A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems

JIE LIU and EDWARD A. LEE
University of California, Berkeley

Systems with both continuous and discrete behaviors can be modeled using a mixed-signal style or a hybrid systems style. This article presents a component-based modeling and simulation framework that supports both modeling styles. The component framework, based on an actor metamodel, takes a hierarchical approach to manage heterogeneity in modeling complex systems. We describe how ordinary differential equations, discrete event systems, and finite-state machines can be built under this metamodel. A mixed-signal system is a hierarchical composition of continuous-time and discrete event models, and a hybrid system is a hierarchical composition of continuous-time and finite-state-machine models. Hierarchical composition and information hiding help build clean models and efficient execution engines. Simulation technologies, in particular, the interaction between a continuous-time ODE solving engine and various discrete simulation engines are discussed. A signal type system is introduced to schedule hybrid components inside a continuous-time environment. Breakpoints are used to control the numerical integration step sizes so that discrete events are handled properly. A "refiring" mechanism and a "rollback" mechanism are designed to manage continuous components inside a discrete event environment. The technologies are implemented in the Ptolemy II software environment. Examples are given to show the applications of this framework in mixed-signal and hybrid systems.

Categories and Subject Descriptors: I.6.5 [**model Development**]: *modeling methodologies*; I.6.8 [**Types of Simulation**]: *continuous, discrete event*; C.3 [**Special-Purpose and Application-Based Systems**]: *real-time and embedded systems*

General Terms: Design

Additional Key Words and Phrases: Component-based modeling, simulation, mixed-signal systems, hybrid systems, actors-oriented design, hierarchical heterogeneity, Ptolemy II

1. INTRODUCTION

Complex engineering systems are usually heterogeneous. In a broad sense, heterogeneity means that the types of components in a system and their interaction

This work is supported in part by the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, National Semiconductor, and Philips.
Author's addresses: J. Liu, Palo Alto Research Center (PARC), Palo Alto, CA 94304; email: jieliu@parc.com; E. A. Lee, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 518 Cory Hall, Berkeley, CA 94720-1770; email: eal@eecs.berkeley.edu.

styles are different. Heterogeneity can be obvious at the implementation level. For example, in an electromechanical system, there are mechanical parts that have certain physical properties, and component interactions are usually characterized by Newton's laws; there are electronic parts, whose physical dimension may not be very important, and component interactions are usually characterized by Kirchoff's laws. Although the ultimate goal of a design is usually the implementation, a system is not necessarily being modeled to the physical details at early design stages. The art of system modeling is to choose the right level of abstraction to capture the aspects worth exploring, and to ignore the irrelevant details.

Over the years, different engineering domains have come up with various modeling abstractions that best suit the design of particular kinds of systems. These abstractions expose the properties that domain engineers like to explore, and hide the information that is of less interest. For example, software engineers think in terms of procedural operations and sequential changes of values in variables. This hides the way that software is executed physically in electronic devices. In another example, most digital signal processing (DSP) engineers think in terms of data samples, which, although they may keep a certain connection with physical time, abstract away the duration between samples. Time is no longer a continuous variable, and a signal becomes a sequence of numbers.

So, even at the modeling level, heterogeneity exists in the formalisms of systems, components, and interactions. This heterogeneity is a mathematical abstraction and reflects the different ways of thinking by domain engineers. As systems become more complex, intelligent, software-enabled, and interconnected, the integration of multiple modeling formalisms becomes a major bottleneck in engineering design [Barker 2000; Mosterman 1999]. An ideal modeling and design framework should allow engineers to express and explore their ideas in their preferred domain-specific ways of thinking, and to integrate small designs systematically to build complex systems.

Multiparadigm modeling and design [Barker 2000; Lee 2000; Liu et al. 2001; Mosterman and Vangheluwe 2000] integrate heterogeneous modeling techniques to achieve scalable designs. Among such techniques, the integration of continuous and discrete dynamics has received attention in system theory [Antsaklis et al. 1995; Henzinger 1996], microelectromechanical devices and systems (MEMS) [Senturia 1998], computer-aided design (CAD) [Alexander et al. 2000; Bakalar and Christen 1999; Tiller 2001], and automatic control [Branicky 1995; Tomlin 1998], among other communities. Two kinds of integration have been widely accepted by modeling and design theories and practices—the *mixed-signal* style that integrates continuous-time differential equations with time-based discrete event models and the *hybrid system* style that integrates differential equations with untimed state machine models. Although it is arguable that these two models can adequately express each other, we respect the fact that they have relative strengths and weaknesses in engineering applications, and complex system designs may need both of them at the same time.

Integrating continuous and discrete models is common, and many tools and engineering design languages have been developed over the past decades. Examples include Simulink with the integration of Stateflow [Harman and Dabney 2001], analog and mixed-signal extensions of hardware description languages such as VHDL-AMS and Verilog-AMS [Bakalar and Christen 1999], object-oriented modeling frameworks such as Omola [Mattsson and Andersson 1993] and Modelica [Elmqvist et al. 1993; Tiller 2001], and the hybrid concurrent constraint language HCC [Gupta et al. 1998]. These approaches typically assume a unified model, such as a continuous-time model with discontinuities, to capture semantically different components, but lack information hiding to help scale up designs. All modeling details have to be exposed to the system assembler to achieve a correct simulation.

Our component-based approach differs from these models in that there is no unified model that covers all possible interactions among components. Component-based approaches decompose complex designs into more manageable pieces and help reuse existing modules. Instead of mapping all components to a grand unified model, we use domain-specific primitive models, such as differential equations and automata, to capture local component interactions, and use hierarchical compositions to build complex designs. Thus the interaction styles among components are well-defined at each level and are constrained to that local scope. Designers can still work in their familiar domain-specific modeling paradigms and the heterogeneity is hidden when composing large systems.

In complex designs, the interaction among components can be significantly diverse for different parts of a system, and this diversity has profound implications on the efficient implementation of simulation engines. For example, the interaction between two continuous-time components is via continuous waveforms, and the simulation engine typically computes the waveform using ODE solving techniques. The interaction between two discrete event components is events, and the simulation engine needs to process events that are transmitted between components in chronological order. A careless aggregation of heterogeneous components not only gives hard-to-understand designs, but makes it difficult to build correct simulation engines. The more scalable hierarchical approaches treat an aggregation of components as an atomic component at a higher level. In particular, hierarchies can help encapsulate component interaction. Correct and efficient simulation engines are easier to build based on homogeneous components in the same scope.

This article presents a hierarchical heterogeneous component metamodel—the *actor model*—and discusses mixed-signal and hybrid system modeling and simulation technologies in actor frameworks. In particular, we address the issues:

—how to model continuous-time, discrete event, and finite-state machine systems in a hierarchical component-based way;
—how to hierarchically compose primitive models to build mixed-signal and hybrid systems;

—how to resolve signal types in a component-based framework and perform signal conversions at the boundaries of continuous and discrete systems;
—how to achieve correct simulations in a component-based framework, including scheduling continuous-time components to solve ODEs with discrete components in the loop, adjusting continuous-time ODE solving step sizes to accurately detect discrete events, and managing the interaction of time progression in continuous and discrete models.

These technologies have been implemented in the Ptolemy II software environment [Davis et al. 2001]. In Ptolemy II, different component interaction styles are characterized as models of computation, and implemented as "domains." Multiple domains can be hierarchically composed to build complex models. Ptolemy II uses hierarchical compositions to hide the implementation details of one component from other components, and keeps the components at the same level of hierarchy interacting in the same way. For example, a mixed-signal model can be built using hierarchical composition of continuous-time and discrete event domains, and hybrid systems can be built using hierarchical composition of continuous-time and finite-state machine domains.

The rest of the article is organized as follows. In Section 2, we give a motivating example to illustrate heterogeneous modeling and to introduce the hierarchical approach. Section 3 describes the actor model that provides the abstraction of components, communication, composition, and models of computation. Section 4 further shows how discrete event, continuous-time, and finite-state machine models can be built in this architecture, and how mixed-signal and hybrid systems can be composed using these primitive models of computation. Section 5 is devoted to the simulation technologies that integrate continuous and discrete executions. Signal conversion mechanisms, time synchronization, and execution scheduling are discussed in detail. Section 6 gives examples to illustrate the modeling and simulation results for some heterogeneous systems.

2. HETEROGENEITY AND HIERARCHY

We motivate hierarchical heterogeneous modeling and design methodologies by an example. Consider an automotive powertrain control (see, e.g., Cho and Hedrick [1989]), depicted in Figure 1. A cylinder of an internal combustion engine has four working phases: intake (I), compress (C), explode (E), and exhaust (H). The engine generates torque through the transmission that drives the car body. Depending on the gear ratio, car body dynamics, the fuel and air supplies, and the spark signal timing, the engine works at different speeds, and makes phase transitions at various time instants. The job of the engine controller is to control the fuel and air supplies as well as the spark signal timing, corresponding to the driver's commands and available sensor information from the engine and the car body.

