9 Continuous-Time Models

Janette Cardoso, Edward A. Lee, Jie Liu, and Haiyang Zheng

Contents

Continuous-time models are realized using the Ptolemy II **continuous-time** (**CT**) domain (also called the **Continuous** domain), which models physical processes. This domain is particularly useful for cyber-physical systems, which are characterized by their mixture of computational and physical processes.

The CT domain conceptually models time as a continuum. It exploits the superdense time model in Ptolemy II to process signals with discontinuities, signals that mix discrete and continuous portions, and purely discrete signals. The resulting models can be combined hierarchically with discrete event models, and modal models can be used to develop hybrid systems.

9.1 Ordinary Differential Equations

The continuous dynamics of physical processes are represented using **ordinary differential equations** (**ODE**s), which are differential equations over a time variable. The Ptolemy II models of continuous-time systems are similar to those used in Simulink (from The MathWorks), but Ptolemy's use of superdense time provides cleaner modeling of mixed signal and hybrid systems (Lee and Zheng, 2007). This section focuses on how continuous dynamics are specified in a Ptolemy II model and how the Continuous director executes the resulting models.

9.1.1 Integrator

In Ptolemy II, differential equations are represented using **Integrator** actors in feedback loops. At time t, the output of an Integrator actor is given by

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau) d\tau, \tag{9.1}$$

where x_0 is the *initialState* of the Integrator, t_0 is the *startTime* of the director, and $\dot{x}$ is the input signal to the Integrator. Note that since the output x of the Integrator is the integral of its input $\dot{x}$, then at any given time, the input $\dot{x}$ is the derivative of the output x,

$$\dot{x}(t) = \frac{d}{dt} x(t). \tag{9.2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use. ODEs can be represented using Integrator actors, as illustrated by the following example.

Example 9.1: The well-known **Lorenz attractor** is a non-linear feedback system that exhibits a style of chaotic behavior known as a **strange attractor**. The model in Figure 9.1 is a block diagram representation of the set of nonlinear ODEs that govern the behavior of this system. Let the output of the top integrator be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3.

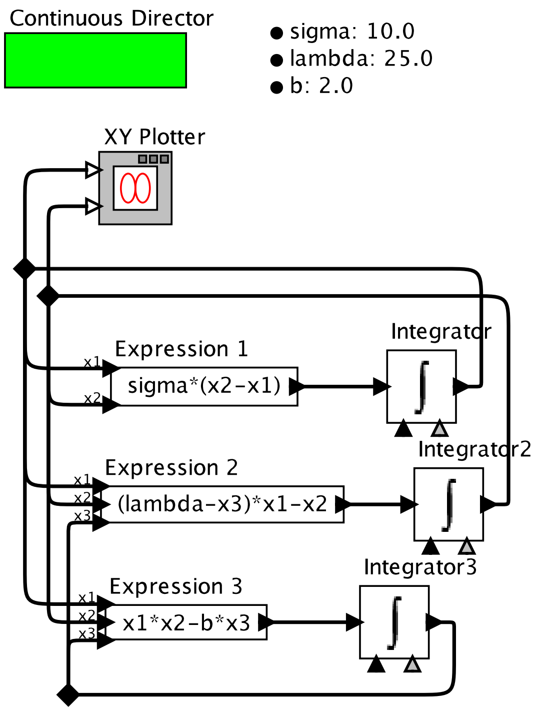


Figure 9.1: A model describing a set of nonlinear ordinary differential equations. [online]

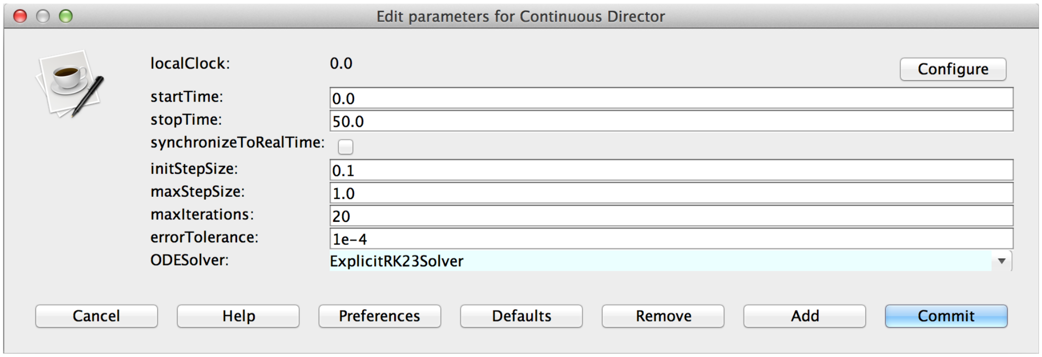
Then the equations described by Figure 9.1 are

$$\begin{aligned}\dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t) \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)\end{aligned} \tag{9.3}$$

where σ, λ, and b are real-valued constants. For each equation, the expression on the right side of the equals sign is implemented by an Expression actor, whose icon shows the expression. Each expression refers to parameters (such as *lambda* for λ and *sigma* for σ) and input ports of the actor (such as x1 for x_1 and x2 for x_2). The expression in each Expression actor can be edited by double clicking on the actor, and the parameter values can be edited by double clicking on the parameters, which are shown next to bullets at the top.

The three integrators specify initial values for x_1, x_2, and x_3; these values can be changed by double-clicking on the corresponding Integrator icon. In this example, all three initial values are set to 1.0 (not shown in the figure).

The Continuous Director, shown at the upper left, manages the simulation of the model. It contains a sophisticated ODE solver with several key parameters. These parameters can be accessed by double clicking on the director, which results in the dialog box shown in Figure 9.2.


Edit parameters for Continuous Director

localClock:	0.0
startTime:	0.0
stopTime:	50.0
synchronizeToRealTime:	☐
initStepSize:	0.1
maxStepSize:	1.0
maxIterations:	20
errorTolerance:	1e-4
ODESolver:	ExplicitRK23Solver

Figure 9.2: Dialog box showing director parameters for the model in Figure 9.1.

The simplest parameters are *startTime* and *stopTime*, which define the region of the time line over which the simulation will execute. The effects of the other parameters are explored in Exercise 1.

The output of the Lorenz model is shown in Figure 9.3. The XY Plotter displays $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

Like the Lorenz model, many continuous-time models contain integrators in feedback loops. Instead of using Integrator actors, however, it is possible to use more elaborate blocks that implement linear and non-linear dynamics, as described below.

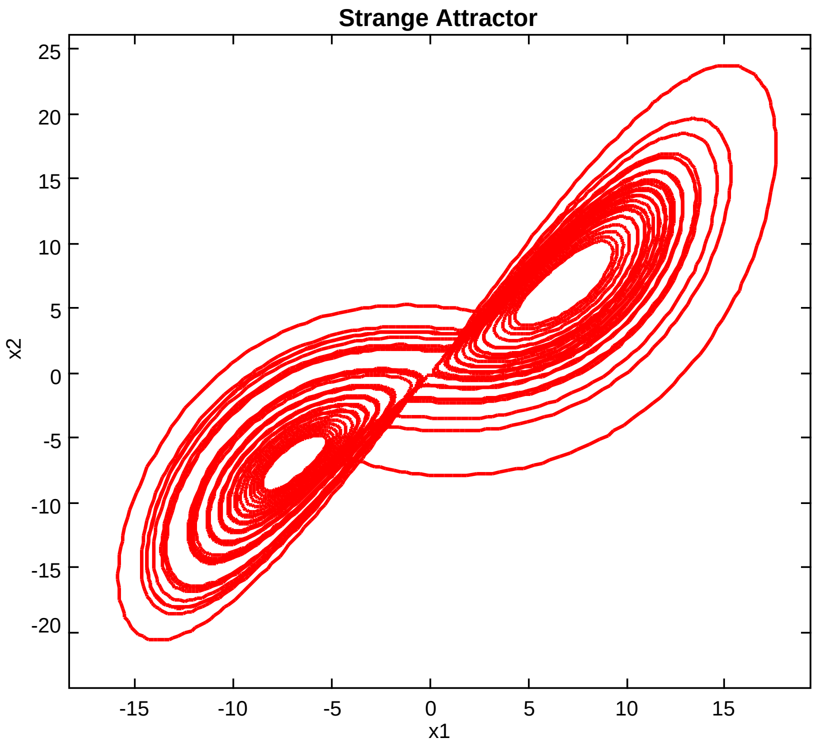


Figure 9.3: Result of running the Lorenz model.

9.1.2 Transfer Functions

When representing continuous-time systems, it is often more convenient to use a higher-level description than individual integrators. For example, for linear time invariant (**LTI**) systems, it is common to characterize their input output behavior in terms of a **transfer function**, which is the Laplace transform of the impulse response. Specifically, for an input x and output y, the transfer function may be given as a function of a complex variable s:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \cdots b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \cdots a_n} \tag{9.4}$$

where Y and X are the Laplace transforms of y and x, respectively. The number n of denominator coefficients is strictly greater than the number m of numerator coefficients. A system that is described by a transfer function can be constructed using individual integrators, but is more conveniently implemented using the **ContinuousTransferFunction** actor, as illustrated by the following example.

Example 9.2: Consider the model in Figure 9.4, which produces the plot in Figure 9.5. This model generates a square wave using a ContinuousClock actor (see sidebar on page 326) and feeds that square wave into a ContinuousTransferFunction actor. The transfer function implemented by the ContinuousTransferFunction actor is given by the following:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{0.001s^2 + 0.01s + 1}.$$

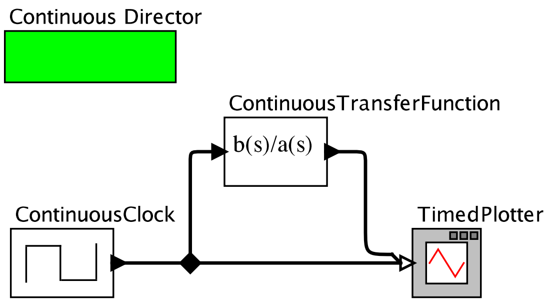


Figure 9.4: Model illustrating the use of ContinuousTransferFunction. [online]

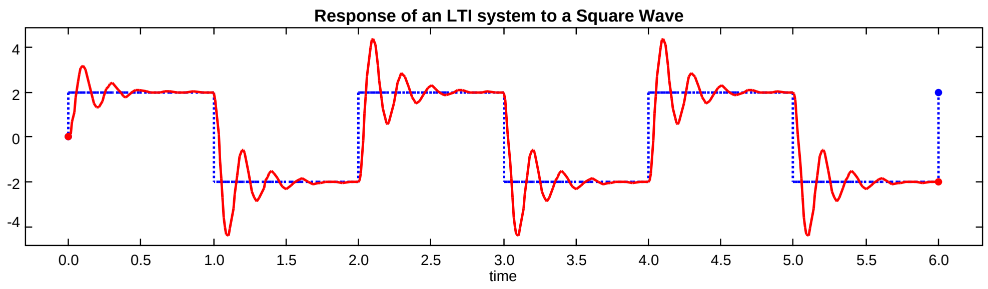


Figure 9.5: Result of running the model in Figure 9.4.

