HYVISUAL: A HYBRID SYSTEM VISUAL MODELER

Authors[1]: Edward A. Lee
Jie Liu
Xiaojun Liu
Haiyang Zheng

Version 2.2-beta[2]
UCB ERL Memorandum Number FIXME
January 28, 2003

FIXME: graphic here

1. With contributions from the entire Ptolemy II team, but most especially Christopher Hylands, Steve Neuendorffer, John Reekie, and Yuhong Xiong.
2. This is the first version of HyVisual, but the version number is selected to match the version of Ptolemy II on which it is based.

1. Introduction

The Hybrid System Visual Modeler (HyVisual) is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid systems. Hybrid systems mix continuous-time dynamics, discrete events, and discrete mode changes. This visual modeler supports construction of hierarchical hybrid systems. It uses a block-diagram representation of ordinary differential equations (ODEs) to define continuous dynamics, and allows mixing of continuous-time signals with events that are discrete in time. It uses a bubble-and-arc diagram representation of finite state machines to define discrete behavior driven by mode transitions.

In this document, we describe how to graphically construct models and how to interpret the resulting models. HyVisual provides a sophisticated numerical solver that simulates the continuous-time dynamics, and effective use of the system requires at least a rudimentary understanding of the properties of the solver. This document provides a tutorial that will enable the reader to construct elaborate models and to have confidence in the results of a simulation of those models. We begin by explaining how to describe continuous-time models of classical dynamical systems, and then progress to the construction of mixed signal and hybrid systems.

The intended audience for this document is an engineer with at least a rudimentary understanding of the theory of continuous-time dynamical systems (ordinary differential equations and Laplace transform representations), who wishes to build models of such systems, and who wishes to learn about hybrid systems and build models of hybrid systems.

HyVisual is built on top of Ptolemy II, a framework supporting the construction of such domain-specific tools. See http://ptolemy.eecs.berkeley.edu for information about Ptolemy II.

1.1 Quick Start

To start HyVisual, click on a Web Start link on the page supporting the web edition:

```
        http://ptolemy.eecs.berkeley.edu/hyvisual/
```

Once you have done this once, then you can select *HyVisual* from the Ptolemy II entry in the Start menu (if you are using a Windows system). You can also start HyVisual on the command line, if you have a command-line oriented computer system, by typing

```
      vergil -hyvisual
```

In all cases, you should see an initial welcome window that looks something like the one in figure 1. Feel free to explore the links in this window.

To create a new model, invoke the New command in the File menu. But before doing this, it is worth understanding how a model works.

2. Continuous-Time Dynamical Systems

In this section, we explain how to read, construct and execute models of continuous-time systems. We begin by examining a demonstration system that is accessible from the welcome window in figure 1, the Lorenz attractor.

2.1 Executing a Pre-Built Model

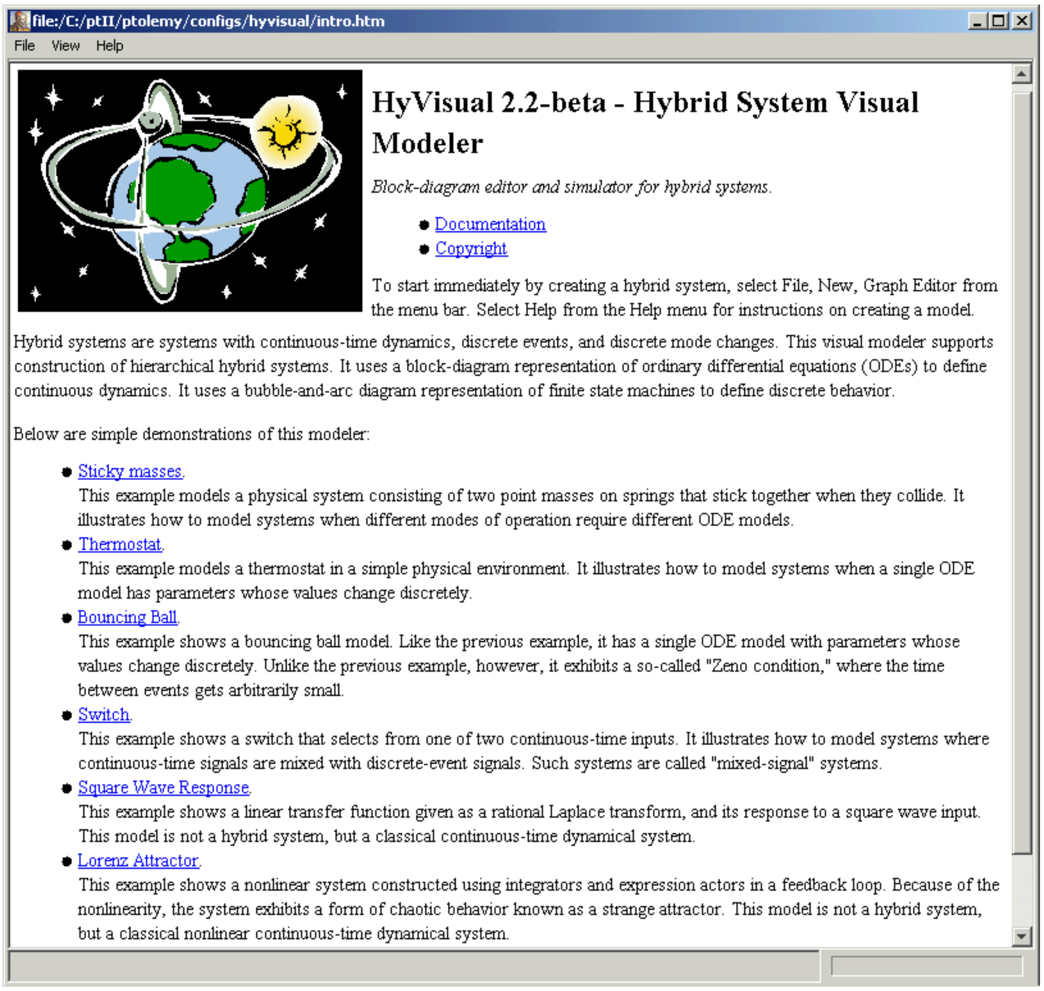
The Lorenz attractor model can be accessed by clicking on the link in the welcome window, which results in the window shown in figure 2. It is a block diagram representation of a set of nonlinear ordinary differential equations. The blocks with integration signs in their icons are integrators. At any given time t, their output is given by

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau) d\tau , \tag{1}$$

where $x(t_0)$ is the initial state of the integrator, t_0 is the start time of the model, and $\dot{x}$ is the input signal. Note that since the output is the integral of the input, then at any given time, the input is the derivative of the output,

$$\dot{x}(t) = \frac{d}{dt} x(t) . \tag{2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use.



HyVisual 2.2-beta - Hybrid System Visual Modeler

Block-diagram editor and simulator for hybrid systems.

- Documentation
- Copyright

To start immediately by creating a hybrid system, select File, New, Graph Editor from the menu bar. Select Help from the Help menu for instructions on creating a model.

Hybrid systems are systems with continuous-time dynamics, discrete events, and discrete mode changes. This visual modeler supports construction of hierarchical hybrid systems. It uses a block-diagram representation of ordinary differential equations (ODEs) to define continuous dynamics. It uses a bubble-and-arc diagram representation of finite state machines to define discrete behavior.

Below are simple demonstrations of this modeler:

- Sticky masses.
  This example models a physical system consisting of two point masses on springs that stick together when they collide. It illustrates how to model systems when different modes of operation require different ODE models.
- Thermostat.
  This example models a thermostat in a simple physical environment. It illustrates how to model systems when a single ODE model has parameters whose values change discretely.
- Bouncing Ball.
  This example shows a bouncing ball model. Like the previous example, it has a single ODE model with parameters whose values change discretely. Unlike the previous example, however, it exhibits a so-called "Zeno condition," where the time between events gets arbitrarily small.
- Switch.
  This example shows a switch that selects from one of two continuous-time inputs. It illustrates how to model systems where continuous-time signals are mixed with discrete-event signals. Such systems are called "mixed-signal" systems.
- Square Wave Response.
  This example shows a linear transfer function given as a rational Laplace transform, and its response to a square wave input. This model is not a hybrid system, but a classical continuous-time dynamical system.
- Lorenz Attractor.
  This example shows a nonlinear system constructed using integrators and expression actors in a feedback loop. Because of the nonlinearity, the system exhibits a form of chaotic behavior known as a strange attractor. This model is not a hybrid system, but a classical nonlinear continuous-time dynamical system.

FIGURE 1. Initial welcome window.

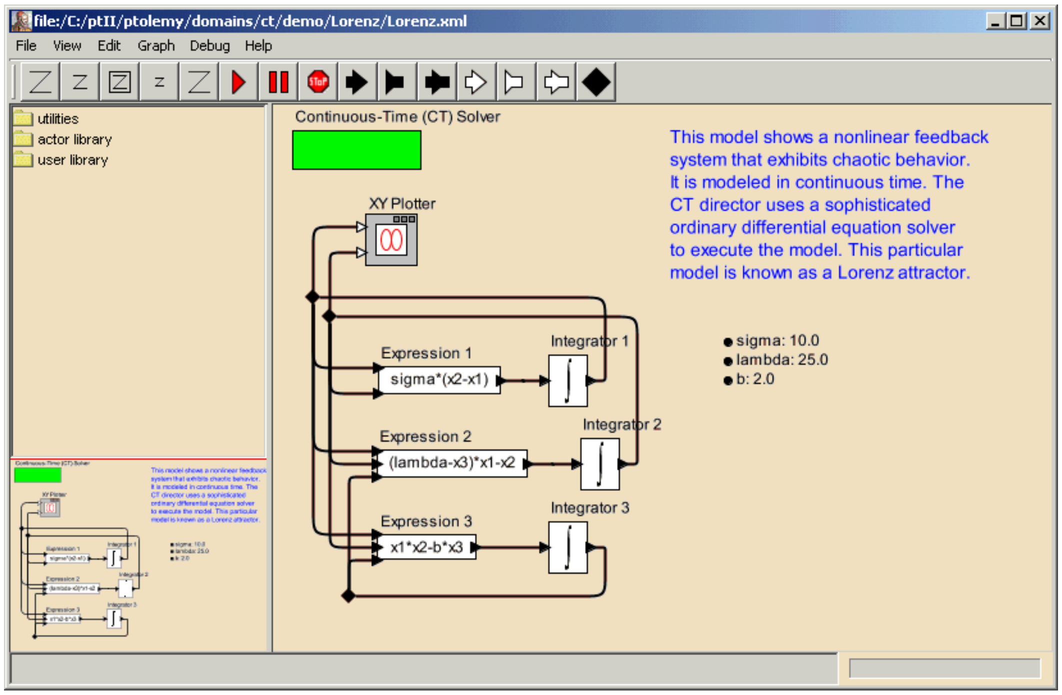
Let the output of the top integrator in figure 2 be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3. Then the equations described by figure 2 are

$$\begin{aligned} \dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t). \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t) \end{aligned} \qquad (3)$$

For each equation, the expression on the right 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 names of the input ports are not shown in the diagram, but if you linger over them with the mouse cursor, the name will pop up in a tooltip. 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 on the right.

The integrators each also have initial values, which you can examine and change by double clicking on the corresponding integrator icon. These define the initial values of x_1, x_2, and x_3, respectively. For this example, all three are set to 1.0.

The Continuous-Time (CT) Solver, shown at the upper right, manages a simulation of the model. It contains a sophisticated ODE solver, and to use it effectively, you will need to understand some of its parameters. The parameters are accessed by double clicking on solver box, which results in the dialog shown in figure 3. The simplest of these parameters are the *startTime* and the *stopTime*, which are self-explanatory. They define the region of the time line over which a simulation will execute.



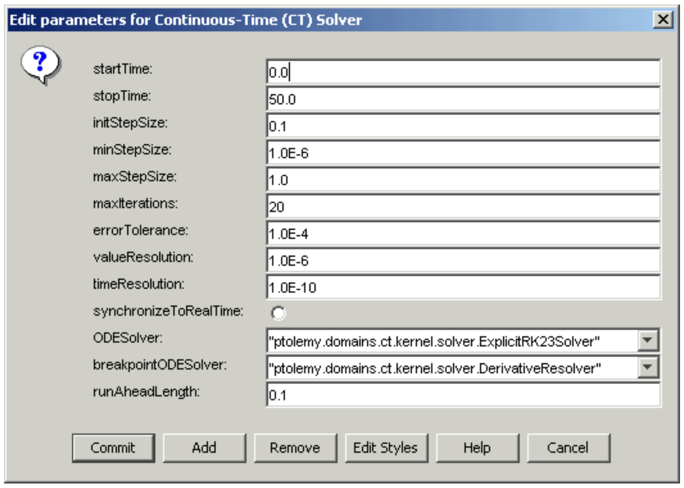
This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

- sigma: 10.0
- lambda: 25.0
- b: 2.0

FIGURE 2. A block diagram representation of a set of nonlinear ordinary differential equations.

To execute the model, you can click on the run button in the toolbar (with a red triangle icon), or you can open the Run Window in the View menu. In the former case, the model executes, and the results are plotted in their own window, as shown in figure 4. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*. The Run Window obtained via the View menu is shown in figure 5.

Like the Lorenz model, a typical continuous-time model contains integrators in feedback loops, or more elaborate blocks that realize linear and non-linear dynamical systems given abstract mathematical representations of them (such as Laplace transforms). In the next section, we will explore how to build a model from scratch.


Edit parameters for Continuous-Time (CT) Solver

startTime:	0.0
stopTime:	50.0
initStepSize:	0.1
minStepSize:	1.0E-6
maxStepSize:	1.0
maxIterations:	20
errorTolerance:	1.0E-4
valueResolution:	1.0E-6
timeResolution:	1.0E-10
synchronizeToRealTime:	○
ODESolver:	"ptolemy.domains.ct.kernel.solver.ExplicitRK23Solver"
breakpointODESolver:	"ptolemy.domains.ct.kernel.solver.DerivativeResolver"
runAheadLength:	0.1

FIGURE 3. Dialog box showing solver parameters for the model in figure 2.

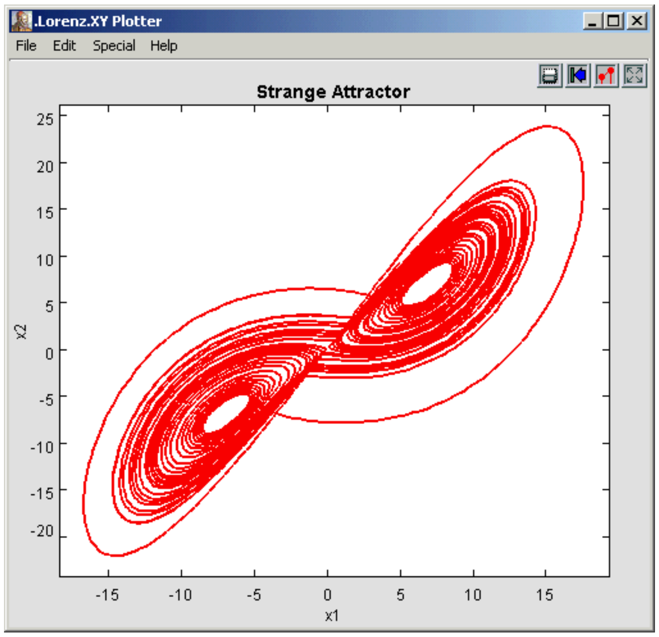
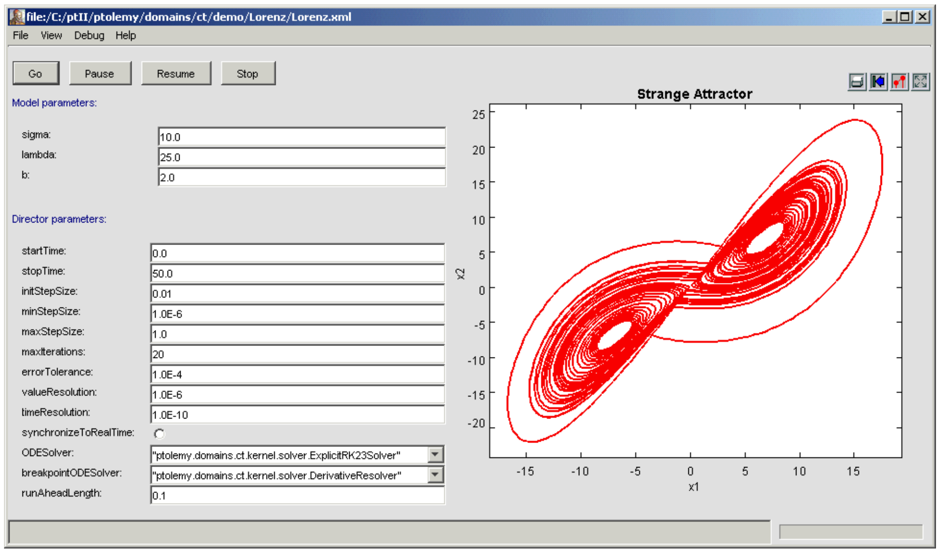


FIGURE 4. Result of running the Lorenz model using the run button in the toolbar.

2.2 Creating a New Model

Create a new model by selecting File, New, and Graph Editor in the welcome window. You should see something like the window shown in figure 6. On the upper left is a library of objects that can be dragged onto the page on the right. These are *actors* (functional blocks) and *utilities* (annotations, hierarchical models, etc.). The page on the right is almost blank, containing only a solver. The lower left corner contains a *navigation area*, which always shows the entire model (which currently consists only



Model parameters:

sigma:	10.0
lambda:	25.0
b:	2.0

Director parameters:

startTime:	0.0
stopTime:	50.0
initStepSize:	0.01
minStepSize:	1.0E-6
maxStepSize:	1.0
maxIterations:	20
errorTolerance:	1.0E-4
valueResolution:	1.0E-6
timeResolution:	1.0E-10
synchronizeToRealTime:	○
ODESolver:	"ptolemy.domains.ct.kernel.solver.ExplicitRK23Solver"
breakpointODESolver:	"ptolemy.domains.ct.kernel.solver.DerivativeResolver"
runAheadLength:	0.1

FIGURE 5. Run Window, obtained via the View menu, for the Lorenz model shown in figure 2.