When designing engine controllers, designers want to quickly validate the control algorithms before considering the implementation details. So, one may start with modeling and simulating the entire system, including the engine and car dynamics, at a high level of abstraction.

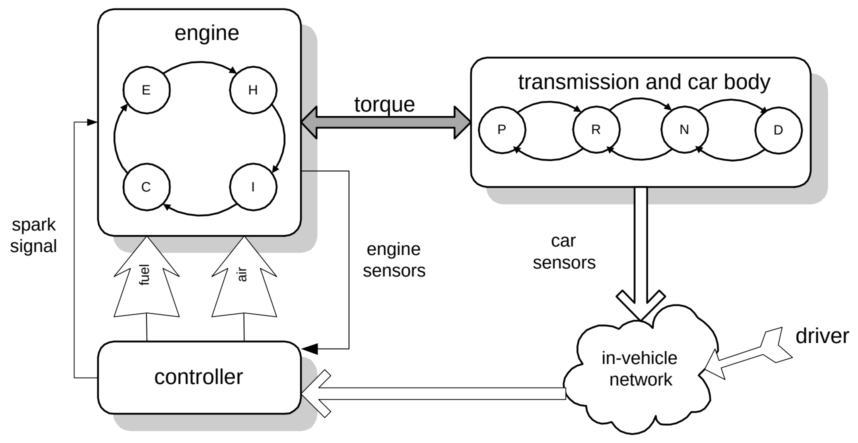

Fig. 1. An example of a power-train control system.

The mechanical properties of the engine and the car body may be naturally modeled using differential equations. But the four phases of the engine, and the different gear ratios in the transmission, may be better modeled as finite-state machines. Although all the mechanical parts interact in a continuous-time style, the embedded controller, most likely implemented by embedded computers with hardware and software, works discretely. In addition, sensor information and driver's commands may come through some communication network. The controller gets this information, computes the control law, drives the air and fuel valves, and produces spark signals, discretely. Discrete event models can be used to model the discrete controller and the communication network. Within the discrete controller, the control algorithms may be implemented as real-time software, and there may be multiple software tasks sharing the same CPU. The real-time scheduling policy among software tasks may have a significant impact on the closed-loop control performance.

In this not so complicated example, we have seen both continuous-time (CT) models and several quite different discrete models: finite-state machines (FSM), discrete events (DE), and priority-driven multitasking (PM). A hierarchical heterogeneous modeling approach decomposes a heterogeneous system such as this into multiple hierarchical components. At each level of hierarchy, it uses a clean model of computation to characterize the interaction of components within that level.

For example, the engine control system can be modeled as in Figure 2. At the top level, a discrete event model may be used to characterize the discrete interaction among the controller, the network, and a discrete abstraction of the car dynamics. Within the controller, a priority-driven multitasking model can be used to model multiple software tasks and their priority-driven execution. Within the discrete abstraction of the car model, there could be a continuous car dynamics, interfaced via event generation and waveform generation components (abstractions of sensors and actuators) so that it can provide a discrete interface to the discrete upper level. The car model may consist of a hybrid engine model and a hybrid transmission model, each of which is a hybrid system

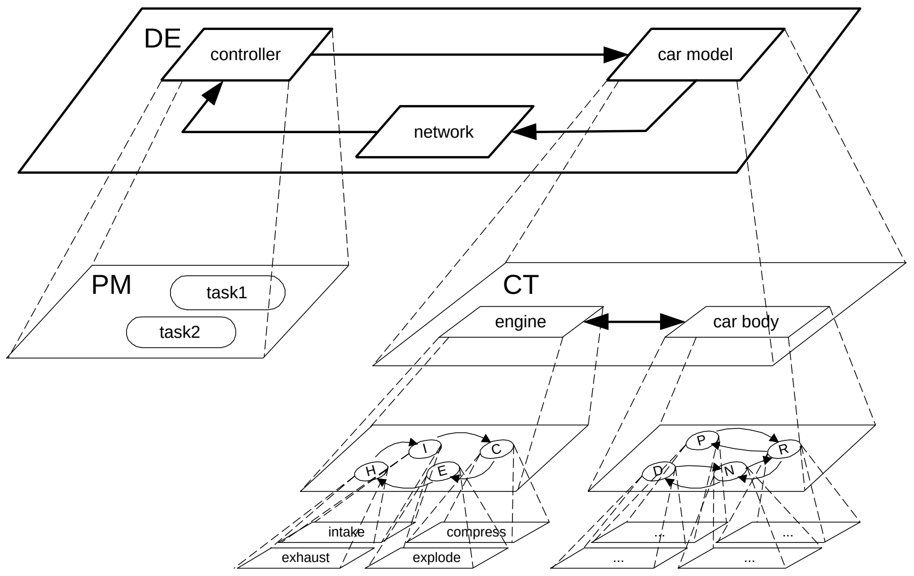

Fig. 2. A hierarchical model for the engine control systems.

consisting of hierarchies of finite-state machines and differential equations. Each individual layer of models may be relatively well understood. But, when integrating these models, the interaction among them requires further study.

3. THE ACTOR METAMODEL

The actor metamodel provides an abstract architecture for components and their composition. Actors encapsulate components; ports represent the communication among components; directors implement models of computation that guard the interaction styles among actors. A more formal and complete discussion of the actor model can be found in Eker et al. [2003] and is beyond the scope of this article. However, we focus on two specific aspects—*modal models* and *signal type systems*—which are essential for modeling mixed-signal and hybrid systems.

3.1 Actors and Ports

In the actor model, the basic building blocks of a system are components called *actors*. Actors encapsulate executions and provide communication interfaces to other actors. Our notion of actors, called *Ptolemy actors* due to its implementation in the Ptolemy project, differs from Agha's actor model [Agha 1986] in the sense that Ptolemy actors do not necessarily associate with a thread of control. An actor can be an *atomic actor*, at the bottom of the hierarchy. An actor can be a *composite actor*, which contains other actors. A composite actor can be contained by another composite actor, so hierarchies can be arbitrarily nested.

Actors have *ports*, which are their communication interfaces. A port is an aggregation of communication channels, which are established when ports are connected. Ports support message passing at an abstract level. Exactly what message passing semantics ports achieve depends on the interaction style among the actors, which is defined by directors. A port can be an input, output, or both. A port of a composite actor can have connections to both the inside and the outside.

3.2 Directors

Actors at the same level of hierarchy are managed by a common *director*. Directors are properties of composite actors. Because of the hierarchy, composite actors may involve two directors. The outside one, managing the composite actor, is called its *executive director*, and the inside one, managing the actors contained by the composite actor, is called its *local director*. A composite actor with a local director is an *opaque composite actor*. In general, the top-level composite actor is always opaque and has no ports. Opaque composite actors are treated as atomic actors by their executive directors, and thus hide the activities inside them from the rest of the system. A composite actor without a local director is a transparent composite actor. The actors contained by a transparent composite actor are managed by the executive director of the composite actor. In a sense, directors can "see through" transparent composite actors.

A director implements a model of computation. More precisely, it defines communication styles between ports, and manages the execution order among actors. By using opaque composite actors, different models of computation can be composed hierarchically.

A director enforces the communication style by providing receivers to input ports and the inside of output ports of opaque composite actors. There is one receiver for each communication channel. A receiver can be a buffer, a queue, a rendezvous point, or a proxy to a global queue depending on the model of computation that the director implements. Hierarchical heterogeneity suggests that receivers provided by the same director be the same. This implies that, for an opaque composite actor, its input ports contains receivers provided by its executive director, and the inside of its output ports contains receivers provided by its local director. There are no receivers inside the ports of transparent composite actors.

For example, in Figure 3, `A0` is a composite actor at the top level of the hierarchy. Actor `A0` directly contains actors `A1`, `A2`, and `A4`. Having director `D2`, actor `A2` is an opaque composite actor, which further contains actor `A3`. Actor `A4` is a transparent composite actor, managed by director `D1`, and thus port `p6` does not have receivers. Actors `A1`, `A3`, and `A5` are atomic actors. In this diagram, ports `p1`, `p4`, and `p5` are output ports, and `p2`, `p3`, `p6`, and `p7` are input ports. Director `D1` manages actors `A1`, `A2`, and `A5`, and ports `p2` and `p7` contain receivers compatible with `D1`. Director `D2` manages actor `A3`, and ports `p3` and the inside of `p5` contain receivers compatible with `D2`. Director `D2` controls the transfer of data from `p2` to `p3`, and director `D1` controls the transfer of data from `p5` to `p7`.

