boxes that conform with the actor abstract semantics. That is, they could be internally implemented as composite actors with SR or DE directors, for example, or as modal models [21]. To evaluate g at $t_{n-1} + 0.5h_n$ and $t_{n-1} + 0.75h_n$, we must *fire* but not *postfire* these actors. Postfiring the actors would erroneously commit them to state updates before we know whether the step size h_n is valid. Thus, in effect, the solver must provide them with tentative inputs at each tag (one tag for each of these time values), as shown in (4) and (5), and find a fixed point at that tag. But it must not commit the actors to any state changes until it is sure of the step size. Avoiding invocation of the *postfire* method successfully avoids these state changes, as long as all actors conform to the actor abstract semantics.

We can now see that CT operates similarly to DE models, with the only real difference being that in addition to using an event queue to determine the advancement of time, we must also consult an ODE solver. The same *fireAt* mechanism that we used in DE would be adequate, but for efficiency we have chosen to use a different mechanism that polls relevant actors for their constraints on the advancement of time and aggregates the results. In our implementation, any actor can assert that it wishes to exert some influence on the passage of time by implementing a *ContinuousStepSizeController* interface. All such actors will be consulted before time is advanced. The *Integrator* actors implement this interface and serve as proxies for the solver. But given this general mechanism, there are other useful actors that also implement this interface. For example, the *LevelCrossingDetector* actor implements this interface. Given a continuous-time input signal, it looks for tags at which the value of the signal crosses some threshold, given as a parameter. If a step size results in a crossing of the threshold, the actor will exert control over the step size, reducing it until the time of the crossing is identified to some specified precision.

Since the CT Director only assumes that component actors conform to the actor abstract semantics, these actors can be opaque composite actors that internally contain SR or DE models. Moreover, a CT model can now form an opaque composite actor that exports the actor abstract semantics, and hence CT models can be included within SR or DE models and vice-versa (subject again to the constraint that if SR is at the top level, then it must be explicit about time).

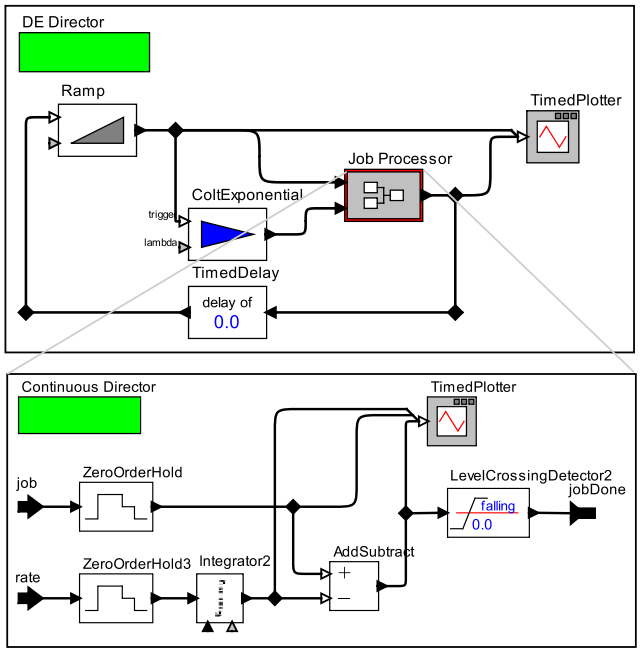

Figure 8: Continuous-time opaque composite actor within a DE model.

A simple example is shown in figure 8. The top-level model is DE representing a sequence of discrete jobs with increasing service requirements. For each job, a random (exponential) service rate is generated. The inside model uses a single integrator to model the (continuous) servicing of the job and a level-crossing detector to detect completion of the job.

8. SOFTWARE IMPLEMENTATION

A prototype of the techniques described here in Ptolemy II is available in open-source form (BSD-style license) at http://ptolemy.org. We started with the *SRDirector* created by Whitaker [48], which was based on an SR director in Ptolemy Classic created by Edwards [17]. We then used this director as a base class for a new *ContinuousDirector*. Unlike the predecessor *CTDirector* created by Liu [37], this new director realizes a fixed point semantics at each discrete time point. The discrete time points are selected from the time continuum, as explained above, in response to actors that invoke *fireAt* and actors that implement *ContinuousStepSizeController*. The latter include Integrator actors, which use an underlying ODE solver with variable step size control.

We modified *SRDirector* and implemented *ContinuousDirector* so that both now rigorously export the actor abstract semantics. That is, when the *fire* method of either director is invoked, the director does not commit to any state

changes, and it does not invoke *postfire* on any actors contained in its composite. Thus, if those actors conform to the actor abstract semantics, then so does the opaque composite actor containing the director.

These improvements led to significant improvements in simplicity and usability. For one, whereas before we had a menagerie of distinct versions of *CTDirector*, we now only need one. Previously, in order to compose continuous-time models with other MoCs (such as DE for mixed signal models and FSM for modal models and hybrid systems), we needed to implement specialized cross-hierarchy operations to coordinate the speculative execution of the ODE solver with the environment. This resulted in distinct directors for use inside opaque composite actors and inside modal models.

We also acquired the ability to put SR inside continuous-time models. This is extremely convenient, because SR can be used to efficiently specify numeric computations and complex decision logic, where the continuous dynamics of CT is irrelevant and distracting. Note that it would be much more difficult to use dataflow models, such as SDF [31] inside CT models. This is because in dataflow models, communication between actors is queued. In order to support the speculative executions that an ODE solver performs, we would have to be able to backtrack the state of the queues. This would add considerable complexity. SR has no such difficulty.