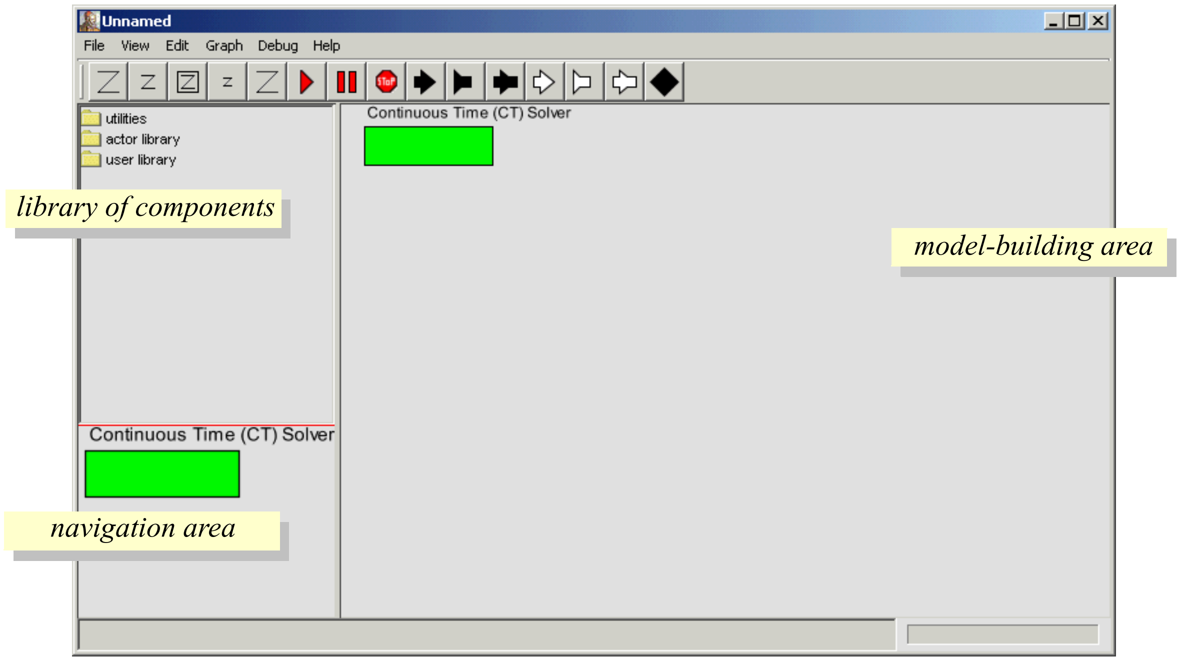


FIGURE 6. A blank model, obtained via File, New, and Graph Editor in the menus.

of a solver). For large models, the navigation area makes it easy to see where you are and makes it easy to get from one part of the model to another.

2.2.1 A Simple Sine Wave Model

We can begin by populating the model with functional blocks. Let's begin with the simple objective of generating and plotting a sine wave. There are a number of ways to do this, and the alternatives illustrate a number of interesting features about HyVisual. Open the *actor library* in the palette, and drag in the *TimedSinewave* actor from the *timed sources* library and the *TimedPlotter* from the *timed sinks* library. Connect the output of the *TimedSinewave* to the input of the *TimedPlotter* by dragging from one port to the other. The result should look something like figure 7.

The model is ready to execute. To execute it, click on the run button in the toolbar, or invoke the Run Window from the view menu. The result of the run should look like figure 8. You can zoom in on the plot by clicking and dragging in the plot window. You can also customize the plot using the buttons at the upper right.

If we zoom in on the plot, turn on stems, and set the marks to "dots," then we can make the plot look like figure 9. In this figure you can see that the sine wave is hardly smooth, and that rather few samples are produced by the simulation. It is worth understanding why this is. Consider the solver parameters shown in figure 3. Notice that the *initStepSize* parameter has value 0.1, which is coincidentally the spacing between samples in figure 9. The spacing between samples is called the *step size* of the solver. If you change *initStepSize* to 0.01 (by double clicking on the solver) and re-run the simulation, then the same region of the plot looks like figure 10. The spacing between samples is now 0.01.

The model shown in figure 7 is atypical of continuous-time models of dynamical systems. It has no blocks that control the step size. Such blocks include those from the *dynamics* and *to discrete* library. For example, another way to get the sine wave to be sampled with a sampling interval of 0.01 is shown in figure 11. The *PeriodicSampler* block has a parameter *samplePeriod* that you can set to

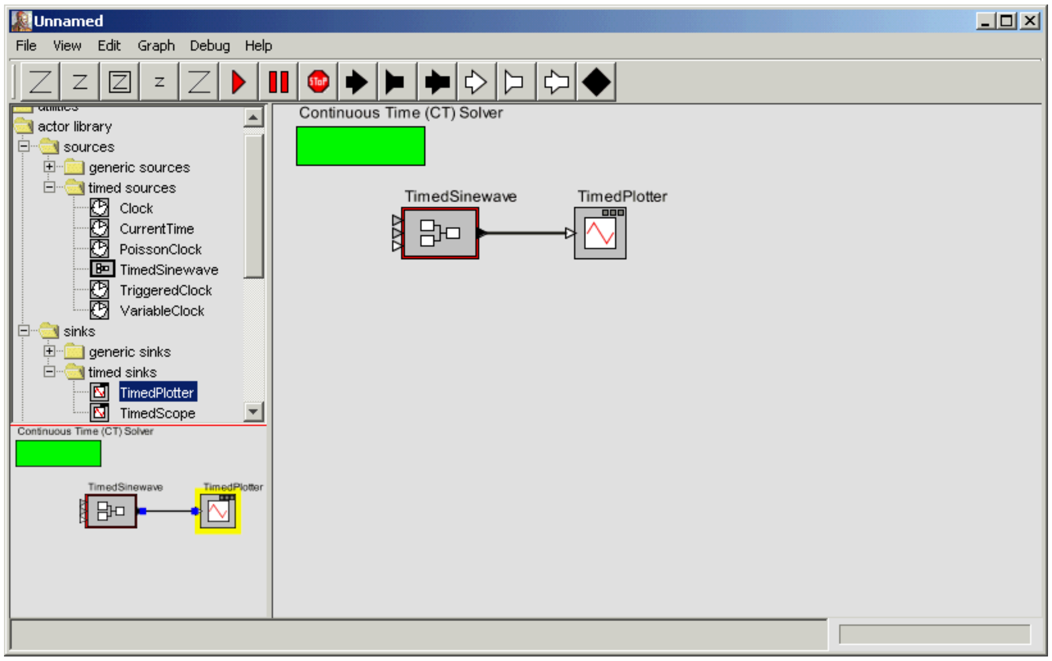


FIGURE 7. A model populated with two actors.

0.01 (by double clicking on the block). This will result in the same plot as shown in figure 10, irrespective of the *initStepSize* parameter of the solver.

The models shown in figures 7 and 11 have no blocks from the *dynamics* library, and hence do not immediately represent an ordinary differential equation. When blocks from the *dynamics* library are used, then the solver uses sophisticated techniques to determine the spacing between samples. The ini-

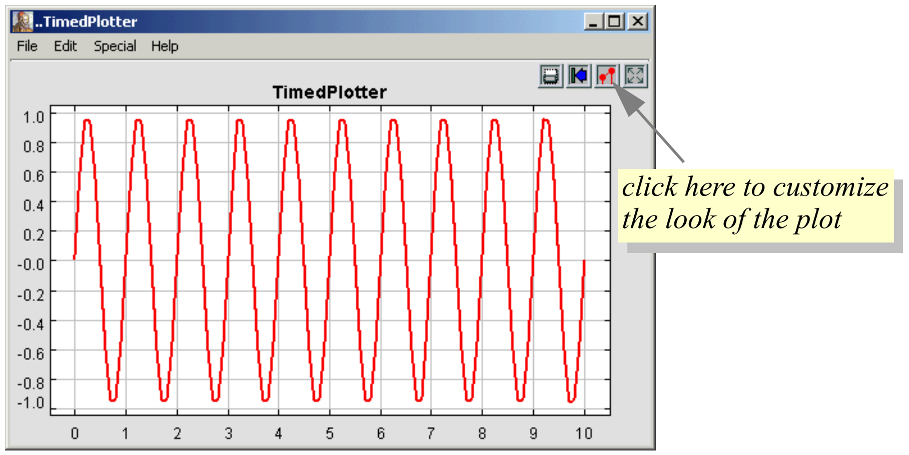


FIGURE 8. Execution of the sine wave example in figure 7, where all parameter values have default.values.

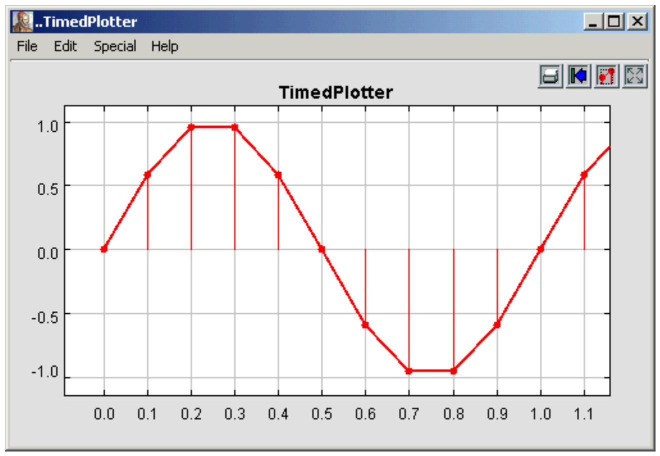


FIGURE 9. Zoomed version of the plot in figure 8, with “dots” and “stems” turned on.

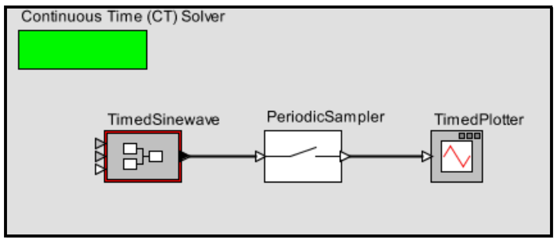


FIGURE 11. Another way to control the step size is to insert a sampler.

tial step size is given by *initStepSize*, but the solver may adjust it to any value between *minStepSize* and *maxStepSize*. In the case of the model in figure 7, there are no blocks with continuous dynamics, and no other blocks that affect the step size and hence there is no basis for the solver to change the step size. Thus, the step size remains at the value given by *initStepSize* for the duration of the simulation.

We will next modify the model to be more typical by describing an ODE whose solution is a sine wave. Before we do that, however, you may want to explore certain features of the user interface:

- You can save your model using commands in the File menu. File names for Ptolemy II models should end in ".xml" or ".moml" so that Vergil will properly process the file the next time you open that file.
- You can obtain documentation for the solver, or any other block in the system, by right clicking on it to get a context menu, and selecting "Get Documentation."
- You can move blocks around by clicking on them and dragging. Connections are preserved.
- You can edit the parameters of any block (including the solver) by either double clicking on it, or right clicking and selecting "Configure."
- You can change the name of a block (or even hide it) by right clicking on the block and selecting "Customize Name."
- If your installation includes the source code, then you can examine the source code for any block by right clicking and choosing "Look Inside."

2.2.2 A Dynamical System Producing a Sine Wave

From the theory of continuous-time dynamical systems, we know that an LTI system with poles on the imaginary axis will produce a sinusoidal output. That is, a system with transfer function of the form

$$H(s) = \frac{\omega_0}{(s - j\omega_0)(s + j\omega_0)} = \frac{\omega_0}{s^2 + \omega_0^2} \tag{4}$$

has an impulse response

$$h(t) = \sin(\omega_0 t)u(t), \tag{5}$$

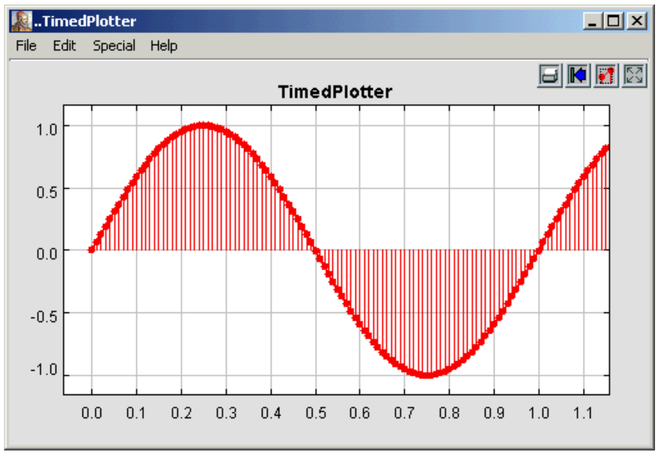


FIGURE 10. The result of running the model in figure 7 with the *initStepSize* parameter of the solver being 0.01.

where $u(t)$ is the unit step function. If the input to this system is a continuous-time signal x and the output is y, then the relationship between the input and output can be described by the differential equation

$$\omega_0 x(t) = \omega_0^2 y(t) + \ddot{y}(t), \tag{6}$$

where $\ddot{y}$ is the second derivative of y. Suppose that the input is zero for all time,

$$\forall t \in \Re, x(t) = 0. \tag{7}$$

Then the output satisfies

$$\ddot{y}(t) = -\omega_0^2 y(t). \tag{8}$$

This output can be generated by the model shown in figure 12. As shown in the annotations in the figure, $\ddot{y}$ is calculated by multiplying y by $-\omega_0$, y is calculated by integrating $\dot{y}$, and $\dot{y}$ is calculated by integrating $\ddot{y}$. If we set the initial state of the left integrator to 1.0 and run the model for 5 time units, we get the result shown in figure 13.

The model in figure 12 shows two additional key features of the user interface, the mechanism for connecting an output to multiple inputs (relations) and the mechanism for defining and using parameters. We discuss these two mechanisms next.

2.2.3 Making Connections

The models in figures 7 and 11 have simple connections between blocks. These connections are made by clicking on one port and dragging to the other. The connection is maintained if either block is moved. We can now explore how to create and manipulate more complicated connections, such as the ones in figure 12, where the output of the right *Integrator* goes to both the *Scale* and the *TimedPlotter* blocks. Such connections are mediated by a *relation*, indicated by a black diamond, as shown in figure 12. A relation can be linked to one output port and any number of input ports.

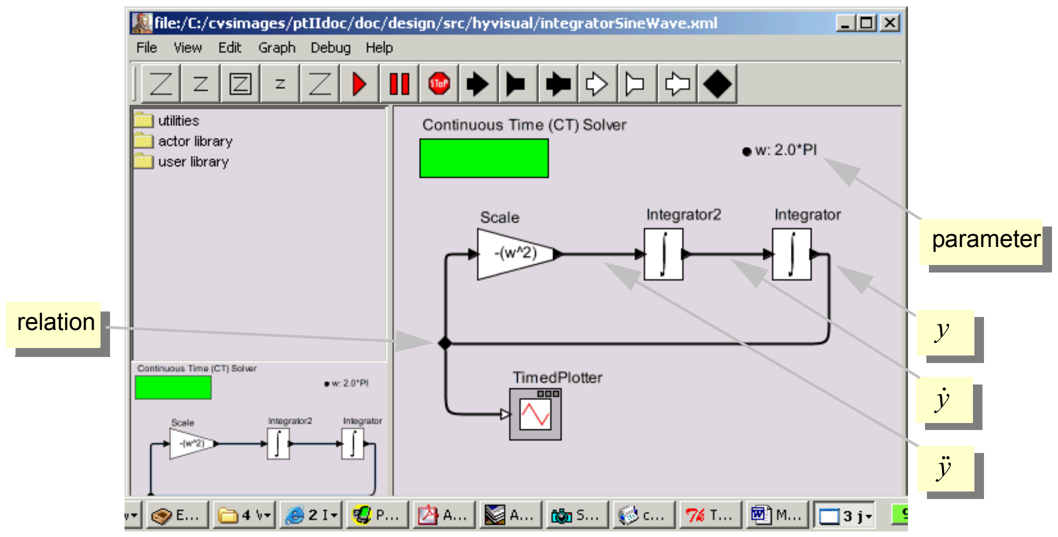

file:/C:/cvsimages/ptIIdoc/doc/design/src/hyvisual/integratorSineWave.xml
File View Edit Graph Debug Help
utilities
actor library
user library
Continuous Time (CT) Solver
w: 2.0*PI
Scale
-(w^2)
Integrator2
Integrator
TimedPlotter
relation
parameter
y
$\dot{y}$
$\ddot{y}$

FIGURE 12. Model that generates a sine wave if the itegrators have a non-zero initial condition.

If we simply attempt to make the connections by clicking and dragging from the *Integrator* output port to the two input ports in sequence, then we get the exception shown in figure 14. Such exceptions can be intimidating, but are the normal and common way of reporting errors in HyVisual. The key line in this exception report is the last one, which says

```
   Attempt to link more than one relation to a single port.
```

The line below that gives the names of the objects involved, which are

```
   in .integratorSineWave.Integrator.output and .integratorSineWave.relation4
```

In HyVisual models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".integratorSineWave.Integrator.output" is an object named "output" contained by an object named "Integrator", which is contained by a model named "integratorSineWave."

