Modeling and Simulating Cyber-Physical Systems using CyPhySim*

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

Mehrdad Niknami
EECS Department
UC Berkeley
Berkeley, CA, USA
mniknami@eecs.berkeley.edu

Thierry S. Nouidui
Lawrence Berkeley
National Laboratory
Berkeley, CA, USA
tsnouidui@lbl.gov

Michael Wetter
Lawrence Berkeley
National Laboratory
Berkeley, CA, USA
mwetter@lbl.gov

ABSTRACT

This paper describes an open-source simulator for cyber-physical systems called CyPhySim that is based on Ptolemy II. This simulator supports classical (Runge-Kutta) and quantized-state simulation of ordinary differential equations, modal models (hybrid systems), discrete-event models, the Functional Mockup Interface (FMI) for model-exchange and co-simulation, discrete-time (periodic) systems, and algebraic loop solvers. CyPhySim provides a graphical editor, an XML file syntax for models, and an open API for programmatic construction of models. It includes an innovation called "smooth tokens," which allow for a blend of numerical and symbolic computation, and for certain kinds of system models, dramatically reducing the computation required for simulation.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: General

1. INTRODUCTION

The heterogeneity of cyber-physical systems presents considerable challenges to software simulation techniques. CyPhySim is a new open-source software simulator (BSD license) that supports the most promising combination of technologies. The core of the system is a discrete-event simulation engine, continuous-time solvers, and state machine modeling infrastrcture from the open source Ptolemy II system [23].

The paper begins in Section 2 with an analysis of the computational aspects of modeling continuous dynamics using numerical solvers. In Section 3, we review models of time that are well suited to modeling cyber-physical systems. In Section 4, we explain how to use modal models to construct hybrid systems. In Section 5, we switch to discrete-event models, and introduce the notion of smooth tokens, which permit modeling continuous dynamics using discrete events. Section 6 leverages this discrete-event approach to show that quantized-state solvers can be very effective for certain kinds of continuous dynamics and particularly for cyber-physical dynamics where digital controllers affect an analog physical plant. Section 7 shows how algebraic loop solvers can be cleanly mixed with the other types of solvers described here. Section 8 concludes with a summary of other issues not covered and open issues.

*This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231, and by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, #1446619 (Mathematical Theory of CPS) and the following companies: Denso, National Instruments, and Toyota.

2. CONTINUOUS DYNAMICS

To model the physical side of cyber-physical systems, we need to represent **dynamics**, the evolution of the state of a system in time. Continuous-time dynamics is typically modeled using **ordinary differential equations** (**ODE**s) or **differential algebraic equations** (**DAE**s).

An example of a nontrivial nonlinear continuous dynamics given by a system of ODEs is the well-known Lorenz attractor, a chaotic system defined by the following equations,

$$\begin{aligned} \dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)), \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t), \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t), \end{aligned}$$

with given initial conditions $x(t_0) = x_0$ and $t \in [t_0, \infty)$. A Ptolemy II implementation of this system is shown in Figure 1, in this case constructed graphically using the Vergil graphical editor for Ptolemy II models. A plot produced by a simulation is shown in Figure 2. Note that this is a chaotic system, so arbitrarily small perturbations can yield arbitrarily large consequences.

In this paper, we are most interested in how to model such continuous dynamics *in combination with* discrete phenomena such as digital controls, abrupt mode changes, and sampling. Moreover, we are interested in the *computational* aspects of the model. Even though the model refers to continuous dynamics, where both time and values exist in a continuum, the model is a **computational artifact**. It is executable on a computer, and hence has an intrinsically

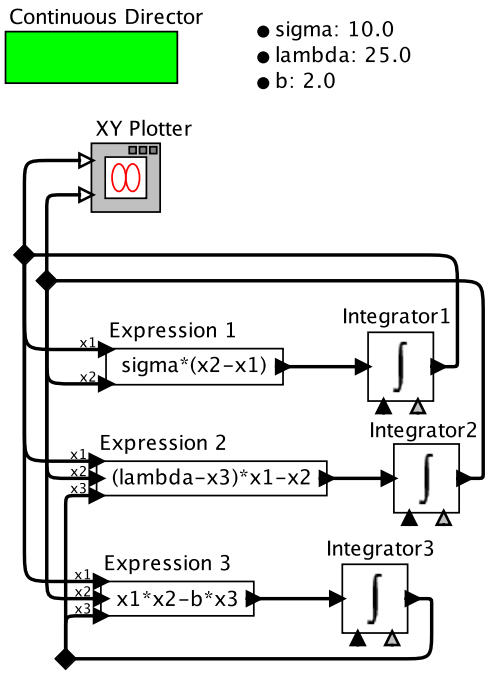

Figure 1: Graphical model of the Lorenz attractor.

discrete semantics. Let us begin with a careful examination of the most primitive element in such models, the integrator.

2.1 Integrators

The Integrator actors in Figure 1 have a *derivative* input and a *state* output, as shown in Figure 3. At time t, the *state* output is

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau) d\tau,$$

where x_0 is the *initialState*, a **parameter** of the actor, t_0 is the start time of the model execution, and $\dot{x}$ is the *derivative* input. Note that this can be written using a derivative

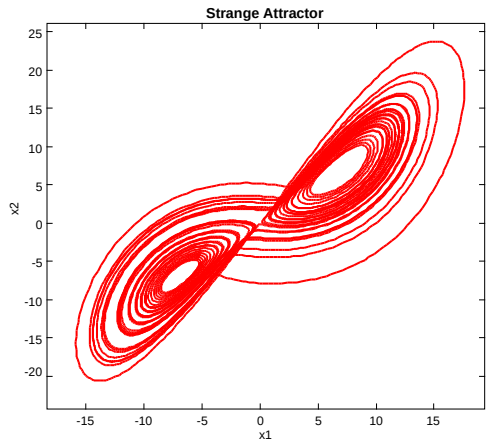

Figure 2: Plot of signals for the Lorenz attractor.

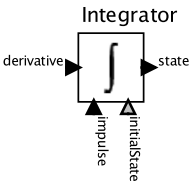

Figure 3: Integrator in Ptolemy II.

instead of an integral as

$$\dot{x}(t) = \frac{d}{dt} x(t),$$

where $\dot{x}$ is the input signal and x is the output. (The relationship between these representations is the **fundamental theorem of calculus**.)