3.3 Modal Models

Actors and ports intrinsically capture concurrent execution and message passing in a system. Sometimes it is useful to explicitly model sequential operation

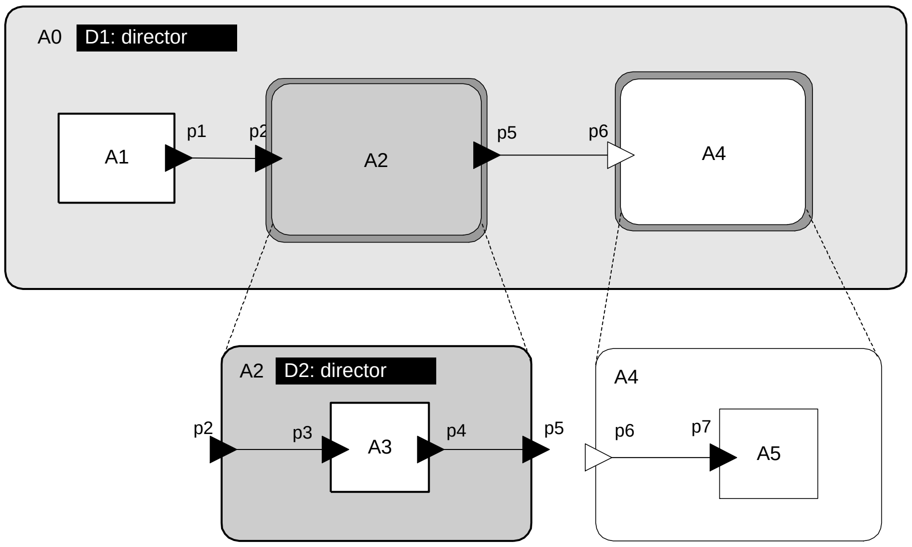

Fig. 3. An illustration of Ptolemy II component architecture.

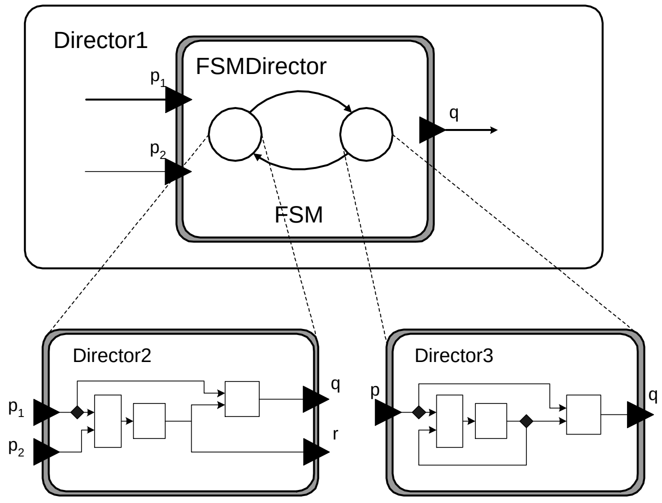

Fig. 4. A modal model uses a finite-state machine to switch between composite actors.

modes in the lifetime of the system and the transitions among these modes. For this, we introduce the notion of a modal metamodel called **-charts* (pronounced star-charts) [Girault et al. 1999] that allows state machines to be composed with many concurrent models in a hierarchical way.

Figure 4 shows an example of a modal model. A composite actor, `FSM`, represents a finite-state machine. Each state of the state machine can associate with a composite actor, which is called the *refinement of the state*. The interpretation of the modal model is that when the state machine is at a particular state, the `FSM` composite actor will be semantically replaced by the refinement of the state. By matching the port names, corresponding inputs to the `FSM` composite actor will be received by the refinement, and corresponding outputs from the refinement will be produced as the output of the FSM. The refinement may have fewer input ports and more output ports than the FSM actor and still define a meaningful model. In the FSM, the transitions among the states can be triggered by both the inputs to the FSM actor and the outputs from the current refinement. A direct use of the *-chart formalism in our context is the formulation of hybrid systems using finite-state machines and continuous-time differential equations, as discussed in Section 4.4.

3.4 Higher-Order Components

Higher-order components (HOCs) are components that manipulate other components. For example, a HOC may construct, change, or destroy other components and connections during the execution of a model. By this definition, the FSM actor in the *-chart formalism is a higher-order component, which, depending on run-time conditions, is replaced by one of its state refinements. The notion of composite actors helps localize these manipulations without affecting the rest of the system.

Some HOCs instantiate other components at the initialization phase of an execution and remain unchanged during the rest of the execution. These components are syntactic shortcuts for constructing complex models from high-level specifications. They can be viewed as model generators. An example is given in Section 4.2 that shows how HOCs may help build complex continuous-time models. Dynamically, automatically, and systematically constructing actors and connections from users' high-level specifications can reduce the burden of model construction and improve the usability of a component-based environment. More sophisticated HOCs can take model manipulation parameters or inputs at run-time and dynamically mutate models. Although the Ptolemy II software environment supports dynamic HOCs, a full discussion of them is beyond the scope of this article.

3.5 Signal Types

In a system that contains both continuous and discrete dynamics, there can be two distinct kinds of signals represented by the connections among the ports: *continuous-time signals* (or *waveforms*) that have meaningful values at all time points and *discrete signals* that are only defined on a discrete subset of the time line. Some components require that specific types of signal be connected to their ports. Among them, *continuous components* only have continuous ports, *discrete components* only have discrete ports, and *hybrid components* can have both continuous and discrete ports. For example, a `WaveformSwitch` actor shown in Figure 5 is a hybrid component that switches between two continuous inputs, $\texttt{input}_a$ and $\texttt{input}_b$ depending on the last event received at the discrete `a/b` port. There are also *domain-polymorphic components* that work with either kind of signal depending on the context to which the component is connected. For example, a `Scale` actor can be used to scale a waveform by a factor, or it can scale all the event values in a set of discrete events.

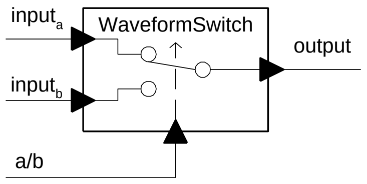

Fig. 5. A WaveformSwitch actor.

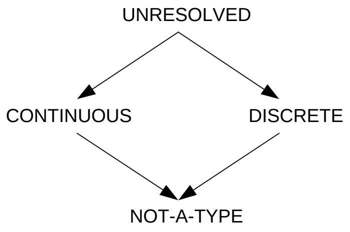

Fig. 6. The signal-type lattice.

For a correct model, all connections among ports must have a definite type of either continuous or discrete. A model is illegal if a port producing continuous signals is connected to a port consuming discrete signals, and vice versa. The type of domain-polymorphic ports must be resolved after the construction of the system depending on its connection context. For the purpose of checking the correctness of mixed-signal specification, we design a signal type system.

The signal type system defines four possible signal types: UNRESOLVED, CONTINUOUS, DISCRETE, and NOT-A-TYPE, forming the lattice in Figure 6, meaning that the type UNRESOLVED can be resolved to either CONTINUOUS or DISCRETE, and the types CONTINUOUS or DISCRETE can be resolved to NOT-A-TYPE. A type that is lower in the lattice is more specific than a type that is higher in the lattice. Some components have fixed signal types at their ports. For example, an integrator has a CONTINUOUS input and a CONTINUOUS output, a periodic sampler has CONTINUOUS inputs and DISCRETE outputs, a zero-order-hold actor has DISCRETE inputs and CONTINUOUS outputs, and many actors only work for DISCRETE inputs and outputs. But for actors that are applicable to both continuous and discrete signals, their signal types are initially UNRESOLVED. The signal type system resolves all the UNRESOLVED types by converting them to either CONTINUOUS or DISCRETE, following these rules.

- If a port p is connected to another port q with a more specific type, then the type of p is resolved to that of the port q. If p is CONTINUOUS but q is DISCRETE, then both of them are resolved to NOT-A-TYPE.
- Unless otherwise specified, the types of the input ports and output ports of an actor are the same.

At the end of the signal-type resolution, if any port is of type UNRESOLVED or NOT-A-TYPE, then the topology of the system is illegal, and the execution is denied.

4. ACTOR MODELS OF CONTINUOUS AND DISCRETE DYNAMICS

The actor metamodel and the notion of models of computation are broad enough to model both continuous and discrete dynamics, and hierarchical composition gives a clean and scalable foundation to build mixed-signal and hybrid systems.

4.1 Modeling Discrete Event Systems

A discrete event (DE) model is a timed model, where time is continuous and global to all components. An event has a value and a timestamp. Actors in this model communicate via a set of events located discretely on the continuous timeline. The execution of actors (via firings) is triggered by events. An actor, when executing, consumes input events and produces output events. These events may further trigger other actors. For source actors, which do not have inputs to trigger them, a self-triggering (also called refiring) mechanism is typically used to register events that trigger the source actor's next execution.

Events in a system are processed in chronological order. This implies that for any actor execution, the output events cannot be earlier in time than the input events that trigger them. This property, called causality, has profound semantics implications on discrete event systems [Lee 1999].

Because of the continuous and global notion of time, and the discrete notion of events, discrete event models are usually used to model systems with discrete actions and timing concerns, such as communication networks, digital circuits, and queueing systems. Many domain-specific modeling tools and languages, such as hardware description languages (VHDL and Verilog) and network simulators [Bajaj et al. 1999], are built using such models.