Comparing the equation above to Equation (9.4), we see that $m = 1$ and $n = 3$, with additional parameters as follows:

$$\begin{aligned} b_1 &= 1 \\ a_1 &= 0.001, \quad a_2 = 0.01, \quad a_3 = 1.0 \end{aligned}$$

The parameters of the actor are therefore set to

$$\begin{aligned} \textit{numerator} &= \{\texttt{1.0}\} \\ \textit{denominator} &= \{\texttt{0.001, 0.01, 1.0}\} \end{aligned}$$

An equivalent model constructed with individual integrators is shown in Figure 9.6 (see Exercise 2 to explore why these are equivalent).

The previous example shows that a complex network of integrators, gains, and adders can be represented compactly using the ContinuousTransferFunction actor. In fact, this actor uses the specified parameter values to construct a hierarchical model similar to the one shown in Figure 9.6. It is possible to view this hierarchical model by right clicking on the ContinuousTransferFunction actor and selecting `Open Actor`. (Select [`Graph`→`Automatic Layout`] so that the actors are shown in a more readable layout.) ContinuousTransferFunction is an example of a higher-order actor, where the parameters specify an actor network that implements the functionality of the actor.

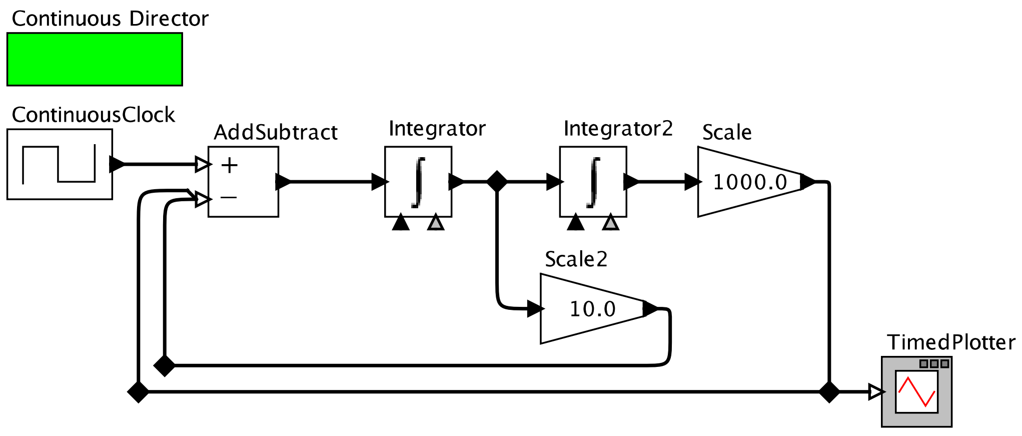


Figure 9.6: A model equivalent to the one in Figure 9.4 assuming the parameters of Example 9.2. [online]

The ContinuousTransferFunction actor and other actors that support higher-level descriptions of dynamics are summarized in the sidebar on page 327.

9.1.3 Solvers

Numerical integration is an old, complex, and deep topic (Press et al., 1992). A complete treatment of this topic is beyond the scope of this text, but it is useful to understand the basic concepts in order to make effective use of the Ptolemy solver functions (which use numerical integration to find solutions to equations). In this section, we will give a brief overview of the solver mechanisms that are implemented in the Ptolemy II Continuous director.

Suppose that w is a continuous-time signal. For the moment, let us ignore the superdense time model used in Ptolemy II, and assume that w is an integrable function of the form $w\colon \mathbb{R} \to \mathbb{R}$. Assume further that for any $t \in \mathbb{R}$, we have a procedure to evaluate $w(t)$. Suppose further that x is a continuous-time signal given by

$$x(t) = x_0 + \int_0^t w(\tau) d\tau,$$

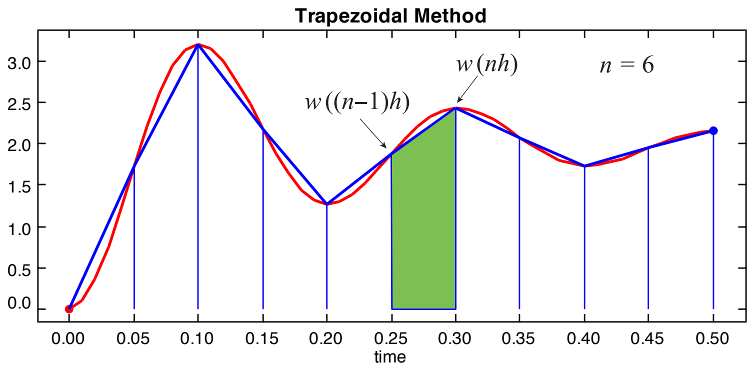


Figure 9.7: Illustration of the trapezoidal method. The area under the curve is approximated by the sum of the areas of the trapezoids. One of the trapezoids is shaded.

where x_0 is a constant. Equivalently, $x(t)$ is the area under the curve formed by $w(\tau)$ from $\tau = 0$ to $\tau = t$, plus an initial value x_0. Note that, consequently, w is the derivative of x, or $w(t) = \dot{x}(t)$. Given w, we can construct x by providing w as the input to an Integrator actor with *initialState* set to x_0; x will then be the output.

Numerical integration is the process of evaluating x at enough points $t \in \mathbb{R}$ to accurately deduce the shape of the function. Of course, the meaning of "accurate" may depend on the application, but one of the key criteria is that the value of x is sufficiently accurate at sufficiently many points that those values of x can be used to calculate values of x at additional points $t \in \mathbb{R}$ in time. A **solver** is a realization of a numerical integration algorithm. The simplest solvers are **fixed step size solvers**. They define a **step size** h, and calculate x at intervals of h, specifically $x(h)$, $x(2h)$, $x(3h)$, etc.

A reasonably accurate fixed step size solver uses the **trapezoidal method**, where it approximates x as follows:

$$x(nh) = \begin{cases} x_0, & \text{if } n = 0 \\ x((n-1)h) + h(w((n-1)h) + w(nh))/2, & \text{if } n \geq 1 \end{cases}$$

This approach is illustrated in Figure 9.7. As defined by the equations above, the area under the curve w from 0 to nh is approximated by the sum of the areas of trapezoids of width h, where the heights of the sides of the trapezoids are given by $w(mh)$ for integers

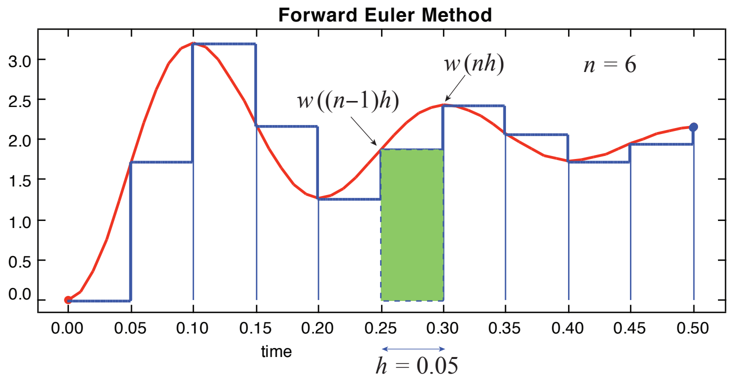


Figure 9.8: Illustration of the forward Euler method. The area under the curve is approximated by the sum of the areas of the rectangles, like the shaded one.

m. The shaded trapezoid in the figure approximates the area under the curve from time $(n-1)h$ to time nh, where $n = 6$ and $h = 0.05$.

A trapezoidal method solver is difficult to use within feedback systems like the one shown in Figure 9.1, however, because the solver needs to compute the outputs of the Integrator actors based on the inputs. To compute the output of an Integrator at time $t_n = nh$, the solver needs to know the value of the input at both $t_{n-1} = (n-1)h$ and $t_n = nh$. But in Figure 9.1, the input to the integrators at any time t_n depends on the output of the same integrators at that same time t_n; there is a circular dependency. Solvers that exhibit a circular dependency are called **implicit method** solvers. One way to use them within feedback systems is to "guess" the feedback value and iteratively refine the guess until some desired accuracy is achieved, but in general there is no assurance that such strategies yield unique answers.

In contrast to implicit method solvers, the **forward Euler method** is an **explicit method** solver. It is similar to the trapezoidal method but is easier to apply to feedback systems. It approximates x by

$$x(nh) = x((n-1)h) + hw((n-1)h).$$

This approach is illustrated in Figure 9.8. The area under each step is approximated as a rectangle rather than as a trapezoid. This method is less accurate, usually, and errors accumulate faster, but it does not require the solver to know the input at time nh.

In general, using a smaller step size h increases the accuracy of the solution, but increases the amount of computation required. The step size required to meet a target level of accuracy depends on how rapidly the signal is varying. Both the trapezoidal method and the forward Euler method can be generalized to become **variable step size solvers** that dynamically adjust their step size based on the variability of the signal. Such solvers evaluate the integral at time instants t_1, t_2, etc., using an algorithm to determine the increment to use between time instants. This algorithm first chooses a step size, then performs the numerical integration, then estimates the error. If the estimate of the error is above some threshold (controlled by the *errorTolerance* parameter of the director), then the director redoes the numerical integration with a smaller step size.

A variable-step-size forward Euler solver will first determine a time increment h_n to define $t_n = t_{n-1} + h_n$ and then calculate

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

The variable-step-size forward Euler method is a special case of the widely used **Runge-Kutta** (**RK**) methods. The Continuous director offers two variants of RK solvers, `ExplicitRK23Solver` and `ExplicitRK45Solver`, selected using the *ODESolver* parameter of the director. These variants are described in more detail in the sidebars on pages 328 and 329. The plot in Figure 9.5 is generated using the `ExplicitRK23Solver`. A closeup with stems that indicate where the solver chose to calculate signal values is shown in Figure 9.9. This figure shows that the solver uses smaller step sizes in regions where the signal is varying more rapidly.