Note that $x(t)$ does not depend on the value of $\dot{x}$ at time t. This is a key **causality** property of the actor, and it ensures that the feedback loops of Figure 1 are **constructive** [11]. At each time t, there is no cyclic dependency in a feedback loop. The values of the signals at time t can be determined without knowing the inputs to the integrators at time t. Once these are known, the inputs to the integrators at time t become known. But these inputs will only affect future outputs of the integrators. This is an example of a *computational* property of these models.

Computers perform discrete calculations. How can they integrate a continuous-time signal? Numeric integration techniques include **implicit methods** (e.g. backward Euler, which computes $x(t_{k+1}) = x(t_k) + \dot{x}(t_{k+1})\,(t_{k+1} - t_k)$) and **explicit methods** (e.g. forward Euler, which computes $x(t_{k+1}) = x(t_k) + \dot{x}(t_k)\,(t_{k+1} - t_k)$). Implicit methods sacrifice the causality property, requiring that at time t, the input $\dot{x}(t)$ be known in order to determine the output $x(t)$. Implicit methods, therefore, change the causality properties of fundamental calculus, and create **causality loops** when used in feedback systems. Causality loops can also arise when models are given by DAEs rather than ODEs. We will say more about this below, but for now, we assume that only explicit solvers will be used.

An **ordinary differential equation** (**ODE**) can be written as

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

where x is the **state variable** (or vector), u is the **input**, t is time, and f is a function giving the derivative in terms of the state, the input, and the time. If x and u are scalars, f has the form $f\colon \mathbb{R}^3 \to \mathbb{R}$. Given an **initial condition** $x_0 = x(t_0)$, this is equivalent to

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau) d\tau = \int_{t_0}^{t} f(x(\tau), u(\tau), \tau) d\tau.$$

The general structure of such a system is shown in Figure 4.

2.2 Classical ODE Solvers

Under certain circumstances, the value of a continuous signal at a future time $t + h$ can be given by a **Taylor series**, which expresses that value in terms of the value and derivatives at the current time t,

$$x(t + h) = x(t) + h\dot{x}(t) + \frac{h^2}{2!}\ddot{x}(t) + \cdots$$

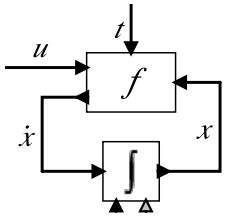

Figure 4: Pattern of ODE problems.

The function f in (1) is said to be **analytic** at t if all the derivatives exist (the function is **smooth**), and this sum converges to x in some neighborhood of t. I.e., there exists a $\delta > 0$ s.t. the Taylor series converges to $x(t+h)$ for all h in $(-\delta, \delta)$.

Various classical ODE solvers are based on truncating and approximating this series. If the series converges, the terms become small quickly, enabling accurate approximations with a finite summations, and also enabling estimating the local approximation error (i.e., for the current time step) by examining the terms just beyond the truncation. Such local error estimates can be used to design **variable step** solvers, which adjust the time offset h to keep the error suitably small.

CyPhySim provides by default classical Runge-Kutta (RK) solvers, which are widely used variable-step-size explicit solvers [6]. RK solvers improve the accuracy in part by evaluating the function f at multiple points in time between t and $t+h$. Upon completing the calculation, these solvers estimate the error, and if it is too high, reject the step size h and redo the step with a smaller step size. In an executable model, of course, the function f is realized by software. RK solvers require being able to *speculatively* evaluate f at various times in the model without committing to a step size. Hence, the software implementation of the function f must either be free of side effects (evaluating it does not change any state), or the software must support rolling back to a previously stored state in order to redo the step. Such rollback is also generally required when the model must respond to unpredictable events such as a zero crossing of a continuous signal. This is another example of a computational property of models.

In Simulink and Ptolemy II, actors are realized using two distinct procedures, one that calculates outputs based on inputs but has no side effects (called `mdl_output` in Simulink and `fire` in Ptolemy), and another that updates the state of the actor based on the inputs (called `mdl_update` in Simulink and `postfire` in Ptolemy). FMI for model exchange [19] provides a function `fmi2CompletedIntegratorStep` that can serve the same role as `mdl_update` and `postfire`.

In principle, the external input u of Figure 4 can be absorbed into the definition of the function f, in which case the problem reduces to an **initial value problem**, which is to find a function x satisfying

$$\dot{x}(t) = f'(x(t), t), \quad x(t_0) = x_0$$

over some interval $t \in [t_0, t_0 + \epsilon]$, where $f'(x(t), t) = f(x(t), u(t), t)$. If f is Lipschitz continuous in its first argument and continuous in the second, then the **Picard-Lindelöf theorem** states that there exists an $\epsilon > 0$ such that there is a unique solution to this problem. This unique solution provides a semantic benchmark against which any numerical solver can be evaluated for accuracy. This semantic benchmark is called the **ideal solver** semantics in [15]. This is the third example of computational property. Just as it is useful to have a theory of real numbers against which to evaluate the accuracy of floating-point numbers, it is useful to have an ideal solver semantics against which to evaluate the accuracy of numerical solvers.

2.3 Differential Algebraic Equations

A more general form of differential equations imposes algebraic constraints as

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), y(t), t), \\ 0 &= g(x(t), y(t), t), \end{aligned}$$

where $x(t) \in \mathbb{R}^n$, $u(t) \in \mathbb{R}^m$, and $y(t) \in \mathbb{R}^k$ for some $n, m, k \geq 1$. These are supported by **Bond graphs** [22], **Modelica** [25], and **SPICE** [20]. The two equations have cycles, in that x depends on y and vice versa, but these cycles do not automatically lead to causality loops. In particular, given a system of DAEs, a Modelica compiler will attempt to remove apparent causality loops through a process known as **index-order reduction**, using for example the Pantelides algorithm [21]. When this process is successful, the model becomes an ODE. CyPhySim currently requires ODEs for modeling continuous dynamics. The conversion from DAEs to ODEs is not automatic, as it is in Modelica, but we believe that open source Modelica compilers such as JModelica can be leveraged sometime in the future.

3. MODEL OF TIME

For models that mix discrete and continuous behaviors, it is well established that the model of time that is used must support sequences of causally-related instantaneous events [11, 3]. CyPhySim uses a **superdense time** model [16, 13]. In this model, a **time stamp** is a pair (t, n), where **model time** t is a digital approximation to a real number, and the **microstep** (or **index**) $n \in \mathbb{N}$ is a non-negative integer. Two time stamps (t_1, n_1) and (t_2, n_2) are **weakly simultaneous** it $t_1 = t_2$ and **strongly simultaneous** if $t_1 = t_2$ and $n_1 = n_2$.

