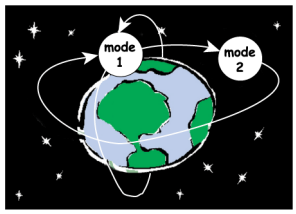
HYVISUAL: A HYBRID SYSTEM VISUAL MODELER

Authors[1]: *Adam Cataldo*
Christopher Hylands
Edward A. Lee
Jie Liu
Xiaojun Liu
Steve Neuendorffer
Haiyang Zheng



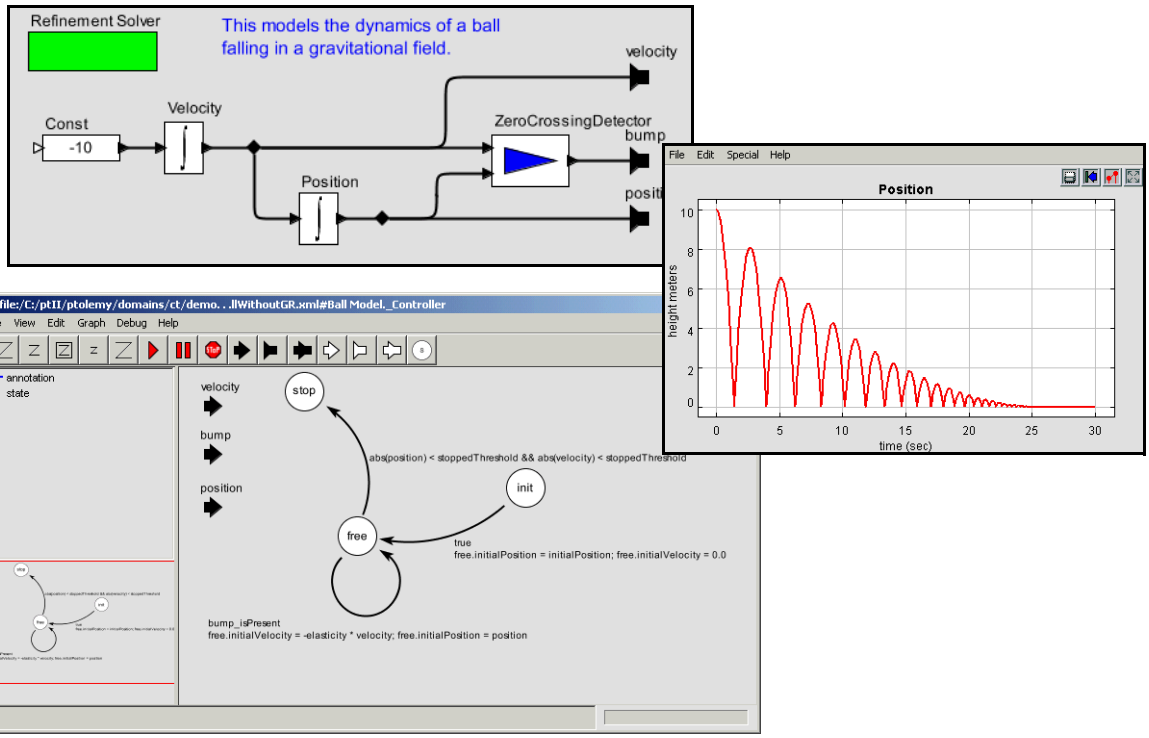
Ptolemy II 4.0-beta

Version 3.0[2]
UCB ERL Memorandum M03/30
July 17, 2003

Note that this document is out of date.

The final release will a new version.

This document describes work that is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Atmel, Cadence Design Systems, Hitachi, National Semiconductor, and Philips.



1. With contributions from the entire Ptolemy II team, but most especially John Reekie and Yuhong Xiong.
2. Version numbers for HyVisual match the version of Ptolemy II on which it is based.

Contents

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 Installation and Quick Start