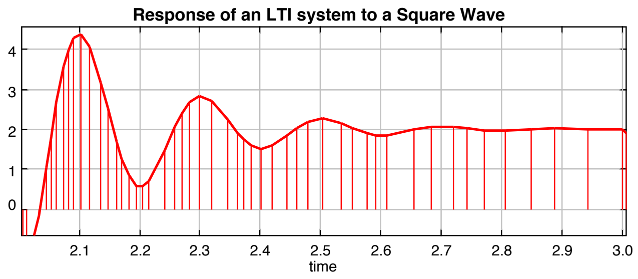
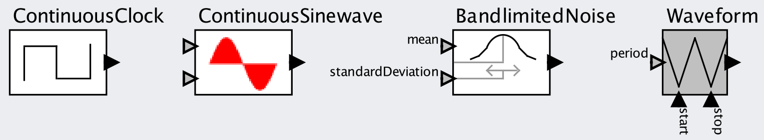


Figure 9.9: A closeup of the plot in Figure 9.5 (with stems showing) reveals that the solver uses smaller step sizes in regions where the signal varies more rapidly.

Sidebar: Continuous-Time Signal Generators

The continuous domain provides several actors that generate continuous-time signals.

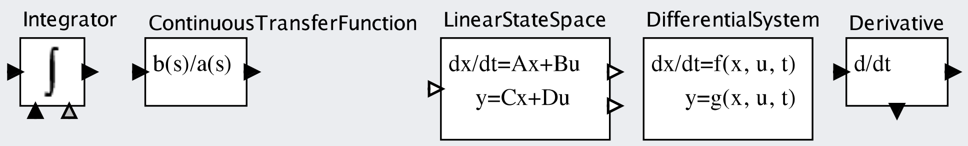


These actors are located in `DomainSpecific→Continuous→SignalGenerators`, except BandlimitedNoise, which is in `DomainSpecific→Continuous→Random`.

- **ContinuousClock** has parameters similar to DiscreteClock, but produces a **piecewise constant** signal. A square wave, like that shown in Figure 9.5, is a simple example of such a signal; the actor is also capable of producing complex repeating or non-repeating patterns.
- **ContinuousSinewave**, as the name implies, produces a sine wave. The frequency, phase, and amplitude of the sine wave are set via parameters. This actor constrains the step size of the solver to ensure that its output is reasonably smooth; the step size is set to be no greater than one tenth of a period of the sine wave.
- **BandlimitedNoise** is the most sophisticated of these signal generators. This actor generates continuous-time noise with a Gaussian distribution and controlled bandwidth. Although a full discussion of the topic is beyond the scope of this text, we note that it is not theoretically possible for a causal system to produce perfectly bandlimited noise (see Lee and Varaiya (2011)). This actor implements a reasonable approximation. Like ContinuousSinewave, this actor affects the step size chosen by the solver; it ensures that the solver samples the signal at least as frequently as twice the specified bandwidth. This is nominally the Nyquist frequency of an ideally bandlimited noise signal.
- **Waveform** produces a periodic waveform from a finite set of samples. It provides two interpolation methods, linear and Hermite, where the latter uses a third-order interpolation technique based on the Hermite curves in Chapter 11 of Foley et al. (1996). Hermite interpolation is useful for generating smooth curves of arbitrary shape. The interpolation assumes that the waveform is periodic. Note that this actor also affects the step sizes taken by the solver. In particular, it ensures that the solver includes the specified samples, though it does not require the solver to include any samples between them.

Sidebar: Actors for Modeling Dynamics

Ptolemy II provides several actors that can be used to model continuous-time systems with complicated dynamics (behavior over time). These actors are shown below, and can be found in `DomainSpecific→Continuous→Dynamics`:



The fundamental actor is the Integrator, described in Sections 9.1.1 and 9.2.4. The others are higher-order actors that construct submodels using instances of Integrator.

- ContinuousTransferFunction, as explained in Section 9.1.2, can realize a continuous-time system based on a transfer function specified as a ratio of two polynomials. The ContinuousTransferFunction actor does not support non-zero initial conditions for the Integrators.
- **LinearStateSpace** specifies the input-output relationship of a system with a set of matrices and vectors that describe a linear constant-coefficient difference equation (**LCCDE**). Unlike ContinuousTransferFunction, this actor supports non-zero initial conditions, but it is similarly constrained to model systems that can be characterized by linear functions.
- **DifferentialSystem** can be used to model complicated nonlinear dynamics. For example, it can be used to specify the Lorenz attractor of Example 9.1, as shown in Exercise 3. See the actor documentation for details.
- **Derivative** provides a crude estimate of the derivative of its input. Use of this actor is discouraged, however, because its output can be very noisy, even if the input is continuous and differentiable. The output is simply the difference between the current input and the previous input divided by the step size. If the input is not differentiable, however, the output is not piecewise continuous, which may force the solver to use the smallest allowed step size. Note that this actor has *two* outputs. The bottom output produces a discrete event when the input has a discontinuity. This event represents a Dirac delta function, explained in Section 9.2.4.

Sidebar: Runge-Kutta Methods

In general, an ODE can be represented by a system of differential equations on a vector-valued state

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

where $x : \mathbb{R} \to \mathbb{R}^n$, $y : \mathbb{R} \to \mathbb{R}^m$, and $u : \mathbb{R} \to \mathbb{R}^l$ are state, output, and input signals. The functions $g : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^m$ are state functions and output functions respectively. The state function g is represented by the Expression actors in the feedback path in Figure 9.1. This function gives the inputs $\dot{x}(t)$ of the Integrator actors as a function of their outputs $x(t)$, external inputs $u(t)$ (of which there are none in Figure 9.1), and the current time t (which is also not used in Figure 9.1).

Given this formulation, an explicit k-stage RK method has the form

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i, \tag{9.5}$$

where

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

and $A_{i,j}$, b_i and c_i are algorithm parameters calculated by comparing the form of a Taylor series expansion of x with (9.5).

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

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

This method is conceptually important but not as accurate as other available methods.

Continued on page 329.

Sidebar: Runge-Kutta Methods - Continued

Continued from page 328.

More accurate Runge-Kutta methods have three or four stages, and also control the step size for each integration step. The `ExplicitRK23Solver` implemented by the Continuous director is a $k = 3$ (three-stage) method and is given by

$$x(t_n) = x(t_{n-1}) + \frac{2}{9}K_0 + \frac{3}{9}K_1 + \frac{4}{9}K_2, \tag{9.6}$$

where

$$\begin{aligned} K_0 &= h_n g(x(t_{n-1}), t_{n-1}), && (9.7)\\ K_1 &= h_n g(x(t_{n-1}) + 0.5K_0, u(t_{n-1} + 0.5h_n), \\ &\quad t_{n-1} + 0.5h_n), && (9.8)\\ K_2 &= h_n g(x(t_{n-1}) + 0.75K_1, u(t_{n-1} + 0.75h_n), \\ &\quad t_{n-1} + 0.75h_n). && (9.9) \end{aligned}$$

Notice that in order to complete one integration step, this method requires evaluation of the function g at intermediate times $t_{n-1} + 0.5h_n$ and $t_{n-1} + 0.75h_n$, in addition to the times t_{n-1}, where h_n is the step size. This fact significantly complicates the design of actors, because they have to tolerate multiple evaluations (firings) that are speculative and may have to be redone with a smaller step size if the required accuracy is not achieved. The validity of a step size h_n is not known until the full integration step has been completed. In fact, any method that requires intermediate evaluations of the state function g, such as the classical fourth-order RK method, linear multi-step methods (LMS), and BulirschStoer methods, will encounter the same issue.

In the Continuous domain, the RK solvers speculatively execute the model at intermediate points, invoking the `fire` method of actors but not their `postfire` method. As a consequence, actors used in the Continuous domain must all conform to the strict actor semantics; they must not change state in their `fire` method.

9.2 Mixed Discrete and Continuous Systems

The continuous domain supports mixtures of discrete and continuous behaviors. The simplest such mixtures produce piecewise continuous signals, which vary smoothly over time except at particular points in time, where they vary abruptly. Piecewise continuous signals are explained in Section 9.2.1.

In addition, signals can be genuinely discrete. In particular, as with the DE domain, a signal in the Continuous domain can be absent at a time stamp. A signal that is *never* absent is a true **continuous-time signal**. A signal that is always absent except at a discrete set of time stamps is a **discrete-event signal**, explained in Section 9.2.2. A model that mixes both types of signals is a **mixed signal** model. It is possible to have signals that are continuous-time over a range of time stamps, and discrete-event over another range. These are called **mixed signals**.

9.2.1 Piecewise Continuous Signals

As shown in Figure 9.9, variable step-size solvers produce more samples per unit time when a signal is varying rapidly. These solvers do not, however, directly support discontinuous signals, such as the square wave shown in Figure 9.5. The Continuous director in Ptolemy II augments the standard ODE solvers with techniques that handle such discontinuities, but the signals must be piecewise continuous. Meeting this prerequisite requires some care, as we will discuss later in the chapter.

Recall that Ptolemy II uses a superdense time model. This means that a continuous-time signal is a function of the form

$$x\colon T \times \mathbb{N} \to V, \tag{9.10}$$

where T is the set of model time values (see Section 1.7.3), $\mathbb{N}$ is the non-negative integers representing the microstep, and V is some set of values (the set V is the data type of the signal). This function specifies that, at each model time $t \in T$, the signal x can have several values, and these values occur in a defined order. For the square wave in Figure 9.5, at time $t = 1.0$, for example, the value of the square wave is first $x(t, 0) = 2$ and then $x(t, 1) = -2$.

In order for time to progress past a model time $t \in T$, we need to ensure that every signal in the model has a finite number of values at t. Thus, we require that for all $t \in T$, there exist an $m \in \mathbb{N}$ such that

$$\forall n > m, \quad x(t, n) = x(t, m). \tag{9.11}$$

This constraint prevents chattering Zeno conditions, where a signal takes on infinitely many values at a particular time. Such conditions prevent an execution from progressing beyond that point in model time, assuming the execution is constrained to produce values in chronological order.

Assuming x has no chattering Zeno condition, then there is a least value of m satisfying (9.11). 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 we say that x has only one value at time t.

Define the **initial value function** $x_i \colon T \to V$ by

$$\forall\, t \in T, \quad x_i(t) = x(t, 0).$$

Define the **final value function** $x_f \colon T \to V$ by

$$\forall\, t \in T, \quad x_f(t) = x(t, m_t),$$

where m_t is the final microstep at time t. Note that x_i and x_f are conventional continuous-time functions if we abstract model time as the real numbers $T = \mathbb{R}$.