For the real number t, CyPhySim uses a representation that has a fixed precision valid throughout a model (the precision is a parameter of the model). Unlike floating point numbers, addition of times is associative, and the precision does not vary with the magnitude of the number. For details, see [23].

One key benefit of the superdense model of time is that it enables better semantics for discontinuous signals. One mechanism that is widely used to model discontinuities in otherwise continuous dynamics relies on **generalized functions** like the **Dirac delta function**, a function $\delta \colon \mathbb{R} \to \mathbb{R}^+$ given by

$$\forall\, t \in \mathbb{R},\ t \neq 0, \quad \delta(t) = 0, \quad \text{and}$$

$$\int_{-\infty}^{\infty} \delta(\tau) d\tau = 1.$$

That is, the signal value is zero everywhere except at $t = 0$, but its integral is unity. Suppose a signal w has a Dirac delta function with weight K occurring at t_1 as follows,

$$w(t) = w_1(t) + K\delta(t - t_1), \tag{2}$$

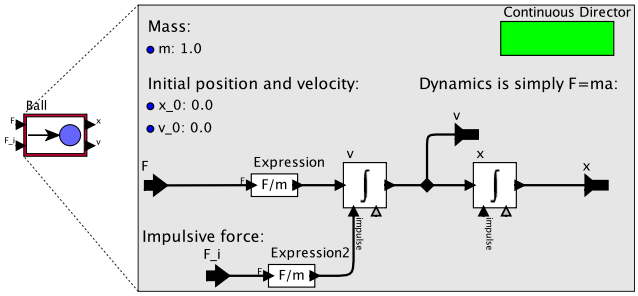

Figure 5: Newtonian model of a mass with impulsive forces.

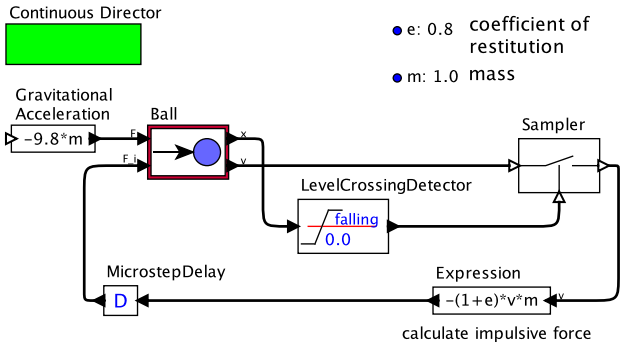

Figure 6: Model of a bouncing ball.

where w_1 is an ordinary continuous-time signal. Then

$$\int_{-\infty}^{t} w(\tau)d\tau = \begin{cases} \int_{-\infty}^{t} w_1(\tau)d\tau & t < t_1 \\ K + \int_{-\infty}^{t} w_1(\tau)d\tau & t \geq t_1 \end{cases}$$

The Dirac delta causes an instantaneous increment in the integral by K at time $t = t_1$. For an integrator, at time t, the *state* output is

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau)d\tau.$$

Suppose that $\dot{x} = w$, given in (2). The integrator, Figure 3, has an *impulse* input to provide the Dirac impulse at time t_1. At time t_1, the effect of the impulse is to add immediately to $x(t)$. The output at time t depends on the *impulse* input at time t, but not on the *derivative* input. This explains why the CyPhySim Integrator provides two distinct input ports, *impulse* and *derivative*. It is imperative that the impulsive input be segregated from the *derivative* input because the causality properties of these inputs are different.

Consider for example modeling a physical object, such as a billiard ball, subject to Newtonian mechanics, $F = ma$, but also subject to collisions. Collisions can be (and should be, for most purposes [11]) modeled using impulsive forces, specifically the Dirac delta function. A model constructed in CyPhySim of such a physical object is shown in Figure 5. The output v depends immediately on the input F_i, if it is present. Subtle issues that arise when modeling discrete phenomena such as collisions this way are considered in [11].

To illustrate the use of such impulsive forces, consider the classic **bouncing ball** problem, which models a ball in free fall (using Figure 5), a **level-crossing detector**, and

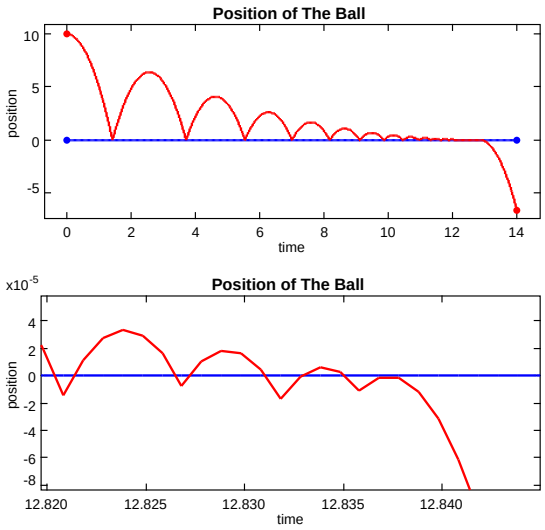

Figure 7: Bouncing ball position vs. time.

an impulsive force to reverse the direction of the ball upon collision with a surface. Such a model is shown in Figure 6. Here, a LevelCrossingDetector actor detects a falling zero crossing, which then triggers a calculation of an impulsive force that is fed back to the impulsive force input of the ball model. To prevent a causality loop, a MicrostepDelay actor is inserted into the feedback loop. This is valid because no (real) time elapses between the detection of the zero crossing and application of the impulsive force.

As we have already mentioned, detecting level crossings of continuous signals is semantically problematic. Computational models cannot directly deal with either time or space continuums. Level-crossing detection ultimately depends on numerical approximations to real numbers. It can only be done up to some precision. In this particular implementation, a level-crossing is asserted when the signal value matches or passes the level, and if it passes the level, it is still within a specified tolerance of the level. Figure 7 shows that this policy inevitably results in the ball traversing the surface in such a way that when the bounce occurs, the impulsive force is insufficient to cause the ball to rise above the surface again. This results in the ball tunneling through the surface.

One could, of course, solve this problem with much more detailed modeling of the physics. Collisions between rigid objects, for example, involve localized plastic deformation, viscous damping in the material, and acoustic wave propagation. Much experimental and theoretical work has been done to refine models of such phenomena, leading to considerable insight into the underlying physical phenomena. Such detailed modeling rarely helps in developing insight about macroscopic system behavior, however [11]. A better approach is to recognize that *every* model is valid only within some circumscribed regime of operation, and that outside that regime, it is inappropriate to use that model. When the height of the bounce is small, the model of Figure 6 is invalid, and we should switch to a new model. Such switching is described below in Section 4.