HyVisual can be quickly downloaded and run using Web Start from the web site:

```
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 should then 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.

HyVisual is also available as a standalone installer for Windows (.exe file), experimental installers for other platforms, and as a CD. This document is included with the software in PDF format, so if you would like to read the HyVisual Documentation online, then you may need to install the Adobe Acrobat Reader. HyVisual-3.0 is also part of the Ptolemy II 3.0 release. HyVisual requires Java 1.4 or later. Java 1.4.1_01 is preferred.

1.1.1 Web Start

Web Start is a tool from Sun Microsystems that makes software installation and updates particularly simple. The Web Start installation works best with Windows, but has also been tried under Solaris, Linux and Mac OS X. The Web Start installation behaves almost exactly like a standalone installation; you can save models locally, and you need not be connected to the net after the initial

installation. The Web Start tool includes a Java Runtime Environment (JRE), and the HyVisual Web Start installer checks that the proper version of the JRE is present.

1.1.2 Standard Installers

The Windows installer and the experimental installers for other platforms are shipped as a single executable. One of the Windows installers includes a 1.4.1_01 Java Runtime Environment (JRE). Note that this JRE will be installed as a private copy and will not be directly accessible by other programs. Under Windows, the installer will create a Ptolemy, HyVisual 2.2-beta menu choice in the Start menu.

1.1.3 CD

The HyVisual CD includes installers for HyVisual, the Java Runtime Environment version 1.4.1_01 and Adobe Acrobat Reader 5.1.

FIGURE 1. Initial welcome window.

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,

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

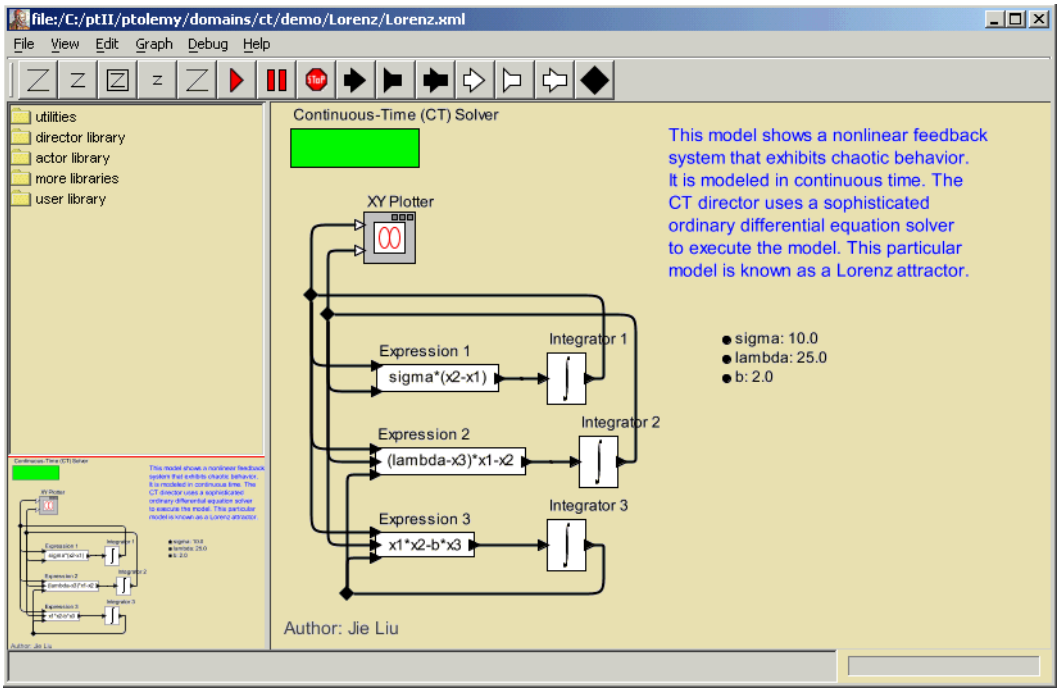


FIGURE 2. A block diagram representation of a set of nonlinear ordinary differential equations.

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

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

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

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.

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 val-

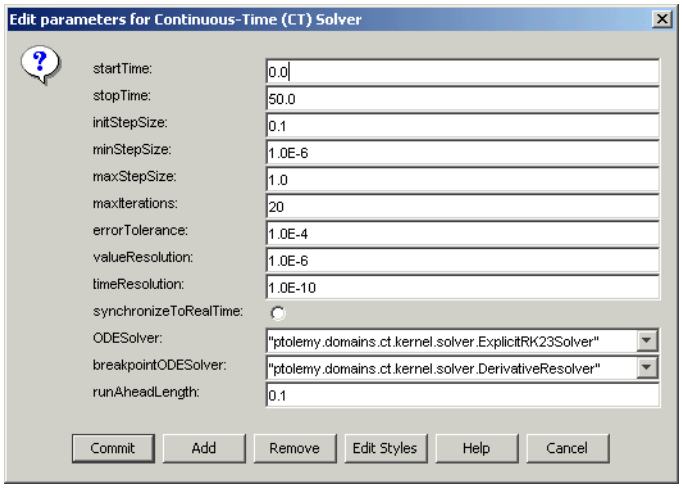


FIGURE 3. Dialog box showing solver parameters for the model in figure 2.

ues 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.

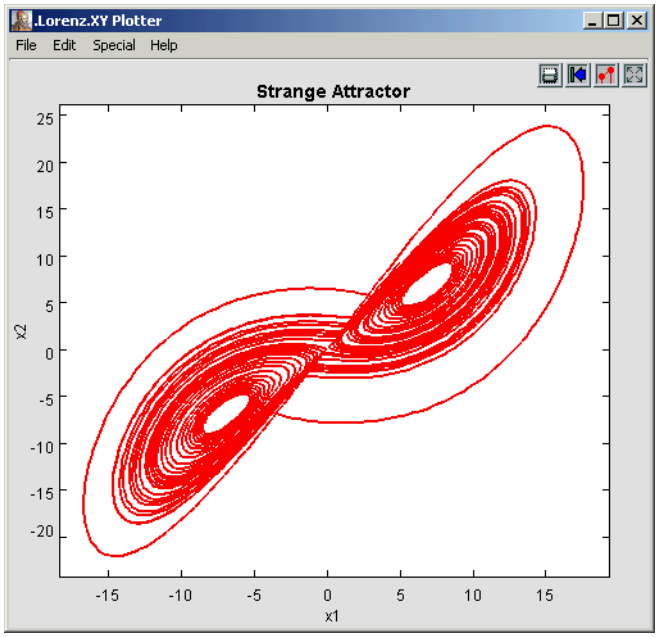


FIGURE 4. Result of running the Lorenz model using the run button in the toolbar.

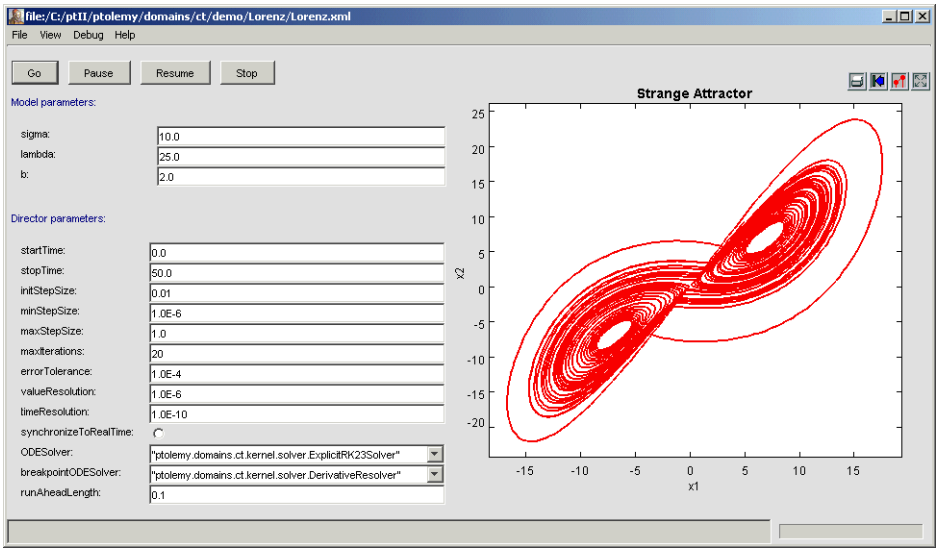


FIGURE 5. Run Window, obtained via the View menu, for the Lorenz model shown in figure 2.

2.2 Creating a New Model

This section walks through the mechanics of constructing simple models. The appendix on page 59 walks through an exercise of constructing a hybrid system example. First, 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 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 *ContinuousSinewave* actor from the *timed sources* library and the *TimedPlotter* from the *timed sinks* library. Connect the output of the *ContinuousSinewave* 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.

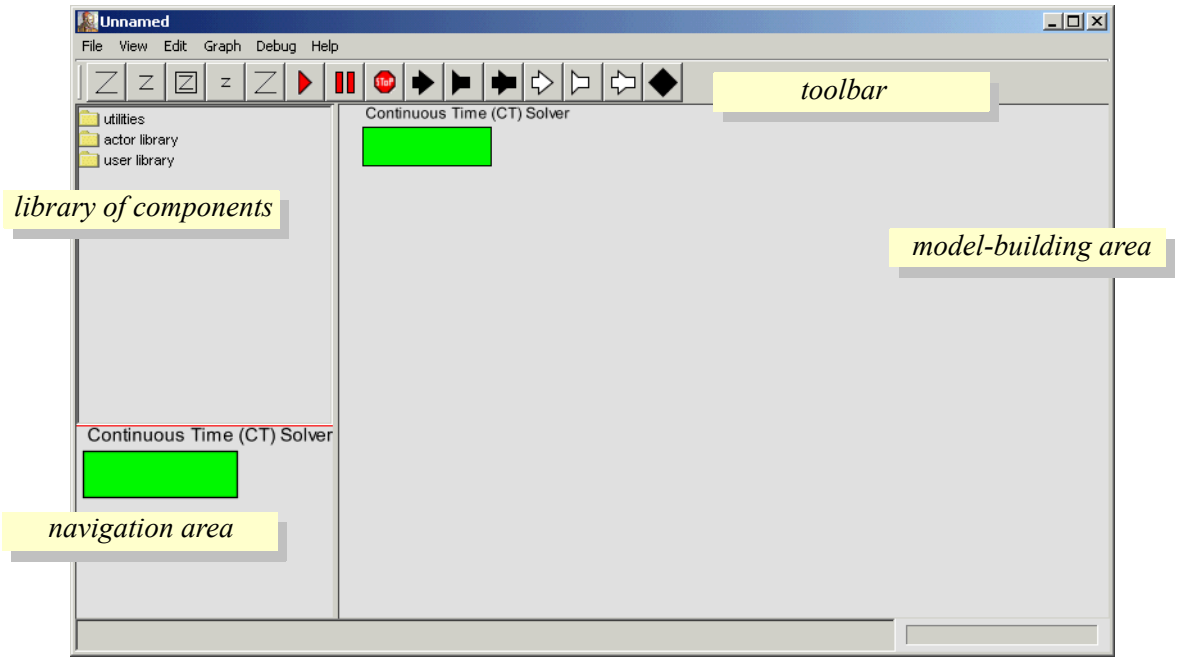


FIGURE 6. A blank model, obtained via File, New, and Graph Editor in the menus.

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*

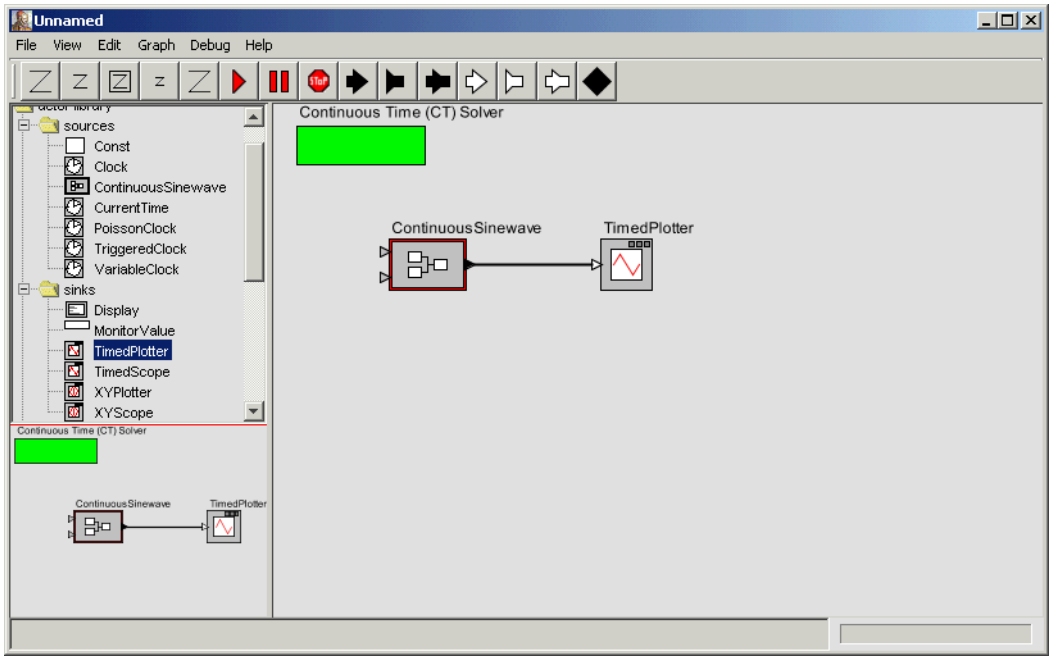


FIGURE 7. A model populated with two actors.

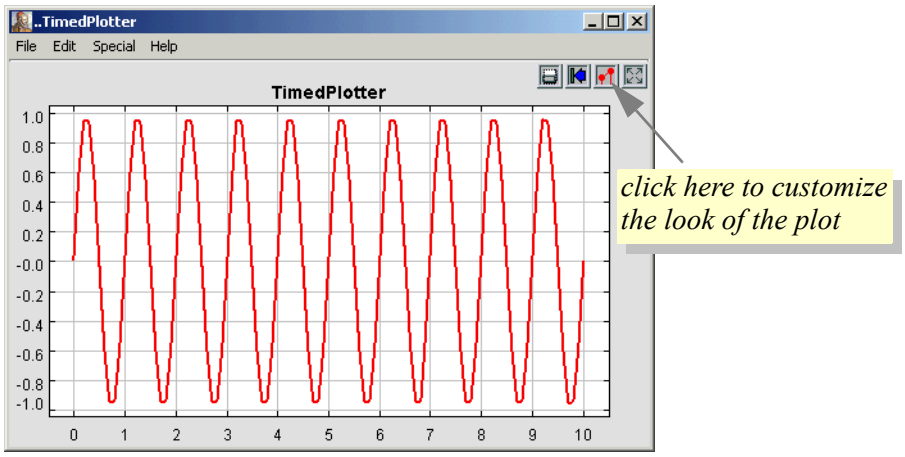


FIGURE 8. Execution of the sine wave example in figure 7, where all parameter values have default.values.

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 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.

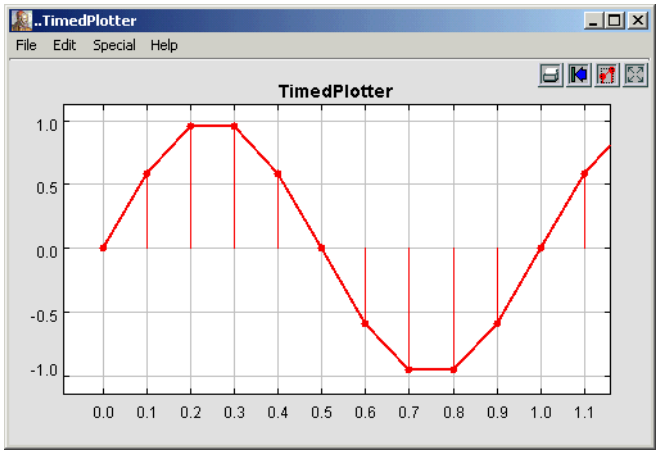


FIGURE 9. Zoomed version of the plot in figure 8, with “dots” and “stems” turned on.

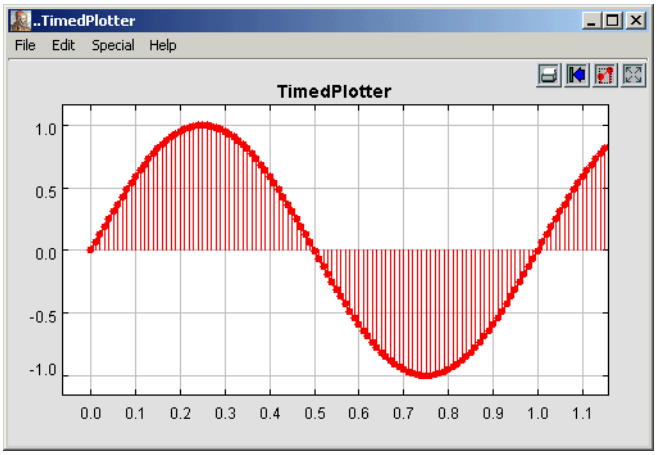


FIGURE 10. The result of running the model in figure 7 with the *initStepSize* parameter of the solver being 0.01.

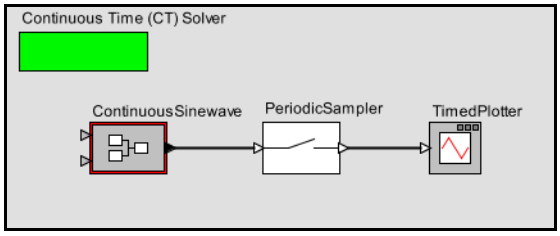


FIGURE 11. Another way to control the step size is to insert a sampler.

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 initial 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

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

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*

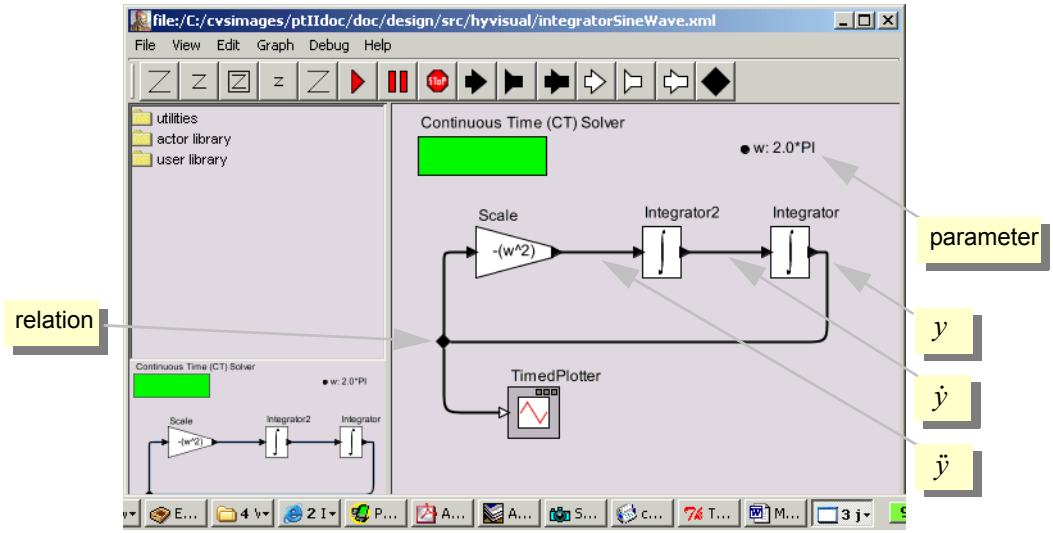


FIGURE 12. Model that generates a sine wave if the integrators have a non-zero initial condition.

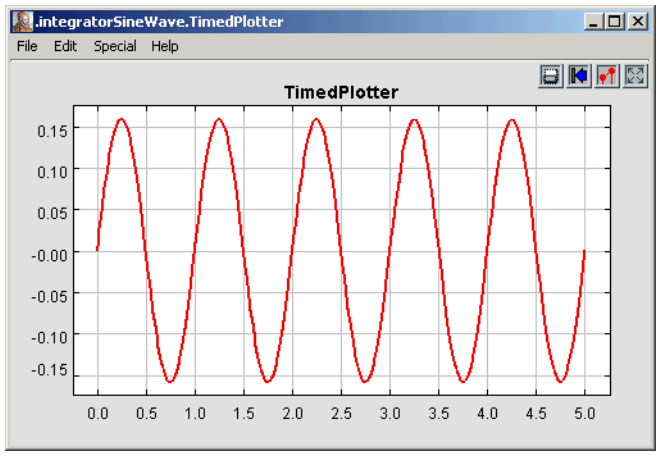


FIGURE 13. Result of running the model in figure 12 with the *initialState* of the left integrator set to 1.0.

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.

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 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*.

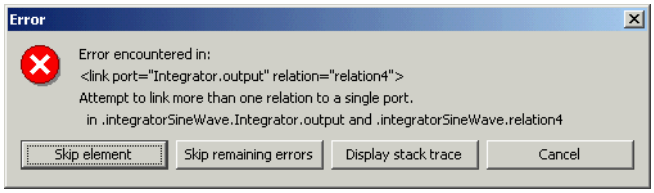


FIGURE 14. An exception that results from attempting to make a multi-way connection without a relation.

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.

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 5.

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.

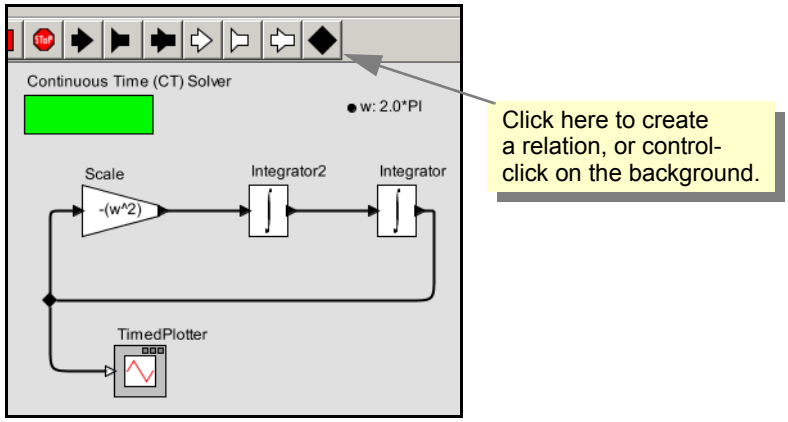


FIGURE 15. A relation can be used to broadcast an output from a single port.

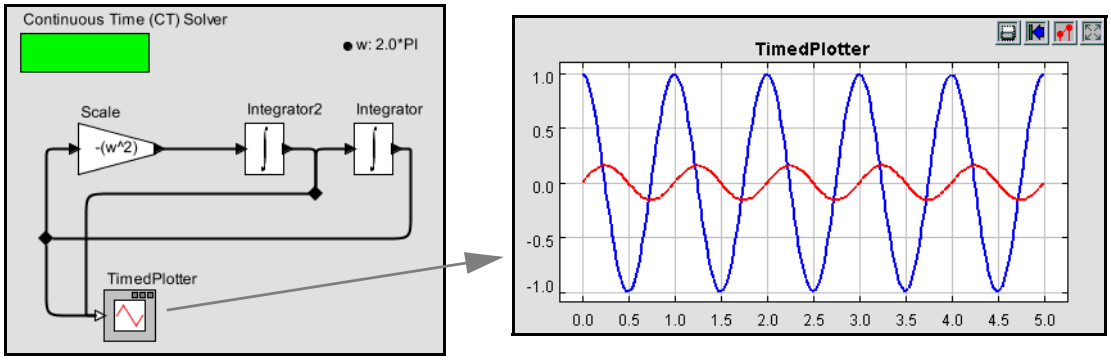


FIGURE 16. Multiple signals can be sent to a multiport, shown with the unfilled triangle on the *TimedPlotter*. In this case, two signals are plotted.

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:

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

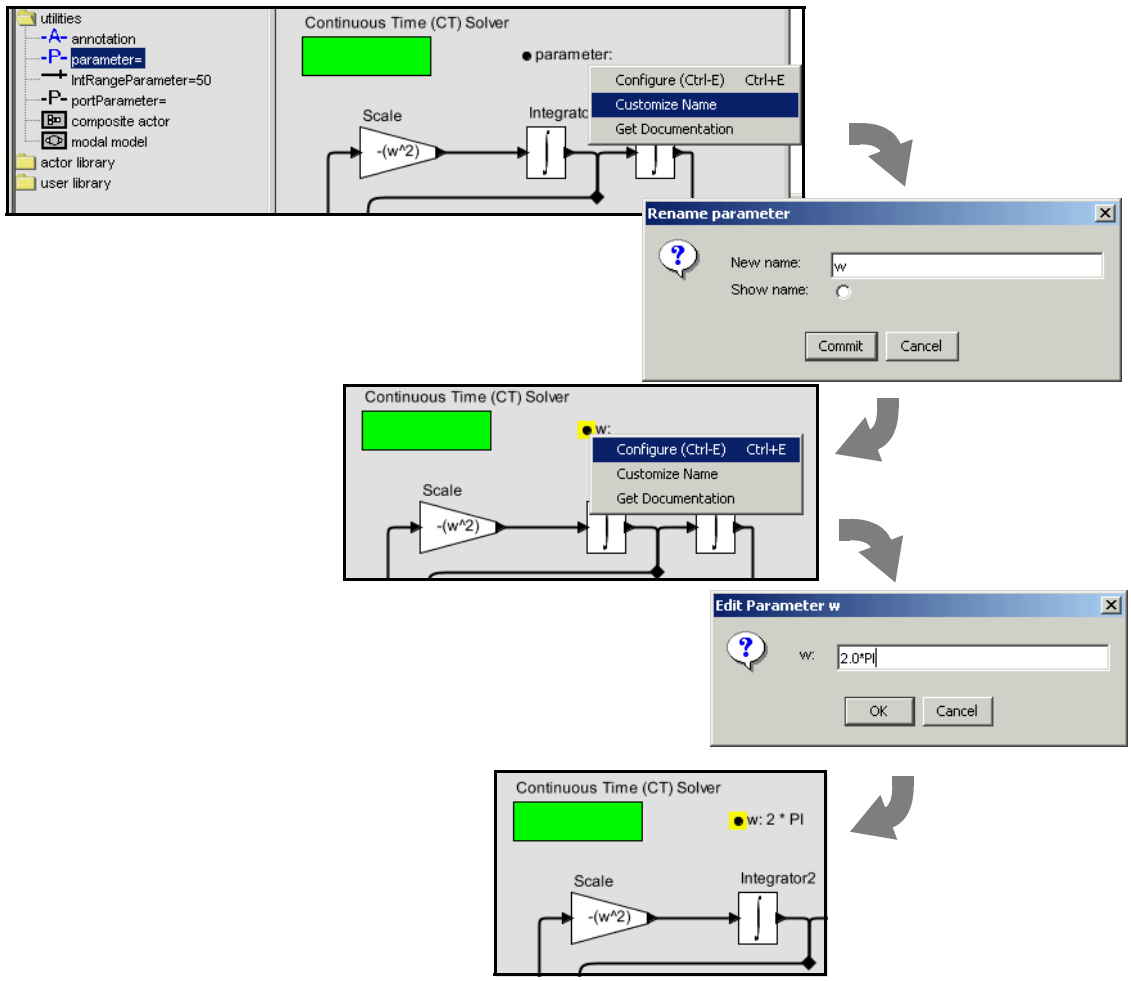


FIGURE 17. Adding a parameter to the channel model.

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 $t = 0$, 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

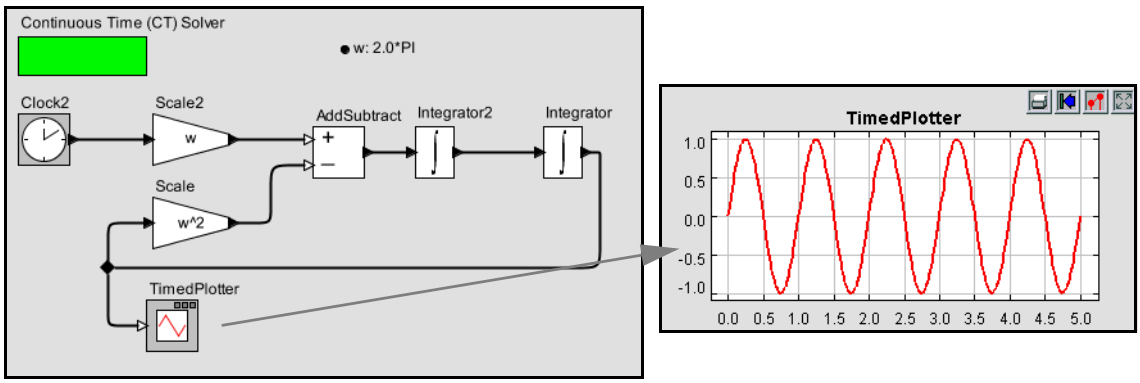


FIGURE 18. Variant of figure 12 that has an input, which is provided by the Clock actor.

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

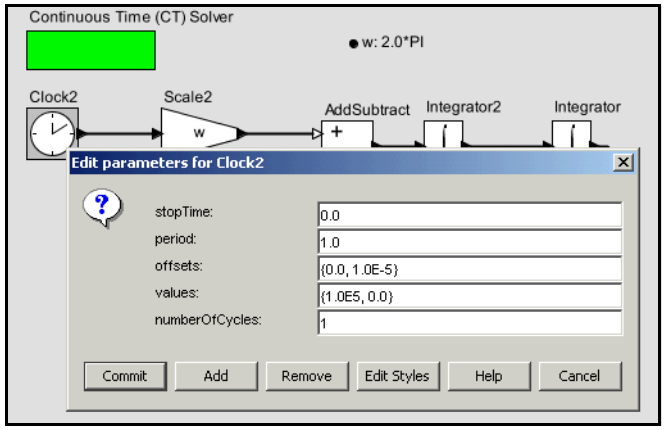


FIGURE 19. Parameters of the Clock actor that get it to output an approximate impulse.

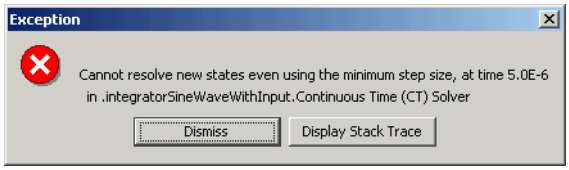


FIGURE 20. Exception due to running the model with the *minStepSize* parameter set too high.

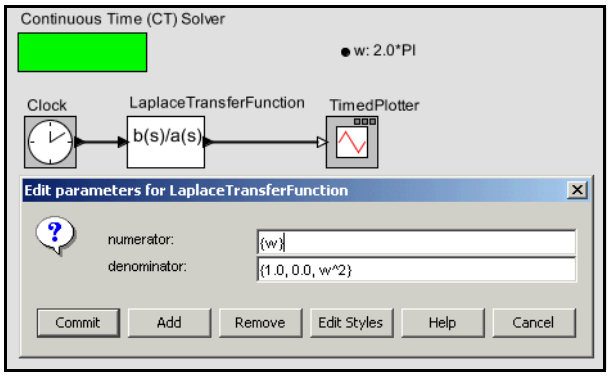


FIGURE 21. A model equivalent to that in figure 18, but using the ContinuousTransferFunction actor.

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

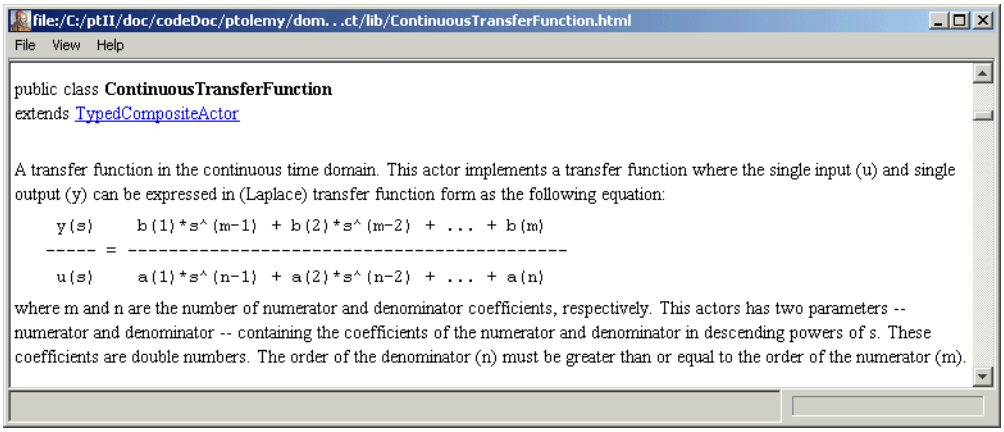


FIGURE 22. A portion of the documentation for the ContinuousTransferFunction actor.

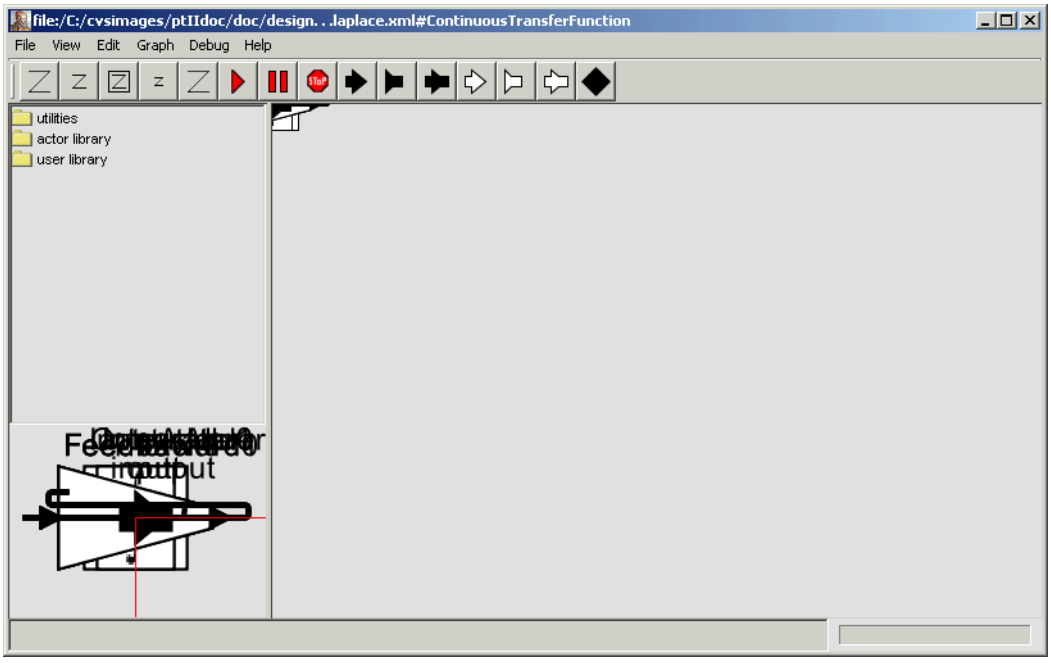


FIGURE 23. Inside the ContinuousTransferFunction actor of figure 21.

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.

2.3 Data Types

In the example of figure 7, the *ContinuousSinewave* 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. 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*. HyVisual includes a sophisticated type system[1] that allows this to be done efficiently and safely. 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.

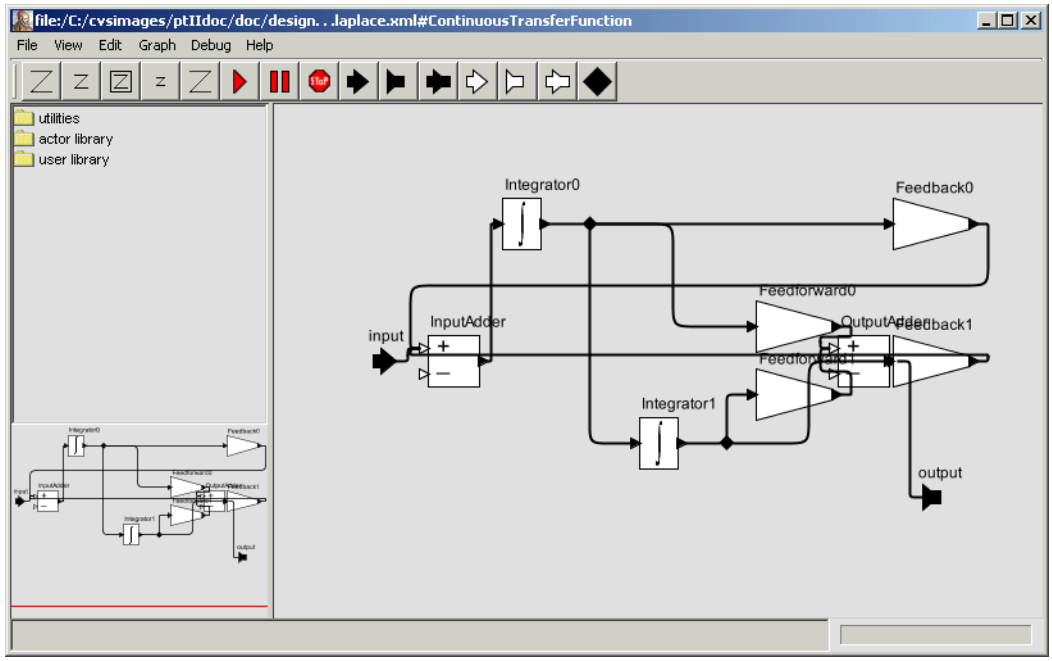


FIGURE 24. The diagram of figure 23, after invoking Automatic Layout from the Graph menu.

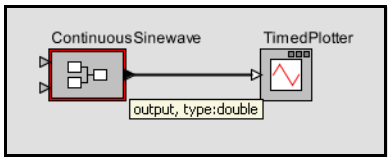


FIGURE 25. Tooltip showing the name and data type of the output port of the ContinuousSinewave of figure 7.

1. Developed by Yuhong Xiong and realized in Ptolemy II.

To explore data types a bit further, try creating the model in Figure 26. The *Const* and *CurrentTime* actors are listed under *timed sources* within *sources*, the *AddSubtract* actor is listed under *math,* and the *MonitorValue* actor is listed under *timed sinks* within *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 5. 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.

We can make 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

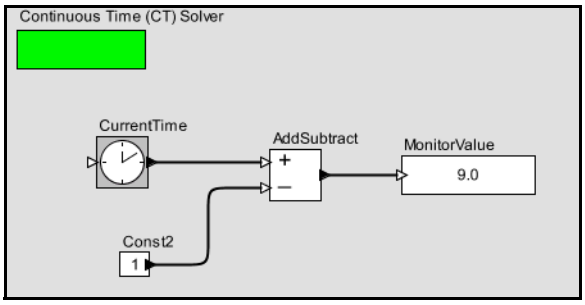


FIGURE 26. Another example, used to explore data types in HyVisual.

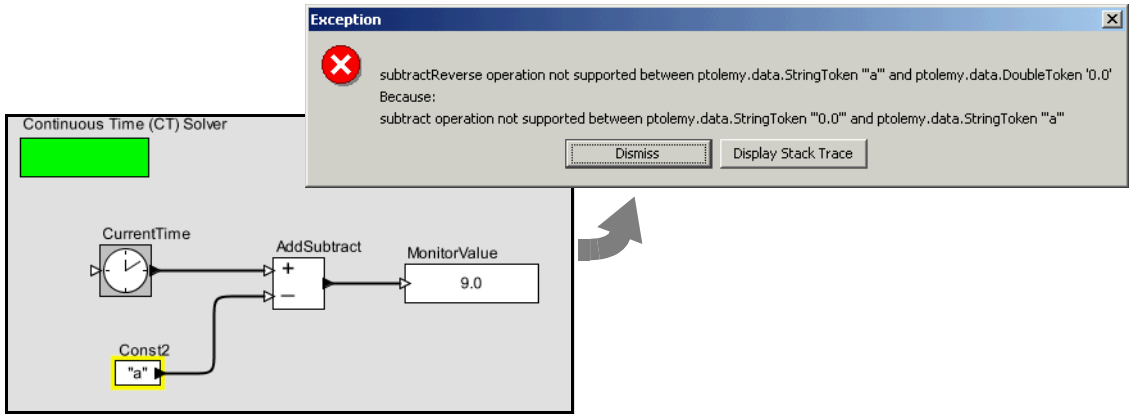


FIGURE 27. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from doubles.

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

HyVisual supports (and encourages) hierarchical models. These are models that contain components that are themselves models. Such components are called *composite actors*. Suppose we wish to take the sine wave generated by the previous examples and send it through a model of a noisy channel. We will create a composite actor modeling the channel, and then use that actor in a model.

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. Notice that the new window has no solver. It will be executed by the solver belonging to the parent model.

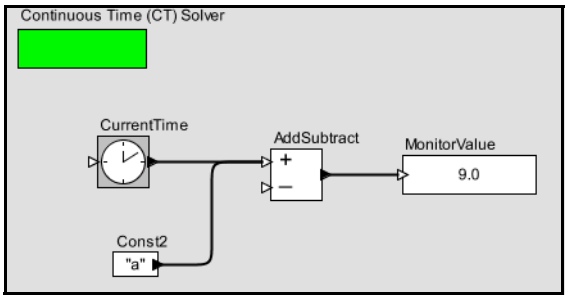


FIGURE 28. Addition of a string to an integer.

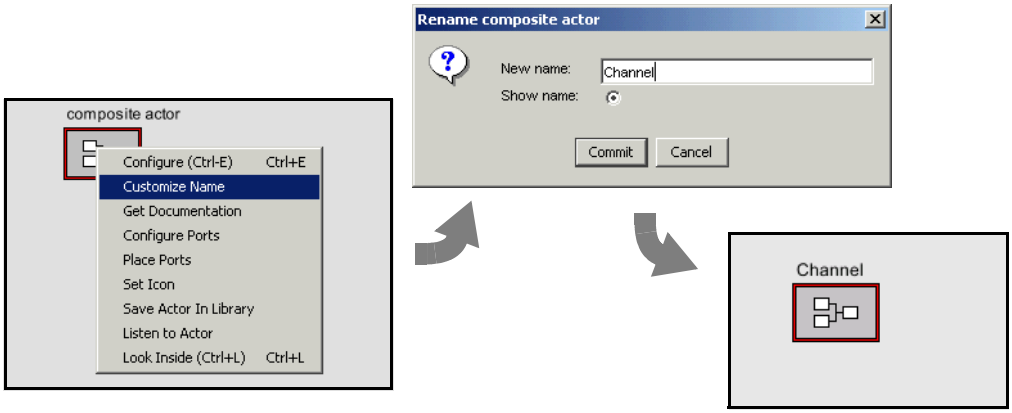


FIGURE 29. Changing the name of an actor.

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 *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,

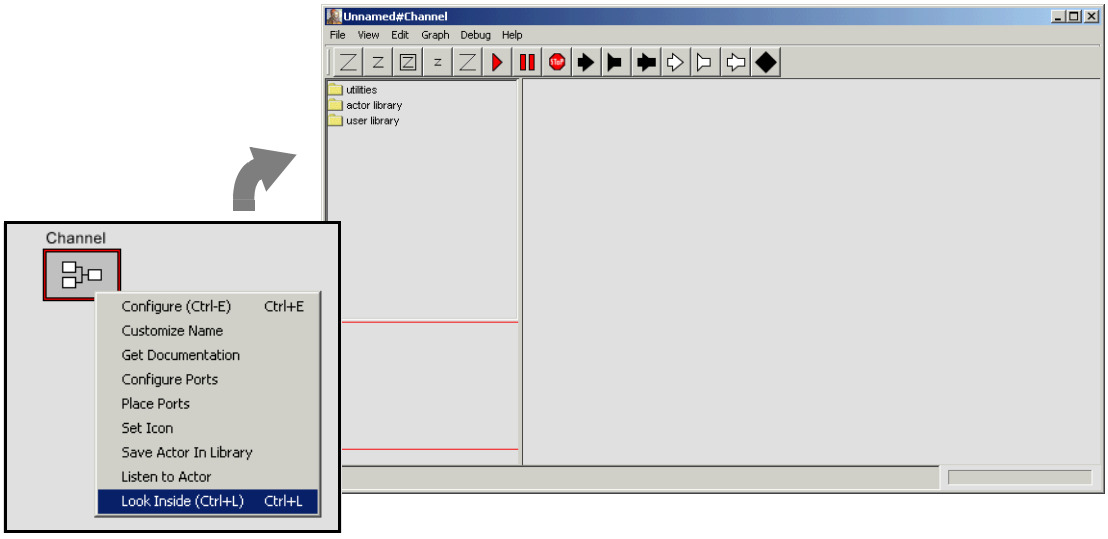


FIGURE 30. Looking inside a composite actor.

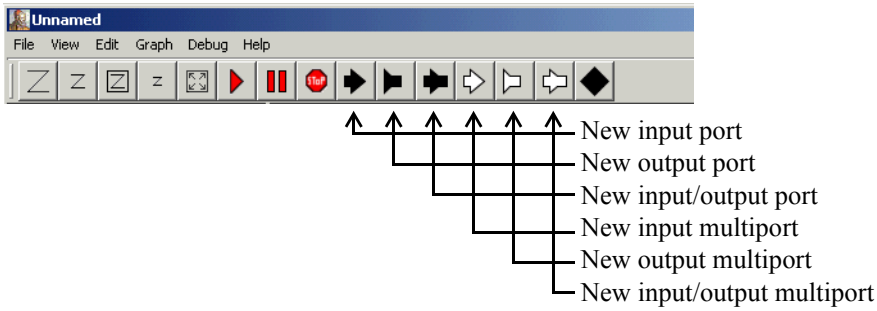


FIGURE 31. Summary of toolbar buttons for creating new ports.

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

similar to the model in figure 7, but with a channel. Notice that both our new *Channel* actor and the *ContinuousSinewave* actor have a red outline. The *ContinuousSinewave* actor is also a composite actor (try looking inside). If you execute this model, you should see something like the plot shown in figure 34.

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*. To set the type to a double matrix, say:

```
[double]
```

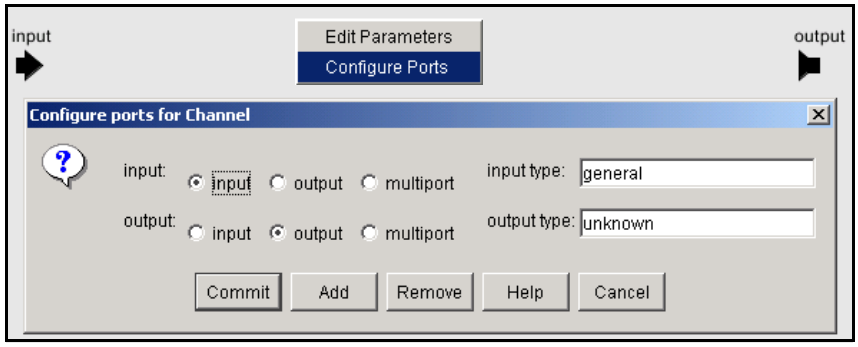


FIGURE 32. Right clicking on the background brings up a dialog that can be used to configure ports.

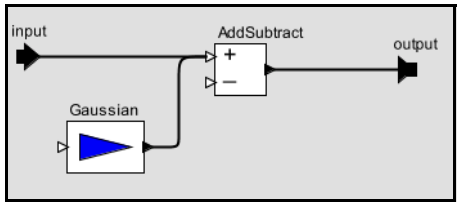


FIGURE 33. A simple channel model defined as a composite actor.

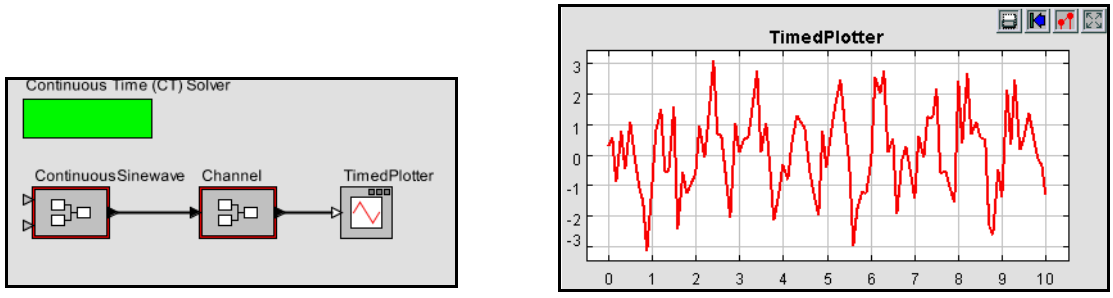


FIGURE 34. A model producing a noisy sine wave.

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