Why did this exception occur? The diagram shows two distinct flavors of ports, indicated in the diagrams by a filled triangle or an unfilled triangle. The output port of the *Integrator* block is a *single port*, indicated by a filled triangle, which means that it can only support a single connection. The input port of the *TimedPlotter* block is a *multiport*, indicated by unfilled triangles. Multiports can support multiple connections, where each connection is treated as a separate *channel*. A channel is a path from an output port to an input port (via relations) that can transport a single stream of tokens.

So how do we get the output of the *Integrator* to the other two actors? We need an explicit *relation* in the diagram. A relation is represented in the diagram by a black diamond, as shown in Figure 15. It

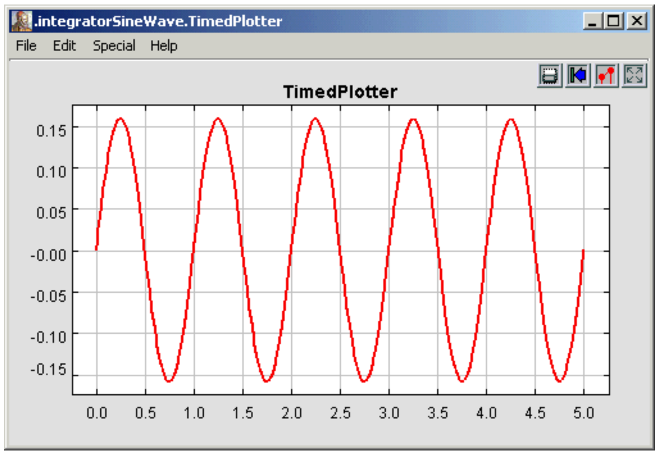
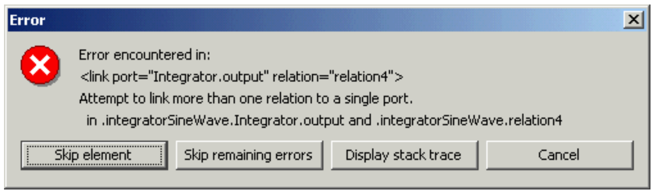


FIGURE 13. Result of running the model in figure 12 with the *initialState* of the left integrator set to 1.0.



Error encountered in:
<link port="Integrator.output" relation="relation4">
Attempt to link more than one relation to a single port.
in .integratorSineWave.Integrator.output and .integratorSineWave.relation4

FIGURE 14. An exception that results from attempting to make a multi-way connection without a relation.

can be created by either control-clicking on the background or by clicking on the button in the toolbar with the black diamond on it.

Making a connection to a relation can be tricky, since if you just click and drag on the relation, the relation gets selected and moved. To make a connection, hold the control button while clicking and dragging on the relation.

In the model shown in figure 15, the relation is used to broadcast the output from a single port to a number of places. The single port still has only one connection to it, a connection to a relation. Relations can also be used to control the routing of wires in the diagram. For example, in figure 15, the relation is placed to the left of all the blocks in order to get a pleasing layout. However, as of this writing, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

The *TimedPlotter* in figure 15 has a multiport input, as indicated by the unfilled triangle. This means that multiple channels of input can be connected directly to it. Consider the modification shown in figure 16, where both y and $\dot{y}$ are connected (via relations) to the input port of the *TimedPlotter*. The resulting plot is shown at the right. The two signals are treated by the block as distinct input signals coming in on separate channels.

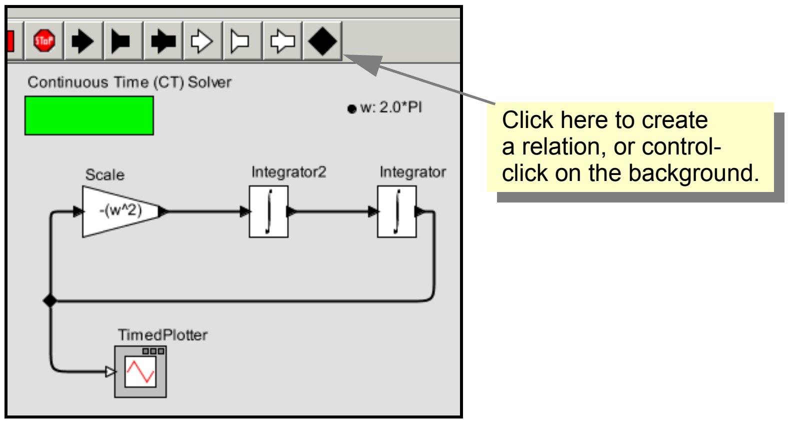

Continuous Time (CT) Solver
w: 2.0*PI
Scale
-(w^2)
Integrator2
Integrator
TimedPlotter
Click here to create a relation, or control-click on the background.

FIGURE 15. A relation can be used to broadcast an output from a single port.

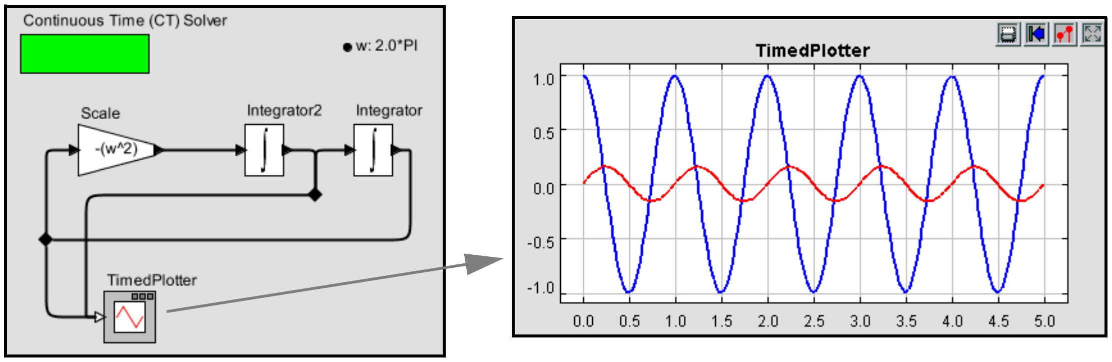


FIGURE 16. Multiple signals can be sent to a multiport, shown with the unfilled triangle on the *TimedPlotter*. In this case, two signals are plotted.

2.2.4 Parameters

Figure 16 shows a parameter named “w” with value “2.0*PI.” That parameter is then used in the *Scale* block to specify that the scale factor is “–(w^2).” This usage illustrates a number of convenient features.

To create a parameter that is visible in the diagram, drag one in from the *utilities* library, as shown in figure 17. Right click on the parameter to change its name to “w”. (In order to be able to use this parameter in expressions, the name cannot have any spaces in it.) Also, right click or double click on the parameter to change its default value to “2.0*PI.” This is an example of the sort of expressions you can use to define parameter values. The expression language is described below in section 3.

The parameter, once created, belongs to the model. If you right click on the grey background and select Configure, then you can edit the parameter value, just as you could by double clicking on the parameter. The resulting dialog also allows you to create parameters that are not visible in the model. The parameter is also visible and editable in the Run Window obtained via the View menu.

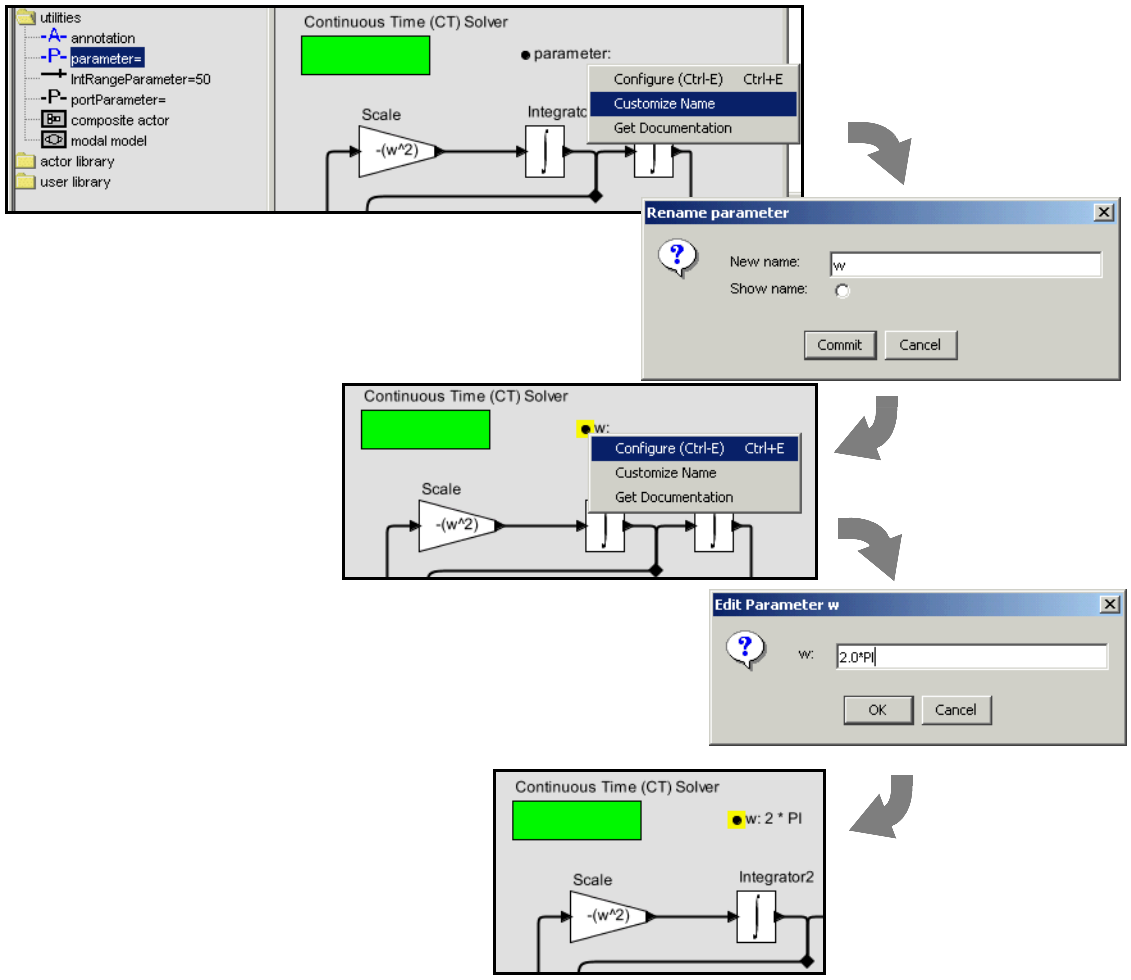


FIGURE 17. Adding a parameter to the channel model.

A parameter of the model can be used in expressions anywhere in the model, including in parameter values for blocks within the model. In figure 16, for instance, the *factor* parameter of the *Scale* actor has the value "–(w^2)," which references the parameter *w*.

2.2.5 Annotations

There are several other useful enhancements you could make to this model. Try dragging an *annotation* from the *utilities* library and creating a title on the diagram. Also, try setting the title of the plot by clicking on the second button from the right in the row of buttons at the top right of the plot. This button has the tool tip "Set the plot format" and brings up the format control window.

2.2.6 Impulse Response

Consider equation (6), which we repeat here for reference:

$$\omega_0 x(t) = \omega_0^2 y(t) + \ddot{y}(t). \tag{9}$$

Figure 12 and subsequent models based on this equation assume that the input is zero for all time, $x(t) = 0$, thus realizing equation (8). We can elaborate on this model by re-introducing the input, and allowing it to be non-zero. The result is shown in figure 18, where the input is provided by a block labeled "Clock." Notice that the input is multiplied by ω_0, and then $\omega_0^2 y(t)$ is subtracted from it to obtain $\ddot{y}(t)$. I.e., it calculates

$$\ddot{y}(t) = \omega_0 x(t) - \omega_0^2 y(t). \tag{10}$$

In the model, both integrators now have *initialState* set to 0.0. The plot in the figure shows the result of running the model with an impulse as the input, resulting in an impulse response that matches (5).

The key question, however, is how can we generate an impulse for the input? In theory, an impulse, also known as a Dirac delta function, is a continuous-time function δ that satisfies

$$\begin{aligned} \delta(t) &= 0, \forall t \neq 0 \\ \int_{-\infty}^{\infty} \delta(t) dt &= 1 \end{aligned} . \tag{11}$$

From these relations, it is easy to see that the Dirac delta function must have infinite value at $t = 0$,

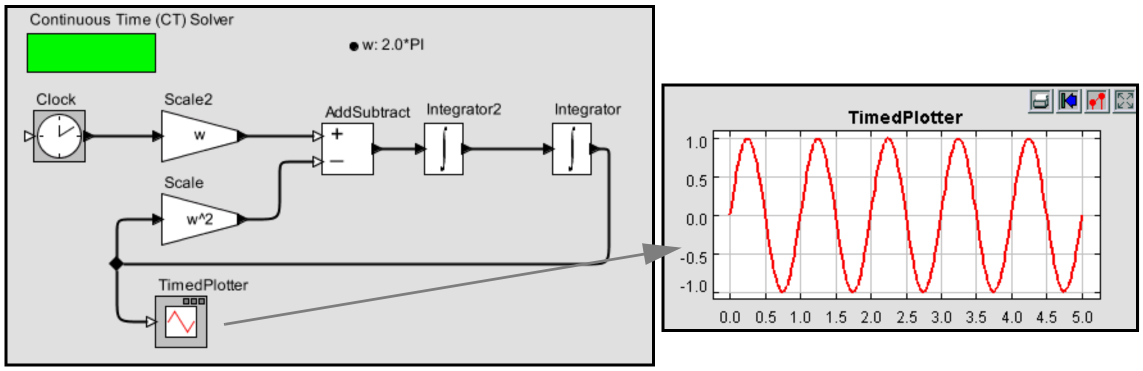


FIGURE 18. Variant of figure 12 that has an input, which is provided by the Clock actor.

because otherwise it could not integrate to one. Hence, it is problematic to generate an impulse in a continuous-time simulator.

The model shown in figure 18 uses a Clock actor to generate an approximate impulse. The parameters of the Clock actor are shown in figure 19. First, notice that *numberOfCycles* is set to 1. This means that only one pulse will be generated. The pulse is defined by the *offsets* and *values* parameters. The *offsets* parameter is set to {0.0, 1.0E-5}, which is an array with two numbers. The *values* parameter is set to {1.0E5, 0.0}. Together, these mean that the output goes to value 1.0×10^5 at time 0.0, and then to value 0.0 at time 1.0×10^{-5}. Thus, the output is a very narrow, very high rectangular pulse, with unit area.

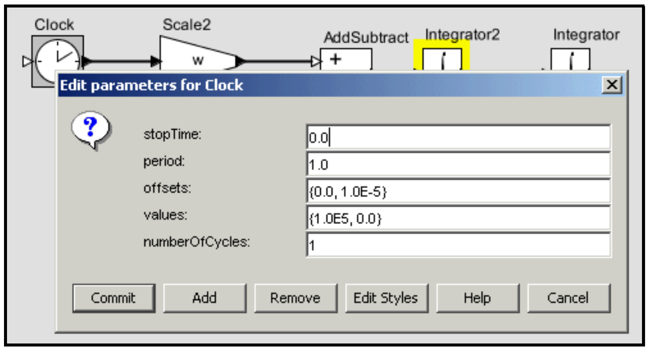
If you create a Clock and set these parameter values, and try to run the model, you are likely to see the exception shown in figure 20. The problem here is that the default *minStepSize* value for the solver, as shown in figure 3, is too large, given the very narrow pulse we are trying to generate. In this case, it is sufficient to change the *minStepSize* parameter to 1.0E-9. Generally, the *minStepSize* parameter needs to be considerably smaller than the smallest phenomenon in time that is being observed. It is worth noting that even with this small value for *minStepSize*, the solver does not actually use step sizes anywhere near this very often. You can examine which points in the output plot are actually computed by the solver by turning on stems in the output plot.

2.2.7 Using Higher-Order Dynamics Blocks

Recall from (4) that a transfer function given by the Laplace transform

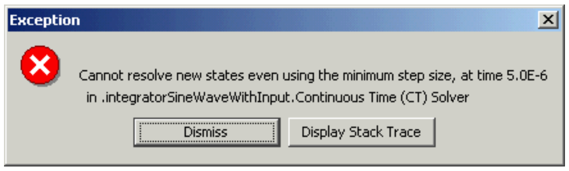
$$H(s) = \frac{\omega_0}{s^2 + \omega_0^2} \tag{12}$$

describes the system shown in figure 18. In fact, we could have constructed the system more easily by using the ContinuousTransferFunction actor in the *dynamics* library, as shown in figure 21. That actor



stopTime: 0.0
period: 1.0
offsets: {0.0, 1.0E-5}
values: {1.0E5, 0.0}
numberOfCycles: 1

FIGURE 19. Parameters of the Clock actor that get it to output an approximate impulse.



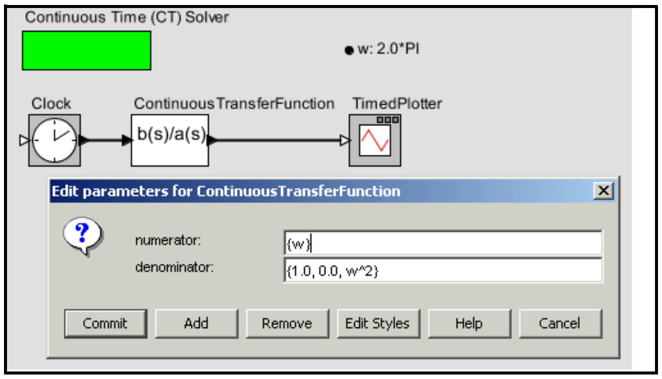
Cannot resolve new states even using the minimum step size, at time 5.0E-6 in .integratorSineWaveWithInput.Continuous Time (CT) Solver

FIGURE 20. Exception due to running the model with the *minStepSize* parameter set too high.

has as parameters two arrays, *numerator* and *denominator*, which are set to {w} and {1.0, 0.0, w^2}, respectively. A portion of the documentation for that actor is shown in figure 22 (you can obtain this documentation by right clicking on the actor icon and selecting Get Documentation). As indicated on that page, the *numerator* and *denominator* parameters give the coefficients of the numerator and denominator polynomials in (12).