3.1 Piecewise Continuity

So that we can leverage standard, well-understood numerical integration methods, we require signals to be piecewise-continuous in a specific technical sense. Consider a real-valued superdense-time signal $x\colon \mathbb{R} \times \mathbb{N} \to \mathbb{R}$. At each real-time $t \in \mathbb{R}$, we require that $x(t, n)$ settle to a final value and stay there. Specifically, we require that for all $t \in \mathbb{R}$, there exist an $m \in \mathbb{N}$ such that

$$\forall n > m, \quad x(t, n) = x(t, m). \tag{3}$$

A violation of this constraint is called a **chattering Zeno** condition. The value of the signal changes infinitely often at a time t. Such conditions would prevent an execution from progressing beyond real time t, assuming the execution is constrained to produce all values in chronological order.

Assuming x has no chattering Zeno condition, then there is a least value of m satisfying (3). We call this value of m the **final microstep** and $x(t, m)$ the **final value** of x at t. We call $x(t, 0)$ the **initial value** at time t. If $m = 0$, then x has only one value at time t.

Define the **initial value function** $x_i\colon \mathbb{R} \to \mathbb{R}$ by

$$\forall\, t \in \mathbb{R}, \quad x_i(t) = x(t, 0).$$

Define the **final value function** $x_f\colon \mathbb{R} \to \mathbb{R}$ by

$$\forall\, t \in \mathbb{R}, \quad x_f(t) = x(t, m_t),$$

where m_t is the final microstep at time t. Note that x_i and x_f are conventional continuous-time functions. A **piecewise continuous** signal is defined to be a function x of the form $x\colon \mathbb{R} \times \mathbb{N} \to \mathbb{R}$ with no chattering Zeno conditions that satisfies three requirements:

1. the initial value function x_i is continuous on the left at all $t \in \mathbb{R}$;
2. the final value function x_f is continuous on the right at all $t \in \mathbb{R}$; and
3. x has only one value at all $t \in \mathbb{R} \backslash D$, where D is a discrete subset of $\mathbb{R}$.

The last requirement is a subtle one that deserves further discussion. First, the notation $\mathbb{R} \backslash D$ refers to a set that contains all elements of the set $\mathbb{R}$ except those in the set D. D is constrained to be a **discrete set**, defined to be one where there exists an order embedding $f\colon D \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.[1] Intuitively, D is a set of time values that can be counted in temporal order. It is easy to see that if $D = \emptyset$ (the empty set), then $x_i = x_f$, and both x_i and x_f are continuous functions. Otherwise each of these functions is piecewise continuous.

Such piecewise-continuous signals coexist nicely with standard ODE solvers. At the time of a discontinuity or discrete event, the final value signal provides the initial boundary condition for the solver. The solver then works with an ordinary continuous signal until the time of the next discontinuity or discrete event, and the solver provides the initial value of the signal at the time of that next event.

Carsdoso et al. [4] give techniques for constructing models that produce only piecewise-continuous signals. For our purposes here, it is sufficient to note that modal models, as defined below, always produce piecewise-continuous signals.

[1] An **order embedding** is a one-to-one monotonic $f\colon X \to Y$, for partially ordered sets X, Y.

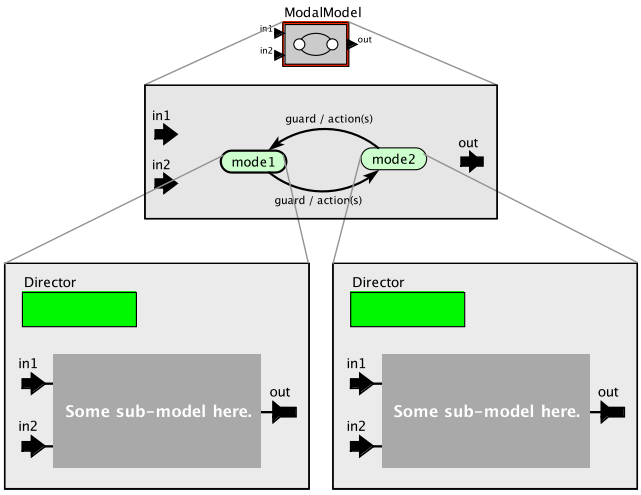

Figure 8: Structure of a modal model.

4. MODAL MODELS

CyPhySim imports the modal models of Ptolemy II [12], which provide hierarchical combinations of state machines and continuous and discrete event (DE) subsystems. This combination supports hybrid system modeling with rigorous deterministic semantics [13]. Modal models can generally be used to solve the problem that *every* model is invalid outside some regime of operation. Typically, the simpler the model, the smaller the regime. Keeping models simple has much value, however, so we are incentivized to build models for more regimes.

The schematic structure of a modal model is shown in Figure 8. A top-level actor, labeled ModalModel, is a component with inputs and outputs. Inside, it contains a state machine, shown here with two states, with guarded transitions between the states. Each state represents a **mode** of operation. The model for each mode is given as a refinement of the states. When the state machine is in mode1, for example, then the behavior of the top-level actor is given by the refinement of mode1. Each refinement has its own director, implying that its internal model of computation can be different from that of its environment.

A solution to the tunneling phenomenon of the bouncing ball problem is shown in Figure 9. Here, a much simpler model of the ball is used when the velocity and position of the ball are both close to zero. Specifically, the mode refinement for the sitting mode simply shows the ball standing still. The guards on the state machine transition govern the switching between modes.

5. DISCRETE-EVENT SIMULATION

In the style of discrete-event (DE) modeling realized in CyPhySim, a model is a network of actors with input and output ports. The actors send each other time-stamped events, and the simulation processes these events in time stamp order. This style of DE is widely used for simulation of large, complex systems [5, 27, 7]. CyPhySim builds on the particular implementation in Ptolemy II, which has a sound, deterministic semantics [10, 17].