A **piecewise continuous** signal is defined to be a function x of the form $x \colon T \times \mathbb{N} \to V$ with no chattering Zeno conditions that satisfies three requirements:

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

The last requirement is a subtle one that deserves further discussion. First, the notation $T \backslash D$ refers to a set that contains all elements of the set T except those in the set D. D is constrained to be a discrete set, described in the sidebar on page 334. 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.

A key constraint of the Continuous domain in Ptolemy II is that all signals are piecewise continuous in the above sense. Based on this definition, the square wave shown in Figure 9.5 is piecewise continuous. At each discontinuity, it has two values: an initial value that matches the values before the discontinuity, and a final value that matches the value after the discontinuity. It can be easy to create signals that are not piecewise continuous, however, as illustrated by the following example.

Example 9.3: The model in Figure 9.10 contains an Expression actor whose input is a continuous-time signal. The expression is shown below:

```
(in > 1.0) ? in + 1 : 0
```

If the input is greater than 1.0, then the output will be the input plus one; otherwise the output will be zero. This output signal is not piecewise continuous in the sense described above. Before or at time 1.0, the output value is zero. But at any time after 1.0, the value is not zero. The signal is not continuous from the right.

Figure 9.11 shows the resulting plot, where the output of the Expression actor is labeled "second." The transition from zero to non-zero is not instantaneous, as

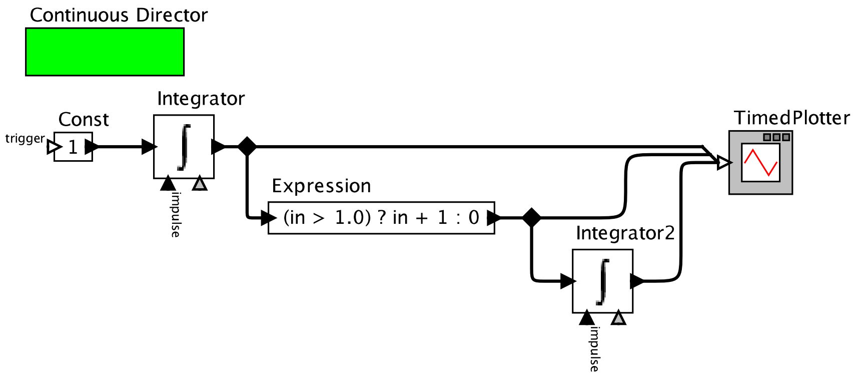


Figure 9.10: A model that produces a signal that is not piecewise continuous, and therefore can exhibit solver-dependent behavior. This problem is eliminated in the model in Figure 9.23. [online]

shown by the slanted dashed line in the middle plot. Worse, the width of the transition depends on seemingly irrelevant details of the model. The model shows the signal connected to a second integrator. If that second integrator is deleted from the

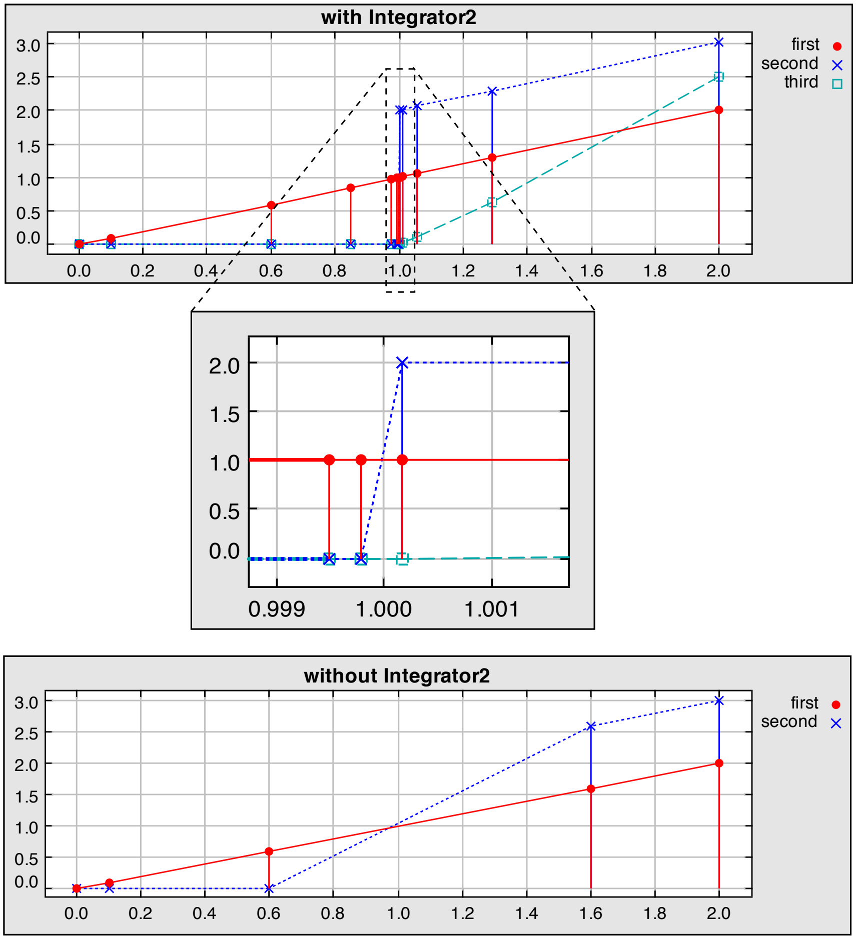


Figure 9.11: A plot of the signal produced by the model in Figure 9.10 with and without the second Integrator. Note that the output of the Expression actor seems to depend on whether the second Integrator is present. The signals labeled “first,” “second,” and “third” are the top-to-bottom inputs of the plotter, respectively.

model, then the width of the transition changes, as shown in the lower plot. This is because the second Integrator affects the step size taken by the solver. In order to achieve adequate integration accuracy, the second Integrator forces a smaller step size in the vicinity of time 1.0.

The problem is not solved by changing the expression to

```
(in >= 1.0) ? in + 1 : 0
```

Here, the transition from zero to non-zero occurs when the input is greater than *or equal to* 1.0. In this case, the resulting signal is not continuous from the left. The resulting plots are identical. The Expression actor simply calculates a specified function of its inputs when it fires. It has no mechanism for generating distinct values at distinct microsteps unless its input already has distinct values at distinct microsteps.

The previous example shows that using an actor whose output is a discontinuous function of the input can create problems if the input is a continuous-time signal. The next few sections describe various mechanisms for properly constructing discontinuous signals. The particular problem with Figure 9.10 is solved using modal models in Section 9.3.1.

Sidebar: Probing Further: Discrete Sets

A set D is a **discrete set** if it is a totally ordered set (for any two elements d_1 and d_2, either $d_1 \leq d_2$ or $d_1 > d_2$) where there exists a one-to-one function $f\colon D \to \mathbb{N}$ that is **order preserving**. Order preserving simply means that for all $d_1, d_2 \in D$ where $d_1 \leq d_2$, we have that $f(d_1) \leq f(d_2)$. The existence of such a one-to-one function ensures that we can arrange the elements of D *in temporal order*. Notice that D is a countable set, but not all countable sets are discrete. For example, the set $\mathbb{Q}$ of rational numbers is countable but not discrete. There is no such one-to-one function.

9.2.2 Discrete-Event Signals in the Continuous Domain

As described earlier, the Continuous domain supports genuinely discrete signals, which are signals that are present only at particular instants. As a consequence, the clock actors that are used in the DE domain (see sidebar on page 241) can be used in the continuous domain.

Example 9.4: The ContinuousClock actor in Figure 9.4 is a composite actor that uses a DiscreteClock and a ZeroOrderHold (see box on page 338), as shown in Figure 9.12. The DiscreteClock produces a discrete-event signal and the ZeroOrderHold converts that signal to a continuous-time signal (see sidebar on page 338).

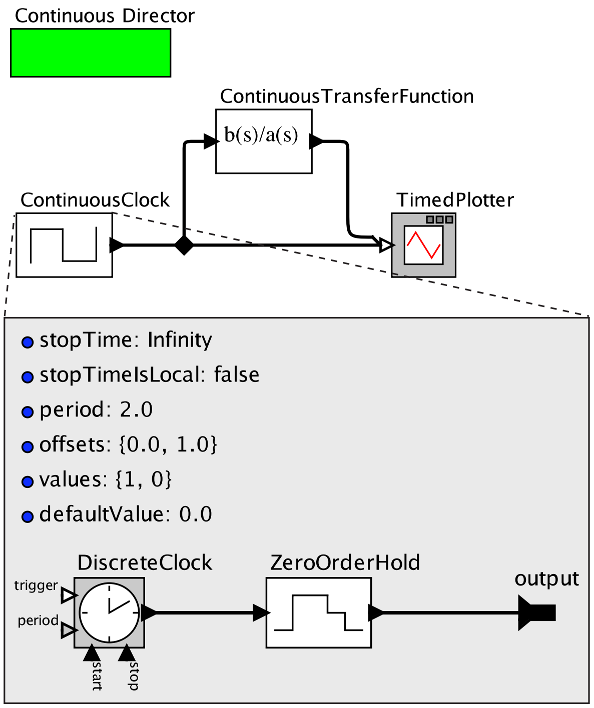


Figure 9.12: The ContinuousClock actor in this model is a composite actor that uses a DiscreteClock and a ZeroOrderHold.

Both signals are piecewise continuous. The output signal from DiscreteClock at the model time of each event is characterized as follows: it is absent at microstep zero (which matches its value at times just before the event); present at microstep one (which is the discrete event); and absent again at microsteps two and higher (which matches its value at larger times until the next discrete event). Therefore, the output of DiscreteClock is piecewise continuous, as required by the solver.

The clock actors described in the sidebar on page 241 all behave in a manner that is similar to DiscreteClock and hence they can all be used in the continuous domain.

Note that although many actors that operate on or produce discrete events have a *trigger* input port, there is rarely a reason to connect that port in the Continuous domain. In the DE domain, a *trigger* port is used to trigger execution of an actor at the time of the input event. But in the Continuous domain, every actor executes every time that there is an execution. Nevertheless, it is sometimes useful to use the *trigger* port, as illustrated by the example in Figure 9.13.

9.2.3 Resetting Integrators at Discrete Times

In addition to its signal input and output, the Integrator actor has two extra ports at the bottom of the icon. The one at the lower right is a PortParameter called *initialState*. When an input token is provided on that port, the state of the Integrator will be reset to the value of the token. The output of the Integrator will change instantaneously to the specified value.