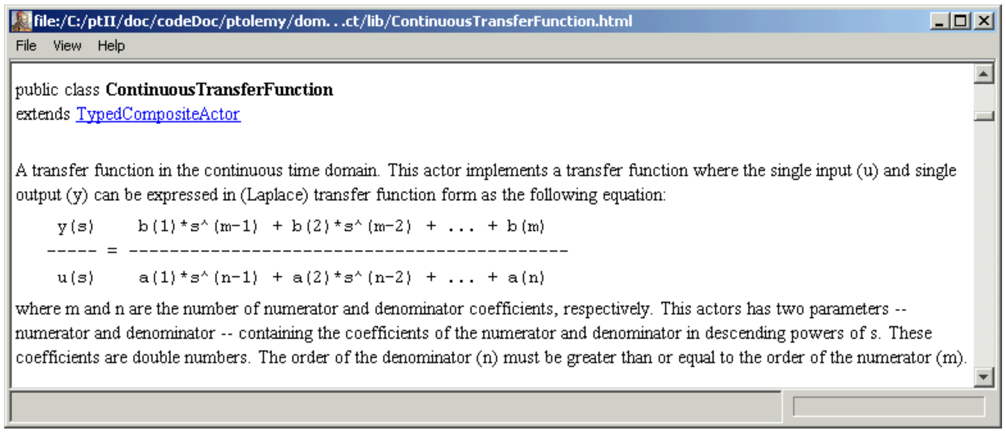
Recall that to run this model, you will need to set the *minStepSize* parameter of the solver to 10^{-9} or smaller.

An interesting curiosity about this actor is how it works. It works by creating a hierarchical model similar to the one that we built by hand. If, after running the model at least once, you right click on the ContinuousTransferFunction icon and select Look Inside (or type Control-L over the icon), you will see an inside model that looks like that shown in figure 23. This model is hard to interpret, since all the icons are placed one on top of the other at the upper left. You can select Automatic Layout from the Graph menu to get something a bit easier to read, shown in figure 24. It is still far from perfect, but with a bit of additional placement effort, you can verify that this model is functionally equivalent to the one we constructed manually in figure 18.


Continuous Time (CT) Solver
w: 2.0*PI
Clock
ContinuousTransferFunction
TimedPlotter
b(s)/a(s)
Edit parameters for ContinuousTransferFunction
Commit
Add
Remove
Edit Styles
Help
Cancel

numerator: {w}
denominator: {1.0, 0.0, w^2}

FIGURE 21. A model equivalent to that in figure 18, but using the LaplaceTransferFunction actor.


file:/C:/ptII/doc/codeDoc/ptolemy/dom. . .ct/lib/ContinuousTransferFunction.html
File View Help

public class **ContinuousTransferFunction**
extends TypedCompositeActor

A transfer function in the continuous time domain. This actor implements a transfer function where the single input (u) and single output (y) can be expressed in (Laplace) transfer function form as the following equation:

```
    y(s)    b(1)*s^(m-1) + b(2)*s^(m-2) + ... + b(m)
   ----- = -------------------------------------------
    u(s)    a(1)*s^(n-1) + a(2)*s^(n-2) + ... + a(n)
```

where m and n are the number of numerator and denominator coefficients, respectively. This actors has two parameters -- numerator and denominator -- containing the coefficients of the numerator and denominator in descending powers of s. These coefficients are double numbers. The order of the denominator (n) must be greater than or equal to the order of the numerator (m).

FIGURE 22. A portion of the documentation for the ContinuousTransferFunction actor.

2.3 Data Types

In the example of figure 7, the *TimedSinewave* actor produces values on its output port. The values in this case are *double*. You can examine the data types of ports after executing a model by simply lingering on the port with the mouse. A tooltip will appear, as shown in figure 25.

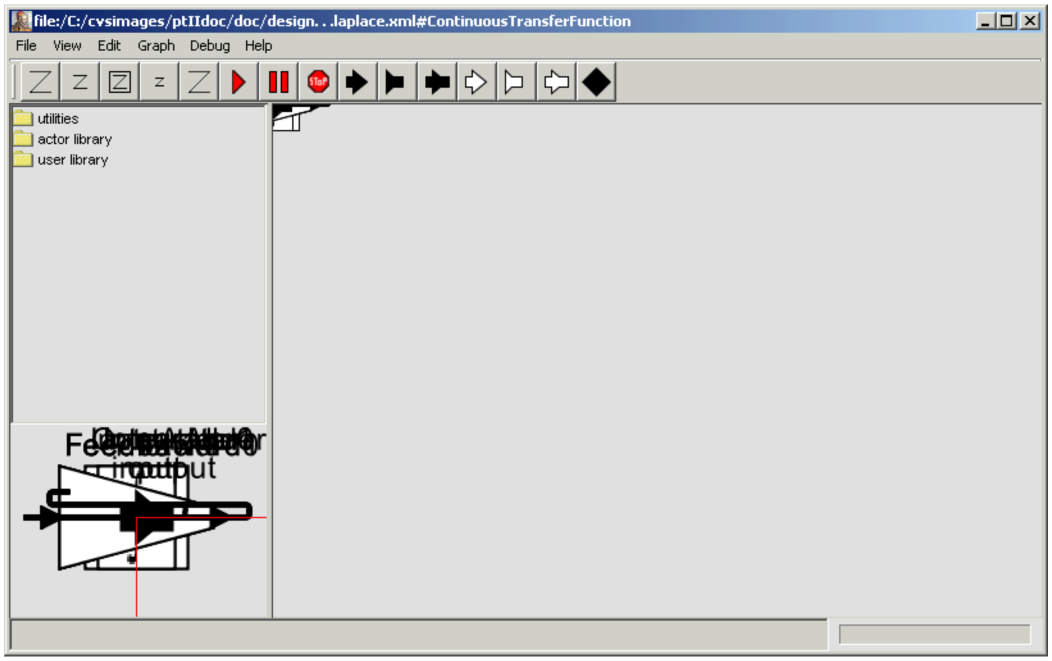

file:/C:/cvsimages/ptIIdoc/doc/design. . .laplace.xml#ContinuousTransferFunction
File
View
Edit
Graph
Debug
Help
utilities
actor library
user library

FIGURE 23. Inside the ContinuousTransferFunction actor of figure 21.

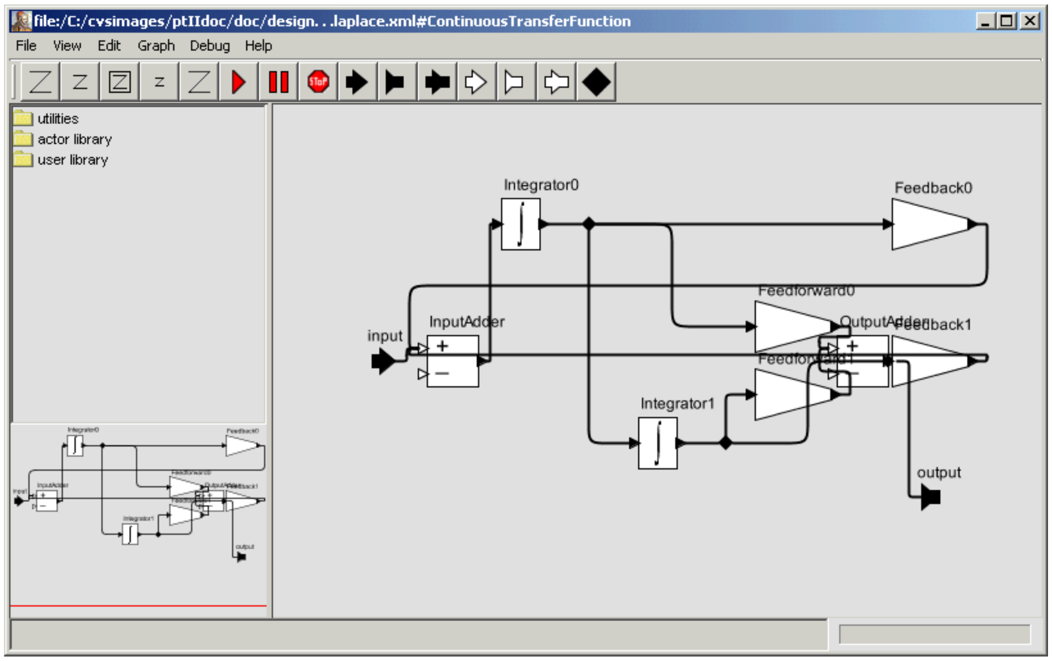


FIGURE 24. The diagram of figure 23, after invoking Automatic Layout from the Graph menu.

HyVisual has quite a sophisticated type system. Actors represent type constraints that relate the types of the their ports and parameters, and the type system resolves the constraints, unless a conflict arises.

, but for now, try giving the value 1 (the integer with value one), or 1.0 (the floating-point number with value one), or {1.0} (An array containing a one), or {value=1, name="one"} (A record with two elements: an integer named “value” and a string named “name”), or even [1,0;0,1] (the two-by-two identity matrix). These are all expressions.

The *Const* actor is able to produce data with different *types*, and the *Display* actor is able to display data with different types. Most actors in the actor library are *polymorphic*, meaning that they can operate on or produce data with multiple types. The behavior may even be different for different types. Multiplying matrices, for example, is not the same as multiplying integers, but both are accomplished by the *MultiplyDivide* actor in the *math library*. Ptolemy II includes a sophisticated type system that allows this to be done efficiently and safely.

To explore data types a bit further, try creating the model in Figure 26. The *Const* and *CurrentTime* actors are listed under *timed sources*, the *AddSubtract* actor is listed under *math,* and the *MonitorValue* actor is listed under *timed sinks*. Set the *value* parameter of the constant to be 1. Running the model for 10.0 time units should result in 9.0 being displayed in *MonitorValue*, as shown in the figure The output of the *CurrentTime* actor is a *double*, the output of the *Const* actor is an *int*, and the *AddSubtract* actor adds these two to get a *double*.

Now for the real test: change the value of the *Const* actor to a string, such as "a" (with the quotation marks). In fact, the *Const* actor can have as its *value* anything that can be given using the expression language, explained below in section 3. When you execute the model, you should see an exception window, as shown in Figure 27. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a *string* value from an *double* value, which doesn’t make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

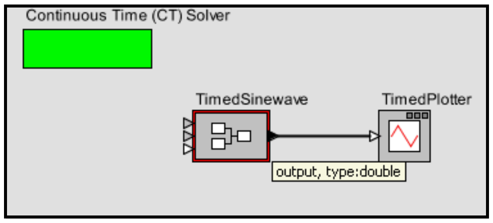


FIGURE 25. Tooltip showing the name and data type of the output port of the TimedSinewave of figure 7.

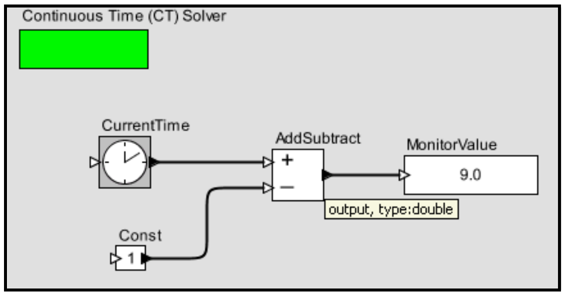


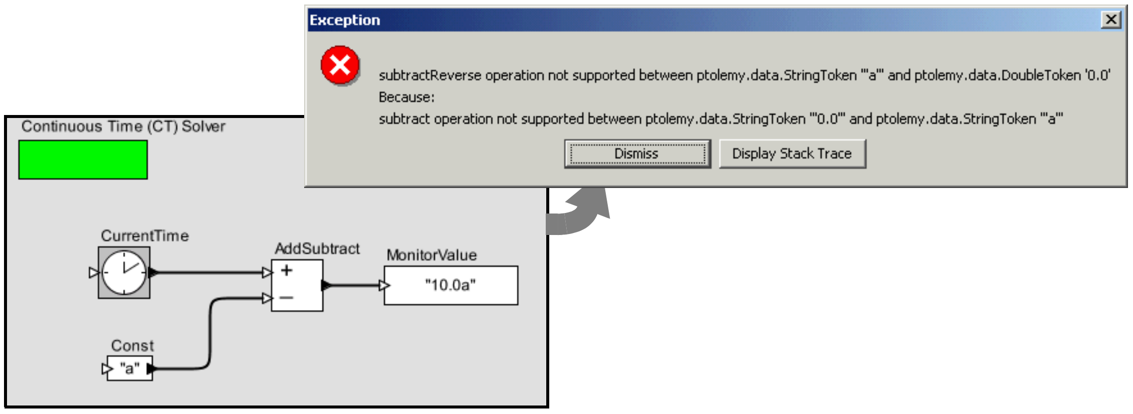
FIGURE 26. Another example, used to explore data types in HyVisual.

Let’s try a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in Figure 28. You can do this by selecting the connection and deleting it (using the delete key), then adding a new connection, or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like “10.0a”, as shown in the figure. This is the result of converting the *double* from *CurrentTime* to a *string* “10.0,” and then adding the strings together (which, following Java, means concatenating them).

As a rough guideline, HyVisual will perform automatic type conversions when there is no loss of information. An integer can be converted to a string, but not vice versa. An integer can be converted to a double, but not vice versa. An integer can be converted to a long, but not vice versa.

2.4 Hierarchy

HiVisual supports (and encourages) hierarchical models. These are models that contain components that are themselves models. Such components are called *composite actors*. Consider a small signal processing problem, where we are interested in recovering a signal based only on noisy measurements of it. We will create a composite actor modeling a communication channel that adds noise, and then use that actor in a model.



```
subtractReverse operation not supported between ptolemy.data.StringToken '"a"' and ptolemy.data.DoubleToken '0.0'
Because:
subtract operation not supported between ptolemy.data.StringToken '"0.0"' and ptolemy.data.StringToken '"a"'
```

FIGURE 27. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from doubles.

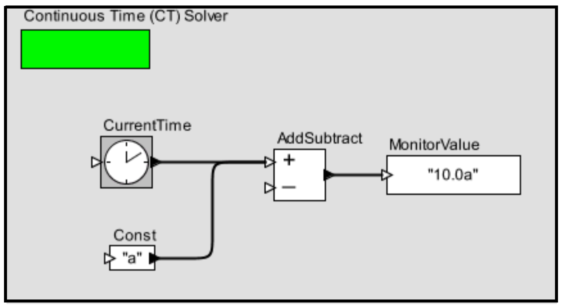


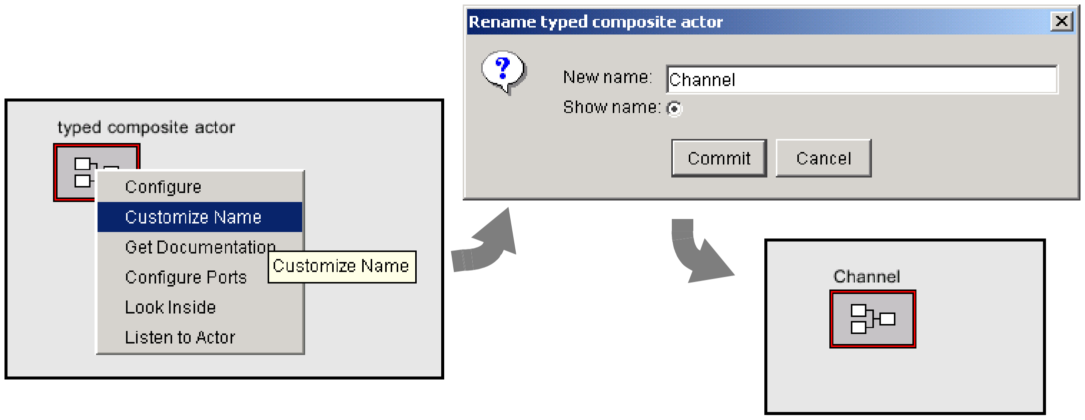
FIGURE 28. Addition of a string to an integer.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in Figure 29. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in Figure 30. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

First we have to add some ports to the composite actor. There are several ways to do this, but clicking on the port buttons in the toolbar is probably the easiest. You can explore the ports in the toolbar by lingering with the mouse over each button in the toolbar. A tool tip pops up that explains the button. The buttons are summarized in Figure 31. Create an input port and an output port and rename them



New name: Channel
Show name: ●

FIGURE 29. Changing the name of an actor.