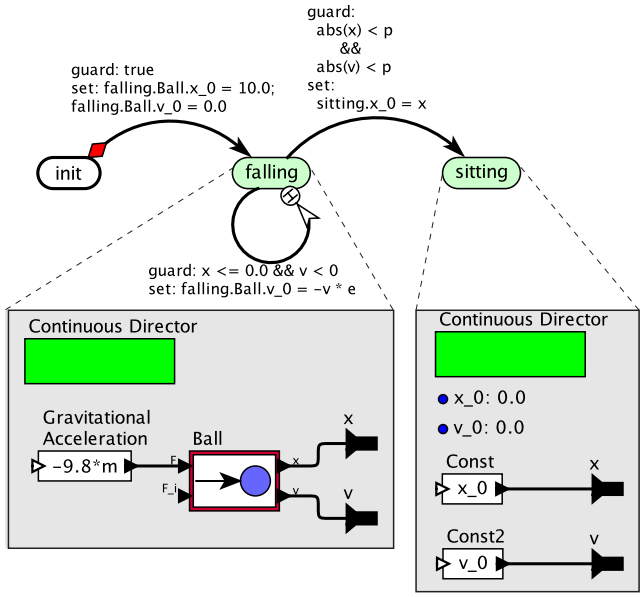

Figure 9: Modal model for the bouncing ball.

5.1 Smooth Tokens

A key innovation in CyPhySim is the introduction of a **smooth token**, which is a discrete event that represents a change in a continuous signal. A smooth token is a time-stamped event that has a real value (approximated as a double) that represents the current sample of a smooth signal. But in addition to the sample value, the smooth token contains zero or more derivatives, representing the value of a function of time at a particular time.

In mathematical analysis, smoothness has to do with how many derivatives a function possesses. A smooth function is one that has derivatives of all orders everywhere in its domain. A smooth token represents a sample of a function at a point in time together with some finite number of derivatives of the function at that same point.

This token will be treated exactly like a real value by any actor or operation that does not specifically support it, and it is represented in the Ptolemy II type system as a "double." But it can (potentially) carry additional information giving one or more derivatives of the function from which it is a sample and giving the time at which it represents a sample of the signal. This token, therefore, gives a way for actors that either generate or use this derivative information to make that information available to other actors that can use it.

An illustration of the implications is shown in Figure 10, where a signal on the left, $\dot{x}$ is piecewise constant as follows,

$$\dot{x}(t) = \begin{cases} 1 & 0 \le t < 2 \\ -1 & 2 \le t < 4 \end{cases}$$

Upon integrating this signal, we get the signal in the middle, which again can be represented by just two smooth tokens. Upon integrating the middle signal, we get the signal on the right.

When a signal is actually smooth over some interval, use of a smooth token makes it unnecessary to convey additional samples until something changes so that the extrapolation implied by a Taylor series expansion is no longer valid. For example, if a signal is piecewise smooth instead of everywhere smooth, then at the point in time where it is not smooth, a new smooth token will need to be conveyed, as shown in Figure 10.

Note that if two smooth tokens are added or subtracted, then the derivatives also add or subtract. If the times of the two tokens that are added or subtracted are not the same, then the one with the lesser time is extrapolated to the larger time, and the result will be the sum at the later time.

If a smooth token is multiplied by a smooth token, then the product rule of calculus is used to determine the derivatives of the product. The product rule stipulates that

$$(xy)' = x'y + xy'. \tag{4}$$

Again, if the times of the two tokens are not equal, then the one with the lesser time will be extrapolated to the larger time before being multiplied, and the time of the result will be the larger time.

Division works similarly:

$$(x/y)' = x'/y + x(1/y)' = x'/y - xy'/y^2 \tag{5}$$

where the last equality follows from the reciprocal rule of calculus. The second derivative of a multiplication or division is obtained by applying the above rules to x' and y' rather than to x and y. Higher-order derivatives are similarly obtained.

You can construct an instance of this token in the Ptolemy expression language using the `smoothToken(double, double, {double})` function, as illustrated in Figure 10. The first argument specifies the value, and the second argument specifies the time, and the third specifies the derivatives.

In a practical implementation, instances of smooth token will need to be truncated to have no more than some number N of derivatives. In our CyPhySim realization, this defaults to $N = 3$.

6. QUANTIZED-STATE SYSTEMS

A relatively recent development in numerical simulation of ordinary differential equations is the emergence of so-called *quantized-state systems* (QSS) [26, 9, 6, 8, 1]. In a classical ODE simulator, a step-size control algorithm determines sample times, and a sample value is computed at those times for all states in the model. In a QSS simulator, each state has its own sample times, and samples are processed using a DE simulation engine in time-stamp order. The sample time of each state is determined by quantizing the value of each state and producing samples only when the value changes by a pre-determined tolerance, called the quantum. Higher-order variants incorporate knowledge of higher-order derivatives of a state to predict trajectories and produce samples only when these higher order prediction differs by more than the quantum [18]. For some systems, QSS yields efficient simulation by producing samples only when predicted state trajectories exceed the quantum. Moreover, state events can be scheduled using an explicit equation, avoiding iteration in time.

6.1 QSS Integrator

Inputs to the QSS integrator are discrete events that indicate significant changes in the input signal, and output events indicate significant changes in the output signal. The value of the input signal is the derivative of the output. Here, for the integrator output, "significant" means that the

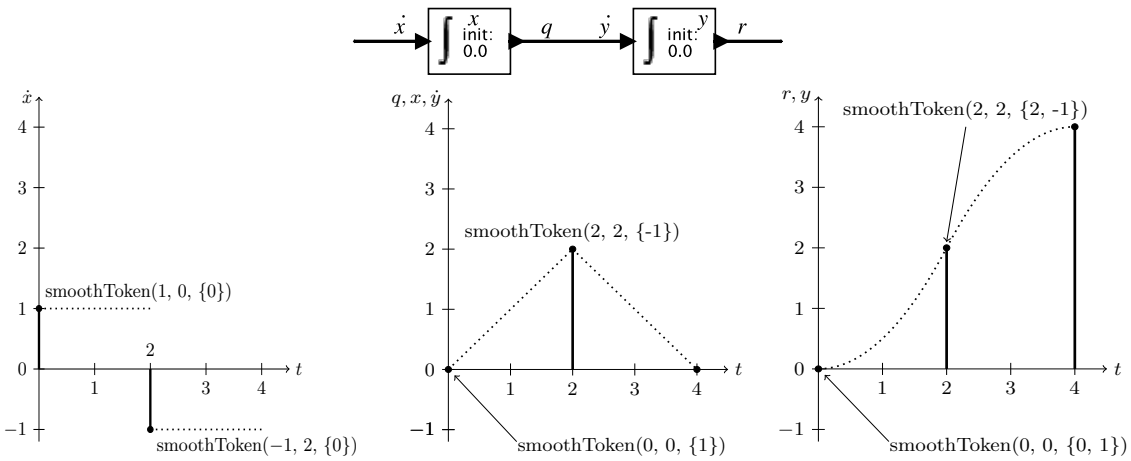

Figure 10: Illustration of smooth tokens representing piecewise-constant, -affine, and -quadratic signals.

signal has changed by more than the specified quantum, as defined by the selected solver, explained in detail below.