Since the CT MoC is a generalization of the SR, in principle, SR becomes unnecessary. However, SR is much simpler, not requiring the baggage of support for ODE solvers, and hence is more amenable to formal analysis, optimization, and code generation.

At the time of this writing, the *DEDirector* is not a subclass of *SRDirector*. Our implementation of the SR semantics presumes that most actors will have something useful to do on every tick of the clock, and hence invokes at least *prefire* for every actor. However, our implementation of DE semantics presupposes that unless an actor has previously called *fireAt*, that it will respond to absent inputs by producing absent outputs. Thus, actors with absent inputs are not invoked at all. Unfortunately, this presupposition is actually an unwritten and unenforced contract with the actors. A consequence is that the same actor may behave significantly differently under the SR director than under the DE director. It is unclear to us at this point whether this is a feature or a bug. We feel that we need more design experience to determine this. We plan to continue evolving these directors, improving their optimizations, and could perhaps unify these distinct behaviors in a sensible way.

9. CONCLUSIONS

In this paper, we developed an operational semantics that supports mixtures of SR, DE, and CT models of computation, and outlined a corresponding denotational semantics. Dialects of DE and CT are developed that generalize SR, but provide complementary modeling and design capabilities. We show that the three MoCs can be combined hierarchically in arbitrary order.

10. ACKNOWLEDGMENTS

Thanks to Jie Liu, Xiaojun Liu, Eleftherios Matsikoudis, Reinhard von Hanxleden, and four anonymous reviewers for helpful comments on the techniques and content of this paper.

11. REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, 2003.

[3] 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.

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

[5] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *EMSOFT*. Springer, 2003.

[6] G. Berry. *The Constructive Semantics of Pure Esterel*. Book Draft, 1996.

[7] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[8] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] R. Boute. Integrating formal methods by unifying abstractions. In E. Boiten, J. Derrick, and G. Smith, editors, *Fourth International Conference on Integrated Formal Methods (IFM)*, volume LNCS 2999, page 441460, Canterbury, Kent, England, 2004. Springer-Verlag.

[10] C. Brooks, A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. Hyvisual: A hybrid system visual modeler. Technical Report UCB/ERL M03/30, University of California, Berkeley, July 17 2003.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, 1994.

[12] L. P. Carloni, M. D. DiBenedetto, A. Pinto, and A. Sangiovanni-Vincentelli. Modeling techniques, programming languages, and design toolsets for hybrid systems. Technical Report IST-2001-38314 WPHS, Columbus Project, June 2004.

[13] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[14] A. Cataldo, E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan, 2006.

[15] A. Deshpande, A. Gollu, and P. Varaiya. The shift programming language for dynamic networks of hybrid automata. *IEEE Trans. on Automatic Control*, 43(4), 1998.

[16] R. Djenidi, C. Lavarenne, R. Nikoukhah, Y. Sorel, and

S. Steer. From hybrid simulation to real-time implementation. In *11th European Simulation Symposium and Exhibition (ESS99)*, page 7478, 1999.

[17] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1), 2003.

[18] 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(2):127–144, 2003.

[19] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag, 2001.

[20] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[21] A. Girault, B. Lee, and E. 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.

[22] G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag.

[23] G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55, 2005.

[24] P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), 1991.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[26] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in systemc. In *Design Automation Conference (DAC)*, San Francisco, 2006. ACM.

[27] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323363, 1977.

[28] A. Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003.

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

[30] E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.

[31] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 1987.

[32] E. A. Lee and S. Neuendorffer. Moml - a modeling markup language in xml. Technical Report UCB/ERL M00/12, UC Berkeley, March 14 2000.

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

[34] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.

[35] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages pp. 25–53, Zurich, Switzerland, 2005. Springer-Verlag.

[36] E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT), Satellite to CONCUR*, San Francisco, CA, 2005.

[37] J. Liu. Responsible frameworks for heterogeneous modeling and design of embedded systems. Ph.D. Thesis Technical Memorandum UCB/ERL M01/41, December 20 2001.

[38] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report EECS-2006-67, UC Berkeley, May 18 2006.

[39] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, volume LNCS 4137, Bonn, Germany, 2006. Springer.

[40] Z. Manna and A. Pnueli. Verifying hybrid systems. *Hybrid Systems*, pages 4–35, 1992.

[41] D. A. Mathaikutty, H. D. Patel, and S. K. Shukla. A functional programming framework of heterogeneous model of computation for system design. In *Forum on Design and Specification Languages (FDL)*, Lille, France, 2004.

[42] P. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In F. Varager and J. H. v. Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 1569, page 165177. Springer-Verlag, 1999.

[43] W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, 1992.

[44] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 23(1):17–32, 2004.

[45] S. Swan. An introduction to system level modeling in systemc 2.0. Technical report, Open SystemC Initiative, May 2001 2001.

[46] M. M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[47] F. D. Torrisi, A. Bemporad, G. Bertini, P. Hertach, D. Jost, and D. Mignone. Hysdel 2.0.5 - user manual. Technical report, ETH, 2002.

[48] P. Whitaker. The simulation of synchronous reactive systems in Ptolemy II. Master's Report Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of California, May 2001.

[49] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.