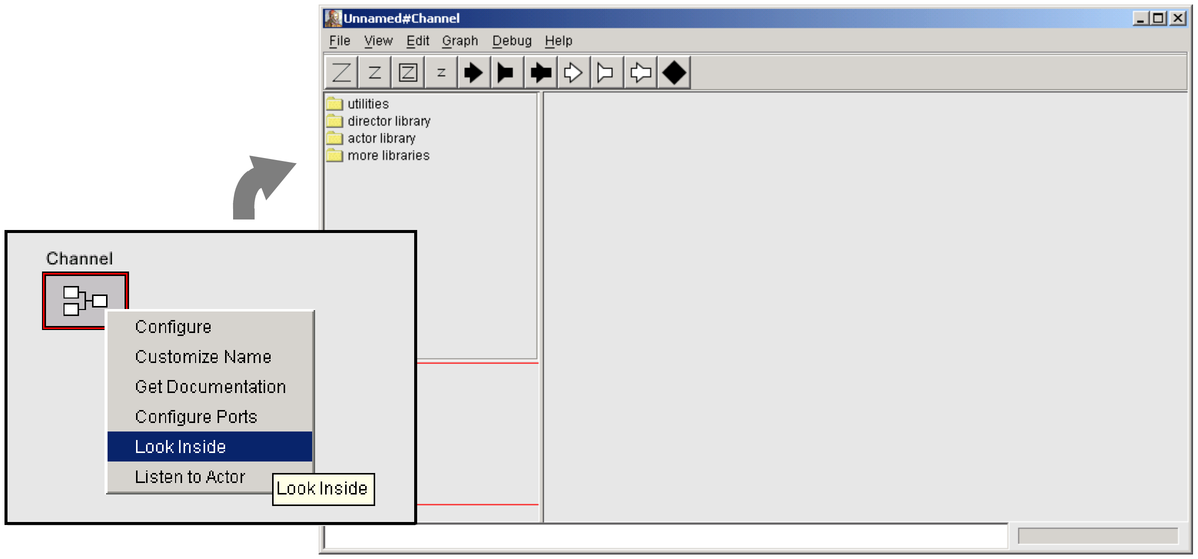


FIGURE 30. Looking inside a composite actor.

input and *output* right by clicking on the ports and selecting “Customize Name”. Note that, as shown in Figure 32, you can also right click on the background of the composite actor and select *Configure Ports* to change whether a port is an input, an output, or a multiport. The resulting dialog also allows you to set the type of the port, although much of the time you will not need to do this, since the type inference mechanism in Ptolemy II will figure it out from the connections.

Then using these ports, create the diagram shown in Figure 33[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close this editor and return to the previous one, you should be able to easily create the model shown in Figure 34.

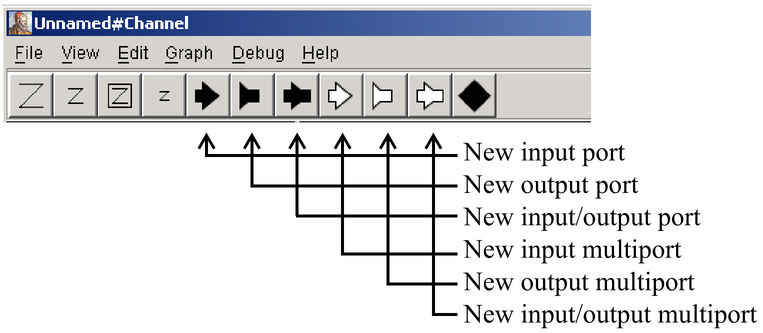
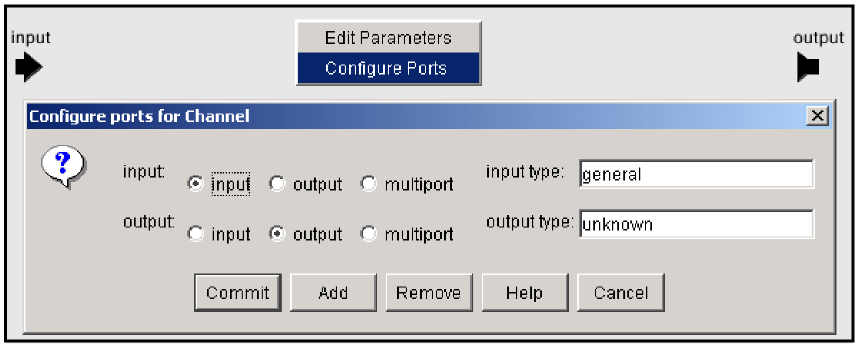


FIGURE 31. Summary of toolbar buttons for creating new ports.



Configure ports for Channel

input: [✓ input] output multiport   input type: general

output: input [✓ output] multiport   output type: unknown

FIGURE 32. Right clicking on the background brings up a dialog that can be used to configure ports.

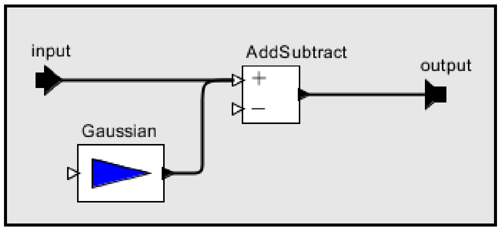


FIGURE 33. A simple channel model defined as a composite actor.

1. **Hint:** to create a connection starting on one of the external ports, hold down the control key when dragging.

The *Sinewave* actor is listed under *sources*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline (try looking inside). If you execute this model (you will probably want to set the iterations to something reasonable, like 100), you should see something like Figure 35.

2.4.3 Setting the Types of Ports

In the above example, we never needed to define the types of any ports. The types were inferred from the connections. Indeed, this is usually the case in Ptolemy II, but occasionally, you will need to set the types of the ports. Notice in Figure 32 that there is a position in the dialog box that configures ports for specifying the type. Thus, to specify that a port has type *boolean*, you could enter *boolean* into the dialog box. There are other commonly used types: *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. Let's take a more complicated case. How would you specify that the type of a port is a double matrix? Easy:

```
[double]
```

This expression actually creates a 1 by 1 matrix containing a double (the value of which is irrelevant). It thus serves as a prototype to specify a double matrix type. Similarly, we can specify an array of complex numbers as

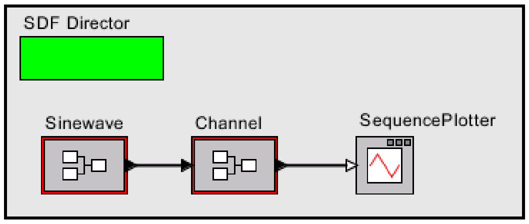
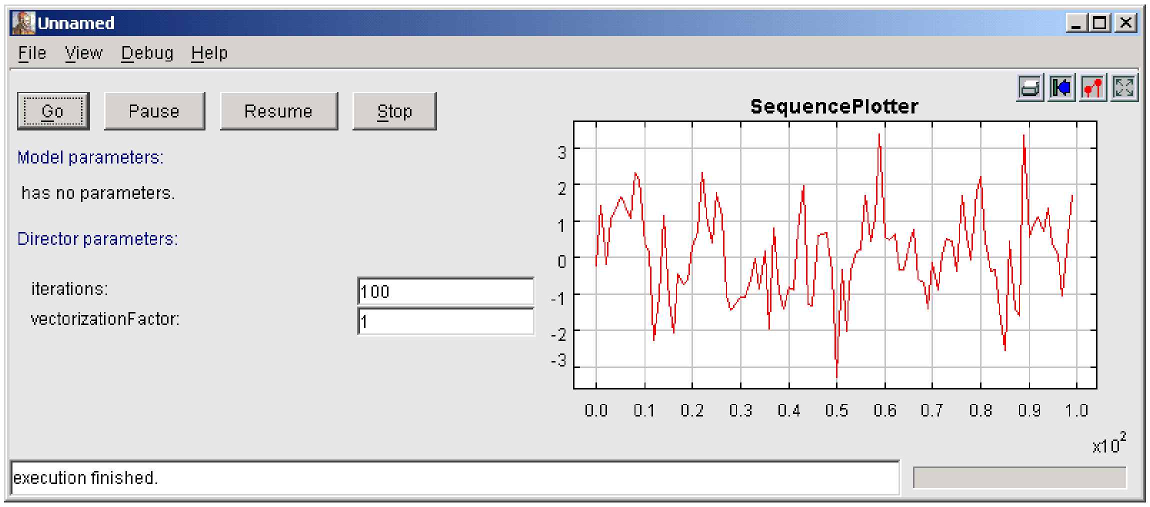


FIGURE 34. A simple signal processing example that adds noise to a sinusoidal signal.



Model parameters:

has no parameters.

Director parameters:

iterations: 100

vectorizationFactor: 1

execution finished.

FIGURE 35. The output of the simple signal processing model in Figure 34.

```
{complex}
```

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. What about a record containing a string named “name” and an integer named “address”? Easy:

```
{name=string, address=int}
```

2.5 Navigating Larger Models

Sometimes, a model gets large enough that it is not convenient to view it all at once. There are four toolbar buttons, shown in Figure 2.27 that help. These buttons permit zooming in and out. The “Zoom reset” button restores the zoom factor to the “normal” one, and the “Zoom fit” calculates the zoom factor so that the entire model is visible in the editor window.

In addition, it is possible to pan over a model. Consider the window shown in Figure 37. Here, we have zoomed in so that icons are larger than the default. The *pan window* at the lower left shows the entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the “Zoom fit” button in the toolbar results in the editor area showing the entire model, just as the pan window does.

2.6 Domains

A key innovation in Ptolemy II is that, unlike other design and modeling environments, there are several available *models of computation* that define the meaning of a diagram. In the above examples, we directed you to drag in an *SDFDirector* without justifying why. A director in Ptolemy II gives meaning (semantics) to a diagram. It specifies what a connection means, and how the diagram should be executed. In Ptolemy II terminology, the director realizes a *domain*. Thus, when you construct a model with an SDF director, you have constructed a model “in the SDF domain.”

The SDF director is fairly easy to understand. “SDF” stands for “synchronous dataflow.” In dataflow models, actors are invoked (fired) when their input data is available. SDF is particularly simple case of dataflow where the order of invocation of the actors can be determined statically from the model. It does not depend on the data that is processed (the tokens that are passed between actors).

But there are other models of computation available in Ptolemy II. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are

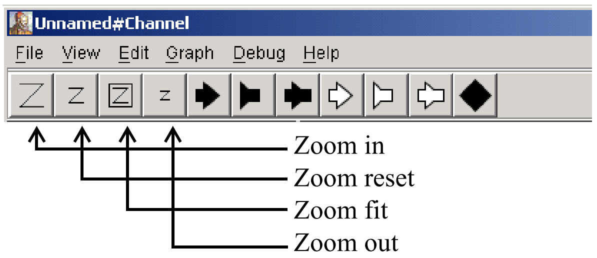


FIGURE 36. Summary of toolbar buttons for zooming and fitting.

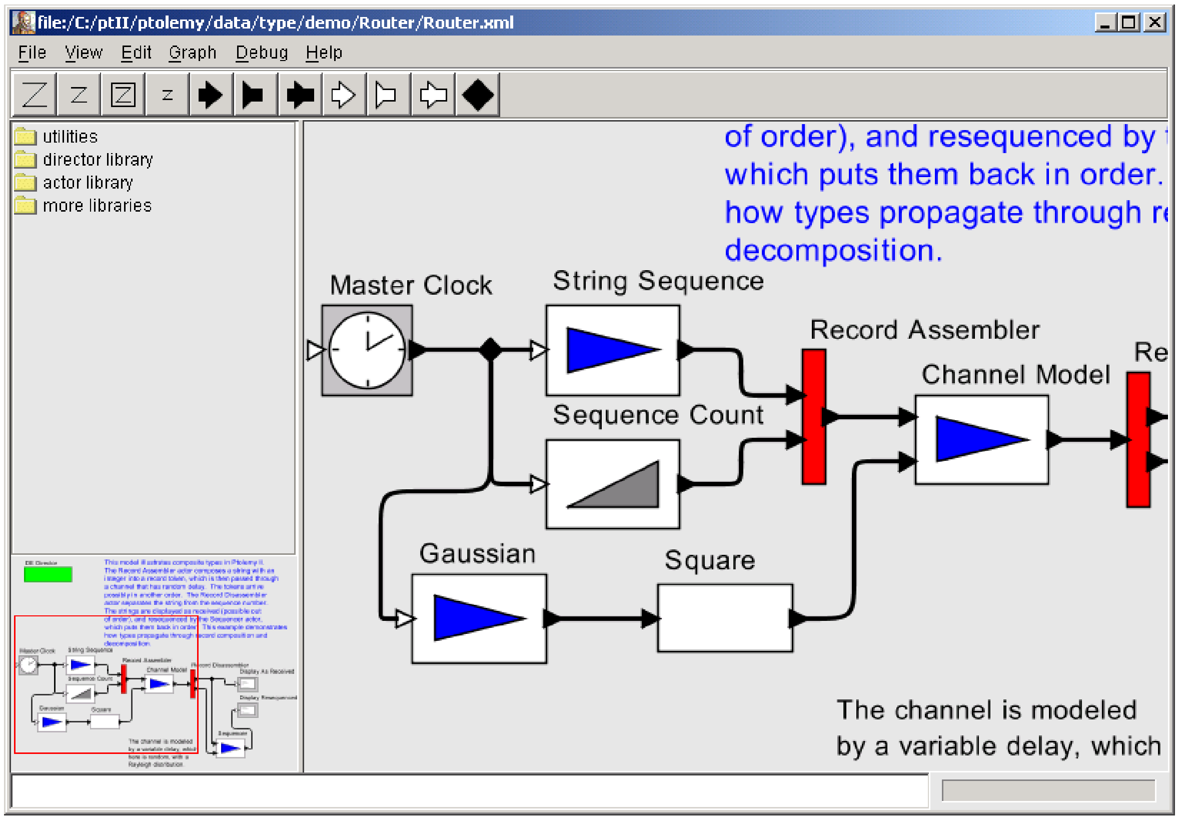
using. Here, we give a very brief introduction to some of the domains. But we begin first by explaining some of the subtleties in SDF.

2.6.1 SDF and Multirate Systems

So far we have been dealing with relatively simple systems. They are simple in the sense that each actor produces and consumes one token from each port at a time. In this case, the SDF director simply ensures that an actor fires after the actors whose output values it depends on. The total number of output values that are created by each actor is determined by the number of iterations, but in this simple case only one token would be produced per iteration.

It turns out that the SDF scheduler is actually much more sophisticated. It is capable of scheduling the execution of actors with arbitrary prespecified data rates. Not all actors produce and consume just a single sample each time they are fired. Some require several input token before they can be fired, and produce several tokens when they are fired.

One such actor is a spectral estimation actor. Figure 38 shows a system that computes the spectrum of the same noisy sine wave that we constructed in Figure 34. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 39 shows the output of the model with *order* set to 8 and the number of *iterations* set to 1. **Note that there are 256 output samples output from the *Spectrum* actor**. This is because the *Spectrum* actor requires 2^8, or 256 input samples to fire, and produces 2^8, or 256 output samples when it fires. Thus, one iteration of the



of order), and resequenced by
which puts them back in order.
how types propagate through r
decomposition.

The channel is modeled
by a variable delay, which

FIGURE 37. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

model produces 256 samples. The *Spectrum* actor makes this a *multirate* model, because the firing rates of the actors are not all identical.

It is common in SDF to construct models that require exactly one iteration to produce a useful result. In some multirate models, it can be complicated to determine how many firings of each actor occur per iteration of the model. See the SDF chapter for details.

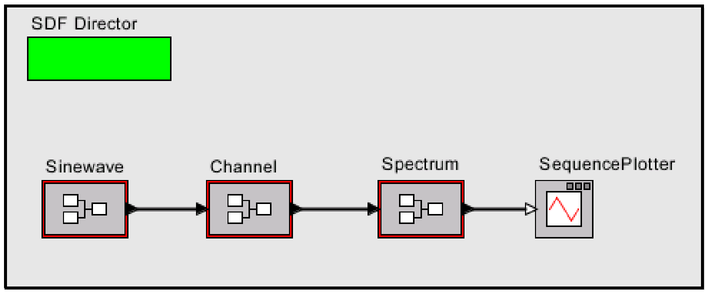
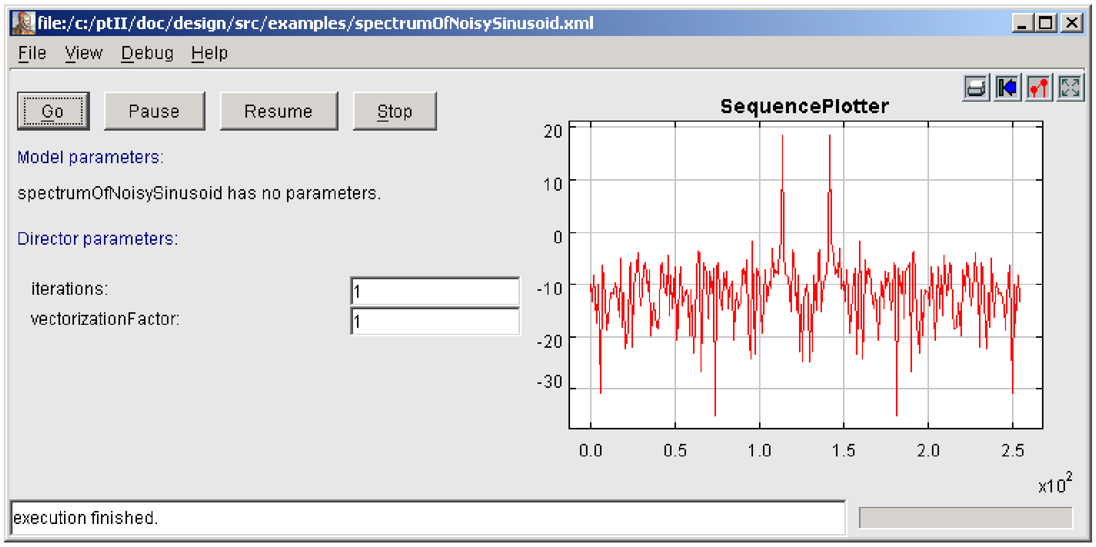


FIGURE 38. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave, Channel,* and *SequencePlotter*, and one firing of *Spectrum*.



Model parameters:

spectrumOfNoisySinusoid has no parameters.

Director parameters:

iterations: 1

vectorizationFactor: 1

execution finished.

FIGURE 39. A single iteration of the SDF model in Figure 38 produces 256 output tokens.

A second subtlety with SDF models is that if there is a feedback loop, as in Figure 40, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *flow control* library). Without this actor, the loop will deadlock. The *SampleDelay* actor produces initial tokens on its output, before the model begins firing. The initial tokens produced are given by a the *initialOutputs* parameter, which specifies an array of tokens. These initial tokens enable downstream actors and break the circular dependencies that would result otherwise from a feedback loop.