```
{complex}
```

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. To specify a record containing a string named "name" and an integer named "address," say:

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

2.5 Discrete Signals and Mixed-Signal Models

Continuous-time signals can be combined with discrete-event signals in the same model. A discrete signal in HyVisual is one that consists of *events* that are placed on the time line and have a value. If a discrete signal is examined at a time where there is no event, then the signal will have no value. For some actors, this is an error. Continuous actors cannot read discrete signals. Most actors can handle either signal, however, so usually it does not require much effort to mix the two types of signals. For example, most of the math actors are equally content working on continuous signals as discrete signals.

Discrete signals are generated by the actors in the *to discrete* library. These include an actor to simply periodically generate discrete events, level-crossing detectors, samplers (like that shown in figure 11), and a threshold monitor. Discrete signals are converted to continuous signals by the actors in the *to continuous* library. These include a zero-order hold and a first-order hold.

Usually, HyVisual will infer automatically whether a signal is discrete or continuous, but occasionally, you will need to help it. Whether a signal is *DISCRETE* or *CONTINUOUS* is referred to as its *signal type* (do not confuse this with the data type of the signal, which indicates, for example, whether it's a *double* or an *int*).

Some actors declare the signal types of their ports. For example, an *Integrator* has a CONTINUOUS input and a CONTINUOUS output; a *PeriodicSampler* has a CONTINUOUS input and a DISCRETE output; a *TriggeredSampler* has one CONTINUOUS input (the *input*), one DISCRETE input (the *trigger*), and a DISCRETE output; and a *ZeroOrderHold* has a DISCRETE input and a CONTINUOUS output.

Certain other actors declare that they operate only on sequences of data tokens (they declare this by implementing the *SequenceActor* interface). Their inputs and outputs are treated as DISCRETE. Unless otherwise specified, the types of the input ports and output ports of an actor are the same.

Sometimes, conflicts arise, and you will need to force a port to be either discrete or continuous. To do this, add a parameter named "signalType" to the port. The signal type system will recognize this parameter (by name) and resolve other types accordingly. To add this parameter, right click on the port, select Configure, then click on Add. Give the parameter as a value either the string "CONTINUOUS" or "DISCRETE", including the quotation marks.

Signal types can become an issue particularly at the boundaries of state or transition refinements, which occur in Hybrid systems, as explained below.

2.6 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 36. Here, we have zoomed in on the Lorenz attractor model of figure 2 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.

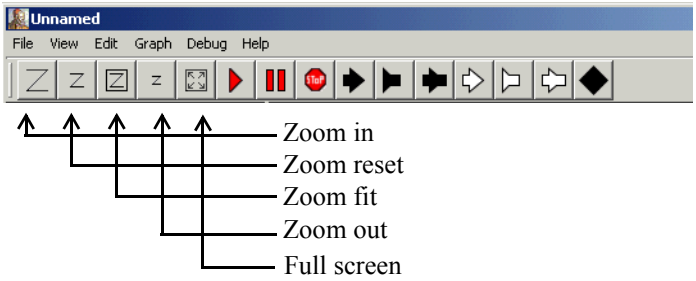


FIGURE 35. Summary of toolbar buttons for zooming and fitting.

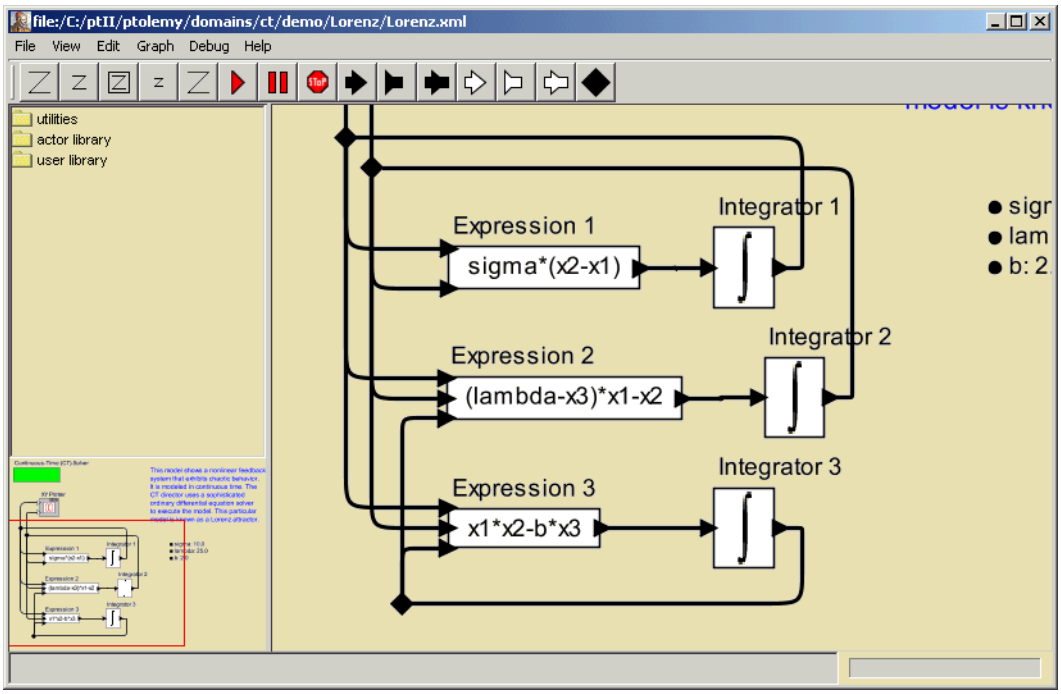


FIGURE 36. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

3. Hybrid Systems

Hybrid systems are models that combine continuous dynamics with discrete mode changes. They are created in HyVisual by creating a *modal model*, found in the *utilities* library. We start by examining a pre-built modal model, and conclude by illustrating how to construct one.

3.1 Examining a Pre-Built Model

The third example in figure 1 is a simple hybrid system model of a bouncing ball. The top-level contents of this model is shown in figure 37. It contains only two actors, a *Ball Model* and a *TimedPlotter*. The *Ball Model* is an instance of the *modal model* found in the *utilities* library, but renamed. If you execute the model, you should see a plot like that in the figure. The continuous dynamics correspond to the times when the ball is in the air, and the discrete events correspond to the times when the ball hits the surface and bounces.

If you look inside the *Ball Model*, you will see something like figure 38. Figure 38 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the left. The circles in figure 38 are states, and the arcs between circles are *transitions* between states. 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 state machine in figure 38 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 39. The *free* state represents the mode of operation where the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

At any time during the execution of the model, the modal model is in one of these three states. When the model begins executing, it is in the *init* state. During the time a modal model is in a state, the behavior of the modal model is specified by the *refinement* of the state. The refinement can be examined by looking inside the state. As shown in figure 40, the *init* state has no refinement.

Consider the transition from *init* to *free*. It is labeled as follows:

```
true
```

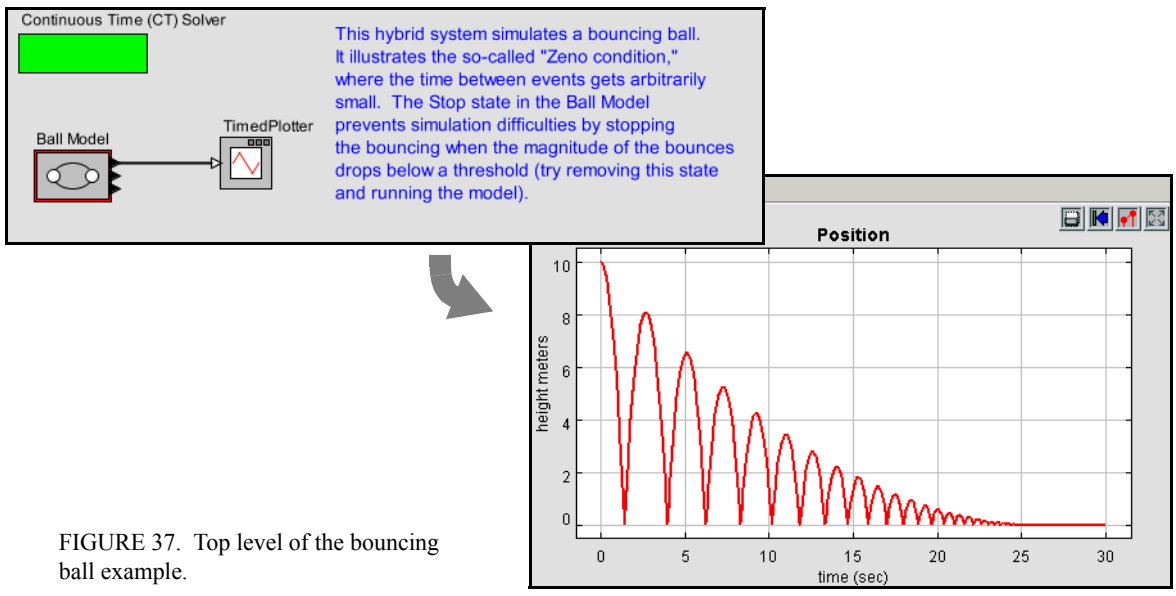


FIGURE 37. Top level of the bouncing ball example.

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

The first line is a *guard*, which is predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value *true*. Thus, the first thing this model will do is take this transition and change modes to *free.* The second line specifies a sequence of *actions*, which in this case set parameters of the destination mode *free*.

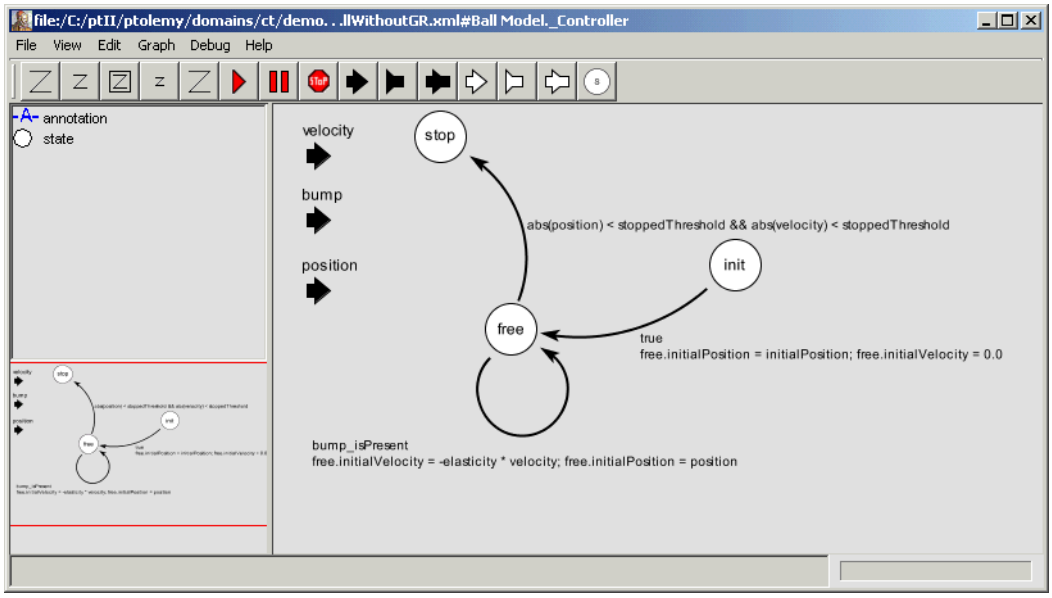


FIGURE 38. Inside the *Ball Model* of figure 37.

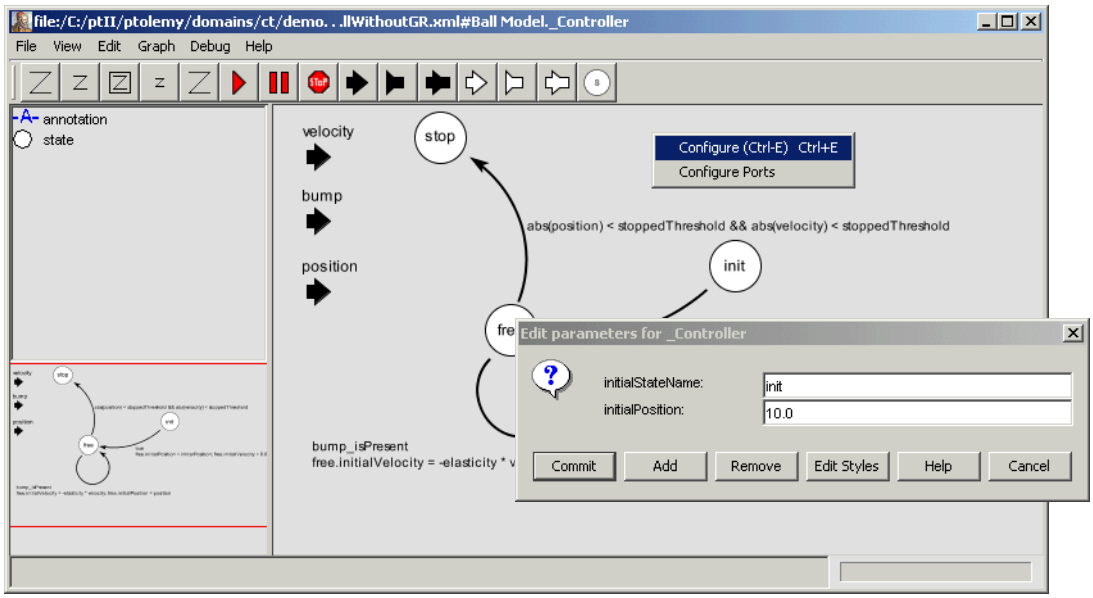


FIGURE 39. The initial state of a state machine is set by right clicking on the background and specifying the state name.

If you look inside the *free* state, you will see the refinement shown in figure 41. This model represents the laws of gravity, which state that an object of any mass will have an acceleration of roughly -10 meters/second^2 (roughly). The acceleration is integrated to get the velocity. which is, in turn, integrated to get the vertical position.

In figure 41, a *ZeroCrossingDetector* actor is used to detect when the vertical position of the ball is zero. This results in production of an event on the (discrete) output *bump*. Examining figure 38, 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 shown by the plot in figure 37.

As you can see from figure 38, 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.

3.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 37 and 38 illustrates an interesting property of hybrid system modeling. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. If you remove the *stop* state from the FSM, and rerun the model, you get the result shown in figure 42. The ball, in effect, falls through the surface on which it is bouncing and then goes into a free-fall in the space below.

The error that occurs here illustrates some fundamental pitfalls with hybrid system modeling. The event detected by the *ZeroCrossingDetector* actor can be missed by the simulator. This actor works with the solver to attempt to identify the precise point in time when the event occurs. It ensures that the simulation includes a sample time at that time. However, when the numbers get small enough, numerical errors take over, and the event is missed.

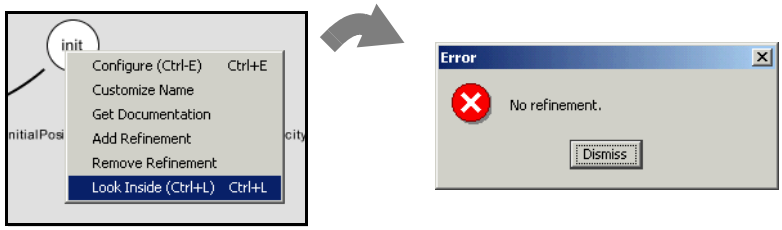


FIGURE 40. A state may or may not have a refinement, which specified the behavior of the model while the model i in that state. In this case, *init* has no refinement.

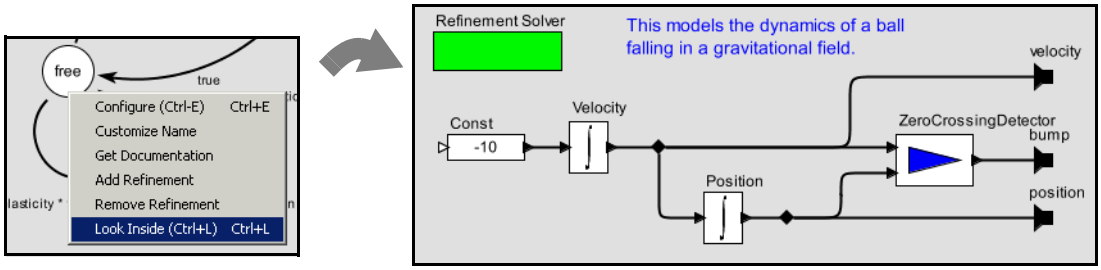


FIGURE 41. The refinement of the *free* state, shown here, is a continuous-model representing the laws of gravity.

A related phenomenon is called the Zeno phenomenon. In the case of the bouncing ball, the time between bounces gets smaller as the simulation progresses. Since the simulator is attempting to capture every bounce event with a time step, we could encounter the problem where the number of time steps becomes infinite over a finite time interval. This makes it impossible for time to advance. In fact, in theory, the bouncing ball example exhibits this Zeno phenomenon. However, numerical precision errors take over, since the simulator cannot possibly keep decreasing the magnitude of the time increments.

The lesson is that some caution needs to be exercised when relying on the results of a simulation of a hybrid system. Use your judgement.

3.3 Constructing Modal Models

A modal model is a component in a larger continuous-time 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 need to add ports. The mechanism for doing that is identical to adding ports to a composite model, and is explained in section 2.4. Figure 37 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. Three output ports have been added to that modal model, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, create an output port and name it *output*, and then look inside, you will get an FSM editor like that shown in figure 43. Note that the output port is (regrettably) located at the upper left, and is only partially visible. The annotation text suggests delete once you no longer need it. You may want to move the port to a more reasonable location (where it is visible).

The output port that you created is in fact indicated in the state machine as being both an output and input port. The reason for this is that guards in the state machine can refer to output values that are produced on this port by refinements. In addition, the output actions of a transition can assign an output value to this port. Hence, the port is, in fact, both an output and input for the state machine.

To create a finite-state machine like that in figure 38, drag in states (white circles), or click on the state icon in the toolbar. You can rename these states by right clicking on them and selecting “Customize Name”. Choose names that are pertinent to your application. In figure 38, 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

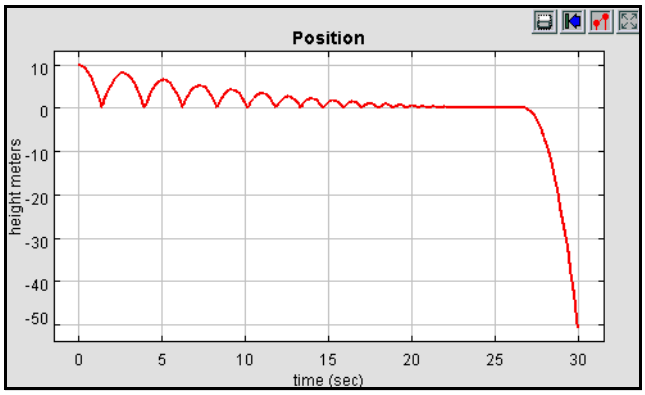


FIGURE 42. Result of running the bouncing ball model without the *stop* state.

FSM Editor, selecting “Edit Parameters”, and specifying an initial state name, as shown in figure 39. In that figure, the initial state is named *init*.

3.3.1 Creating Transitions

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 right clicking and selecting “Configure”) allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in Figure 44. 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.

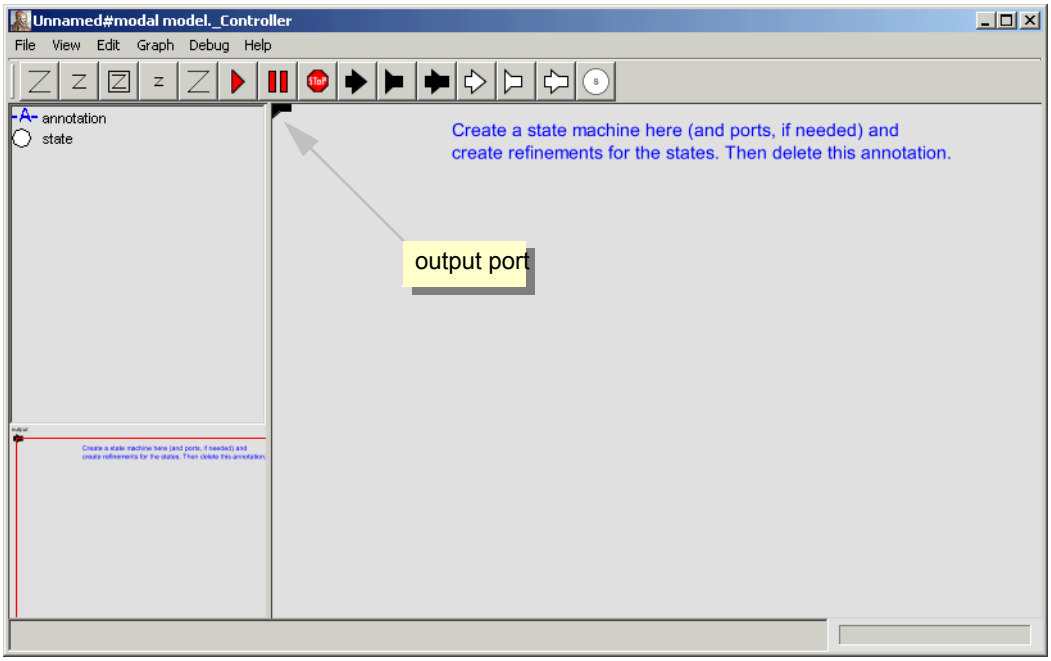


FIGURE 43. Inside of a new modal model that has had a single output port added.

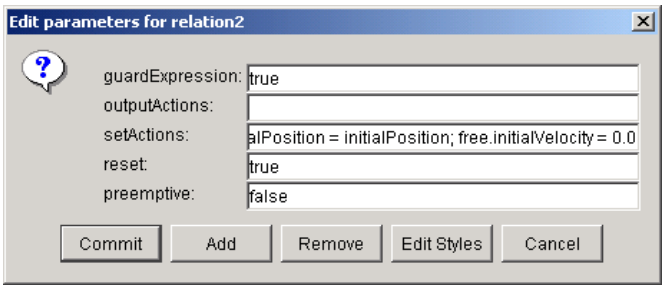


FIGURE 44. Transition dialog for the transition from *init* to *free* in Figure 37.

- 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 field contains 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 refinement will 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 executing the refinement. Thus, the refinement will not affect the outputs of the modal model.

A state may have several outgoing transitions. However, it is up to the model builder to ensure that at no time does more than one guard on these transitions evaluate to true. In other words, HyVisual does not allow nondeterministic state machines, and will throw an exception if it encounters one.

3.3.2 Creating 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 45. As shown in the figure, you will be offered the alternatives of a "Continuous Time Refinement" or a "State Machine Refinement." The first of these provides a continuous-time model as the refinement. The second provides another finite state machine as the refinement. In the former case (the default), an almost blank refinement model will open, as shown in the figure. As before, the output port will appear in an inconvenient location. You will almost certainly want to move it to a more convenient location.

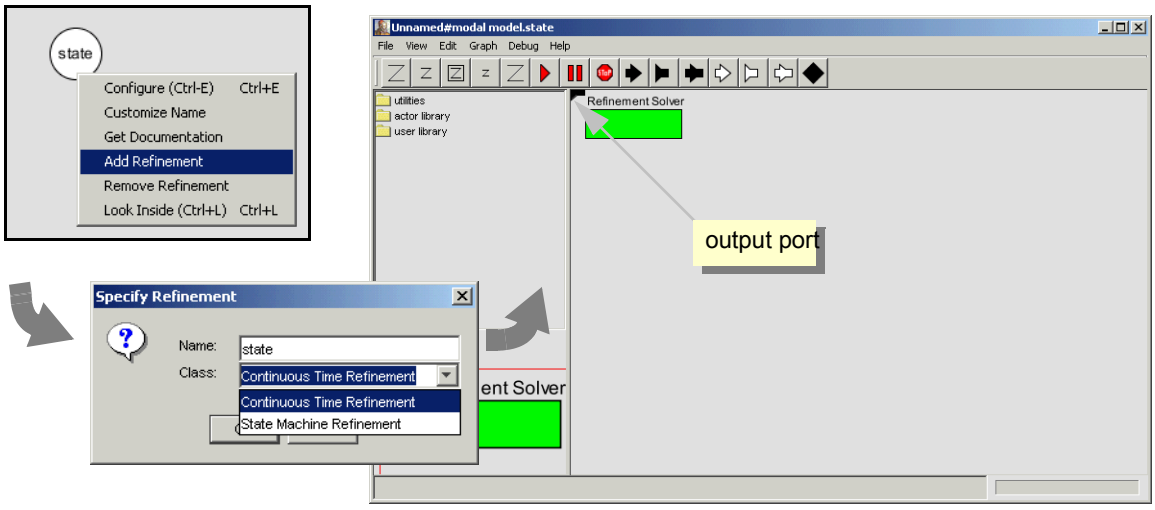


FIGURE 45. Adding a refinement to a state.

You can also create refinements for transitions, but these have somewhat different behavior. They will execute exactly once when the transition is taken. For this reason, they offer an entirely different "solver." In fact, transition refinements will execute according to dataflow semantics. They are typically used to perform arithmetic computations that are too elaborate to be conveniently specified as an action on the transition.

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 41. This model exhibits certain key properties of refinements:

- Refinements must contain solvers, unlike composite actors. In this case, the solver is named "Refinement Solver" and is an instance of the CTEmbeddedDirector class.
- 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.

3.4 Execution Semantics

The behavior of a refinement is simple. When the modal model is executed, the following sequence of events occurs:

- For any transitions out of the current state for which *preemptive* is *true*, the guard is evaluated. If exactly one such guard evaluates to *true*, then that transition is chosen. The *output actions* of the transition are executed, and the *refinements* of the transition (if any) are executed, followed by the *set actions*.
- If no preemptive transition evaluated to true, then the refinement of the current state, if there is one, is evaluated at the current time step.
- Once the refinement has been evaluated (and it has possibly updated its output values), the guard expressions on all the outgoing transitions of the current state are evaluated. If none is true, the execution is complete. If one is true, then that transition is taken. If more than one is true, then an exception is thrown (the state machine is nondeterministic). What it means for the transition to be "taken" is that its *output actions* are executed, its *refinements* (if any) are executed, and its *set actions* are executed.
- If *reset* is true on a transition that is taken, then the refinement of the destination mode (if there is one) is initialized.

There is a subtle distinction between the *output actions* and the *set actions*. The intent of these two fields on the transition is that *output actions* are used to define the values of output ports, while *set actions* are used to define state variables in the refinements of the destination modes. The reason that these two actions are separated is that while solving a continuous-time system of equations, the solver may speculatively execute models at certain time steps before it is sure what the next time step will be. The *output actions* make no permanent changes to the state of the system, and hence can be executed during this speculative phase. The *set actions*, however, make permanent changes to the state variables of the destination refinements, and hence are not executed during the speculative phase.

4. 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 16 has the default (uninformative) title, the axes are not labeled, and there is no legend.

The *TimedPlotter* actor has some pertinent parameters, shown in figure 46. The *fillOnWrapup* parameter specifies whether the plot should fill the available screen area when the model completes execution. The default value is *true*, and for finite executions, this is often the most useful value. The *legend* parameter can be used to identify signals. It is a comma-separated listed of names that will be attached to signals. With the values shown in figure 46, the resulting plot looks like figure 47. The *startingDataset* parameter is used when two actors share the same plot, and is beyond the scope of this discussion.

The plot in figure 47 is better, but it is 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 48, filled in with values that result in the plot shown in figure 49. 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.”

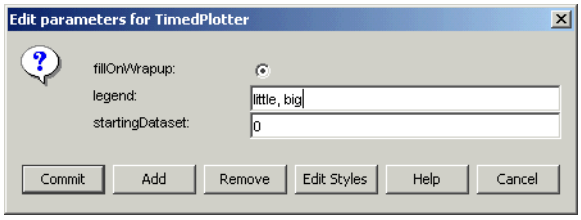


FIGURE 46. Parameters of the *TimedPlotter* actor.

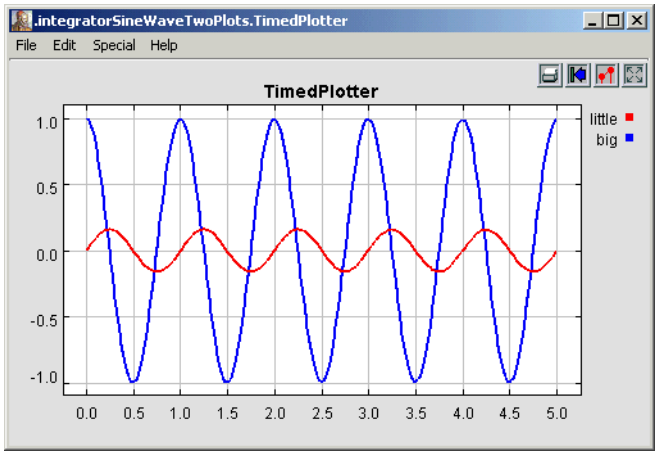


FIGURE 47. A plot with a legend.

- Individual sample points can be shown by clicking on “dots.”
- Connecting lines can be eliminated by deselecting “connect.”
- The Y axis label has been changed to text rather than numbers. This is done by entering the following in the Y Ticks field:

```
"minus one" -1.0, zero 0.0, one 1.0
```

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.

5. Expressions

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

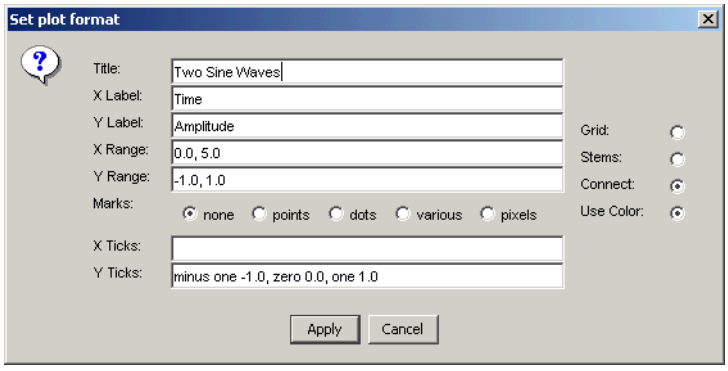


FIGURE 48. Format control window for a plot.

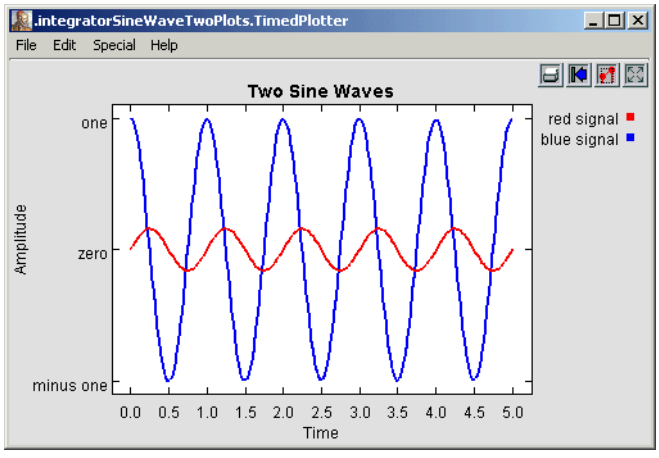


FIGURE 49. Still better labeled plot.

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.

5.1 Expression Evaluator

Vergil provides an interactive *expression evaluator*, which is accessed through the File:New menu. This operates like an interactive command shell, and is shown in figure 5.1. It supports a command history. To access the previously entered expression, type the up arrow or Control-P. To go back, type the down arrow or Control-N. The expression evaluator is useful for experimenting with expressions.

5.2 Simple Arithmetic Expressions

5.2.1 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, j, NaN, Infinity, PositiveInfinity, NegativeInfinity, MaxUnsignedByte, MinUnsignedByte, MaxInt, MinInt, MaxLong, MinLong, MaxDouble, MinDouble. 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 the imaginary number with value equal to the square root of –1. The constant NaN is “not a number,” which for example is the result of dividing 0.0/0.0. The constant Infinity is the result of dividing 1.0/0.0. The constants that start with “Max” and “Min” are the maximum and minimum values for their corresponding types.

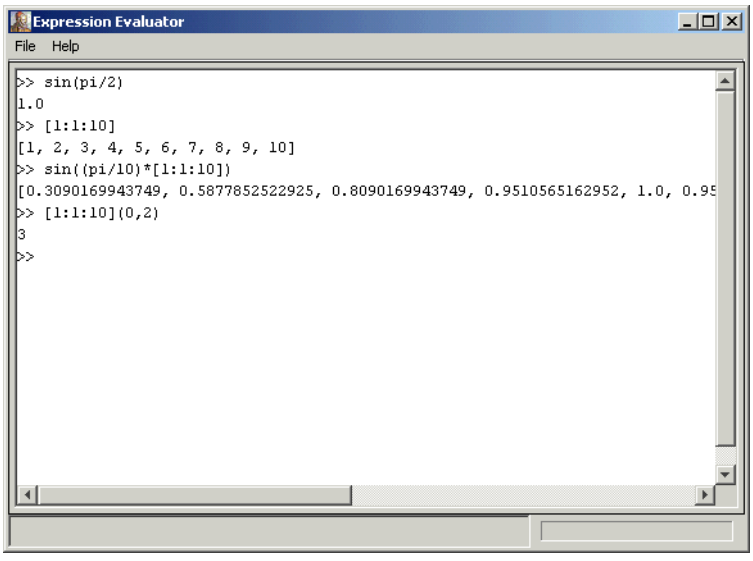


FIGURE 5.1. Expression evaluator, which is accessed through the File:New menu.

Numerical values without decimal points, such as “10” or “–3” are integers (type *int*). Numerical values with decimal points, such as “10.0” or “3.14159” are of type *double*. Numerical values without decimal points followed by the character “l” (el) or “L” are of type *long*. Unsigned integers followed by “ub” or “UB” are of type *unsignedByte*, as in “5ub”. An *unsignedByte* has a value between 0 and 255; note that it not quite the same as the Java byte, which has a value between -128 and 127.

Numbers of type *int*, *long*, or *unsignedByte* can be specified in decimal, octal, or hexadecimal. Numbers beginning with a leading “0” are octal numbers. Numbers beginning with a leading “0x” are hexadecimal numbers. For example, “012” and “0xA” are both equal to the integer 10.

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`”.

Literal string constants are also supported. Anything between double quotes, “...”, is interpreted as a string constant. The following built-in string-valued constants are defined:

TABLE 1: String-valued constants defined in the expression language.

Variable name	Meaning	Property name	Example under Windows
PTII	The directory in which HyVisual is installed	ptolemy.ptII.dir	c:\tmp
HOME	The user home directory	user.home	c:\Documents and Settings\you
CWD	The current working directory	user.dir	c:\ptII

The value of these variables is the value of the Java virtual machine property, such as *user.home*. The properties *user.dir* and *user.home* are standard in Java. Their values are platform dependent; see the documentation for the java.lang.System.getProperties() method for details. Note that *user.dir* and *user.home* are usually not readable in unsigned applets, in which case, attempts to use these variables in an expression will result in an exception. Vergil will display all the Java properties if you invoke JVM Properties in the View menu of a Graph Editor.

The *ptolemy.ptII.dir* property is set automatically when HyVisual is started up. The constants() utility function returns a record with all the globally defined constants. If you open the expression evaluator and invoke this function, you will see that its value is something like:

```
{CWD="C:\ptII\ptolemy\data\expr", E=2.718281828459, HOME="C:\Documents
and Settings\eal", Infinity=Infinity, MaxDouble=1.7976931348623E308,
MaxInt=2147483647, MaxLong=9223372036854775807L,
MaxUnsignedByte=255ub, MinDouble=4.9E-324, MinInt=-2147483648,
MinLong=-9223372036854775808L, MinUnsignedByte=0ub, NaN=NaN,
NegativeInfinity=-Infinity, PI=3.1415926535898, PTII="c:\ptII",
PositiveInfinity=Infinity, boolean=false, complex=0.0 + 0.0i,
double=0.0, e=2.718281828459, false=false, fixedpoint=fix(0.0,2,1),
```

```
general=present, i=0.0 + 1.0i, int=0, j=0.0 + 1.0i, long=0L, matrix=[],
object=object(null), pi=3.1415926535898, scalar=present, string="",
true=true, unknown=present, unsignedByte=0ub}
```

5.2.2 Variables

Expressions can contain identifiers that are 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 expression evaluator, the variables that are in scope include the built-in constants plus any assignments that have been previously made. For example,

```
>> x = pi/2
1.5707963267949
>> sin(x)
1.0
>>
```

In the context of HyVisual 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 functional programming languages.

5.2.3 Operators

The arithmetic operators are +, –, *, /, ^, and %. Most of these operators operate on most data types, including arrays, records, and matrices. The ^ operator computes "to the power of" or exponentiation where the exponent can only be an *int* or an *unsignedByte*.

The *unsignedByte*, *int* and *long* types can only represent integer numbers. Operations on these types are integer operations, which can sometimes lead to unexpected results. For instance, 1/2 yields 0 if 1 and 2 are integers, whereas 1.0/2.0 yields 0.5. The exponentiation operator '^' when used with negative exponents can similarly yield unexpected results. For example, 2^–1 is 0 because the result is computed as 1/(2^1).

The % operation is a *modulo* or *remainder* operation. The result is the remainder after division. The sign of the result is the same as that of the dividend (the left argument). For example,

```
>> 3.0 % 2.0
1.0
>> -3.0 % 2.0
-1.0
>> -3.0 % -2.0
-1.0
>> 3.0 % -2.0
1.0
```

The magnitude of the result is always less than the magnitude of the divisor (the right argument). Note that when this operator is used on doubles, the result is not the same as that produced by the remainder() function (see Table 4 on page 74). For instance,

```
>> remainder(-3.0, 2.0)
1.0
```

The remainder() function calculates the IEEE 754 standard remainder operation. It uses a rounding division rather than a truncating division, and hence the sign can be positive or negative, depending on complicated rules (see page 56). For example, counterintuitively,

```
>> remainder(3.0, 2.0)
-1.0
```

When an operator involves two distinct types, the expression language has to make a decision about which type to use to implement the operation. If one of the two types can be converted without loss into the other, then it will be. For instance, *int* can be converted losslessly to *double*, so 1.0/2 will result in 2 being first converted to 2.0, so the result will be 0.5. Among the scalar types, *unsignedByte* can be converted to anything else, *int* can be converted to *double*, and *double* can be converted to *complex*. Note that *long* cannot be converted to *double* without loss, nor vice versa, so an expression like `2.0/2L` yields the following error message:

```
Error evaluating expression "2.0/2L"
  in .Expression.evaluator
Because:
divide method not supported between ptolemy.data.DoubleToken '2.0' and
ptolemy.data.LongToken '2L' because the types are incomparable.
```

All scalar types have limited precision and magnitude. As a result of this, arithmetic operations are subject to underflow and overflow.

- For *double* numbers, overflow results in the corresponding positive or negative infinity. Underflow (i.e. the precision does not suffice to represent the result) will yield zero.
- For integer types and *fixedpoint*, overflow results in wraparound. For instance, while the value of `MaxInt` is 2147483647, the expression `MaxInt + 1` yields –2147483648. Similarly, while `MaxUnsignedByte` has value 255ub, `MaxUnsignedByte + 1ub` has value 0ub. Note, however, that

`MaxUnsignedByte + 1` yields 256, which is an *int*, not an *unsignedByte*. This is because `MaxUnsignedByte` can be losslessly converted to an *int*, so the addition is *int* addition, not *unsignedByte* addition.

The bitwise operators are `&`, `|`, `#`, and `~`. They operate on *boolean*, *unsignedByte*, *int* and *long* (but not *fixedpoint*, *double* or *complex*). The operator `&` is bitwise AND, `~` is bitwise NOT, and `|` is bitwise OR, and `#` is bitwise XOR (exclusive or, after MATLAB).

The relational operators are <, <=, >, >=, == and `!=`. They return type *boolean*. Note that these relational operators check the values when possible, irrespective of type. So, for example,

```
1 == 1.0
```

returns *true*. If you wish to check for equality of both type and value, use the equals() method, as in