4.2 Modeling Continuous Dynamics

We focus on continuous-time (CT) models that can be written as a set of ordinary differential equations (ODE) with initial conditions:

$$\dot{x} = f(x, u, t) \tag{1}$$

$$y = g(x, u, t) \tag{2}$$

$$x(t_0) = x_0, \tag{3}$$

where

—$t \in R, t \geq t_0$, a real number, representing continuous time;

—x are the n-dimensional state variables of the system; x_0 is the initial value of state variable at time t_0;

—u are the m-dimensional input variables;

—y are the l-dimensional output variables;

—$\dot{x}$ is the derivative of x with respect to time t;

—$f : R^n \times R^m \times R \to R^n$ gives the derivative of the states;

—$g : R^n \times R^m \times R \to R^l$ gives the output.

Using components, the ODE system (1) to (3) can be modeled by a block diagram as shown in Figure 7. In this model, components communicate via

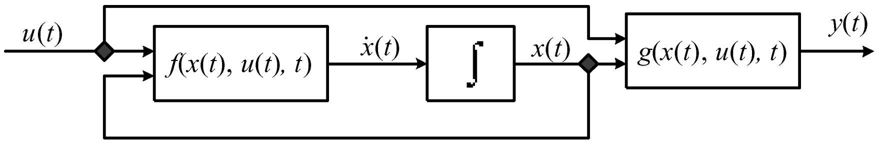

Fig. 7. A conceptual block diagram for continuous-time systems.

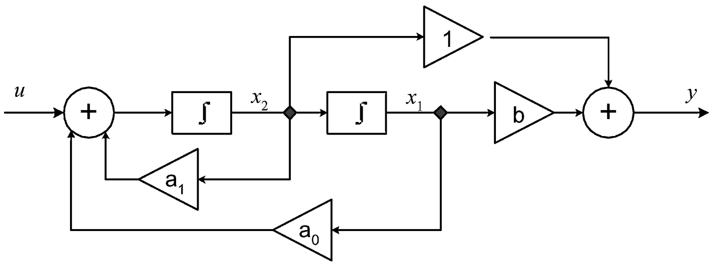

Fig. 8. An implementation for a transfer function in the controllable canonical form.

(piecewise-) continuous waveforms. The components are continuous maps from input waveforms to output waveforms. A special component, the *integrator*, makes a feedback loop an ODE. The functions f and g can consist of a feedforward composition of components that implement piecewise-continuous maps. Higher-order ODEs may involve multiple integrators connected in serial or parallel ways.

4.2.1 *Higher-Order Continuous Components.* Continuous-time models are usually highly structured. For example, linear systems are always built using integrators, scale actors, and adders. And there are canonical realizations in which the actors are connected in similar ways. For these systems, wiring primitive components from scratch is tedious and error prone. For example, a transfer function

$$\frac{Y(s)}{U(s)} = \frac{s+b}{s^2 - a_1 s - a_0} \tag{4}$$

has zero-initial-state controllable canonical form (see, e.g., Callier and Desoer [1991]):

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ a_0 & a_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{5}$$

$$y = \begin{bmatrix} b & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tag{7}$$

which can be built as the composite actor shown in Figure 8.

Some higher-order components can take advantage of the regular structures of CT systems and build continuous-time systems automatically from their higher-level specifications. For example, a transfer function HOC can take a set of parameters such as the coefficients of the numerator and the denominator in

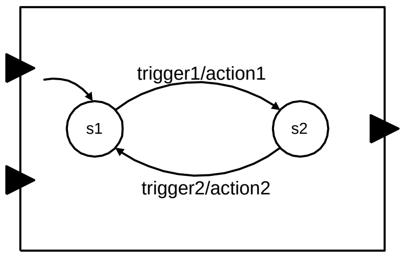

Fig. 9. A finite-state machine actor.

(4) and automatically build the composite actor that contains the model shown in Figure 8 at initialization time. In fact, using generic expression actors, which encapsulate arbitrary mathematical expressions, any pure (linear or nonlinear) CT system can be built using HOCs from their ODE specifications.

4.2.2 *Signal Conversion Components.* Signals in continuous and discrete models are fundamentally different. When combining these models, appropriate signal conversion mechanisms need to be introduced. Many signal conversion algorithms are application-specific, and must be implemented as separate components. Event generators are components that generate discrete events from piecewise-continuous waveforms. A key job for event generators is to find event timestamps. We classify two kinds of events:

—*time events*. These are events whose timestamps are known beforehand. A typical case is the sampling events in a sampled-data system.

—*state events*. The timestamps of this type of event depend on the values of the state variables in the CT system. An example is a zero-crossing event, which is triggered when the input waveform crosses zero.

In general, the timestamps of state events cannot be predicted accurately in advance. Special treatments are required in the ODE solvers.

Waveform generators are components that convert discrete events into piecewise-continuous waveforms. These components typically provide values in the waveform between successive event timestamps. One useful waveform generator is the *zero-order hold*, which is consistent with the common features of D/A converters. In general, any extrapolation of previous event values is a valid waveform generation algorithm.

4.3 Modeling Finite-State Machines

Finite-state machines (FSMs) can be used at two levels. Within an atomic actor, they can be used as primitives to build a discrete event actor, as shown in Figure 9. There is a finite set of states (the bubbles), a finite set of events, an initial state, and transitions from states to states (the arcs). The set of events, received and produced, respectively, by the input and output ports, does not necessarily have a notion of time. A transition is associated with a trigger and some actions. A trigger is a predicate on input events, and an action might produce output events. The interpretation is that the execution starts from the

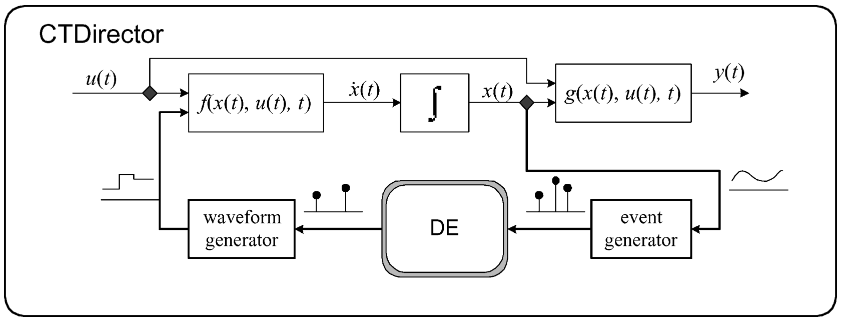

Fig. 10. A discrete event component inside a continuous-time model.

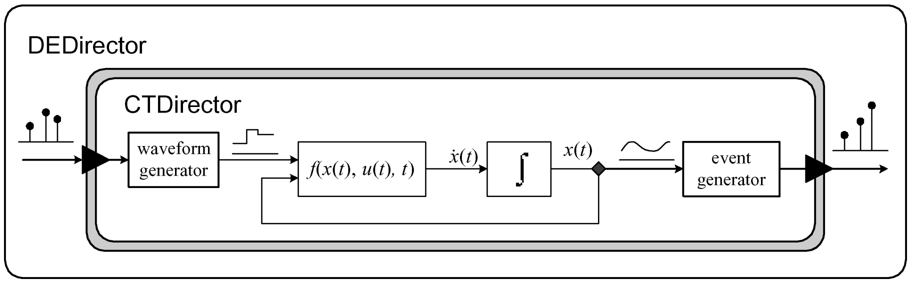

Fig. 11. A continuous subsystem inside a discrete event model.

initial state, and if an outgoing trigger is valid, it makes the corresponding transition to another state and performs the associated actions.

A more powerful use of FSMs is to compose them with concurrent models at a composite actor level to build modal models, as discussed in Section 3.3. The information hiding in the hierarchical composition suggests that even in the *-chart formalism, the semantics of states and transitions at the state machine level be the same as those in primitive state machines.

4.4 Hierarchical Composition

When building hierarchical models with continuous and discrete dynamics, we always put hybrid components, event generators and waveform generators inside the CT model. This significantly simplifies the interfaces between these models.

A mixed-signal system can be built by hierarchically composing CT and DE domains. For example, Figure 10 shows a scenario where a DE model is embedded in a CT system. An event generator produces discrete events that trigger the execution of the DE subsystem. The response, another set of events, is fed through the waveform generator and converted to waveforms. Figure 11 shows a scenario where a continuous model is embedded in a DE system. Event generators and waveform generators are used again at the boundaries of these models.