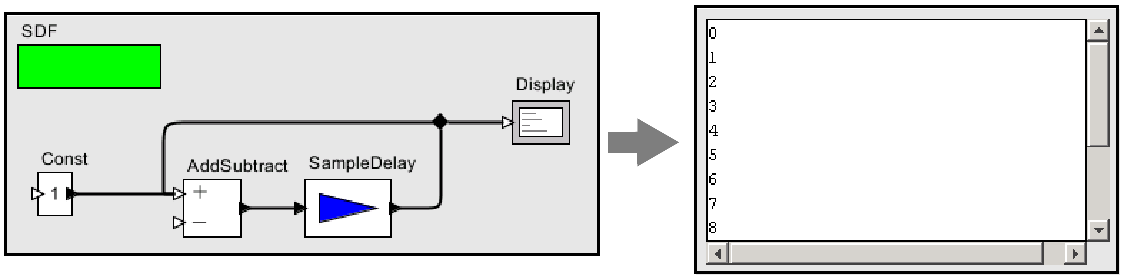
A final issue to consider with the SDF domain is time. Notice that in all the examples above we have suggested using the *SequencePlotter* actor, not the *TimedPlotter* actor, which is in the same *sinks* library. This is because the SDF domain does not include in its semantics a notion of time. Time does not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.6.2 Discrete-Event Systems

In discrete-event (DE) systems, the connections between actors carry signals that consist of *events* placed on a time line. Each event has both a value and a time stamp, where its time stamp is a double-precision floating-point number. This is different from dataflow, where a signal consists of a sequence of tokens, and there is no time significance in the signal.

A DE model executes chronologically, processing the oldest events first. Time advances as events are processed. There is potential confusion, however, between *model time*, the time that evolves in the model, and *real time*, the time that elapses in the real world while the model executes (also called *wall-clock time*). Model time may advance more rapidly than real time or more slowly. The DE director has a parameter, *synchronizeToRealTime*, that, when set to true, attempts to synchronize the two notions of time. It does this by delaying execution of the model, if necessary, allowing real time to catch up with model time.

Consider the DE model shown in Figure 41. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead, these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the



```
0
1
2
3
4
5
6
7
8
```

FIGURE 40. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

time stamp of the input). The *WallClockTime* actor an event with the same time stamp as the input, but whose value is the current real time, in seconds since initialization of the model.

The plot in Figure 41 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

A significant subtlety in using the DE domain is in how simultaneous events are handled. Simultaneous events are simply events with the same time stamp. We have stated that events are processed in chronological order, but if two events have the same time stamp, then there is some ambiguity. Which one should be processed first? If the two events are on the same signal, then the answer is simple: process first the one that was produced first. However, if the two events are on different signals, then the answer is not so clear.

Consider the model shown in Figure 42, which produces a histogram of the interarrival times of events from the *PoissonClock* actor. In this model, we calculate the difference between the current event time and the previous event time, resulting in the plot that is shown in the figure. The *Previous* actor is a *zero-delay* actor, meaning that it produces an output with the same time stamp as the input (except on the first firing, where in this case it produces no output). Thus, when the *PoissonClock* actor produces an output, there will be two simultaneous events, one at the input to the *plus* port of the *AddSubtract* actor, and one at the input of the *Previous* actor. Should the director fire the *AddSubtract* actor or the *Previous* actor? Either seems OK if it is to respect chronological order, but it seems intuitive that the *Previous* actor should be fired first.

It is helpful to know how the *AddSubtract* actor works. When it fires, it adds all available tokens on the *plus* port, and subtracts all available tokens on the *minus* port. If the *AddSubtract* actor fires before the *Previous* actor, then the only available token will be the one on the *plus* port, and the

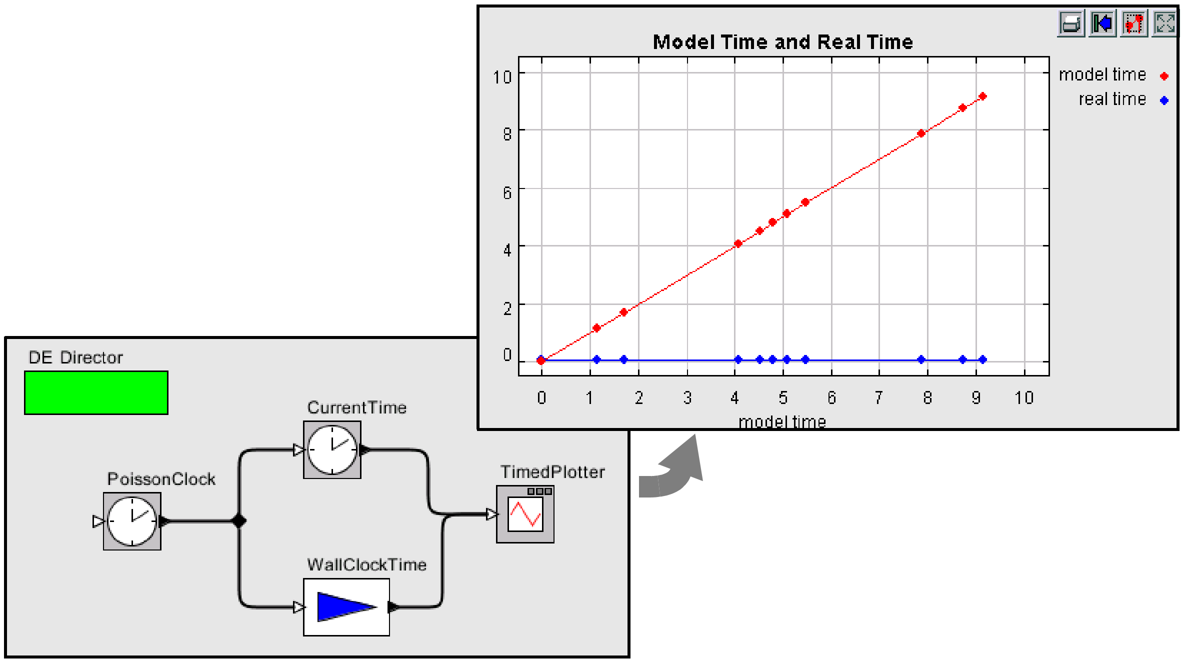


FIGURE 41. Model time vs. real time (wall clock time).

expected subtraction will not occur. Intuitively, we would expect the director to invoke the *Previous* actor before the *AddSubtract* actor so that the subtraction occurs.

How does the director deliver on the intuition that the *Previous* actor should be fired first? Before executing the model, the DE director constructs a *topological sort* of the model. A topological sort is simply a list of the actors in data-precedence order. For the model in Figure 42, there is only one allowable topological sort:

- *PoissonClock*, *CurrentTime*, *Previous*, *AddSubtract*, *HistogramPlotter*

In this list, *AddSubtract* is after *Previous*. So the when they have simultaneous events, the DE director fires *Previous* first.

Thus, the DE director, by analyzing the structure of the model, usually delivers the intuitive behavior, where actors that produce data are fired before actors that consume their results, even in the presence of simultaneous events.

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *domain specific* library under *discrete-event* (this library is shown on the left in Figure 43). Consider for example the model shown in Figure 43. That model has a *Clock* actor, which is set to produce events every 1.0 time units. Those events trigger the *Ramp* actor, which produces outputs that start at 0 and increase by 1 on each firing. In this model, the output of the *Ramp* goes into an *AddSubtract* actor, which subtracts from the *Ramp* output its own prior output delayed by one time unit. The result is shown in the plot in the figure.

Occasionally, you will need to put a *TimedDelay* actor in a feedback loop with a delay of 0.0. This is particularly true if you are building complex models that mix domains, and there is a delay inside a composite actor that the DE director cannot recognize as a delay. The *TimedDelay* actor with a delay of 0.0 can be thought of as a way to let the director know that there is a time delay in the preceding actor, without specifying the amount of the time delay.

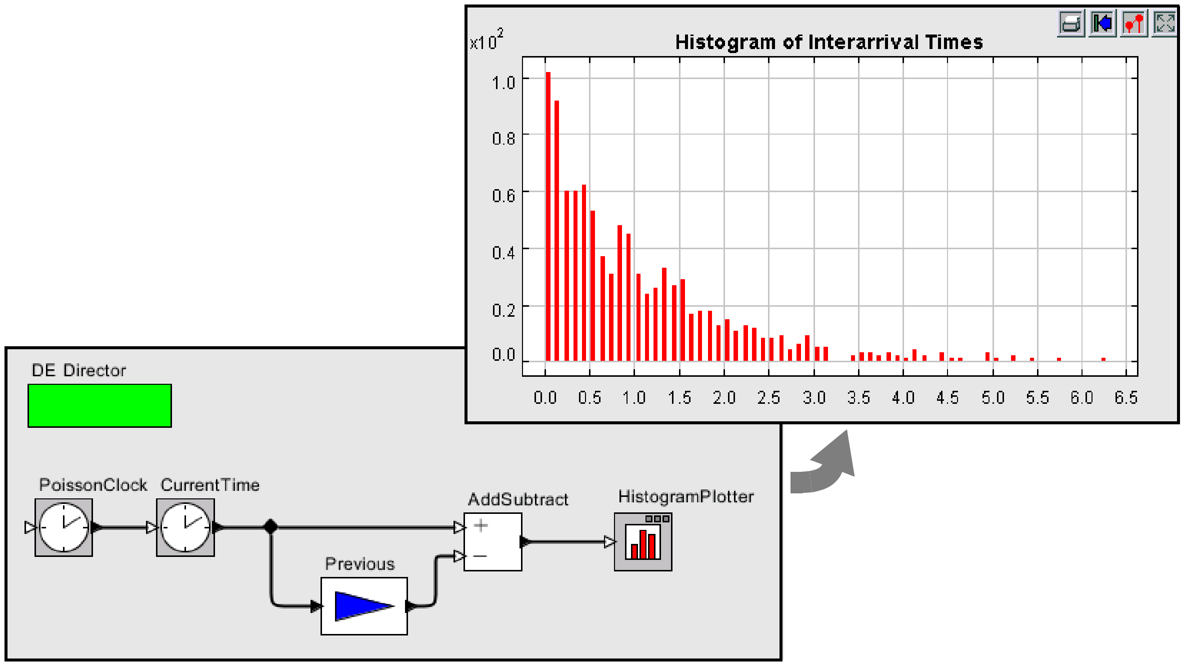


FIGURE 42. Histogram of interarrival times, illustrating handling of simultaneous events.

2.6.3 FSM and Modal Models

The finite-state machine domain (FSM) in Ptolemy II is a relatively less mature domain (but mature enough to be useful) with semantics very different from the domains covered so far. An FSM model looks different in Vergil. An example is shown in Figure 44. Notice that the component library on the left and the toolbar at the top are different for this model. We will explain how to construct this model.

First, the FSM domain is almost always used in combination with other domains in Ptolemy II to create *modal models*. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a *state* in a finite-state machine. The circles in Figure 44 are states, and the arcs between circles are *transitions* between states.

A modal model is typically a component in a larger model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will probably need to add ports. Figure 45 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. It represents a bouncing ball. Three output ports have been added, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

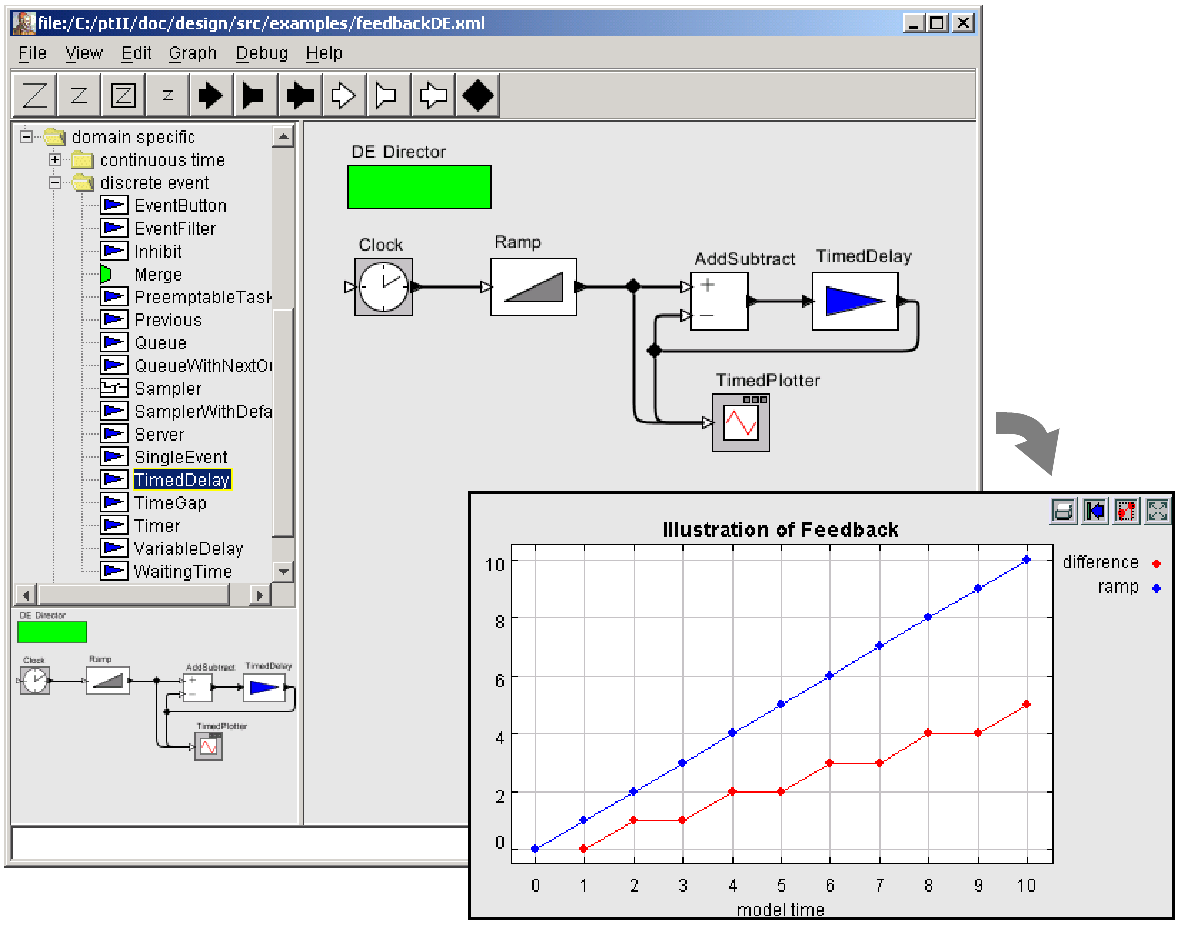
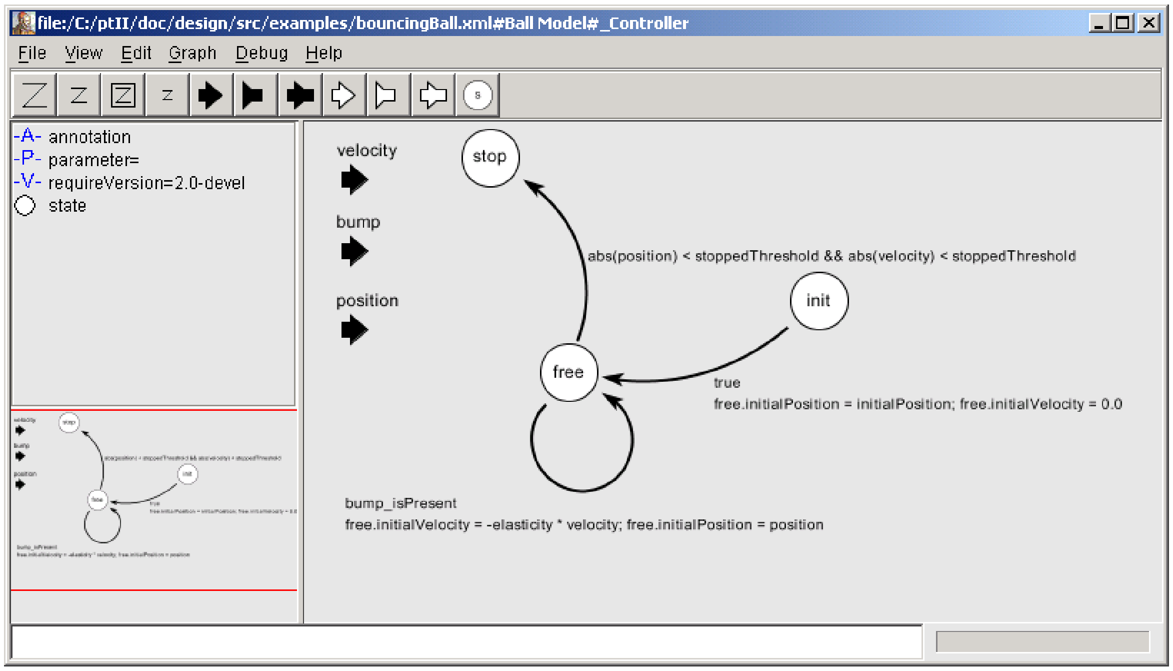


FIGURE 43. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

If you create a new modal model by dragging it in from the *utilities* library, and then look inside, you will get an FSM editor like that in Figure 44, except that it will be almost blank. The only items in it will be the ports you have added, and possibly some text that you can delete once you no longer need it. You may want to move the ports to reasonable locations.

To create a finite-state machine like that in Figure 44, drag in states (white circles). You can rename these states by right clicking on them and selecting "Customize Name". Choose names that are pertinent to your application. In Figure 44, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name. In this example, the initial state is *init*.