Example 9.5: The model in Figure 9.14 uses a DiscreteClock actor to periodically reset an Integrator.

The input events on the *initialState* port are required to be purely discrete. This means that at all model times, the input signal must be absent at microstep 0. Any attempt to feed a continuous signal into this port results in an exception similar to the one below:

```
IllegalActionException:  Signal at the initialState port is not purely
discrete.
```

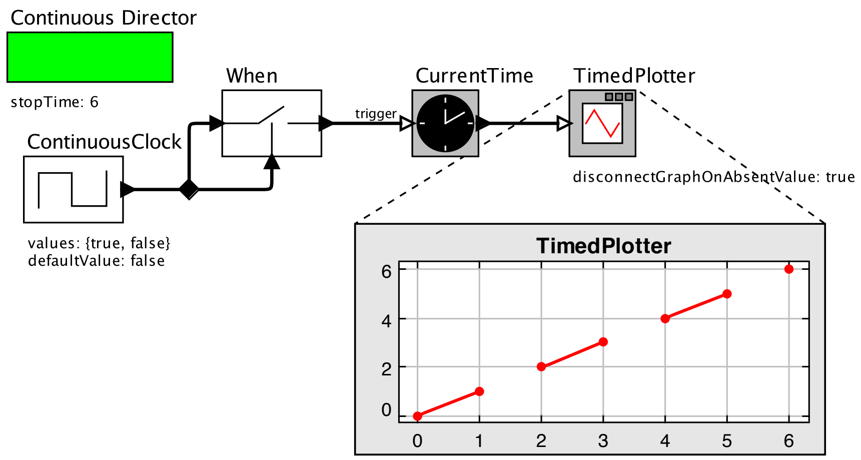


Figure 9.13: The *trigger* port of the CurrentTime actor in this model is used to turn on and off its output. During the time intervals where the output of the DiscreteClock actor is `false`, the CurrentTime actor is disabled, and hence its output will be absent. [online]

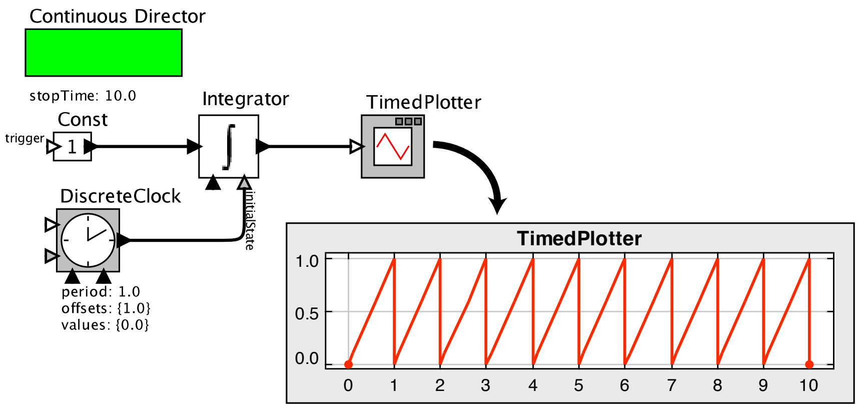


Figure 9.14: Illustration of the use of the *initialState* port of the Integrator actor. [online]

```
    in Integrator
```

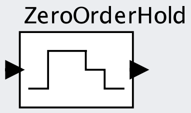
This check ensures that the output of the Integrator is piecewise continuous. In Figure 9.14, for example, at the points of discontinuity the signal first takes the value of the Integrator state prior to applying the reset. In the subsequent microstep, it takes the value after the reset. In contrast, if the output of the Integrator had changed abruptly at microstep zero, then the output could have been a different value at a time infinitesimally earlier — thus violating the requirement for piecewise continuity.

9.2.4 Dirac Delta Functions

The other Integrator input port is called *impulse*. Like the *initialState* port discussed in the previous section, the signal at this port is required to be purely discrete. When an impulse event arrives, it causes an instantaneous increment or decrement of the state (and output)

Sidebar: Continuous Signals from Discrete Events

The **ZeroOrderHold** actor, shown below, takes a discrete-event signal in and produces a continuous-time signal on its output:



This actor is in `DomainSpecific→Continuous→Discrete to Continuous`.

At times between input events, the value of the output is the value of the most recent event, so the output is piecewise constant. At the time of each input event, the output at microstep zero is the value of the previous event, and at microstep one, it takes the value of the current event. Hence, the output signal is piecewise continuous.

It may seem desirable to define an actor that interpolates between the values of the input events, as is done by the Waveform actor (see sidebar on page 326). However, in order to interpolate, the actor would have to know the value of a *future* event. Actors in the continuous domain are required to be **causal**, meaning that their outputs depend only on current and past inputs. The outputs cannot depend on future inputs. Hence, no such interpolation is possible. The Waveform actor is able to perform interpolation because the values that it is interpolating are specified as parameters, not as input events.

of the Integrator. That is, rather than resetting the state to a specified value, it adds to (or subtracts from) the current state.

Mathematically, such functionality is often represented as a **Dirac delta function** in signals and systems. A Dirac delta function is a function $\delta\colon \mathbb{R} \to \mathbb{R}^+$ given by

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

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. At $t = 0$, therefore, its value cannot be finite. Any finite value would yield an integral of zero. This is indicated by $\mathbb{R}^+$ in the form of the function, $\delta\colon \mathbb{R} \to \mathbb{R}^+$, where $\mathbb{R}^+$ represents the **extended reals**, which includes infinity. Dirac delta functions are widely used in modeling continuous-time systems (see Lee and Varaiya (2011), for example), so it is important to be able to include them in simulations.

Suppose that a signal y has a Dirac delta function occurring at time t_1 as follows,

$$y(t) = y_1(t) + K\delta(t - t_1),$$

where y_1 is an ordinary continuous-time signal, and K is a scaling constant. Then

$$\int_{-\infty}^{t} y(\tau) d\tau = \begin{cases} \int_{-\infty}^{t} y_1(\tau) d\tau & t < t_1 \\ K + \int_{-\infty}^{t} y_1(\tau) d\tau & t \geq t_1 \end{cases}$$

The component $K\delta(t - t_1)$ is a Dirac delta function at time t_1 with weight K, and it causes an instantaneous increment in the integral by K at time $t = t_1$.

Example 9.6: LTI systems can be characterized by their impulse response, which is their response to a Dirac delta function. The model in Figure 9.15 is an LTI system with transfer function

$$H(s) = \frac{1}{1 + as^{-1} + bs^{-2}}.$$

The model provides a Dirac delta function at time 0.2, producing the impulse response shown in the plot.

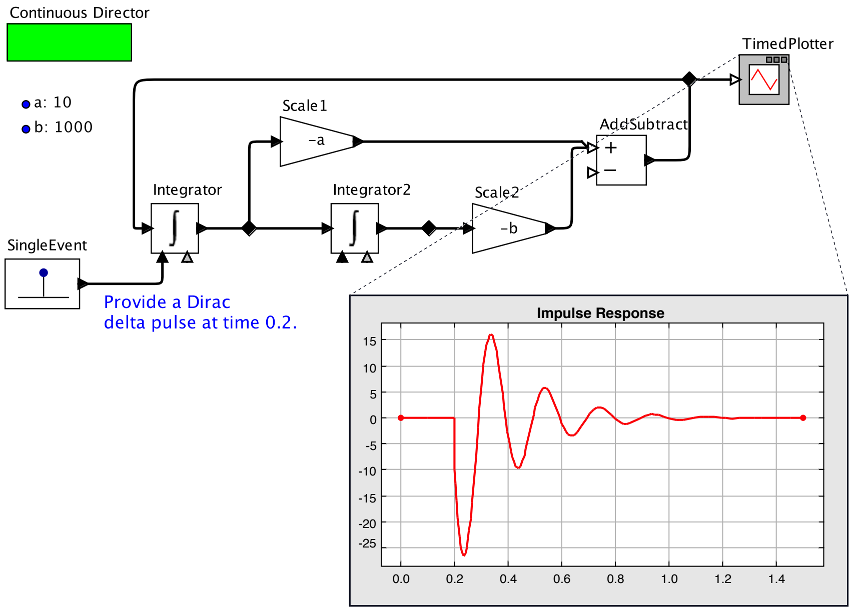
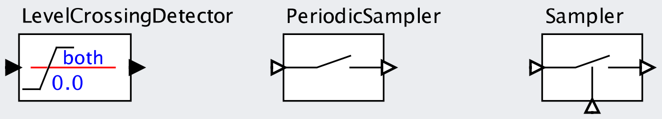


Figure 9.15: Response of an LTI system to a Dirac delta function. [online]

It is difficult to model Dirac delta functions in computing systems because of their instantaneous and infinite nature. The superdense time model of Ptolemy II coupled with the semantics of the Continuous domain provide a rigorous, unambiguous model that can support Dirac delta functions.

Sidebar: Generating Discrete Events

Several actors convert continuous-time signals to discrete-event signals (these actors are located in `DomainSpecific→Continuous→Continuous to Discrete`):



- **LevelCrossingDetector** converts continuous signals to discrete events when the input signal crosses a threshold specified by the *level* parameter. A *direction* parameter constrains the actor to detect only rising or falling transitions. This actor introduces a one-microstep delay before it produces an output. That is, when a level crossing is detected, the actor requests a refiring in the next microstep at the current time, and produces the output during that refiring. This ensures that the output satisfies the piecewise continuity constraint; it is always absent at microstep 0. The one-microstep delay enables the actor to be used in a feedback loop. An example is shown in Figure 9.16, where the feedback loop resets the Integrator each time it reaches a threshold (1.0 in the example).
- **PeriodicSampler** generates discrete events by periodically sampling an input signal. The sampling rate is given by a parameter. By default, the actor reads the initial value of the input signal (the input value at microstep 0), but sends it to the output port one microstep later (at microstep 1). This ensures that the output at microstep 0 is always absent, thus ensuring that the output signal is piecewise continuous. (The input is absent prior to the sample time, so piecewise continuity requires that it be absent at microstep 0 at the sample time.) Because of the one-step delay, the PeriodicSampler can also be used in a feedback loop. For example, it can be used to periodically reset an Integrator, as shown in the example in Figure 9.17.
- **Sampler** is a simple actor. Whenever the *trigger* signal (at the bottom port on the icon) is present, it copies the input from the left port to the output. There is no microstep delay. If the signal at the *trigger* port is a piecewise continuous discrete-event signal, then the output will also be a piecewise continuous discrete-event signal. Sampler will normally read its inputs at microstep 1 because the *trigger* input is discrete. (PeriodicSampler will behave in the same way if its *microstep* parameter is set to 1.)