A hybrid system is shown in Figure 12, where each state is refined into a CT composite actor. Notice that by including the event generation facilities in CT

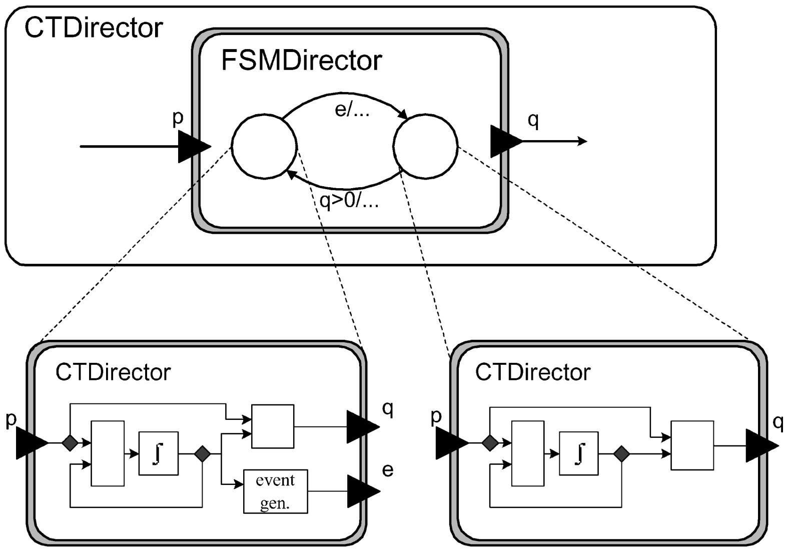

Fig. 12. A hybrid system model.

models, a CT subsystem that refines an FSM state can produce discrete events as their outputs, like the port e in the figure. State machine transition triggers can be built using these events, as well as continuous signals.

5. COMPONENT-BASED SIMULATION TECHNOLOGIES

5.1 Discrete Simulation

The simulation of pure discrete models is fairly straightforward. For discrete event models, the simulation engine typically has a global event queue, which sorts events by their timestamps. When a component generates an output event, the event is placed in the queue. At each iteration of the simulation, the event with the smallest timestamp is taken out of the queue, and its destination component is executed. Additional subtleties exist for handling events with the same timestamp. We take an approach proposed in Chang et al. [1997], which applies topological sorting to actors and yields a deterministic ordering on simultaneous events.

An actor can schedule itself to be executed at a particular future time by placing a pure event (an event without a value) in the event queue, with the component itself as the destination. This mechanism is called *refiring*. When this pure event is taken out of the queue, the component can be executed. Pure events are useful for source components (components without input ports) to produce events at a reasonable rate, and they are also essential for interacting with other timed models.

The execution of state machine models starts from the initial state. The simulator keeps track of the current state. For each input event from the outside model or from the refinement of the current state, the trigger on the outgoing transitions is evaluated. If a trigger evaluates to true, then the transition is taken and the associated action will be performed. The end state of the transition will be the new current state.

5.2 Component-Based Simulation of Mixed-Signal ODEs

5.2.1 *Simulating Pure ODEs.* Simulating pure continuous-time systems requires solving the initial value ODE problems numerically. A widely used class of algorithms, called *time-marching* algorithms, discretizes the continuous timeline into an increasing set of discrete time instants, and numerically computes state variable values at these time instants in increasing order. The discretization of time reflects the trade-off between speed and accuracy of a simulation, and is handled based on the error tolerance of the solutions and the order of the algorithms. To compute the values of state variables at each time instant, the right-hand side (RHS) of the ODE needs to be evaluated with different values. For example, the *trapezoidal-rule* algorithm solves the ODE in (1) using the formula:

$$x(t+h) = x(t) + \frac{h}{2}(f(x(t), u(t), t) + f(x(t+h), u(t+h), t+h)), \tag{8}$$

where t is the last time instant where the solution has been computed, so $x(t)$ and $f(x(t), u(t), t)$ are known, h is the step size; and $t+h$ is the time instant where the new values of x are to be computed. Notice that (8) is an algebraic equation that involves evaluating the RHS of the ODE at $t+h$. Numerically, algebraic equations are typically solved by fixed-point iterations or Newton iterations. For example, in fixed-point iterations, the RHS of the ODE will be evaluated with a new approximation of $x(t+h)$ for each iteration. Theorems about contraction mappings and the continuity of x imply that if the step size is sufficiently small, the fixed-point iteration always converges, and the converged value is the solution for x at time $t+h$ (see, e.g., Lambert [1991]).

Under the component-based approach, the evaluation of the RHS of the ODE can be achieved by firing actors that construct the f function. For example, for the model shown in Figure 7, an evaluation of $f(x(t), u(t), t)$ at time t_1 corresponds to letting the integrator emit its current state $x(t_1)$, firing actor u to produce $u(t_1)$, and firing actor $f(x(t), u(t), t)$, which consumes $x(t_1)$ and $u(t_1)$, and produces $f(x(t_1), u(t_1), t_1)$. The data received by the integrator are $\dot{x}(t_1)$. Obviously, if the RHS of the ODE is built using chains of actors, these actors need to be executed in a data dependency order. By determining what values the integrators emit and what computations the integrators perform on the received token, different numerical ODE solving algorithms can be implemented. After resolving the state of the system, the actors that construct the output map are fired in their topological order to produce the output of the ODE system.

5.2.2 *Handling Discrete Events in ODE Solving.* Traditional ODE solvers assume sufficient smoothness of the RHS functions between integration time points. This may not be the case when the inputs of a CT subsystem are generated from discrete events. Furthermore, the event generation components require that the solvers find the states of the CT system at the time points when an event occurs, which may not coincide with the discretization of time in CT simulation. In order to handle discrete events in continuous-time simulation, we develop the notion of breakpoints.

A *breakpoint* is a time instant in the CT execution when the right-hand side of the ODE is not smooth, or the output map is not continuous. The numerical ODE solvers cannot cross breakpoints in one integration step since either the smooth-RHS assumption is violated or an event needs to be produced.

According to whether a breakpoint is known before an integration step is taken, we classify two kinds of breakpoints: *predictable* ones and *unpredictable* ones. For example, time events and some unsmoothness in input signals are predictable, whereas state events and unsmoothness depending on state variables are unpredictable.

Predictable breakpoints can be stored in a table in their chronological order and handled efficiently. Before each integration step, the breakpoint table is queried and the intended step size (possibly adjusted for error-control concerns) may be reduced so that it does not cross any predictable breakpoints. After taking care of the predictable breakpoints, the smoothness assumption is always taken to proceed for one integration step. Having the new states resolved, the actors that may generate unpredictable breakpoints (which are called *step-size-control actors*) are queried as to whether the integration results are valid. If there is one or more of these actors that reports missing discrete events, then the actors will be asked for a *refinement* on the last integration step size. The integration process will be recalculated with the smallest refined step size. This process is repeated until the discrete event is located within a specified accuracy.

5.2.3 *Hybrid Components in CT: Two-Phase Execution.* A CT system with discrete or hybrid components needs to be scheduled in clusters. The clustering of a mixed-signal ODE into continuous and discrete parts helps build correct and efficient simulation engines, since within each cluster the simulation technologies are relatively well understood. Notice that during the solving of the ODEs, there are many intermediate firings for the continuous parts of the system. During these firings, the discrete actors should not be fired. Instead, the mixed-signal semantics requires that the CT solver always step on the time instants when the discrete events happen, and perform an event-triggered execution on discrete components only at that time.

The signal type system helps partition a mixed-signal CT system into two clusters, a continuous cluster and a discrete cluster. Any actor with a CONTINUOUS port is in the continuous cluster, and any actor with a DISCRETE port is in the discrete cluster. Notice that there may be actors, namely, the hybrid components, in both clusters. Waveform generators, not requiring an input to produce the output, are treated as a source actor in the continuous cluster. Similarly, event generators are treated as sinks. Thus the system in Figure 10 is scheduled into a continuous cluster shown in Figure 13 and a discrete cluster that consists of the DE composite actor together with the event generator and

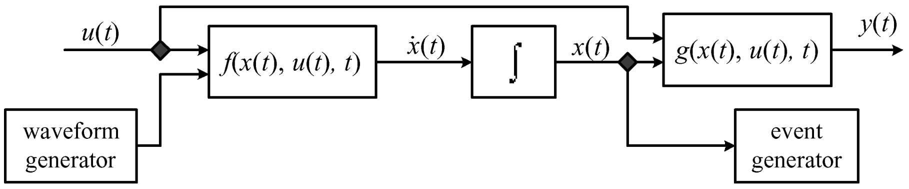

Fig. 13. The continuous cluster of the system in Figure 10.