To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or

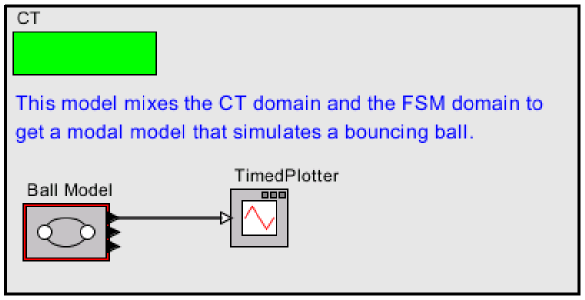

file:/C:/ptII/doc/design/src/examples/bouncingBall.xml#Ball Model#_Controller
File View Edit Graph Debug Help
-A- annotation
-P- parameter=
-V- requireVersion=2.0-devel
state
velocity
bump
position
stop
init
free

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

true
free.initialPosition = initialPosition; free.initialVelocity = 0.0

bump_isPresent
free.initialVelocity = -elasticity * velocity; free.initialPosition = position

FIGURE 44. Finite-state machine model used in the bouncing ball example.



This model mixes the CT domain and the FSM domain to get a modal model that simulates a bouncing ball.

FIGURE 45. Top-level of the bouncing ball example. The *Ball Model* actor is an instance of *modal model* from the *utilities* library. It has been renamed.

right clicking and selecting “Configure”) allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in Figure 46. In that dialog, we see the following:

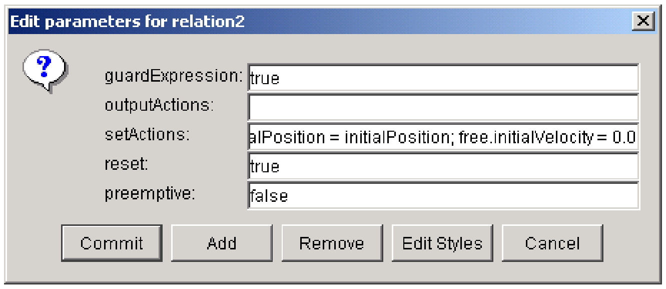
- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.
- The output actions are empty, meaning that when this transition is taken, no output is specified. This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.
- The set actions contain the following statements:

  ```
  free.initialPosition = initialPosition; free.initialVelocity = 0.0
  ```

  The “free” in these expressions refers to the mode refinement in the “free” state. Thus, “free.initialPosition” is a parameter of that mode refinement. Here, its value is assigned to the value of the parameter “initialPosition”. The parameter “free.initialVelocity” is set to zero.
- The *reset* parameter is set to *true*, meaning that the destination mode should be initialized when the transition is taken.
- The *preemptive* parameter is set to *false*. In this case, it makes no difference, since the *init* state has no refinement. Normally, if a transition out of a state is enabled and *preemptive* is *true*, then the transition will be taken without first firing the refinement.

Refinements. Both states and transitions can have *refinements.* To create a refinement, right click on the state or transition, and select “Add Refinement”. You will see a dialog like that in Figure 50. You can specify the class name for the refinement, but for now, it is best to accept the default. Once you have created a refinement, you can look inside a state or transition. For the bouncing ball example, the refinement of the *free* state is shown in Figure 48. This model exhibits certain key properties of refinements:

- Refinements must contain directors. In this case, the CTEmbeddedDirector is used. When a continuous-time model is used inside a mode, this director must be used instead of the default CTDirector (see the CT chapter for details).
- The refinement has the same ports as the modal model, and can read input value and specify output values. When the state machine is in the state of which this is the refinement, this model will be executed to read the inputs and produce the outputs.


Edit parameters for relation2

guardExpression: true
outputActions:
setActions: alPosition = initialPosition; free.initialVelocity = 0.0
reset: true
preemptive: false

FIGURE 46. Transition dialog for the transition from *init* to *free* in Figure 44.

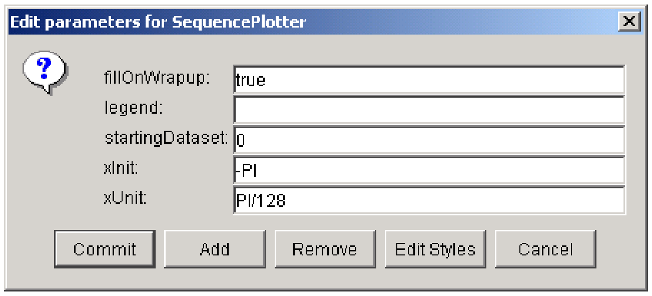
- In this case, the refinement simply defines the laws of gravity. An acceleration of -10 m/sec^2 (roughly) is integrated to get the velocity. This, in turn, is integrated to get the vertical position.
- A *ZeroCrossingDetector* actor is used to detect when the vertical position of the actor is zero. This results in production of an event on the (discrete) output *bump*. Examining Figure 2.36, you can see that this event triggers a state transition back to the same *free* state, but where the *initialVelocity* parameter is changed to reverse the sign and attenuate it by the *elasticity*. This results in the ball bouncing, and losing energy.

As you can see from Figure 44, 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. This results in the model producing no further output. The result of an execution is shown in Figure 49. Notice that the ball bounces until it stops, after which there are no further outputs.

This model illustrates an interesting property of the CT domain. 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. Try removing the *stop* state from the FSM, and re-run the model. What happens? Why?

Modal models can be used in any domain. Their behavior is simple. When the modal model is fired, the following sequence of events occurs:

- The refinement of the current state, if there is one, is fired (unless *preemptive* is true, and one of the guards on outgoing transitions evaluates to true).
- The guard expressions on all the outgoing transitions are evaluated. If none are true, the firing is complete. If one is true, then that transition is taken. If more than one is true, then an exception is thrown (the FSM is nondeterministic).
- When a transition is taken, its output actions and set actions are evaluated.


Edit parameters for SequencePlotter

fillOnWrapup:	true
legend:	
startingDataset:	0
xInit:	-PI
xUnit:	PI/128

FIGURE 47. Parameters of the SequencePlotter actor.

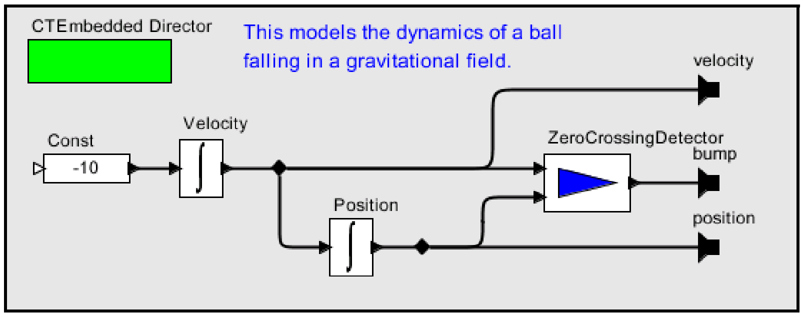


FIGURE 48. Refinement of the *free* state of the modal model in Figure 44.

- If *reset* is true, then the refinement of the destination mode (if there is one) is initialized.

Execution Semantics. The execution of a modal model follows a sequence of carefully defined steps. An *iteration* of a modal model consists of one invocation of prefire(), one or more invocations of fire(), and exactly one invocation of postfire(). The prefire() method always returns *true*, which indicates that the modal model can always be fired (at a minimum, the control logic for state transitions is executed). In each fire() invocation, the following occurs:

1. Evaluate the guard on each preemptive transition out of the current state. If exactly one guard is true, then the corresponding transition is chosen. The *output actions* of the transition are executed, and the *refinements* of the transition are iterated.
2. If no guard on a preemptive transition is true, then:

   2a. The refinements of the current state are iterated.

   2b. Evaluate the guard on each non-preemptive transition out of the current state. If exactly one of these guards is true, then the corresponding transition is chosen. The output actions of the transition are executed, and the refinements of the transition are iterated.

   If guards on multiple transitions are true, the modal model actor raises an exception that terminates the simulation. The intent is to guard against non-deterministic modal models.
   In the postfire phase:

1. The transition chosen in the last fire phase is committed.
2. The *set actions* of the transition are executed.
3. The destination state of the transition becomes the current state of the modal model, and the iteration concludes.

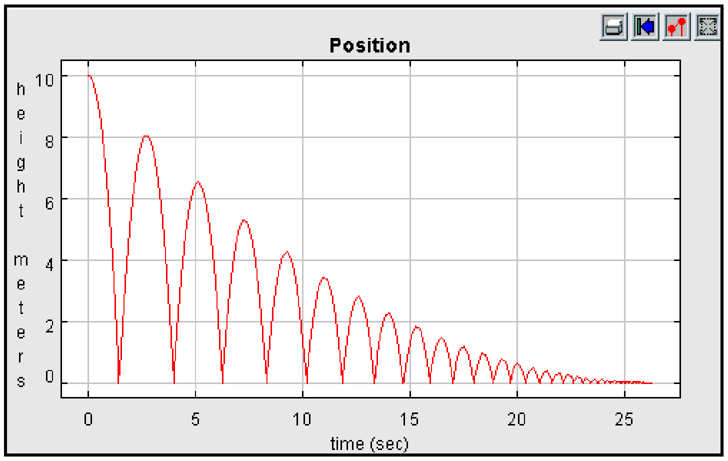


FIGURE 49. Result of execution of the bouncing ball model.

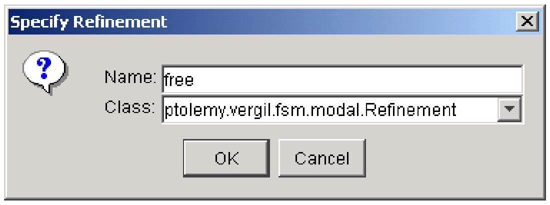
2.7 Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in Figure 39 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces 256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

The *SequencePlotter* actor has some pertinent parameters, shown in Figure 47. The *xInit* parameter specifies the value to use on the horizontal axis for the first token. The *xUnit* parameter specifies the value to increment this by for each subsequent token. Setting these to “-PI” and “PI/128” respectively results in the plot shown in Figure 51.

This plot is better, but still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in Figure 52, filled in with values that result in the plot shown in Figure 53. Most of these are self-explanatory, but the following pointers may be useful:

- The grid is turned off to reduce clutter.
- Titles and axis labels have been added.
- The X range and Y range are determined by the fill button at the upper right of the plot.
- Stem plots can be had by clicking on “Stems”


Specify Refinement

Name: free

Class: ptolemy.vergil.fsm.modal.Refinement

FIGURE 50. Dialog for creating a refinement of a state.

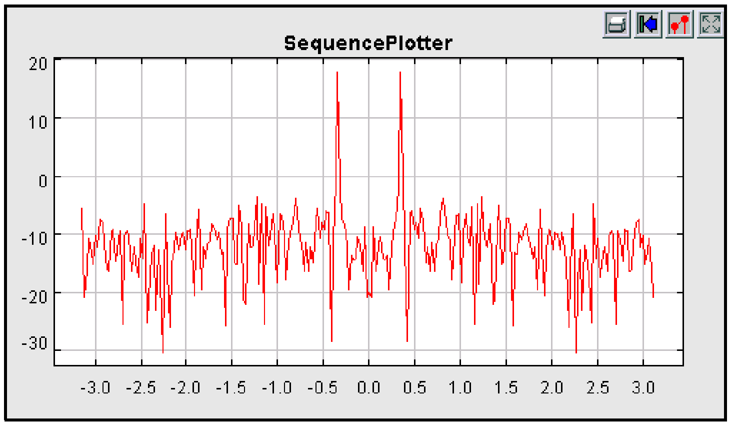


FIGURE 51. Better labeled plot, where the horizontal axis now properly represents the frequency values.

1. **Hint:** Notice the “x10^2” at the bottom right, which indicates that the label “2.5” stands for “250”.

- Individual tokens can be shown by clicking on “dots”
- Connecting lines can be eliminated by deselecting “connect”
- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

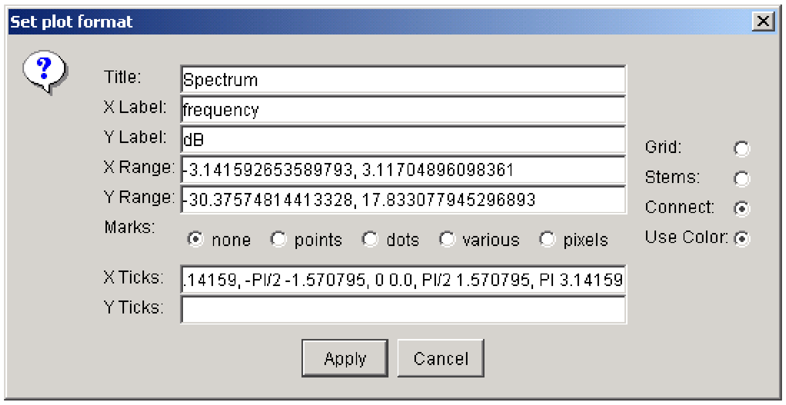
  -PI -3.14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159

  The syntax in general is:

  label value, label value, ...

  where the label is any string (enclosed in quotation marks if it includes spaces), and the value is a number.

3. Expressions



Title: Spectrum
X Label: frequency
Y Label: dB
X Range: -3.141592653589793, 3.11704896098361
Y Range: -30.37574814413328, 17.833077945296893
Marks: [✓ none] points dots various pixels
X Ticks: .14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159
Y Ticks:
Grid: ○
Stems: ○
Connect: [✓]
Use Color: [✓]

FIGURE 52. Format control window for a plot.

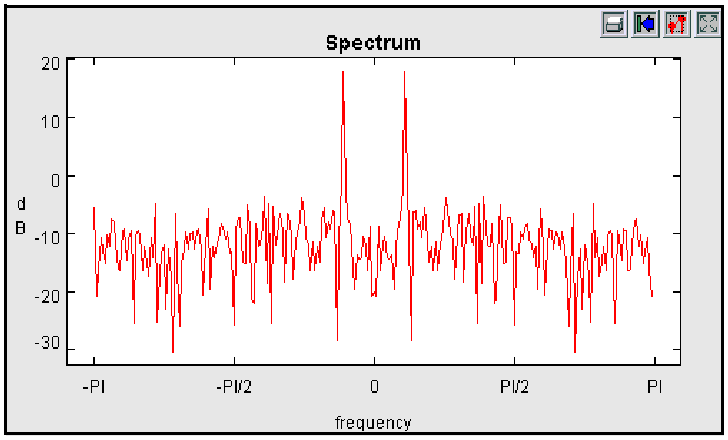


FIGURE 53. Still better labeled plot.

In HyVisual, models specify computations by composing actors. Many computations, however, are awkward to specify this way. A common situation is where we wish to evaluate a simple algebraic expression, such as “sin(2π (*x*-1)).” It is possible to express this computation by composing actors in a block diagram, but it is far more convenient to give it textually.

The expression language provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the *Expression* actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, upon which HyVisual is built.

3.1 Simple Arithmetic Expressions

3.1.1 Constants and Literals

The simplest expression is a constant, which can be given either by the symbolic name of the constant, or by a literal. By default, the symbolic names of constants supported are PI, pi, E, e, true, false, i, and j. for example,

```
PI/2.0
```

is a valid expression that refers to the symbolic name “PI” and the literal “2.0.” The constants i and j are complex numbers with value equal to 0.0 + 1.0i.

Numerical values without decimal points, such as “10” or “-3” are integers. Numerical values with decimal points, such as “10.0” or “3.14159” are doubles. Integers followed by the character “l” (el) or “L” are long integers. Integers beginning with a leading “0” are octal numbers. Integers beginning with a leading “0x” are hexadecimal numbers. For example, “012” and “0xA” are both the integer 10. In releases later than Ptolemy II 2.0.1, but not including 2.0.1 itself, integers followed by “ub” or “UB” are unsigned bytes, as in “5ub”. Literal string constants are also supported. Anything between quotes, “...”, is interpreted as a string constant.

A complex is defined by appending an “i” or a “j” to a double for the imaginary part. This gives a purely imaginary complex number which can then leverage the polymorphic operations in the Token classes to create a general complex number. Thus “`2 + 3i`” will result in the expected complex number. You can optionally write this “`2 + 3*i`”.

3.1.2 Summary of Supported Types

The types currently supported in the expression language are *boolean*, *unsigned byte*, *complex*, *fixedpoint*, *double*, *int*, *long*, *array*, *matrix*, *record*, and *string*. The composite types, array, matrix, and record, are described below in section 3.3. Note that there is no float (as yet). Use double or int instead. A long is defined by appending an integer with “l” (lower case L) or “L”, as in Java. A fixed point number is defined using the “fix” function, as will be explained below in section 3.5.

3.1.3 Variables

Expressions can contain references to variables within the *scope* of the expression. For example,

```
PI*x/2.0
```

is valid if “x” is a variable in scope. In the context of Ptolemy II models, the variables in scope include all parameters defined at the same level of the hierarchy or higher. So for example, if an actor has a parameter named “x” with value 1.0, then another parameter of the same actor can have an expression with value “`PI*x/2.0`”, which will evaluate to π /2.

Consider a parameter *P* in actor *X* which is in turn contained by composite actor *Y*. The scope of an expression for *P* includes all the parameters contained by *X* and *Y*, plus those of the container of *Y*, its container, etc. That is, the scope includes any parameters defined above in the hierarchy.

You can add parameters to actors (composite or not) by right clicking on the actor, selecting “Configure” and then clicking on “Add”, or by dragging in a parameter from the *utilities* library. Thus, you can add variables to any scope, a capability that serves the same role as the “let” construct in many programming languages.

3.1.4 Operators

The arithmetic operators are +, –, *, /, ^, and %. Most of these operators operate on most data types, including matrices. The ^ operator computes “to the power of” where the power can only be an integer. The bitwise operators are &, |, #, and ~. They operate on integers, where & is bitwise and, ~ is bitwise not, and | is bitwise or, and # is bitwise exclusive or (after MATLAB).