Three types of solvers are provided:

1. **QSS1**: Input u is assumed to be piecewise constant.
2. **QSS2**: Input u is assumed to be piecewise affine.
3. **QSS3**: Input u is assumed to be piecewise quadratic.

An input event can be a smooth token, which carries not only a value, but also zero or more derivatives of the input signal at that time. To provide a piecewise affine input to a QSS2 integrator, for example, you can specify an input with the expression `smoothToken(2, 0, {1})`, which specifies a value of 2 at time 0 with a first derivative of 1. All other derivatives are assumed to be zero. A QSS1 integrator will ignore all derivatives on the input. A QSS2 integrator will ignore all but the first derivative on the input. A QSS3 integrator will ignore all but the first and second derivatives on the input.

The integrator produces an output whenever a *quantization event* occurs. For QSS1, a quantization event occurs when the state of the integrator changes by the quantum (see below for an explanation of the quantum). For example, if the input is a constant 1 and the quantum is 0.1, then an output will be produced every 0.1 seconds, because the input specifies that the state has slope 1, so it will increase by the quantum every 0.1 seconds.

For QSS2, a quantization event occurs when a piecewise linear approximation of the state trajectory deviates from a piecewise quadratic approximation by the quantum. For example, suppose we have $t = t_0$, input $u(t)$, current state $x(t_0)$ and quantum $q > 0$, for some $q \in \mathbb{R}$. Then, the piecewise linear approximation to the state trajectory is

$$l(t) = x(t_0) + \dot{x}(t_0)\,(t - t_0) \tag{6}$$

on $t \in [t_0, \tau]$ for some, yet to be determined τ, whereas the piecewise quadratic approximation is

$$x(t) = x(t_0) + \dot{x}(t_0)\,(t - t_0) + \ddot{x}(t_0)\,(t - t_0)^2/2. \tag{7}$$

We seek

$$\begin{aligned} \tau &\in \arg\min\{t > t_0 \,|\, q = |l(t) - x(t)|\} \\ &= \arg\min\{t > t_0 \,|\, q = |\dot{u}(t_0)\,(t - t_0)^2/2|\}. \end{aligned} \tag{8}$$

Hence, the next quantization happens at $\tau = t_0 + \sqrt{2\,q/|\dot{u}(t_0)|}$, at which time an output will be produced, unless the function u changes earlier.

For QSS3, a quantization event occurs when a piecewise quadratic approximation of the state trajectory deviates from a piecewise cubic approximation by the quantum.

When an output is produced, its value will be the current state of the integrator. In addition, it may contain derivative information. For QSS1, the input is semantically piecewise constant, so the output is piecewise affine; hence each output event will be a smooth token that is piecewise affine, with a first derivative equal to the most recently received input value (see Section 6.4 below for an elaboration of this behavior). For QSS2, the output will have a first and second derivative obtained from the input. For QSS3, the output will have first, second, and third derivatives.

6.2 Quantums

The frequency with which the output q of this integrator is produced depends on the *solver* choice and the *absoluteQuantum* and *relativeQuantum* parameter values. These determine when a quantization event occurs, as explained above. The *quantum* is equal to the larger of *absoluteQuantum* and the product of *relativeQuantum* and the current state value. The simplest case is where the solver is QSS1 and *relativeQuantum* is zero. In this case, a quantization event occurs whenever the integral of the input signal changes by the *absoluteQuantum*. For QSS1, the input is assumed to be piecewise constant. If the input is a smooth token, the derivatives of the input are ignored.

When an *impulse* input is received, the value of that event is added to the current state of this integrator (any derivatives provided on the *impulse* input are ignored). Then an output event is produced and the integrator is reinitialized so that the next output quantum is relative to the new state value.

6.3 Performance and Accuracy

In our implementation in CyPhySim, the Lorenz attractor of Figure 1 executes approximately 2.5 times faster using QSS3 integrators vs. an RK 2-3 solver. However, since this system is chaotic, it is not really meaningful to compare accuracy. Chaotic systems often have the property that ar-

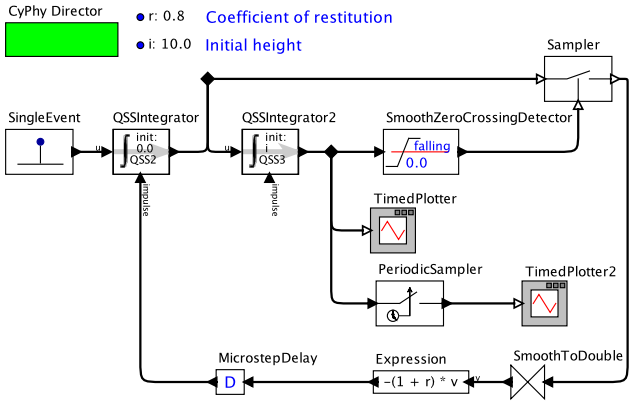

Figure 11: QSS version of a bouncing ball model.

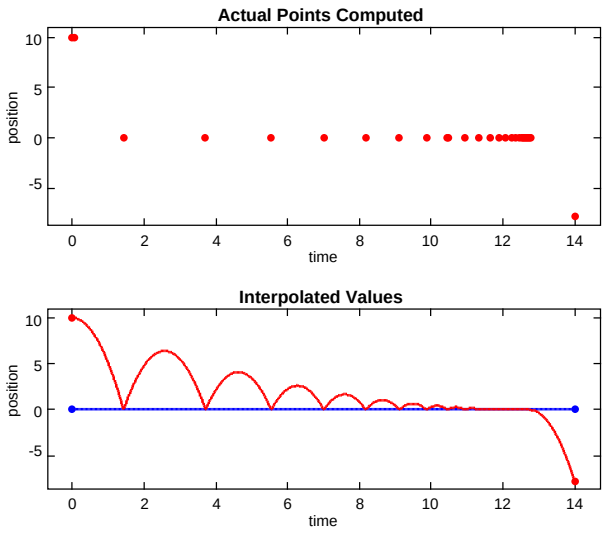

Figure 12: Position vs. time for the QSS bouncing ball.

bitrarily small perturbations can have arbitrarily big consequences, a phenomenon known as the **butterfly effect**.