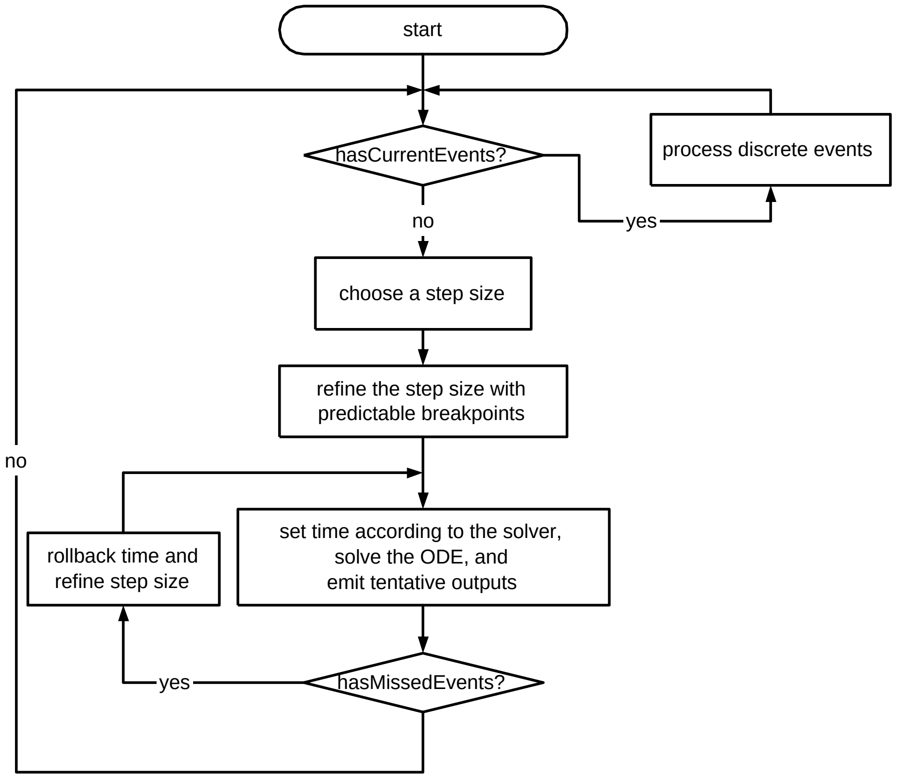
start
hasCurrentEvents?
no
yes
process discrete events
choose a step size
refine the step size with predictable breakpoints
no
rollback time and refine step size
set time according to the solver, solve the ODE, and emit tentative outputs
hasMissedEvents?
yes

Fig. 14. The flow of the two-phase execution of mixed-signal CT systems.

the waveform generator. A conceptual flow of the two-phase execution is shown in Figure 14.

5.3 Continuous Components in a Discrete Event Environment

When a CT component is contained in a DE system, as shown in Figure 11, the CT component is required to be causal, like all other components in the DE system. Consider that the CT component has local time t when it receives an input event with timestamp τ. Since time is continuous in the CT model, it will execute from its local time, and may generate events at any time greater than or equal to t. Thus we need

$$t \geq \tau \tag{9}$$

to ensure the causality of the CT subsystem. This means that the local time of the CT component should be greater than or equal to the global time whenever it executes.

Lookahead executions are not entirely new in coordinating discrete event simulators [Jha and Bagrodia 2000]. In the mixed-signal context, this implies that the CT component should be able to remember its past states and be ready to roll back if the input event time is smaller than its current local time. The state it needs to remember is the state of the component after it has processed the last input event, since no events earlier than that can come. Consequently, the CT component should not emit its detected events to the outside DE system before the global time reaches the event time, since a later input event may change the CT trajectory and have an impact on the detected events. Instead, it should send a pure event to the DE system at the event time, and wait until it is safe to emit it.

Optimizations can prevent unnecessary rollbacks if the discrete event simulator can provide more information. For example, if the DE simulator provides for a CT component the component's next input event time, then the CT component can safely execute to that time instant. This can always be done if the DE system is reduced to a cycle-driven model, where events only occur at predefined time instants. In general cases, this next input time is still predictable if there is no feedback from the output of the CT component to its input. To further reduce the impact introduced by feedback, the CT component can request a refiring at the *current* time after emitting its output. The DE simulator can process this "zero-delay refiring" request after processing all other events at the current time. By that time, the effect of the output from the CT component has taken place, so that we can have a conservative estimation about the next input event time for the CT component. If there is no feedback in the DE system that involves two CT components, then no rollback is necessary.

5.4 Hierarchical Hybrid System Simulation

Simulating hybrid systems, such as the one shown in Figure 12, requires the coordination of two layers of CT models across the FSM model in the middle. Although FSM is an untimed model, its composition with a timed model requires it to transfer the notion of time from its external model to its internal model. The actions associated with transitions can be used to set parameters in the destination state. In particular, setting the initial values of integrators corresponds to a continuous-time state jump.

The execution again proceeds in two phases. During the continuous phase, the system is simulated as a CT system where the FSM is replaced by the continuous component refining its current state. After each time point of CT simulation, the triggers on the transitions starting from the current FSM state are evaluated. If a trigger is enabled, the simulation goes to the discrete phase. The FSM makes the corresponding transition. The continuous dynamics of the destination state may be initialized by the actions on the transition. The simulation continues with the transition time treated as a breakpoint.

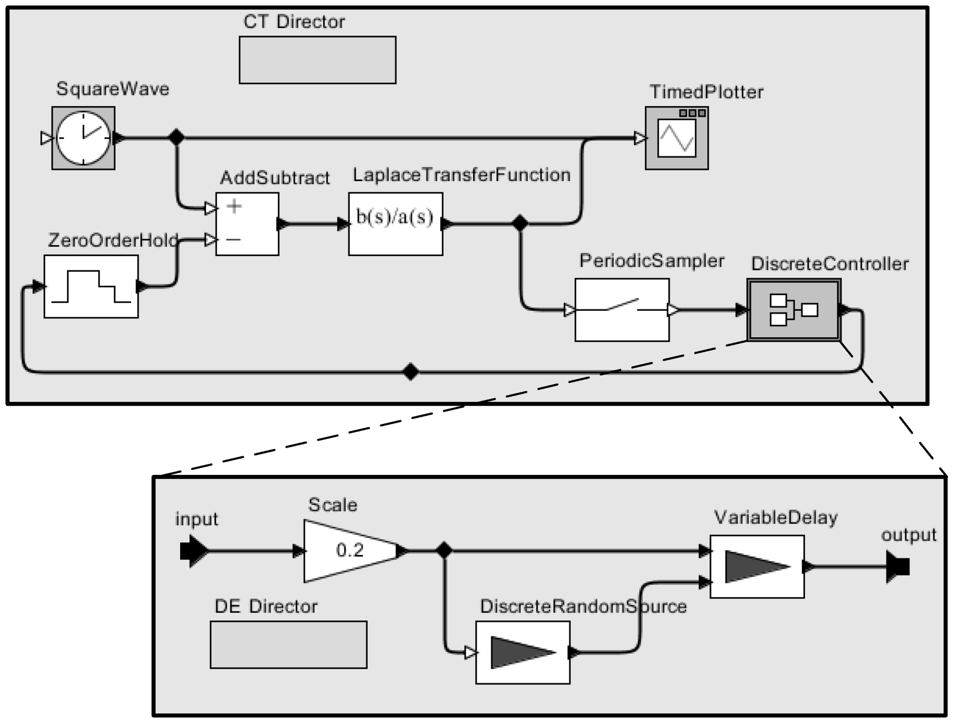

Fig. 15. Modeling a continuous plant with a random delay discrete controller.

6. EXAMPLES

In this section, some examples are given that use the component-based framework for modeling and simulating heterogeneous systems. All the models are built in Ptolemy II v2.0.

6.1 Mixed-Signal Modeling: Controller with Time Delay

This example models a discrete controller that controls a continuous plant, as shown in Figure 15. The control algorithm is simply a proportional feedback. We assume that the controller is implemented in software so it introduces a time delay from receiving the input sample to producing the control value. Depending on other software tasks that may be running, this delay varies for each sample input.

The model has two levels of hierarchy, a CT top-level containing a DE composite actor. A transfer function actor is used to model the differential equations. The output of that actor is periodically sampled and fed into the discrete controller. Inside the discrete controller, the control law is applied. The input event also triggers a random number generator and a variable delay actor, which delays its input events by the amount of time indicated by the value of the lower input.We model the delay as a random number that takes two values 0.05 and 0.1 with equal probability. One execution trace is shown in Figure 16.

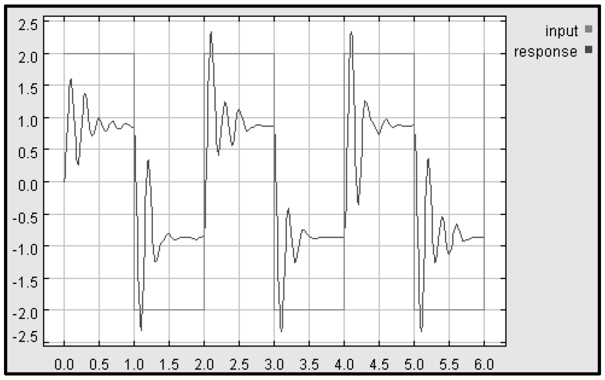

Fig. 16. One execution trace of the discrete control system with random delay.

The random delays introduced by the controller make the settling time vary from time to time.

6.2 Mixed-Signal Modeling: Σ/Δ Modulated A/D Conversion

Sigma-delta (Σ/Δ) *modulation* [Candy 1974], also called *pulse density modulation*, is an oversampling A/D conversion technology. The analog input is oversampled N times faster than the requested digital output frequency, and quantized to one bit, ± 1. The quantized value is fed back to the analog part, as well as accumulated by a digital accumulator. For every N samples, the converter produces the digital output and resets the accumulator. Due to its robustness, Σ/Δ modulated A/D convertors have been extensively developed. Recently, this technique has been applied to microelectromechanical accelerometers to reduce noise and improve stability and sensing ranges [Lemkin 1997].