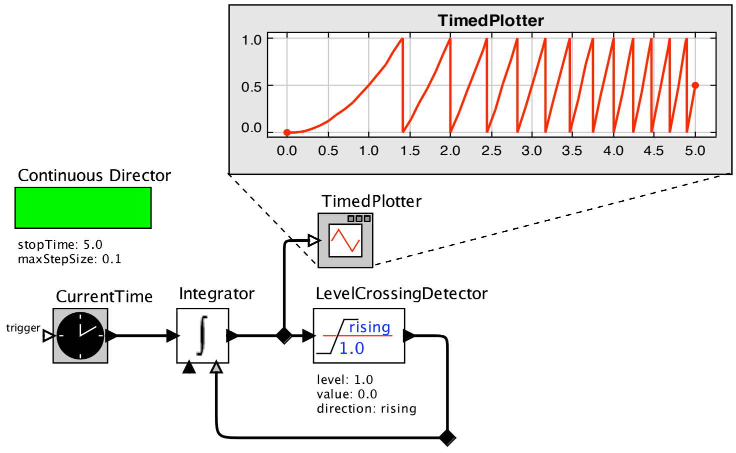


Figure 9.16: Illustration of the LevelCrossingDetector, which can be put in a feedback loop. In this case, whenever the output of the Integrator reaches 1.0, it is reset to zero. [online]

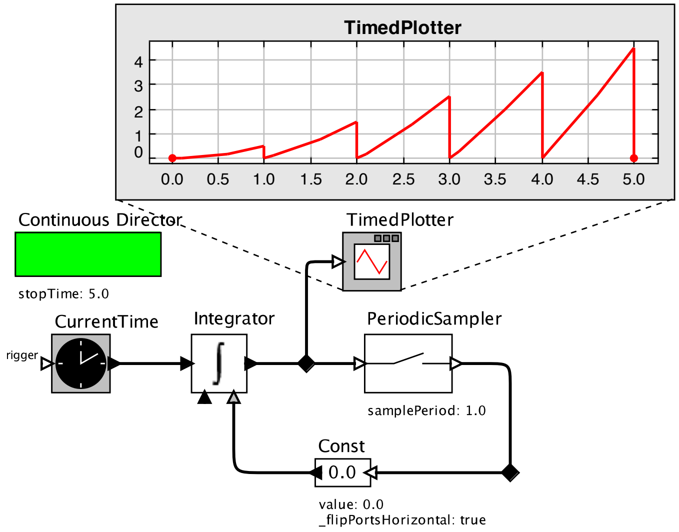


Figure 9.17: The PeriodicSampler actor, placed in a feedback loop. In this case, the Integrator will be reset to zero at intervals of one time unit regardless of its state. [online]

9.2.5 Interoperating with DE

The Continuous and DE domains both support discrete-event signals. There is a subtle but important difference between these domains, however. In the Continuous domain, at a time stamp selected by the solver, *all* actors are fired. In the DE domain, an actor is only fired if either it has an event at an input port or it has previously asked to be fired. As a consequence, DE models can be much more efficient, particularly when events are sparse.

It can be useful to build models that combine the two domains. Such combinations are suitable for many cyber-physical systems, for example, which combine continuous dynamics with software-based controllers. Constructing models with a mixture of Continuous and DE domains is easy, as illustrated by the following example.

Example 9.7: Consider the model in Figure 9.18. The top level of the model is implemented in the DE domain, and includes an opaque composite actor that is a Continuous model. This example models a "job shop," where job arrivals are discrete events, the processing rate is given by an exponential random variable, and the job processing is modeled in continuous time.

The model assigns an integer number to each job. It then approaches that number with a slope given by the (random) rate. The higher the rate, the faster the job is completed. The job is complete when the blue (dashed) line in the plot reaches the red (solid) line in the lower plot. The upper plot shows the times at which each job is generated and completed. Note that this model has a feedback loop such that each time a job is finished, a new one is started with a new service time.

This example is somewhat contrived, however, in the sense that it does not actually require the use of the Continuous domain (see Exercise 4). In fact, models where continuous-time signals linearly increase or decrease can usually be realized within the DE domain alone, without the need for a solver. That said, there is still value in constructing the mixed-domain model because it can easily evolve to support more complex dynamics in the Continuous portion.

Continuous models can be placed within DE models, as shown in the previous example. Conversely, DE models can be placed within Continuous models. The choice of top level domain is often determined by emphasis. If the emphasis is on a discrete controller, then

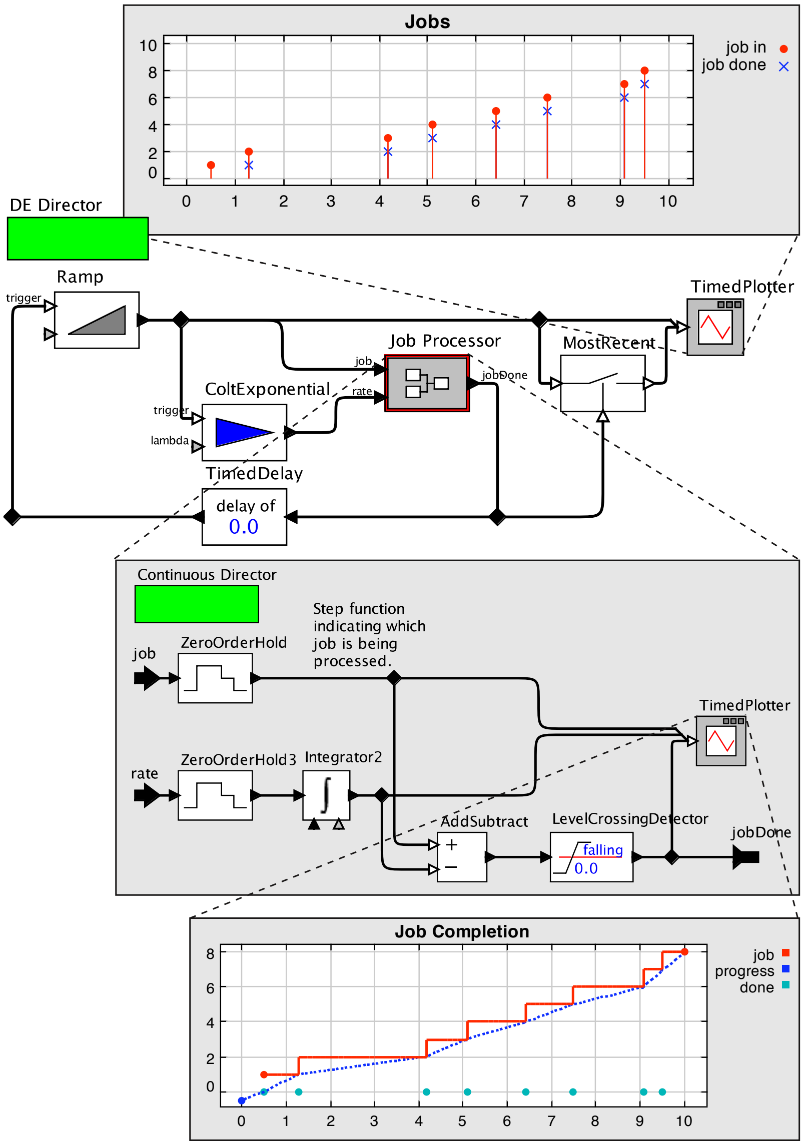


Figure 9.18: Illustration of a hierarchical combination of DE and Continuous models, as considered in Example 9.7. [online]

using DE at the top level often makes sense. If the emphasis is on the physical plant, then using Continuous at the top level may be better.

If multiple Continuous submodels are placed within a DE model, then the solvers in the submodels are decoupled. This can be useful for modeling systems with widely disparate time scales; one solver can use small step sizes without forcing the other submodel to use small step sizes.

The ability to combine DE and Continuous domains relies on a key property of DE, which is that events in DE normally occur at microstep one, not at microstep zero. When these events cross the boundary into a Continuous model, they preserve this microstep. Hence, a signal that passes from the DE domain to the Continuous domain will normally be absent at microstep zero, thus ensuring piecewise continuity. When a signal goes from the Continuous domain to the DE domain, however, it is important that the signal be discrete, as would be produced by a Sampler or LevelCrossingDetector (see page 341).

9.2.6 Fixed-Point Semantics

Recall from Section 7.3.4 that, as of this writing, the DE director in Ptolemy II implements an approximation of the fixed-point semantics described by Lee and Zheng (2007). In contrast, the Continuous director implements an exact fixed-point semantics, and can therefore execute some models that DE cannot.

Example 9.8: Consider the model shown in Figure 9.19. This model is identical to the model considered in Example 7.14, except that the Continuous director is used instead of the DE director. The Continuous director, unlike the DE director, is able to fire actors multiple times at a given time stamp. As a consequence, it does not need to know whether an event is present or absent at the input of the composite actor before it is fired. The director can fire the composite actor, obtain an event from the DiscreteClock, and then later fire the composite actor again once that event has been fed back.

9.3 Hybrid Systems and Modal Models

A **hybrid system** is a model that combines continuous dynamics with discrete mode changes. Such models are created in Ptolemy II using ModalModel actors, found in the `Utilities` library and explained in Chapter 8. This section starts by examining a prebuilt hybrid system, and concludes by explaining the principles that make hybrid models work. Chapter 8 explains how to construct such models, and explains how time is handled in mode refinements.