Consider again the bouncing ball model of Figure 6. This problem is ideally suited to QSS because the force on the ball is piecewise constant, and accuracy comparisons are more meaningful. The velocity of the ball is therefore piecewise affine, and the position is piecewise quadratic. A QSS model in CyPhySim is shown in Figure 11, and a plot of its execution is shown in Figure 12. The upper plot shows the actual points in time computed by the simulation.

The model in Figure 6 uses an RK 2-3 solver with an error tolerance of 10^{-4}. Figure 11 uses a QSS2 solver for the left integrator (which outputs the velocity) and a QSS3 solver for the right integrator (which outputs the position).[2] Both simulations are run for 14 seconds, taking the model just past the point of tunneling.

In the QSS model, computation is performed only at 46 points in time, as illustrated in the upper plot in Figure 12. At each of these points, a smooth token is produced. The zero crossings are predictable from these points, so no rollback is needed to identify the points in time where the collisions occur. The lower plot is created by sampling the position signal with a sample interval of 0.01. The samples between computed points of the upper plot are extrapolated automatically from the smooth tokens. In the RK 2-3 model, computation is performed at 14,072 time points, not counting rejected time points due to rollback that is caused by the zero-crossing detector. On a Mac Powerbook Pro, the RK 2-3 simulation completes in 3.3 seconds, whereas the QSS simulation completes in 0.085 seconds, on average, so the QSS simulation is approximately 38 times faster.

As discussed above in Section 2.2, most classical ODE solvers require rollback to adjust step sizes based on error estimates and to iterate to the time of events such as for zero crossings. QSS solvers have the interesting property that if the QSS1, 2, and 3 assumptions about the integrator inputs, given above in Section 6.1, are indeed valid, then rollback is never required. If these assumptions are valid, then there is no error due to numerical approximation of the integration, and events such as zero crossings are predictable in advance. The bouncing ball example illustrates this well. Fortunately, for certain cyber-physical systems, these assumptions are indeed valid. Most digital actuators do, in fact, produce piecewise-constant outputs, so if the physical reaction to these actuations is sufficiently simple, then **computationally exact** simulation is possible, where the only source of errors is numerical roundoff errors. There is no error due to numerical integration. When the QSS1, 2, and 3 assumptions are not satisfied, then truncating the derivatives does, in fact, introduce errors in the numerical integration. Management of these errors appears to be a relatively unstudied problem, at least to the knowledge of these authors, except for asymptotically stable linear time invariant systems for which the global error can be computed [6].

6.4 Derivative Propagation

Mathematically, integration is an operation that always occurs over a measurable interval. Semantically, this means that integration always introduces an (infinitesimal) delay between its input and output. In CyPhySim, this means that the output of an integrator can be only based on previous inputs up to, but *not including*, the current timestamp. Semantically, QSS integrator is a composition of an ordinary integrator with a *hysteretic quantizer*, the latter of which suppresses changes in its output until it has deviated from its old value by one quantum. However, since a QSS integrator outputs derivative information inside its smooth tokens, and the derivative is precisely the input itself, one may ask whether the output should propagate changes in the input *instantaneously* at the following microstep, rather than suppress them until a quantum is reached at a future time.

While propagating derivative information instantaneously alters QSS semantics slightly, it can only increase the accuracy of the model, since downstream components are notified *earlier* of the changes than they otherwise would be. Of course, it would also come at a computational cost, as the model is now evaluated much more frequently than the quantum size would otherwise suggest. Preliminary results, however, seem to suggest that it is possible for the accu-

[2]This highlights another advantage QSS, which is that the choice of QSS order can be optimized for each state variable independently, although in this case, there is very little observable difference if QSS3 is used throughout.

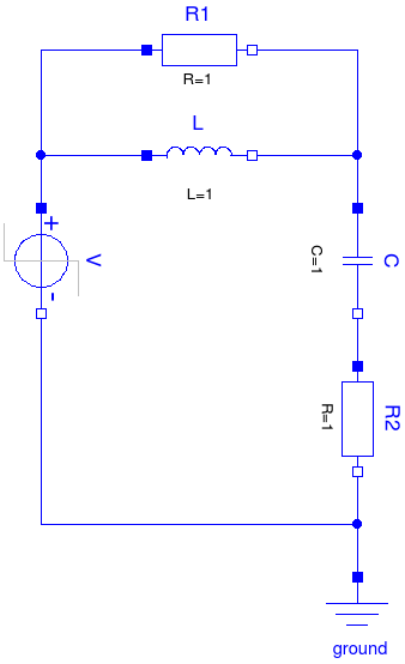

Figure 13: RLC circuit requiring an algebraic loop solver (after Figure 12.8 in [6]).

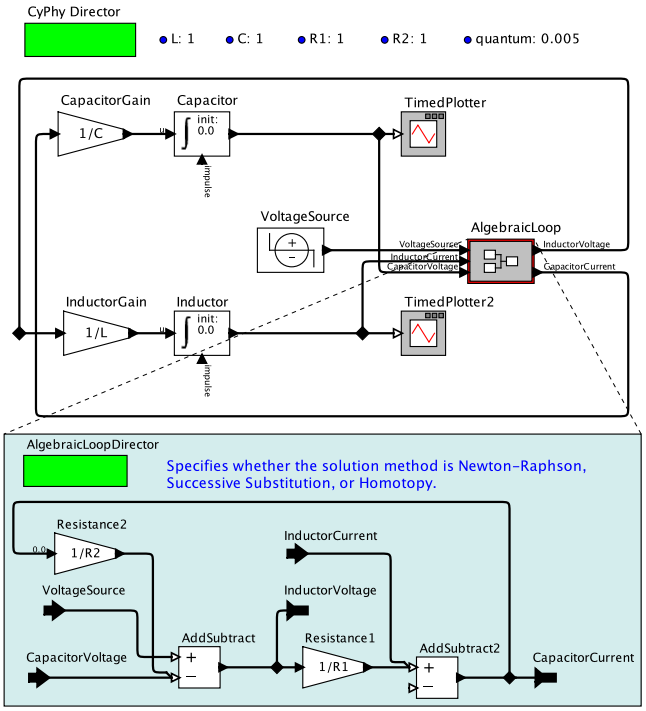

Figure 14: Model of the RLC circuit of Figure 13.

racy improvement to compensate for the performance loss by causing the model to correct its errors sooner rather than later, and hence do so less often. Therefore, to allow for the potential benefits of such behavior, a QSS integrator in CyPhySim has a `propagateInputDerivatives` parameter that dictates whether changes in the input should be propagated to the output at the current model time.