Figure 17 shows a model for a Σ/Δ modulated microaccelerometer. A CT composite actor, `CTSubsystem`, is used to model the mechanical dynamics of the accelerometer, which is built by silicon beams and anchors. A second-order ODE is used as the simplified dynamics. The sensing signal is sampled by the `PeriodicSampler`, filtered by a lead compensator (`FIR Filter`), and fed to an one-bit `Quantizer`. A delay actor is used to model the time delay introduced by filtering and quantization. In addition to feeding back to the analog part, the outputs of the quantizer are filtered again by the `MovingAverage` actor and accumulated. A `DigitalClock`, which produces a trigger every N sampling period, triggers the accumulator to produce the digital output, as well as to reset the accumulator.

Figure 18 shows an execution result of the model for a sine wave input. The upper plot in the figure shows the discrete signals. The dense events, with values ± 1, are the quantization result. The sparse events are produced by the accumulator, that is, the digital outputs, and as expected, they have a sinsoidal shape. The lower plot shows the continuous signals, where the low-frequency sine wave is the system input signal, the high-frequency waveform is the analog sensing signal, and the square wave is the zero-order hold of the feedback from the digital part.

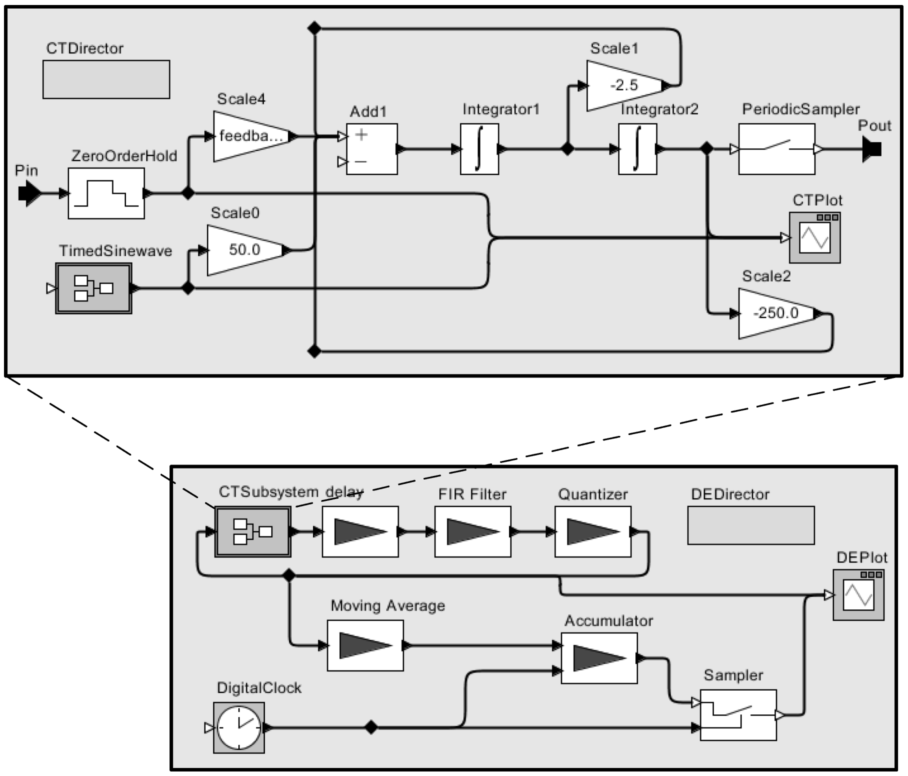

Fig. 17. A mixed-signal model for S/D modulated micro-accelerometer.

6.3 Hybrid System Modeling: Sticky Point Masses

This example shows a simple hybrid system. As shown in Figure 19, there are two point masses on a frictionless surface with two springs attaching them to fixed walls. Given initial positions other than the equilibrium points, the point masses oscillate. The distance between the two walls is small enough that the two point masses may collide. The point masses are sticky, so that when they collide, they will stick together and become one point mass with two springs attached to it. Assume the stickiness decays exponentially after the collision, such that eventually the pulling force between the two springs is big enough to pull the point masses apart. This gives the two point masses a new set of initial positions and speeds, and they oscillate freely until they collide again.

The system model, as shown in Figure 20, has three levels of hierarchy: CT, FSM, and CT. The top level is a continuous-time model with two actors, a composite actor, `Sticky Mass Model`, which outputs the position of the two point masses, and a plotter that simply plots the trajectories. The composite actor is a finite-state machine with two modes, `separate` and `together`.

In the `separate` state, there are two ODEs modeling two independently oscillating point masses. There is an event detection mechanism, implemented by subtracting one position from the other and detecting the zero crossing. If the positions are equal, meaning that the two point masses collide, then a

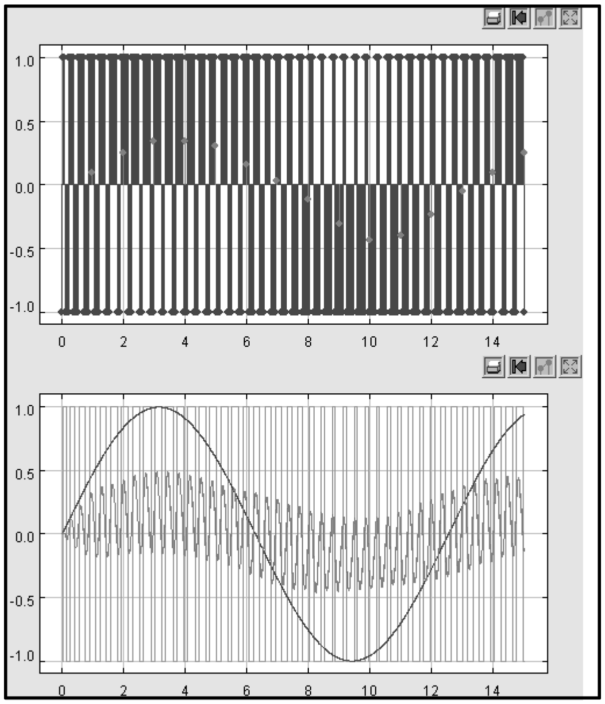

Fig. 18. A simulation result for the micro-accelerometer.

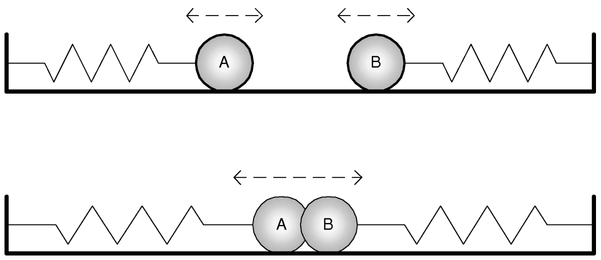

Fig. 19. A sticky point mass system.

collision event, touched, is produced. This event will trigger a FSM transition from the separate state to the together state. The actions on the transition set the velocity of the stuck point mass based on the law of conservation of momentum.

In the together state, there is one differential equation modeling the stuck point masses, and another first-order differential equation modeling the exponentially decaying stickiness. An expression computes the pulling force between the two springs. The trigger on the transition from the together state to the separated state compares the pulling force to the stickiness. If the pulling force is bigger than the stickiness, then the transition is taken. The positions and velocities of the two separated point masses are equal to those before the separation. The simulation result is shown in Figure 21.

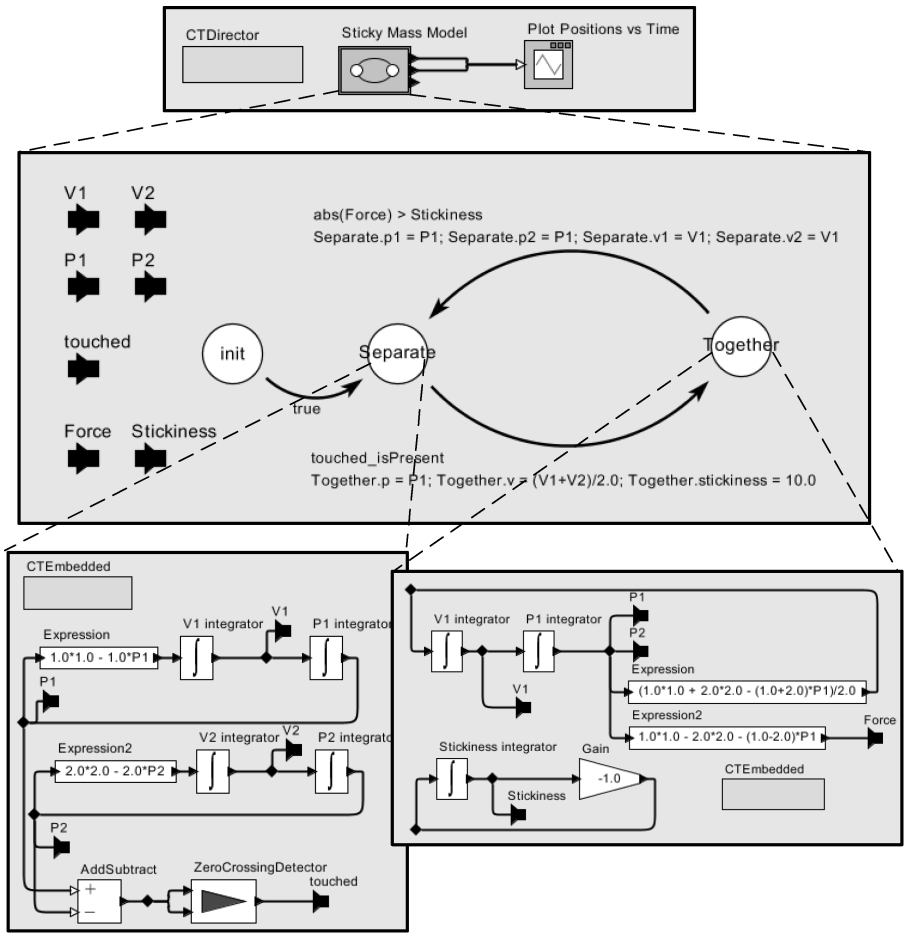

Fig. 20. Modeling the sticky point mass system.