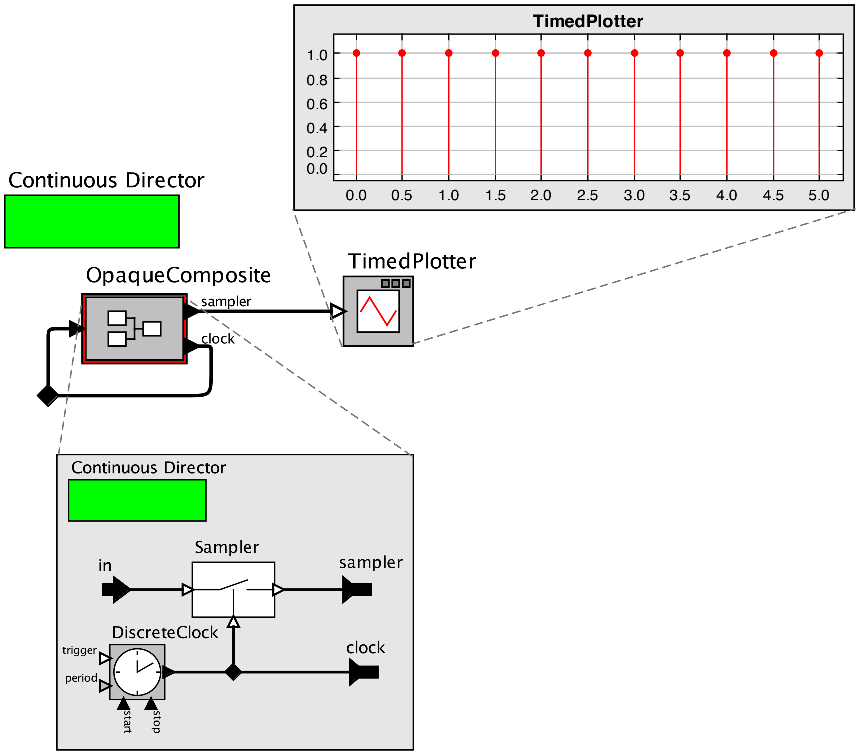


Figure 9.19: A discrete-event model that is executable using the Continuous director, but not using the DE director, as shown in Example 7.14. [online]

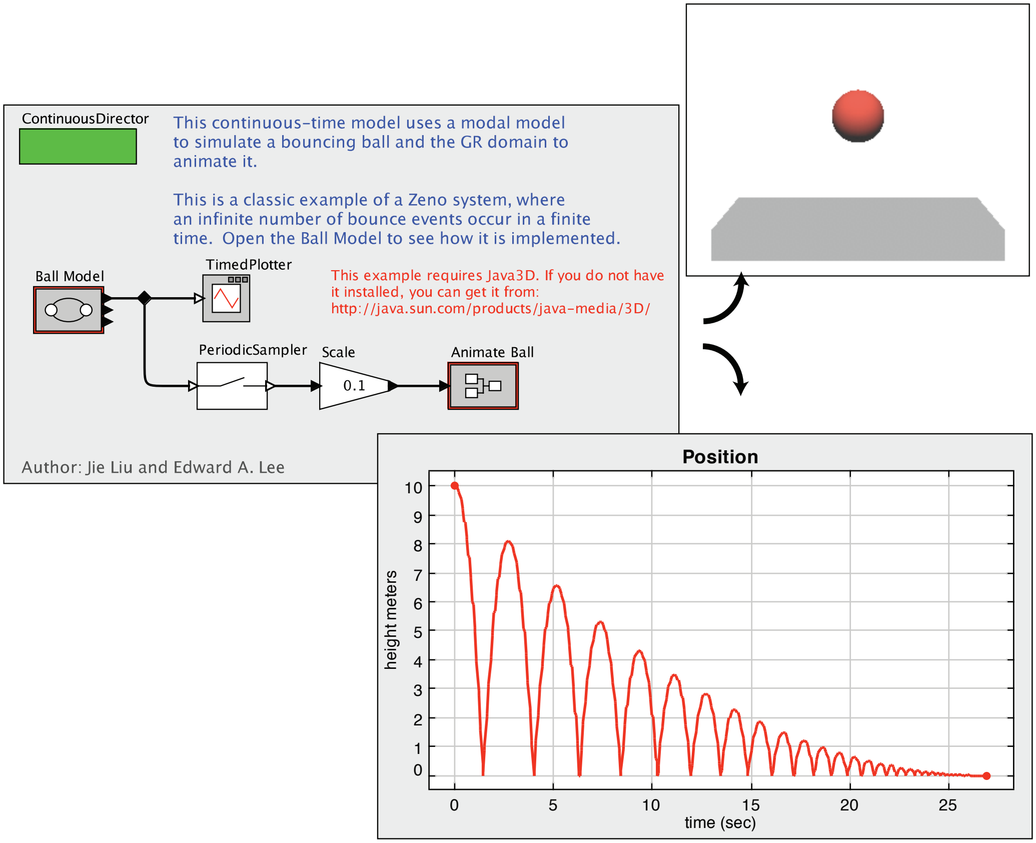

ContinuousDirector
This continuous-time model uses a modal model to simulate a bouncing ball and the GR domain to animate it.
This is a classic example of a Zeno system, where an infinite number of bounce events occur in a finite time. Open the Ball Model to see how it is implemented.
Ball Model
TimedPlotter
This example requires Java3D. If you do not have it installed, you can get it from: http://java.sun.com/products/java-media/3D/
PeriodicSampler
Scale
0.1
Animate Ball
Author: Jie Liu and Edward A. Lee
Position
height meters
time (sec)

Figure 9.20: Top level of the bouncing ball hybrid system example. [online]

Example 9.9: A bouncing ball model is shown in Figure 9.20. It can be found under “Bouncing Ball” in the Tour of Ptolemy II (Figure 2.3, in the “Hybrid Systems” entry). The bouncing ball model uses a ModalModel component named Ball Model. Executing the model yields a plot like that in the figure (along with 3-D animation that is constructed using the GR (graphics) domain, which is not covered here). This model has continuous dynamics during times when the ball is in the air, and discrete events when the ball hits the surface and bounces.

Figure 9.21 shows the contents of Ball Model, which is a modal model with three states: init, free, and stop. During the time a modal model is in a state, its behavior is specified by the mode refinement. In this case, only the free state has a refinement,

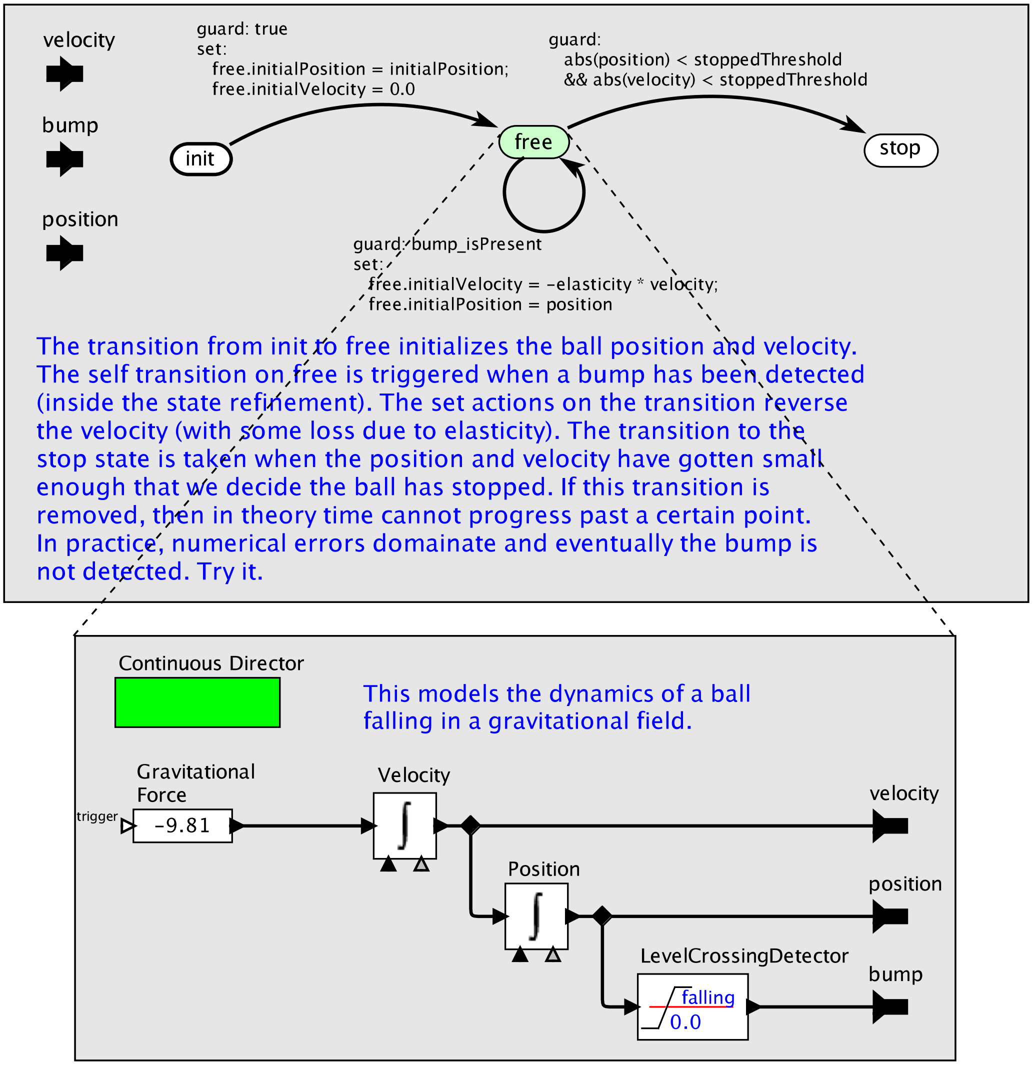

velocity
bump
position
guard: true
set:
free.initialPosition = initialPosition;
free.initialVelocity = 0.0
guard:
abs(position) < stoppedThreshold
&& abs(velocity) < stoppedThreshold
init
free
stop
guard: bump_isPresent
set:
free.initialVelocity = -elasticity * velocity;
free.initialPosition = position
The transition from init to free initializes the ball position and velocity. The self transition on free is triggered when a bump has been detected (inside the state refinement). The set actions on the transition reverse the velocity (with some loss due to elasticity). The transition to the stop state is taken when the position and velocity have gotten small enough that we decide the ball has stopped. If this transition is removed, then in theory time cannot progress past a certain point. In practice, numerical errors domainate and eventually the bump is not detected. Try it.
Continuous Director
This models the dynamics of a ball falling in a gravitational field.
Gravitational Force
trigger
-9.81
Velocity
velocity
Position
position
LevelCrossingDetector
falling
0.0
bump

Figure 9.21: Inside the Ball Model of Figure 9.20.

shown at the bottom of Figure 9.21. The *init* state is the initial state, which is used only for its outgoing transition, and has set actions to initialize the ball model. Specifically, the transition is labeled as follows:

```
guard: true
set:
 free.initialPosition = initialPosition;
 free.initialVelocity = 0.0
```

The first line is a guard, which is a predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value `true`. Thus, the model immediately transitions to mode *free*. This transition occurs in microstep zero at the start of the execution. The "set:" line indicates that the successive lines define set actions (see Section 6.2). The third and fourth lines set the parameters of the destination mode *free*. The *free* state represents the mode of operation when the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

When the model begins executing, it is in the *init* state. Since the *init* state has no refinement, the outputs of the Ball Model will be absent while the modal model is in that state. The outgoing transition has a guard that is always enabled, so the Ball Model will be in that state for only one microstep.