The relational operators are <, <=, >, >=, == and !=. They return booleans. Boolean-valued expressions can be used to give conditional values. The syntax for this is

```
   boolean ? value1 : value2
```

If the boolean is true, the value of the expression is `value1`; otherwise, it is `value2`.

The logical boolean operators are &&, ||, !, & and |. They operate on booleans and return booleans. The difference between logical && and logical & is that & evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical || and |. This approach is borrowed from Java.

The << and >> operators performs arithmetic left and right shifts respectively. The >>> operator performs a logical right shift, which does not preserve the sign.

3.1.5 Comments

In expressions, anything inside /*...*/ is ignored, so you can insert comments.

3.2 Uses of Expressions

3.2.1 Parameters

The values of most parameters of actors can be given as expressions[1]. The variables in the expression refer to other parameters that are in scope, which are those contained by the same container or some container above in the hierarchy. Adding parameters to actors is straightforward, as explained in section 2.2.4.

3.2.2 Port Parameters

It is possible to define a parameter that is also a port. Such a *PortParameter* provides a default value, which is specified like the value of any other parameter. When the corresponding port receives

1. The exceptions are parameters that are strictly string parameters, in which case the value of the parameter is the literal string, not the string interpreted as an expression, as for example the *function* parameter of the *TrigFunction* actor, which can take on only “sin,” “cos,” “tan”, “asin”, “acos”, and “atan” as values.

data, however, the default value is overridden with the value provided at the port. Thus, this object function like a parameter and a port. The current value of the PortParameter is accessed like that of any other parameter. Its current value will be either the default or the value most recently received on the port.

A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from from the *utilities* library, as shown in figure 54. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parameter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

To be useful, a PortParameter has to be given a name (the default name, “portParameter,” is not very compelling). To change the name, right click on the icon and select “Customize Name,” as shown in figure 54. In the figure, the name is set to “noiseLevel.” Then set the default value by either double clicking or selecting “Configure.” In the figure, the default value is set to 10.0.

In a continuous-time model, if a PortParameter is supplied with discrete data at the port, then it must be declared DISCRETE. To do this, create a parameter in the PortParameter with name *signal-Type* and value "DISCRETE" (with the quotation marks).

3.2.3 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output an no inputs, as shown in Figure 55(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).

3.2.4 State Machines

Expressions give the guards for state transitions, as well as the values used in actions that produce

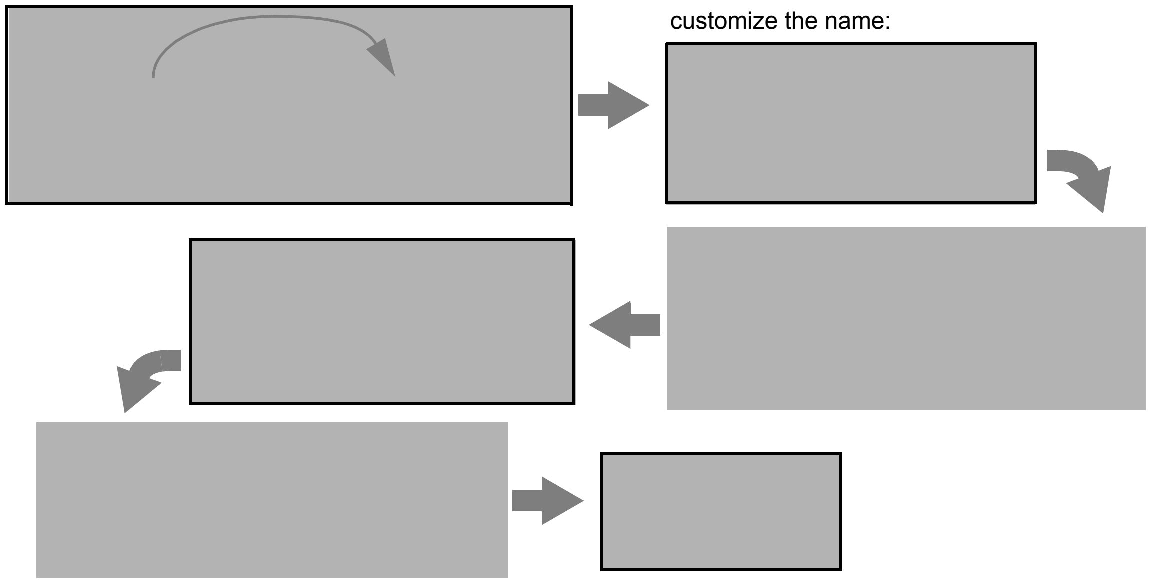


FIGURE 54. A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful, and set its default value.

outputs and actions that set values of parameters in the refinements of destination states. This mechanism was explained earlier.

3.3 Composite Data Types

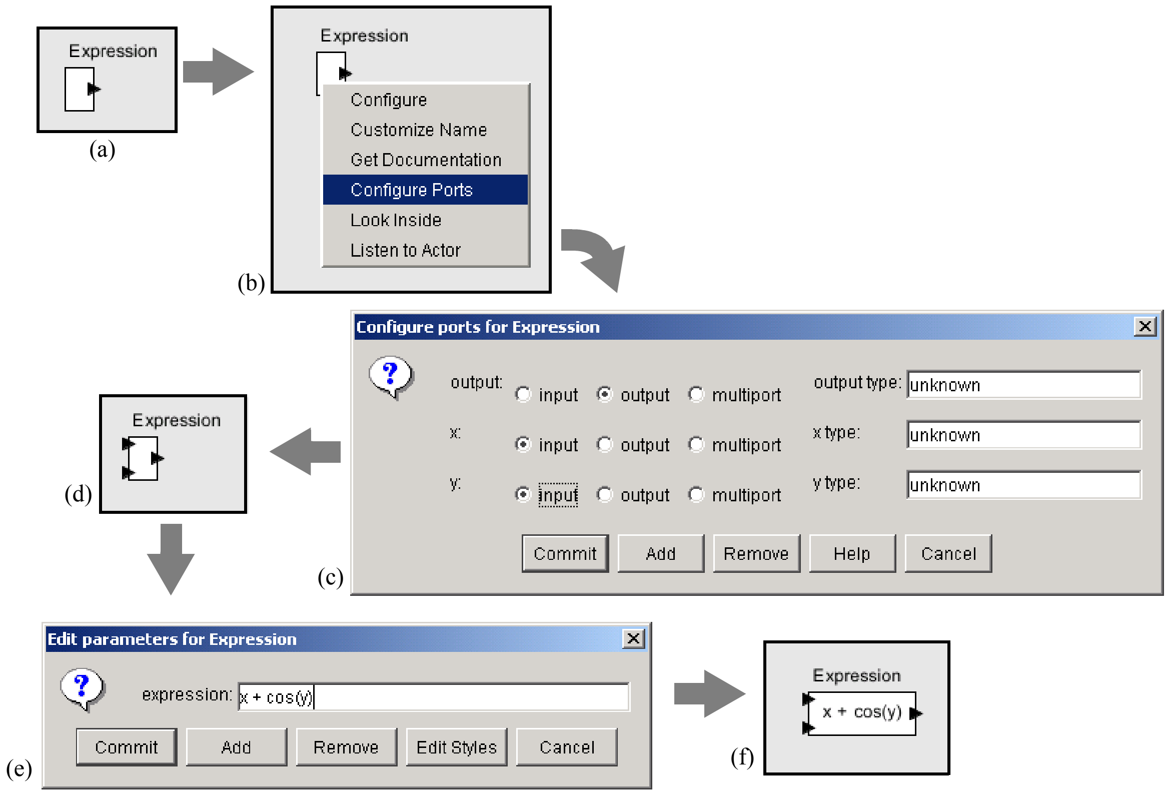
3.3.1 Arrays

Arrays are specified with curly brackets, e.g., “{1, 2, 3}” is an array of integers, while “`{"x", "y", "z"}`” is an array of strings. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have the same type. Thus, for example, “{1, 2.3}” is illegal because the first element is an integer and the second is a double. The elements of the array can be given by expressions, as in the example “{2*pi, 3*pi}.” Arrays can be nested; for example, “{{1, 2}, {3, 4, 5}}” is an array of arrays of integers.

3.3.2 Matrices

In HyVisual, arrays are ordered sets of tokens. Ptolemy II also supports matrices, which are more specialized than arrays. They contain only primitive types, currently *boolean*, *complex*, *double*, *fixed-point*, *int*, and *long*. Matrices cannot contain arbitrary tokens, so they cannot, for example, contain matrices. They are intended for data intensive computations.

Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., “[1, 2, 3; 4, 5, 5+1]” gives a two by three integer matrix (2 rows and 3 col-



output: input [✓ output] multiport   output type: unknown
x: [✓ input] output multiport   x type: unknown
y: [✓ input] output multiport   y type: unknown

expression: x + cos(y)

FIGURE 55. Illustration of the *Expression* actor.

umns). Note that an array or matrix element can be given by an expression, but all elements must have the same type, and that type must be one of the types for which matrices are defined. A row vector can be given as “[1, 2, 3]” and a column vector as “[1; 2; 3]”. Some MATLAB-style array constructors are supported. For example, “[1:2:9]” gives an array of odd numbers from 1 to 9, and is equivalent to “[1, 3, 5, 7, 9].” Similarly, “[1:2:9; 2:2:10]” is equivalent to “[1, 3, 5, 7, 9; 2, 4, 6, 8, 10].” In the syntax “[*p*:*q*:*r*]”, *p* is the first element, *q* is the step between elements, and *r* is an upper bound on the last element. That is, the matrix will not contain an element larger than *r*.

Reference to matrices have the form “*name*(*n*, *m*)” where *name* is the name of a matrix variable in scope, *n* is the row index, and *m* is the column index. Index numbers start with zero, as in Java, not 1, as in MATLAB.

3.3.3 Records

A record token is a composite type where each element is named, and each element can have a distinct type. Records are delimited by curly braces, with each element given a name. For example, “`{a=1, b="foo"}`” is a record with two elements, named “a” and “b”, with values 1 (an integer) and “foo” (a string), respectively. The value of a record element can be an arbitrary expression, and records can be nested (an element of a record token may be a record token).

Fields of records may be accessed using the period operator. For example,

```
{a=1,b=2}.a
```

yields 1. You can optionally write this as if it were a method call:

```
{a=1,b=2}.a()
```

3.4 Functions and Methods

3.4.1 Functions

The expression language includes an extensible set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the java.lang.Math class and the ptolemy.data.expr.UtilityFunctions class. This can easily be extended by registering another class that includes static methods. The functions currently available are shown in Figures 56 and 57, with the argument types and return types[1].

One slightly subtle function is the random() function shown in Figure 56. It takes no arguments, and hence is written “`random()`”. It returns a random number. However, this function is evaluated only when the expression within which it appears is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. The random() function is not called again. Thus, for example, if the *value* parameter of the *Const* actor is set to “`random()`”, then its output will be a random constant, i.e., it will not change on each firing.

The property() function accesses system properties by name. Some possibly useful system properties are:

- ptolemy.ptII.dir: The directory in which Ptolemy II is installed.
- ptolemy.ptII.dirAsURL: The directory in which Ptolemy II is installed, but represented as a URL.
- user.dir: The current working directory, which is usually the directory in which the current executable was started.

1. At this time, in release 2.0, the types must match exactly for the expression evaluator to work. Thus, “sin(1)” fails, because the argument to the sin() function is required to be a double.

function	argument type(s)	return type	description
abs	double	double	absolute value
abs	int	int	absolute value
abs	long	long	absolute value
acos	double	double	arc cosine
asin	double	double	arc sine
atan	double	double	arc tangent
atan2	double, double	double	angle of a vector
ceil	double	double	ceiling function
cos	double	double	cosine
exp	double	double	exponential function (e^argument)
floor	double	double	floor function
IEEEremainder	double, double	double	remainder after division
log	double	double	natural logarithm
max	double, double	double	maximum
max	int, int	int	maximum
max	long, long	long	maximum
min	double, double	double	minimum
min	int, int	int	minimum
min	long, long	long	minimum
pow	double, double	double	first argument to the power of the second
random		double	random number between 0.0 and 1.0
rint	double	double	round to the nearest integer
round	double	long	round to the nearest integer
sin	double	double	sine function
sqrt	double	double	square root
tan	double	double	tangent function
toDegrees	double	double	convert radians to degrees
toRadians	double	double	convert degrees to radians

FIGURE 56. Functions available to the expression language from the java.lang.Math class.

3.4.2 Methods

Every element and subexpression in an expression represents an instance of the Token class in Ptolemy II (or more likely, a class derived from Token). The expression language supports invocation of any method of a given token, as long as the arguments of the method are of type Token and the return type is Token (or a class derived from Token, or something that the expression parser can easily convert to a token, such as a string, double, int, etc.). The syntax for this is (*token*).*methodName*(*args*), where *methodName* is the name of the method and *args* is a comma-separated set of arguments. Each argument can itself be an expression. Note that the parentheses around the *token* are not required, but might be useful for clarity. As an example, the ArrayToken class has a getElement(int) method, which can be used as follows:

```
{1, 2, 3}.getElement(1)
```

This returns the integer 2. Another useful function of array token is illustrated by the following example:

```
{1, 2, 3}.length()
```

function	argument type(s)	return type	description
constants	none	record	Record identifying all the globally defined constants in the expression language.
freeMemory	none	long	Return the approximate number of bytes available for future memory allocation.
gaussian	double, double	double	Gaussian random variable with the specified mean, and standard deviation
gaussian	double, double, int, int	double matrix	Gaussian random matrix with the specified mean, standard deviation, rows, and columns
property	string	string	Return a system property with the specified name from the environment, or an empty string if there is none.
readFile	string	string	Get the string text in the specified file. Return an empty string if the file is not found.
readMatrix	string	double matrix	Read a file that contains a matrix of reals in MATLAB notation.
repeat	int, general	array	Create an array by repeating the specified token the specified number of times.
totalMemory	long	none	Return the approximate number of bytes used by current objects plus those available for future object allocation.
findFile	string	string	Return an absolute file name given one that is relative to the user directory or the classpath.

FIGURE 57. Functions available to the expression language from the ptolemy.data.expr.UtilityFunctions class. This class is still at a preliminary stage, and the function it provides will grow over time.

which returns the integer 3.

The MatrixToken classes have three particularly useful methods, illustrated in the following examples:

```
[1, 2; 3, 4; 5, 6].getRowCount()
```

which returns 3, and

```
[1, 2; 3, 4; 5, 6].getColumnCount()
```

which returns 2, and

```
[1, 2; 3, 4; 5, 6].toArray()
```

which returns {1, 2, 3, 4, 5, 6}. The latter function can be particularly useful for creating arrays using MATLAB-style syntax. For example, to obtain an array with the integers from 1 to 100, you can enter:

```
[1:1:100].toArray()
```

The get() method of RecordToken accesses a record field, as in the following example:

```
{a=1, b=2}.get("a")
```

which returns 1.

The Token classes from the data package form the primitives of the language. For example the number 10 becomes an IntToken with the value 10 when evaluating an expression. Normally this is invisible to the user. The expression language is object-oriented, of course, so methods can be invoked on these primitives. A sophisticated user, therefore, can make use of the fact that "10" is in fact an object to invoke methods of that object.

In particular, the convert() method of the Token class might be useful, albeit a bit subtle in how it is used. For example:

```
double.convert(1)
```

creates a DoubleToken with value 1.0. The variable *double* is a built-in constant with type double. The convert() method of DoubleToken converts the argument to a DoubleToken, so the result of this expression is 1.0. A more peculiar way to write this is

```
(1.2).convert(1)
```

Any double constant will work in place of 1.2. Its value is irrelevant.

The convert() method supports only lossless type conversion. Lossy conversion has to be done explicitly via a function call.

3.5 Fixed Point Numbers

Ptolemy II includes a preliminary fixed point data type. We represent a fixed point value in the expression language using the following format:

```
fix(value, totalBits, integerBits)
```

Thus, a fixed point value of 5.375 that uses 8 bit precision of which 4 bits are used to represent the integer part can be represented as:

```
fix(5.375, 8, 4)
```

The value can also be a matrix of doubles. The values are rounded, yielding the nearest value representable with the specified precision. If the value to represent is out of range, then it is saturated, meaning that the maximum or minimum fixed point value is returned, depending on the sign of the specified value. For example,

```
fix(5.375, 8, 3)
```

will yield 3.968758, the maximum value possible with the (8/3) precision.

In addition to the fix() function, the expression language offers a quantize() function. The arguments are the same as those of the fix() function, but the return type is a DoubleToken or DoubleMatrixToken instead of a FixToken or FixMatrixToken. This function can therefore be used to quantize double-precision values without ever explicitly working with the fixed-point representation.

To make the FixToken accessible within the expression language, the following functions are available:

- To create a single FixPoint Token using the expression language:
  ```
  fix(5.34, 10, 4)
  ```
  This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.
- To create a Matrix with FixPoint values using the expression language:
  ```
  fix([ -.040609, -.001628, .17853 ], 10,  2)
  ```
  This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a FixPoint value with precision(10/2). The resulting FixMatrixToken will try to fit each element of the given double matrix into a 10 bit representation with 2 bits used for the integer part. By default the round quantizer is used.
- To create a single DoubleToken, which is the quantized version of the double value given, using the expression language:
  ```
  quantize(5.34, 10, 4)
  ```
  This will create a DoubleToken. The resulting DoubleToken contains the double value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.
- To create a Matrix with doubles quantized to a particular precision using the expression language:
  ```
  quantize([ -.040609, -.001628, .17853 ], 10,  2)
  ```

This will create a DoubleMatrixToken with 1 row and 3 columns. The elements of the token are obtained by fitting the given matrix elements into a 10 bit representation with 2 bits used for the integer part. Instead of being a fixed point value, the values are converted back to their double representation and by default the round quantizer is used.