7. CONCLUSION

This article presents a component-based framework for modeling and simulating systems with continuous and discrete dynamics. The framework systematically and modularly integrates different models using an actor metamodel and hierarchical composition of heterogeneous models of computation. Both modeling and simulation technologies for mixing continuous-time differential equations, timed discrete event models, and finite-state machines in this framework are studied. Signal conversions, signal type systems, and breakpoints are used to handle mixed-signal components in a continuous-time framework. The execution coordination among continuous and discrete models allows us to achieve correct and efficient simulations. The technologies are implemented in the Ptolemy II software environment. Examples are given to show the modeling and simulation capabilities of Ptolemy II for mixed-signal and hybrid systems.

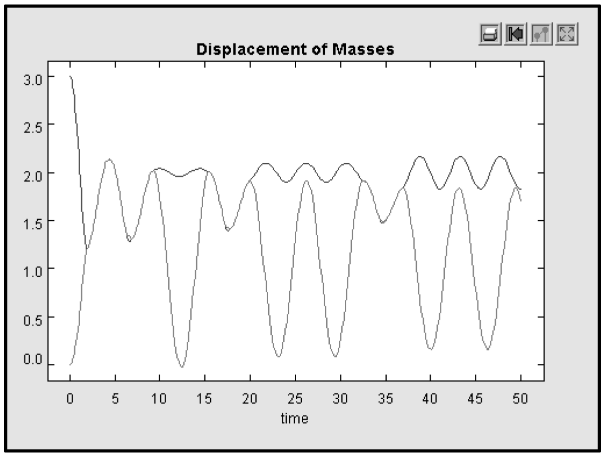

Fig. 21. An execution result of the sticky point mass system.

ACKNOWLEDGMENTS

The authors would like to thank the entire Ptolemy team, especially Xiaojun Liu, who created the Ptolemy II FSM domain, Lukito Muliadi, who created the Ptolemy II DE domain, and Johan Eker, who gave valuable comments on an early version of this article.

REFERENCES

AGHA, G. A. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass.

ALEXANDER, P., KAMATH, R., AND BARTON, D. 2000. System specification in Rosetta. In *Proceedings of IEEE Engineering of Computer-Based Systems Symposium*. IEEE (Edinburgh, UK), 299–307.

ANTSAKLIS, P., KOHN, W., NERODE, A., AND SASTRY, S. 1995. *Hybrid Systems II*. In Lecture Notes in Computer Science, vol. 999, Springer-Verlag, Heidelberg.

BAJAJ, S., BRESLAU, L., ESTRIN, D., FALL, K., FLOYD, S., HALDAR, P., HANDLEY, M., HELMY, A., HEIDEMANN, J., HUANG, P., KUMAR, S., MCCANNE, S., REJAIE, R., SHARMA, P., VARADHAN, K., XU, Y., YU, H., AND ZAPPALA, D. 1999. Improving simulation for network research. Tech. Rep. 99-702b, University of Southern California. March.

BAKALAR, K. AND CHRISTEN, E. 1999. VHDL 1076.1: Analog and mixed signal extensions for VHDL. Tech. Rep., IEEE.

BARKER, D. 2000. Requirements modeling technology: A vision for better, faster, and cheaper systems. In *Proceedings of the VHDL International Users Forum Fall Workshop (VIUF'00)*. IEEE (Orlando, FL), 3–6.

BRANICKY, M. S. 1995. Studies in hybrid systems: Modeling, analysis, and control. Ph.D. thesis, Laboratory of Information and Decision, MIT, Cambridge, Mass.

CALLIER, F. M. AND DESOER, C. A. 1991. *Linear System Theory*. Springer-Verlag, Germany.

CANDY, J. C. 1974. A use of limit cycle oscillations to obtain robust analog-to-digital converters. *IEEE Trans. Commun. COM-2,* 3, 298–305.

CHANG, W.-T., HA, S., AND LEE, E. A. 1997. Heterogeneous simulation—mixing discrete-event models with dataflow. *J. VLSI Signal Process. 15,* 1 (Jan.), 127–144.

CHO, D. AND HEDRICK, J. 1989. Automotive powertrain modeling for control. *ASME J. Dynamic Syst. Meas. Control 111,* 4 (Dec.), 568–576.

DAVIS, J., HYLANDS, C., KIENHUIS, B., LEE, E., LIU, J., LIU, X., MULIADI, L., NEUENDORFFER, S., TSAY, J., VOGEL, B., AND XIONG, Y. 2001. Ptolemy II: Heterogeneous concurrent modeling and design in Java. Tech. Rep. UCB/ERL M01/12, EECS, University of California, Berkeley.

EKER, J., JANNECK, J., LEE, E., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity, the Ptolemy approach. *Proceedings IEEE 91,* 1 (Jan.).

ELMQVIST, H., CELLIER, F., AND OTTER, M. 1993. Object-oriented modeling of hybrid systems. In *Proceedings of ESS'93, SCS European Simulation Symposium* (Delft, The Netherlands), xxxi–xli.

GIRAULT, A., LEE, B., AND LEE, E. A. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. Comput. Aid. Des. Integ. Circ. Syst. 18,* 6 (June), 742–760.

GUPTA, V., JAGADEESAN, R., AND SARASWAT, V. 1998. Computing with continuous change. *Sci. Comput. Program. 30,* 1-2 (Jan.), 3–49.

HARMAN, T. AND DABNEY, J. 2001. *Mastering Simulink 4*. Prentice Hall, Englewood Cliffs, N.J.

HENZINGER, T. 1996. The theory of hybrid automata. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, IEEE Computer Society Press, New Brunswick, N.J., 278–292.

JHA, V. AND BAGRODIA, R. 2000. Simultaneous events and lookahead in simulation protocols. *ACM Trans. Model. Comput. Simul. 10,* 3 (July), 241–267.

LAMBERT, J. 1991. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problems*. Wiley, New York.

LEE, E. A. 1999. Modeling concurrent real-time processes using discrete events. *Ann. Softw. Eng. 7,* 1-4, 25–45.

LEE, E. A. 2000. What's ahead for embedded software? *IEEE Computer 33,* 9 (Sept.), 18–26.

LEMKIN, M. A. 1997. Micro accelerometer design with digital feedback control. Ph.D. thesis, University of California, Berkeley.

LIU, J., JEFFERSON, S., AND LEE, E. 2001. Motivating hierarchical run-time models in measurement and control systems. In *Proceedings of the 2001 American Control Conference (ACC01)* (Arlington, VA), 3457–3462.

MATTSSON, S. AND ANDERSSON, M. 1993. Omola—an object-oriented modeling language. In *Recent Advances in Computer-Aided Control Systems Engineering,* M. Jamshidi and C. Herget, Eds., North-Holland, Amsterdam, 291–310.

MOSTERMAN, P. 1999. An overview of hybrid simulation phenomena and their support by simulation packages. In *Hybrid Systems: Computation and Control (HSCC99)*, Lecture Notes in Computer Science, vol. 1569, F. W. Vaandrager and J. H. van Schuppen, Eds. Springer-Verlag, New York, 165–177.

MOSTERMAN, P. AND VANGHELUWE, H. 2000. Computer automated multi-paradigm modeling in control system design. In *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control Systems Design* (Anchorage, AK), 65–70.

SENTURIA, S. 1998. CAD challenges for microsensors, microactuators, and microsystems. *Proceedings IEEE 86,* 8 (Aug.), 1611–1626.

TILLER, M. 2001. *Introduction to Physical Modeling with Modelica*. Kluwer Academic, Reading, Mass.

TOMLIN, C. 1998. Hybrid control of air traffic management systems. Ph.D. thesis, University of California, Berkeley.

Received August 2001; revised July 2002; accepted December 2002