Inside the *free* state, the refinement represents the law of gravity, which states that an object of any mass will have an acceleration of about $9.81 \text{meters}/\text{second}^2$. The acceleration is integrated to find the velocity, which is, in turn, integrated to find the vertical position. In the refinement, a LevelCrossingDetector actor is used to detect when the vertical position of the ball is zero. Its output produces events on the (discrete) output port *bump*. Figure 9.21 shows that this event triggers a state transition back to the same *free* state, but now the *initialVelocity* parameter is changed to reverse the sign and attenuate its value by the *elasticity*. The ball loses energy when it bounces, as shown by the plot in Figure 9.20.

Figure 9.21 shows that when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop*, which has no refinement. At this point, the model produces no further outputs.

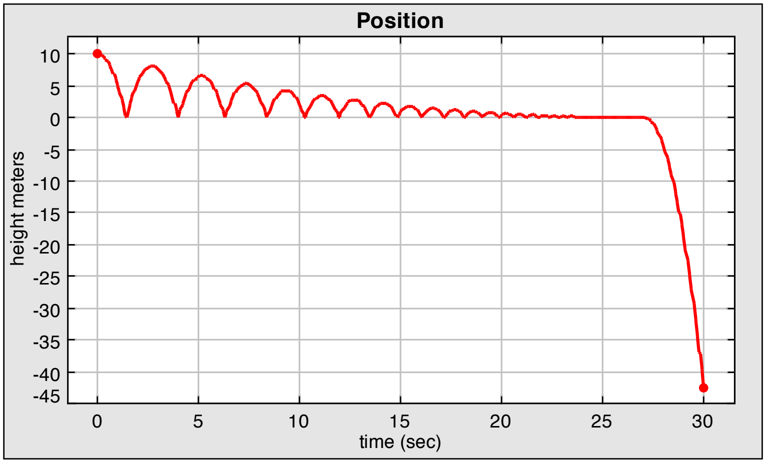


Figure 9.22: Result of running the bouncing ball model without the stop state.

The bouncing ball model illustrates an interesting property of hybrid system modeling. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. Removing the *stop* state from the FSM and re-running the model yields the result shown in Figure 9.22. In effect, the ball falls through the surface on which it is bouncing and then goes into a free-fall in the space below.

The error that occurs here illustrates a fundamental pitfall that can occur with hybrid system modeling. In this case, the event detected by the LevelCrossingDetector actor can be missed by the simulator. This actor works with the solver to attempt to identify the precise point in time when the event occurs. It ensures that the simulation includes a sample at that time. However, when the numbers become sufficiently small they are dominated by numerical errors, and the event is missed.

The bouncing ball is an example of a Zeno model (see Section 7.4). The time between bounces gets smaller as the simulation progresses, and it gets smaller fast enough that, with infinite precision, an infinite number of bounce events would occur in a finite amount of time.

9.3.1 Hybrid Systems and Discontinuous Signals

Recall from Example 9.3 that actors whose outputs are a discontinuous function of the input can create signals that are not piecewise continuous. This can result in solver-dependent behavior, in which arbitrary step-size decisions made by the solver strongly affect the execution of the model. These problems can be solved using modal models, as illustrated in the following example.

Example 9.10: Figure 9.23 shows a variant of the model in Figure 9.10 that correctly produces a piecewise continuous signal. This variant uses a ModalModel, which specifies a transition at the discontinuity of the signal. The transitions of a modal model are instantaneous, in that model time does not advance. The microstep, however, does advance. In this model, the transition occurs within the *errorTolerance* (a director parameter) after time 1.0. At the time of the transition, the refinement of the *zero* state fires first, producing output 0 at microstep 0, and then the refinement of the *increment* state fires at microstep 1, producing output 2.0 (or within the *errorTolerance* of 2.0). Hence, the output signal is piecewise continuous.

The operation of ModalModel actors is explained in Chapter 8. When combined with the Continuous director, such operation translates naturally into an effective and useful semantics for hybrid systems. To fully understand the interoperation of modal models and the Continuous domain, it is useful to review the execution semantics of modal models, described in Section 6.2. Specifically, a firing of the modal model consists of firing of the refinement of the current state (if there is one), evaluating the guards, and taking a transition if a guard is true. It is also important to understand that while a mode is inactive, time does not advance in the refinement. Thus, the local notion of time within a refinement lags the global notion of time in its environment.

In modal models, transitions are allowed to have output actions (see Section 6.2). Such actions should be used with care because the transition may be taken in microstep 0, and the resulting output will not be piecewise continuous. If output actions are used to produce discrete events, the transition must be triggered by a discrete event from a piecewise continuous signal.

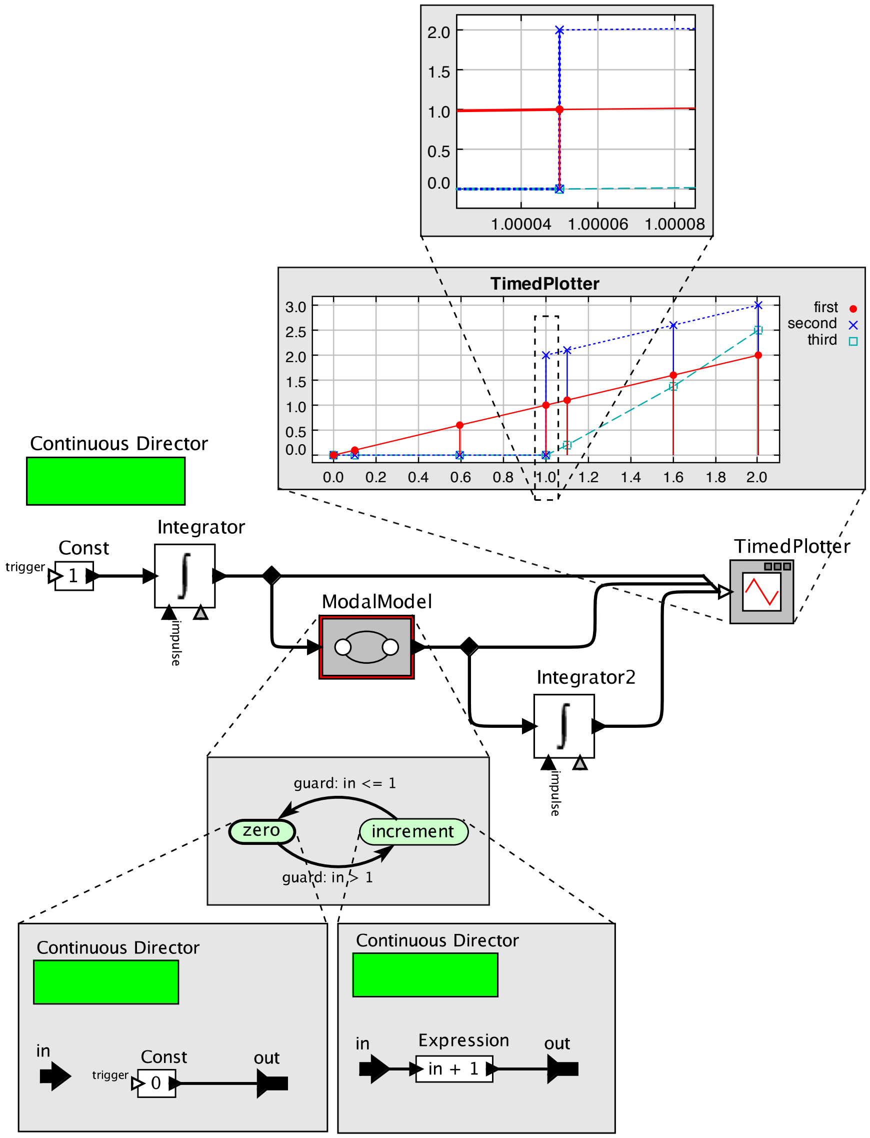


Figure 9.23: A variant of the model in Figure 9.10 that correctly produces a piece-wise continuous signal. [online]

9.4 Summary

Modeling continuous-time systems and approximating their behavior on digital computers can be tricky. The superdense time model of Ptolemy II makes it easier to accurately model a large class of systems, and is particularly useful for systems that mix continuous and discrete behaviors. The Continuous domain, described in this chapter, exploits this model of time to deliver sophisticated modeling and simulation capabilities.

Exercises

1. Let x be a continuous-time signal where $x(0) = 1$ and $\ddot{x}(t) = -x(t)$, where $\ddot{x}$ is the second derivative of x with respect to time t. It is easy to verify that a solution to this equation is $x(t) = \cos(t)$.

   (a) Use Integrator actors to construct this signal x without using any actors or expressions involving trigonometric functions. Plot the execution over some reasonable time to verify that the solution matches what theory predicts.

   (b) Change the solver that the director uses from ExplicitRK23Solver to ExplicitRK45Solver. Describe qualitatively the difference in the results. Which solver gives a better solution? What criteria are you using for "better"? Give an explanation for the differences.

   (c) All numerical ODE solvers introduce errors. Although the theory predicts that the amplitude of the solution $x(t) = \cos(t)$ remains constant for all time, a numerical solver will be unable to sustain this. Describe qualitatively how the ExplicitRK23Solver and ExplicitRK45Solver perform over the long run, leaving other parameters of the director at their default values. Which solver is better? By what criteria?

   (d) Experiment with some of the other director parameters. How does the *errorTolerance* parameter affect the solution? How about *maxStepSize*?

2. Example 9.2 shows the use of ContinuousTransferFunction to specify a transfer function for a continuous-time system. Show that with the parameters given in the example, that the models in Figures 9.4 and 9.6 are equivalent. **Hint:** This problem is easy if you have taken a typical electrical engineering signals and systems class, but it is doable without that if you recognize the following fact: If a signal w has Laplace transform W, then the integral of that signal has Laplace transform W' where for all complex numbers s, $W'(s) = W(s)/s$. That is, dividing by s in the Laplace domain is equivalent to integrating in the time domain.

3. Consider the Lorenz attractor in Example 9.1. Implement the same system using the DifferentialSystem higher-order actor. Give the parameter names and values for your DifferentialSystem.

4. The model in Example 9.7 does not actually require the Continuous domain to achieve the same functionality. Construct an equivalent model that is purely a DE model.