7. ALGEBRAIC LOOP SOLVERS

CyPhySim includes a mechanism for specifying algebraic loop solvers, including a simple successive substitution mechanism, a Newton-Raphson solver, and a homotopy method. The model builder is given explicit control over the solution method and initial guesses in order to ensure deterministic results.

Figure 13 shows the electrical circuit with an algebraic loop, which is the problem shown in Figure 12.8 in [6]. For this system, an algebraic loop needs to be solved whenever a QSS integrator changes its output, or when VoltageSource triggers an event. Figure 14 shows the CyPhySim implementation. The actor labeled AlgebraicLoop implements the algebraic loop that needs to be solved. It has an input for the voltage source, and two pairs of inputs and outputs that connect the algebraic loop to the capacitor and the inductor, each having a state variable.

8. FURTHER WORK

Some of the capabilities of CyPhySim are beyond the scope of this paper. For example, CyPhySim includes the ability to import **functional mockup units** (FMUs), which are components designed in some foreign modeling language or tool (such as Modelica), and exported with an interface defined by the FMI standard. The Functional Mockup Interface (FMI) is a standard for model exchange and co-simulation of dynamical models [19]. CyPhySim supports importing FMUs designed for model exchange or co-simulation. For FMU for model-exchange, it provides a QSS simulation engine for numerical integration. This strategy also enables co-simulation of DE models with FMI, a combination also described in [24].

Many CPS applications include sampled-data subsystems with regular, periodic sample rates. CyPhySim leverages the synchronous-reactive (SR) domain of Ptolemy II to provide such models, which permits specification of a sample rate and enables structured multi rate systems. Such SR models interoperate well with DE, QSS, and Continuous models [14]. CyPhySim incorporates a new innovation that enables arbitrary hierarchical nesting of these models. Sampled data systems can contain continuous or DE subsystems and vice versa.

Nevertheless, much work remains to be done. The approach used to handle algebraic loops in Section 7 is semantically sound, but the graphical syntax of such models is not satisfactory. The requirement that algebraic loops be factored out into a different level of hierarchy in the model leads to good separation of concerns, but it makes the models considerably less readable. A better approach would be to graphically or textually describe the structure of the models in the most natural way, including where appropriate using acausal ports as in Modelica, and then to automatically transform the model for analysis and execution into a form that separates concerns and implements the correct semantics.

The operations on derivatives that are described in Section 5.1 are particularly convenient to implement in CyPhySim, because in the underlying Ptolemy II system, arithmetic operations on tokens exchanged between actors are polymor-

phic, which means that the token itself defines what addition, multiplication, division, etc. mean. An interesting possibility, which is not so trivial to implement in the software, would extend the manipulation of derivatives to transcendental functions and other common operations on signals. In effect, this would endow a numerical solver with a measure of symbolic computation capability, creating an interesting blend of symbolic and numerical simulation.

9. REFERENCES

[1] Bliudze, S., and Furic, S. An operational semantics for hybrid systems involving behavioral abstraction. In *Modelica Conference* (2014). QSS. Quantizing signals rather than time.

[2] Broman, D., Brooks, C., Greenberg, L., Lee, E. A., Masin, M., Tripakis, S., and Wetter, M. Determinate composition of fmus for co-simulation. In *International Conference on Embedded Software (EMSOFT)* (2013), IEEE.

[3] Broman, D., Greenberg, L., Lee, E. A., Masin, M., Tripakis, S., and Wetter, M. Requirements for hybrid cosimulation standards. In *Hybrid Systems: Computation and Control (HSCC)* (2015).

[4] Cardoso, J., Lee, E. A., Liu, J., and Zheng, H. Continuous-time models. In *System Design, Modeling, and Simulation using Ptolemy II*, C. Ptolemaeus, Ed. Ptolemy.org, Berkeley, CA, 2014.

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

[6] Cellier, F. E., and Kofman, E. *Continuous System Simulation*. Springer, 2006.

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

[8] Floros, X., Cellier, F. E., and Kofman, E. Discretizing time or states? a comparative study between dassl and qss - work in progress paper -. In *Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)* (2010), Linköping University.

[9] Kofman, E., and Junco, S. Quantized-state systems: A DEVS approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International 18*, 1 (2001), 2–8.

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

[11] Lee, E. A. Constructive models of discrete and continuous physical phenomena. *IEEE Access 2*, 1 (2014), 1–25.

[12] Lee, E. A., and Tripakis, S. Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)* (Oslo, Norway, 2010), vol. 47, Linköping University Electronic Press, Linköping University, pp. 11–21.

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

[14] Lee, E. A., and Zheng, H. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT* (Salzburg, Austria, 2007), ACM, pp. 114 – 123.

[15] Liu, J., and Lee, E. A. On the causality of mixed-signal and hybrid models. In *6th International Workshop on Hybrid Systems: Computation and Control (HSCC '03)* (Prague, Czech Republic, 2003), O. Maler and A. Pnueli, Eds., vol. LNCS 2623, Springer-Verlag, pp. 328–342.

[16] Maler, O., Manna, Z., and Pnueli, A. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop* (1992), Springer-Verlag, pp. 447–484.

[17] Matsikoudis, E., and Lee, E. A. On fixed points of strictly causal functions. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)* (Buenos Aires, Argentina, 2013), vol. LNCS 8053, Springer-Verlag, pp. 183–197.

[18] Migoni, G., Bortolotto, M., Kofman, E., and Cellier, F. E. Linearly implicit quantization-based integration methods for stiff ordinary differential equations. *Simulation Modelling Practice and Theory 35* (2013), 118–136.

[19] Modelica Association. Functional mock-up interface for model exchange and co-simulation. Report Version 2.0, July 25 2014.

[20] Nagel, L. W., and Pederson, D. O. SPICE (simulation program with integrated circuit emphasis). Technical memorandum, Electronics Research Laboratory, University of California, Berkeley, April 1973.

[21] Pantelides, C. C. The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing 9*, 2 (1988), 213–231.

[22] Paynter, H. M. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, MA, 1961.

[23] Ptolemaeus, C., Ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, 2014.

[24] Savicks, V., Butler, M., and Colley, J. Co-simulating Event-B and continuous models via FMI. In *Summer Simulation Multi-Conference (SummerSim)* (2014), pp. 259–256.

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

[26] Zeigler, B. P., and Lee, J. S. Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment. In *SPIE Conference on Enabling Technology for Simulation Science* (1998), vol. SPIE Vol. 3369, pp. 49–58.

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