```
>> 1.equals(1.0)
false
```

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 type *boolean* and return type *boolean*. 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. Thus, for example, the expression "`false && x`" will evaluate to *false* irrespective of whether `x` is defined. On the other hand, "`false & x`" will throw an exception.

The << and >> operators performs arithmetic left and right shifts respectively. The >>> operator performs a logical right shift, which does not preserve the sign. They operate on *unsignedByte*, *int*, and *long*.

5.2.4 Comments

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

5.3 Uses of Expressions

5.3.1 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. They can also reference variables in a *scope-extending attribute*, which includes variables defining units. Adding parameters to actors is straightforward, as explained in the previous chapter.

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.

5.3.2 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 data, however, the default value is overridden with the value provided at the port. Thus, this object functions 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 5.2. 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 5.2. 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.

An example of a library actor that uses a PortParameter is the Sinewave actor, which is found in the *sources* library in Vergil. It is shown in figure 5.3. If you double click on this actor, you can set the default values for *frequency* and *phase*. But both of these values can also be set by the corresponding ports, which are shown with grey fill.

5.3.3 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output and no inputs, as shown in Figure 5.4(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

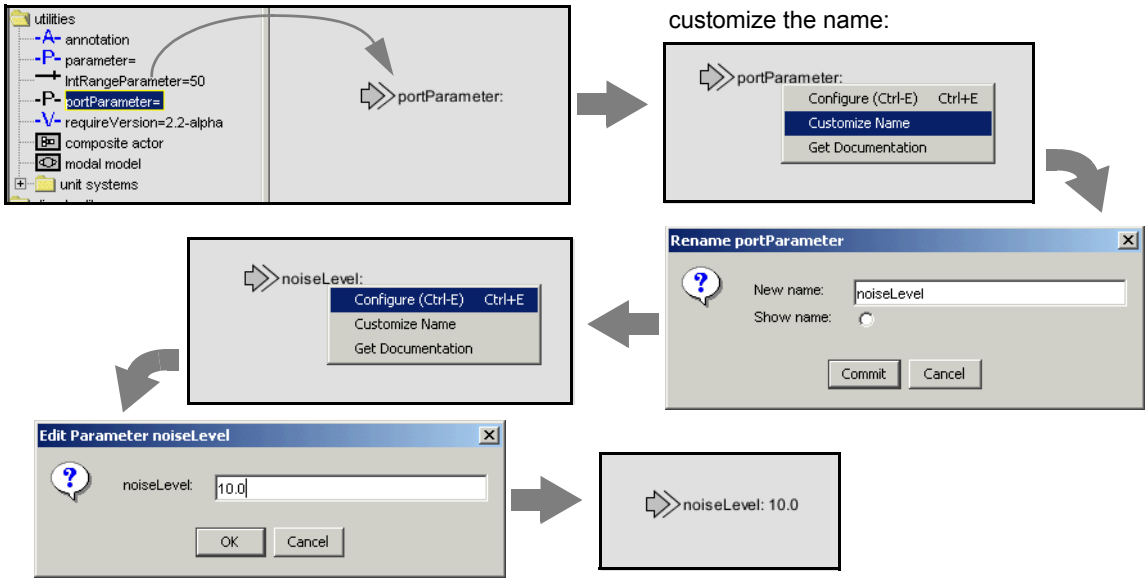


FIGURE 5.2. A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful, and set its default value.

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).

5.3.4 State Machines

Expressions give the guards for state transitions, as well as the values used in actions that produce outputs and actions that set values of parameters in the refinements of destination states. This mechanism was explained in the previous chapter.

5.4 Composite Data Types

5.4.1 Arrays

Arrays are specified with curly brackets, e.g., "`{1, 2, 3}`" is an array of *int*, while "`{"x", "y", "z"}`" is an array of *string*. The types are denoted "`{int}`" and "`{string}`" respectively. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have

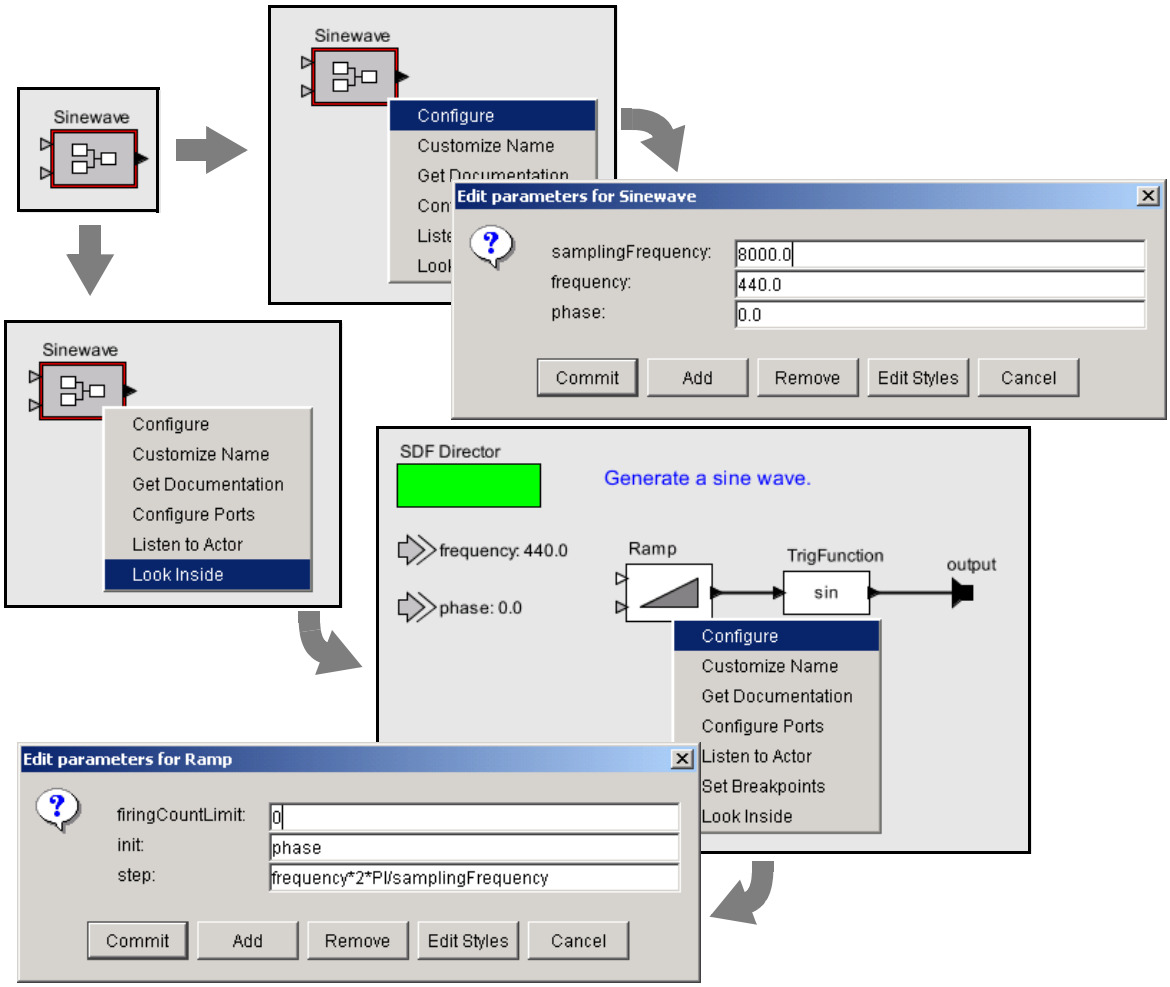


FIGURE 5.3. Sinewave actor, showing its port parameters, and their use at the lower level of the hierarchy.

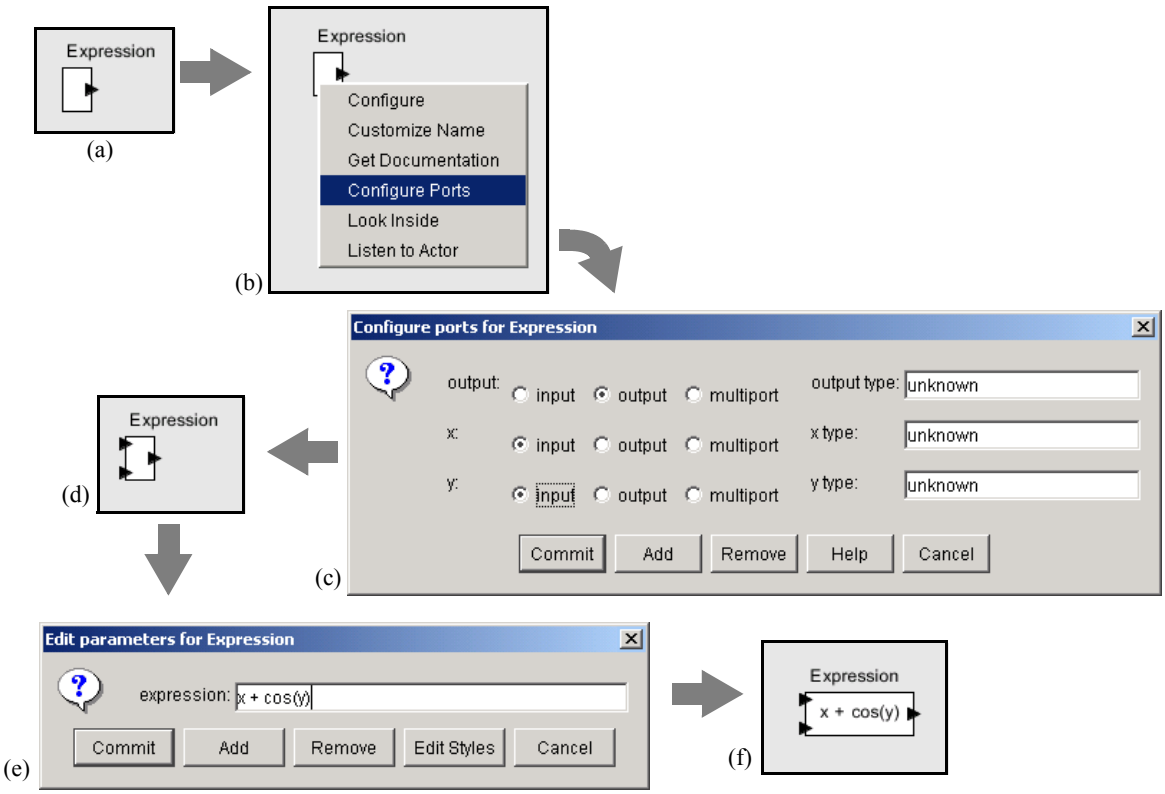


FIGURE 5.4. Illustration of the *Expression* actor.

the same type. If an array is given with mixed types, the expression evaluator will attempt to losslessly convert the elements to a common type. Thus, for example,

```
{1, 2.3}
```

has value

```
{1.0, 2.3}
```

Its type is `{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. The elements of an array can be accessed as follows:

```
>> {1.0, 2.3}(1)
2.3
```

which yields 2.3. Note that indexing begins at 0. Of course, if *name* is the name of a variable in scope whose value is an array, then its elements may be accessed similarly, as shown in this example:

```
>> x = {1.0, 2.3}
{1.0, 2.3}
>> x(0)
1.0
```

Arithmetic operations on arrays are carried out element-by-element, as shown by the following examples:

```
>> {1, 2}*{2, 2}
{2, 4}
>> {1, 2}+{2, 2}
{3, 4}
>> {1, 2}-{2, 2}
{-1, 0}
>> {1, 2}^2
{1, 4}
>> {1, 2}%{2, 2}
{1, 0}
```

An array can be checked for equality with another array as follows:

```
>> {1, 2}=={2, 2}
false
>> {1, 2}!={2, 2}
true
```

For other comparisons of arrays, use the compare() function (see Table 4 on page 74). As with scalars, testing for equality using the == or != operators tests the values, independent of type. For example,

```
>> {1, 2}=={1.0, 2.0}
true
```

5.4.2 Matrices

In HyVisual, *arrays* are ordered sets of tokens. HyVisual also supports *matrices*, which are more specialized than arrays. They contain only certain primitive types, currently *boolean*, *complex*, *double*, *fixedpoint*, *int*, and *long*. Currently *unsignedByte* matrices are not supported. 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 columns). Note that an array or matrix element can be given by an expression. 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*. If a matrix with mixed types is specified, then the elements will be converted to a common type, if possible. Thus, for example, "[1.0, 1]" is equivalent to "[1.0, 1.0]," but "[1.0, 1L]" is illegal (because there is no common type to which both elements can be converted losslessly).

Reference to elements of matrices have the form "*matrix*(*n*, *m*)" or "*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. For example,

```
>> [1, 2; 3, 4](0,0)
1
>> a = [1, 2; 3, 4]
[1, 2; 3, 4]
>> a(1,1)
4
```

Matrix multiplication works as expected. For example, as seen in the expression evaluator (see figure 5.1),

```
>> [1, 2; 3, 4]*[2, 2; 2, 2]
[6, 6; 14, 14]
```

Of course, if the dimensions of the matrix don't match, then you will get an error message. To do elementwise multiplication, use the multipyElements() function (see Table 5 on page 76). Matrix addition and subtraction are elementwise, as expected, but the division operator is not supported. Elementwise division can be accomplished with the divideElements() function, and multiplication by a matrix inverse can be accomplished using the inverse() function (see Table 5 on page 76). A matrix can be

raised to an *int* or *unsignedByte* power, which is equivalent to multiplying it by itself some number of times. For instance,

```
>> [3, 0; 0, 3]^3
[27, 0; 0, 27]
```

A matrix can also be multiplied or divided by a scalar, as follows:

```
>> [3, 0; 0, 3]*3
[9, 0; 0, 9]
```

A matrix can be added to a scalar. It can also be subtracted from a scalar, or have a scalar subtracted from it. For instance,

```
>> 1-[3, 0; 0, 3]
[-2, 1; 1, -2]
```

A matrix can be checked for equality with another matrix as follows:

```
>> [3, 0; 0, 3]!=[3, 0; 0, 6]
true
>> [3, 0; 0, 3]==[3, 0; 0, 3]
true
```

For other comparisons of matrices, use the compare() function (see Table 4 on page 74). As with scalars, testing for equality using the == or `!=` operators tests the values, independent of type. For example,

```
>> [1, 2]==[1.0, 2.0]
true
```

To get type-specific equality tests, use the equals() method, as in the following examples:

```
>> [1, 2].equals([1.0, 2.0])
false
>> [1.0, 2.0].equals([1.0, 2.0])
true
>>
```

5.4.3 Records

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

Fields 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()
```

The arithmetic operators +, –, *, /, and % can be applied to records. If the records do not have identical fields, then the operator is applied only to the fields that match, and the result contains only the fields that match. Thus, for example,

```
{foodCost=40, hotelCost=100} + {foodCost=20, taxiCost=20}
```

yields the result

```
{foodCost=60}
```

You can think of an operation as a set intersection, where the operation specifies how to merge the values of the intersecting fields. You can also form an intersection without applying an operation. In this case, using the intersect() function, you form a record that has only the common fields of two specified records, with the values taken from the first record. For example,

```
>> intersect({a=1, c=2}, {a=3, b=4})
{a=1}
```

Records can be joined (think of a set union) without any operation being applied by using the merge() function. This function takes two arguments, both of which are record tokens. If the two record tokens have common fields, then the field value from the first record is used. For example,

```
merge({a=1, b=2}, {a=3, c=3})
```

yields the result `{a=1, b=2, c=3}`.

Records can be compared, as in the following examples:

```
>> {a=1, b=2}!={a=1, b=2}
false
>> {a=1, b=2}!={a=1, c=2}
true
```

Note that two records are equal only if they have the same field labels and the values match. As with scalars, the values match irrespective of type. For example:

```
>> {a=1, b=2}=={a=1.0, b=2.0+0.0i}
true
```

The order of the fields is irrelevant. Hence

```
>> {a=1, b=2}=={b=2, a=1}
true
```

Moreover, record fields are reported in alphabetical order, irrespective of the order in which they are defined. For example,

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

To get type-specific equality tests, use the equals() method, as in the following examples:

```
>> {a=1, b=2}.equals({a=1.0, b=2.0+0.0i})
false
>> {a=1, b=2}.equals({b=2, a=1})
true
>>
```

5.5 Invoking Methods

Every element and subexpression in an expression represents an instance of the Token class in HyVisual (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 and RecordToken classes have a length() method, illustrated by the following examples:

```
{1, 2, 3}.length()
{a=1, b=2, c=3}.length()
```

each of 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()
```

5.6 Defining Functions

The expression language supports definition of functions. The syntax is:

```
function(arg1:Type, arg2:Type...)
    function body
```

where “function” is the keyword for defining a function. The type of an argument can be left unspecified, in which case the expression language will attempt to infer it. The function body gives an expres-

sion that defines the return value of the function. The return type is always inferred based on the argument type and the expression. For example:

```
function(x:double) x*5.0
```

defines a function that takes a *double* argument, multiplies it by 5.0, and returns a double. The return value of the above expression is the function itself. Thus, for example, the expression evaluator yields:

```
>> function(x:double) x*5.0
(function(x:double) (x*5.0))
>>
```

To apply the function to an argument, simply do

```
>> (function(x:double) x*5.0) (10.0)
50.0
>>
```

Alternatively, in the expression evaluator, you can assign the function to a variable, and then use the variable name to apply the function. For example,

```
>> f = function(x:double) x*5.0
(function(x:double) (x*5.0))
>> f(10)
50.0
>>
```

Functions can be passed as arguments to certain “higher-order functions” that have been defined (see table Table 8 on page 80). For example, the iterate() function takes three arguments, a function, an integer, and an initial value to which to apply the function. It applies the function first to the initial value, then to the result of the application, then to that result, collecting the results into an array whose length is given by the second argument. For example, to get an array whose values are multiples of 3, try

```
>> iterate(function(x:int) x+3, 5, 0)
{0, 3, 6, 9, 12}
```

The function given as an argument simply adds three to its argument. The result is the specified initial value (0) followed by the result of applying the function once to that initial value, then twice, then three times, etc.

Another useful higher-order function is the map() function. This one takes a function and an array as arguments, and simply applies the function to each element of the array to construct a result array. For example,

```
>> map(function(x:int) x+3, {0, 2, 3})
{3, 5, 6}
```

A typical use of functions in a HyVisual model is to define a parameter in a model whose value is a function. Suppose that the parameter named "`f`" has value "`function(x:double) x*5.0`". Then within the scope of that parameter, the expression "`f(10.0)`" will yield result 50.0.

Functions can also be passed along connections in a HyVisual model. Consider the model shown in figure 5.5. In that example, the Const actor defines a function that simply squares the argument. Its output, therefore, is a token with type *function*. That token is fed to the "f" input of the Expression actor. The expression uses this function by applying it to the token provided on the "y" input. That token, in turn, is supplied by the Ramp actor, so the result is the curve shown in the plot on the right.

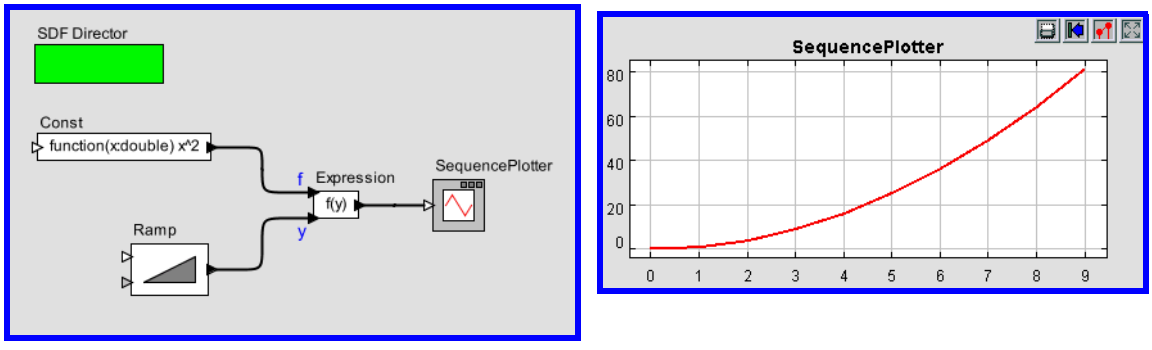


FIGURE 5.5. Example of a function being passed from one actor to another.

A more elaborate use is shown in figure 5.6. In that example, the Const actor produces a function, which is then used by the Expression actor to create new function, which is then used by Expression2 to perform a calculation. The calculation performed here adds the output of the Ramp to the square of the output of the Ramp.

Functions can be recursive, as illustrated by the following (rather arcane) example:

```
>> fact = function(x:int,f:(function(x,f) int)) (x<1?1:x*f(x-1,f))
(function(x:int, f:function(a0:general, a1:general) int)
(x<1)?1:(x*f((x-1), f)))
>> factorial = function(x:int) fact(x,fact)
(function(x:int) (function(x:int, f:function(a0:general, a1:general)
int) (x<1)?1:(x*f((x-1), f)))(x, (function(x:int, f:function(a0:gen-
eral, a1:general) int) (x<1)?1:(x*f((x-1), f)))))
>> map(factorial, [1:1:5].toArray())
{1, 2, 6, 24, 120}
>>
```

The first expression defines a function named “fact” that takes a function as an argument, and if the argument is greater than or equal to 1, uses that function recursively. The second expression defines a new function “factorial” using “fact.” The final command applies the factorial function to an array to compute factorials.

5.7 Built-In Functions

The expression language includes a set of functions, such as sin(), cos(), etc. The functions currently available are shown in the tables in the appendix, which also show the argument types and return types.

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices.

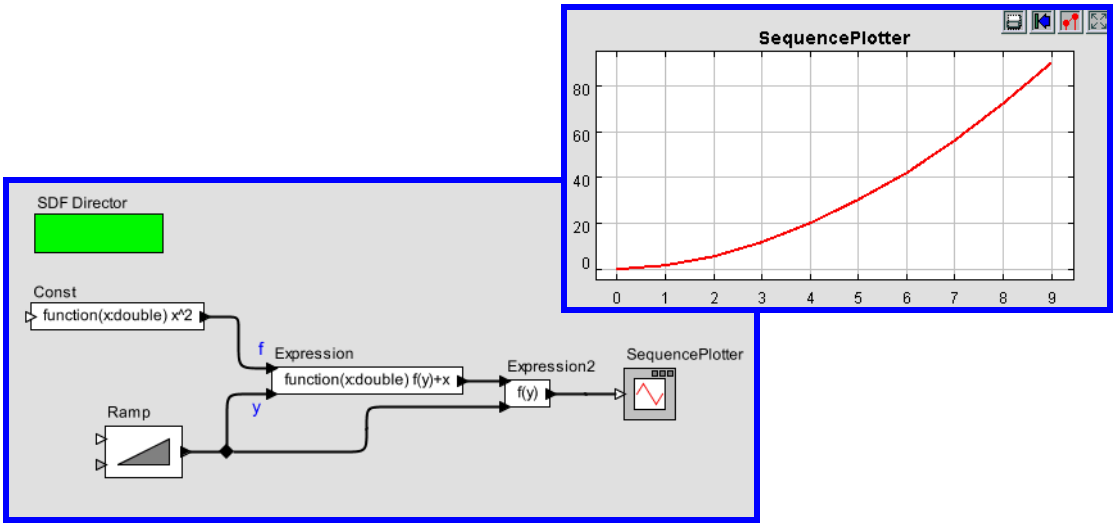


FIGURE 5.6. More elaborate example with functions passed between actors.

Thus, for example, you can fill a row vector with a sine wave using an expression like

```
sin([0.0:PI/100:1.0])
```

Or you can construct an array as follows,

```
sin({0.0, 0.1, 0.2, 0.3})
```

Functions that operate on type *double* will also generally operate on *int* or *unsignedByte*, because these can be losslessly converted to *double*, but not generally on *long* or *complex*.

Tables of available functions are shown in the appendix. For example, Table 3 on page 73 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be double. For example,

```
>> cos(0)
1.0
```

These functions will also operate on matrices and arrays, in addition to the scalar types shown in the table, as illustrated above. The result will be a matrix or array of the same size as the argument, but always containing elements of type *double*

Table 4 on page 74 shows other arithmetic functions beyond the trigonometric functions. As with the trigonometric functions, those that indicate that they operate on *double* will also work on *int* and *unsignedByte*, and unless they indicate otherwise, they will return whatever they return when the argument is *double*. Those functions in the table that take scalar arguments will also operate on matrices and arrays. For example, since the table indicates that the max() function can take *int, int* as arguments, then by implication, it can also take {*int*}, {*int*}. For example,

```
>> max({1, 2}, {2, 1})
{2, 2}
```

Notice that the table also indicates that max() can take {*int*} as an argument. E.g.

```
>> max({1, 2, 3})
3
```

In the former case, the function is applied pointwise to the two arguments. In the latter case, the returned value is the maximum over all the contents of the single argument.

Table 5 shows functions that only work with matrices, arrays, or records (that is, there is no corresponding scalar operation). Recall that most functions that operate on scalars will also operate on arrays and matricesTable 6 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the eval() function is the most flexible (see page 55).

A few of the functions have sufficiently subtle properties that they require further explanation. That explanation is here.

eval() and traceEvaluation()

The built-in function eval() will evaluate a string as an expression in the expression language. For example,

```
eval("[1.0, 2.0; 3.0, 4.0]")
```

will return a matrix of doubles. The following combination can be used to read parameters from a file:

```
eval(readFile("filename"))
```

where the *filename* can be relative to the current working directory (where HyVisual was started, as reported by the property user.dir), the user's home directory (as reported by the property user.home), or the classpath, which includes the directory tree in which HyVisual is installed.

Note that if eval() is used in an Expression actor, then it will be impossible for the type system to infer any more specific output type than *general*. If you need the output type to be more specific, then you will need to cast the result of eval(). For example, to force it to type *double*:

```
>> cast(double, eval("pi/2"))
1.5707963267949
```

The traceEvaluation() function evaluates an expression given as a string, much like eval(), but instead of reporting the result, reports exactly how the expression was evaluated. This can be used to debug expressions, particularly when the expression language is extended by users.

random(), gaussian()

The functions random() and gaussian() shown in Table 4 on page 74 return one or more random numbers. With the minimum number of arguments (zero or two, respectively), they return a single number. With one additional argument, they return an array of the specified length. With a second additional argument, they return a matrix with the specified number of rows and columns.

There is a key subtlety when using these functions in HyVisual. In particular, they are evaluated only when the expression within which they appear is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. 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 output will change, however, on successive runs of the model. In contrast, if this is used in an Expression actor, then each firing triggers an evaluation of the expression, and consequently will result in a new random number.

property()

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

- ptolemy.ptII.dir: The directory in which HyVisual is installed.
- ptolemy.ptII.dirAsURL: The directory in which HyVisual 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.

remainder()

This function computes the remainder operation on two arguments as prescribed by the IEEE 754 standard, which is not the same as the modulo operation computed by the % operator. The result of `remainder(x, y)` is $x - yn$, where n is the integer closest to the exact value of x/y. If two integers are equally close, then n is the integer that is even. This yields results that may be surprising, as indicated by the following examples:

```
>> remainder(1,2)
1.0
>> remainder(3,2)
-1.0
```

Compare this to

```
>> 3%2
1
```

which is different in two ways. The result numerically different and is of type *int*, whereas `remainder()` always yields a result of type *double*. The `remainder()` function is implemented by the java.lang.Math class, which calls it `IEEEremainder()`. The documentation for that class gives the following special cases:

- If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN.
- If the first argument is finite and the second argument is infinite, then the result is the same as the first argument.

DCT() and IDCT()

The DCT function can take one, two, or three arguments. In all three cases, the first argument is an array of length $N > 0$ and the DCT returns an

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos\left((2n+1)k\frac{\pi}{2D}\right) \tag{13}$$

for k from 0 to $D - 1$, where N is the size of the specified array and D is the size of the DCT. If only one argument is given, then D is set to equal the next power of two larger than N. If a second argu-

ment is given, then its value is the *order* of the DCT, and the size of the DCT is 2^{order}. If a third argument is given, then it specifies the scaling factors s_k according to the following table:

TABLE 2: Normalization options for the DCT function

Name	Third argument	Normalization
Normalized	0	$s_k = \begin{cases} 1/\sqrt{2}; & k = 0 \\ 1; & \text{otherwise} \end{cases}$
Unnormalized	1	$s_k = 1$
Orthonormal	2	$s_k = \begin{cases} 1/\sqrt{D}; & k = 0 \\ \sqrt{2/D}; & \text{otherwise} \end{cases}$

The default, if a third argument is not given, is “Normalized.”

The IDCT function is similar, and can also take one, two, or three arguments. The formula in this case is

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos\left((2n+1)k\frac{\pi}{2D}\right). \tag{14}$$

5.8 Fixed Point Numbers

HyVisual 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 (signed) 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 DoubleMa-

trixToken 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.

Appendix A: Water Tanks Example

In this section, we step through construction of a simple hybrid system model in HyVisual. In the first model, there are two tanks, and a hose fills only one tank at a time. It switches to fill the other tank whenever the level of that tank goes below a certain value. To build this model, start HyVisual and use File...New...Graph Editor to get a new screen like this:

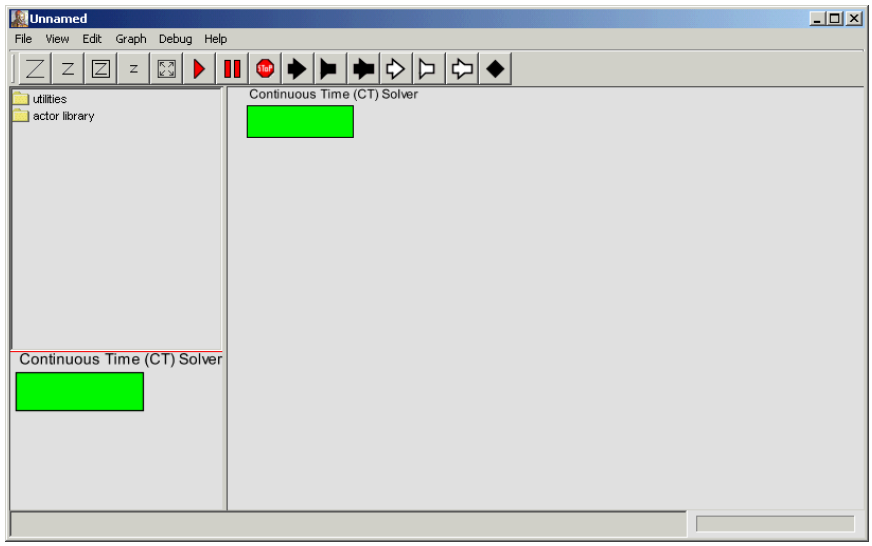


FIGURE 50. An empty HyVisual model

We will use a modal model to represent the water source. In the library on the left, open the *utilities* library. Then drag out a *modal model*. Your model should look like this:.

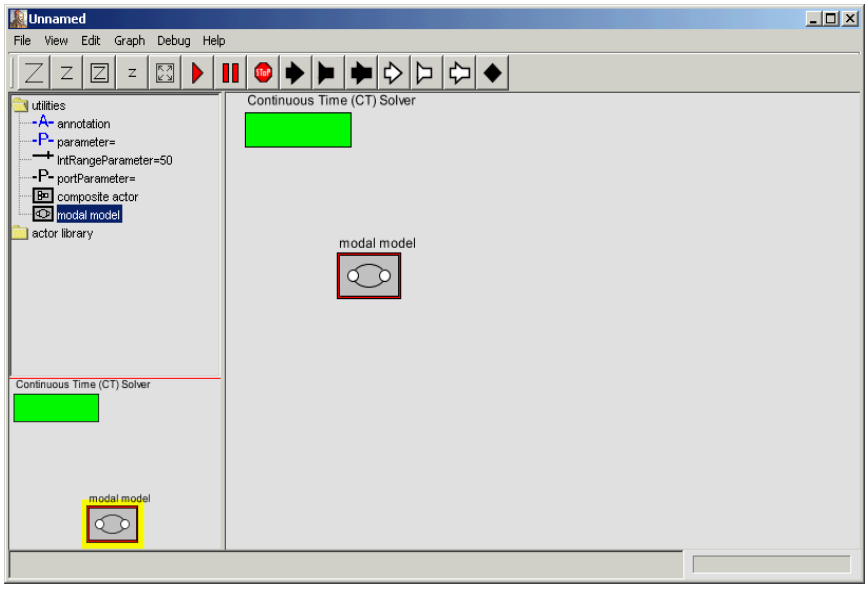


FIGURE 51. Model with a single modal model.

The water tanks themselves will be represented by integrators, where the output of each integrator is the level of the corresponding tank, and the input is the net flow rate into the tank. Find the integrator in the *dynamics* library, and create two instances, one for each water tank as follows:

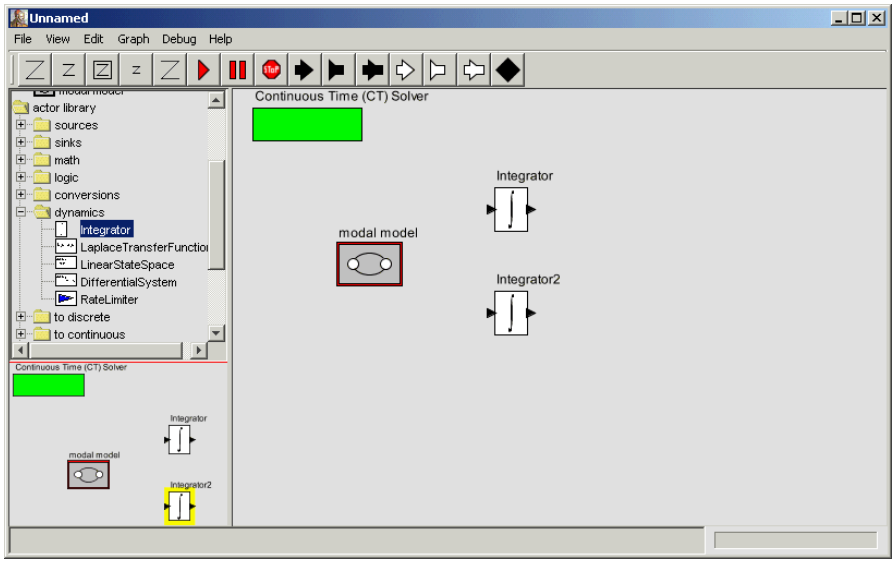


FIGURE 52. Two integrators, each representing one water tank.

The modal model needs to provide the flow rates for each of the tanks. These flow rates will depend on the levels, which will be the outputs of the integrators. Thus, we need four ports on the modal model. Call them *level1*, *level2*, *flow1*, *flow2*. To create them, right click on the modal model and select "Configure Ports," which yields the dialog shown below:

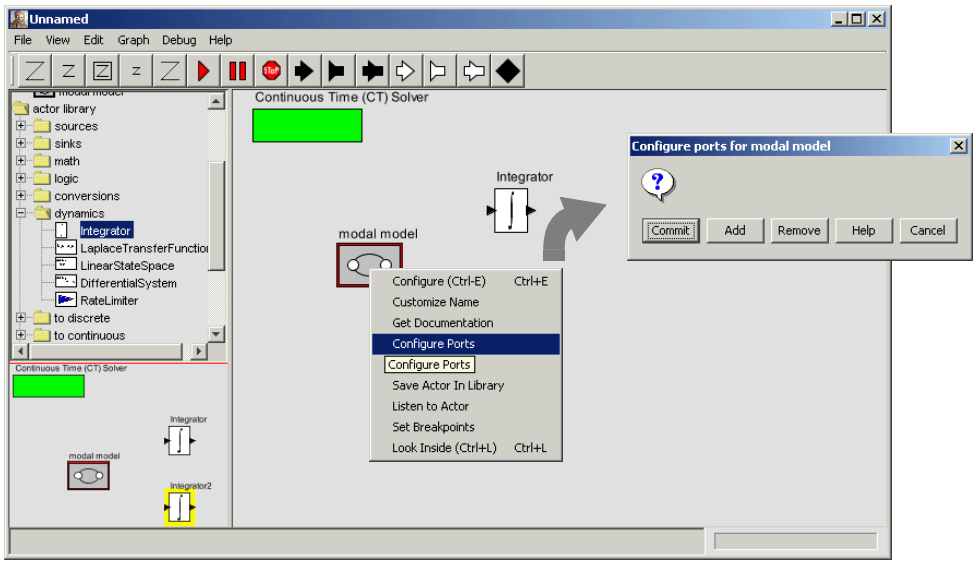


FIGURE 53. Invocation of the dialog to add ports to the modal model.

Click on the Add button four times to add the four ports, and specify whether they are inputs or outputs as follows:

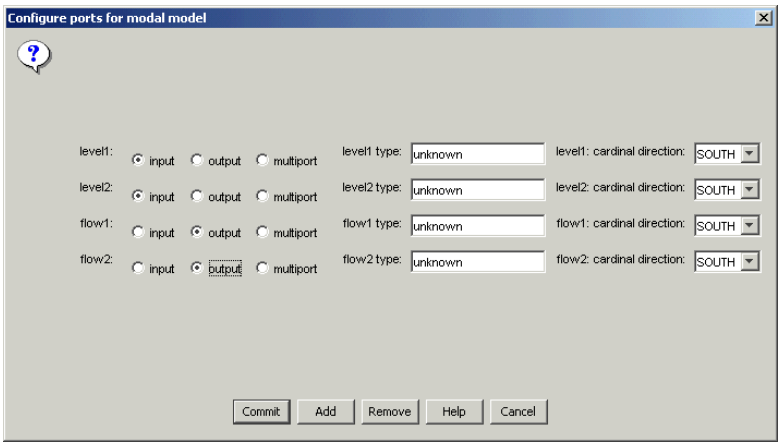


FIGURE 54. Dialog for adding ports to the modal model.

Your model should look like this:

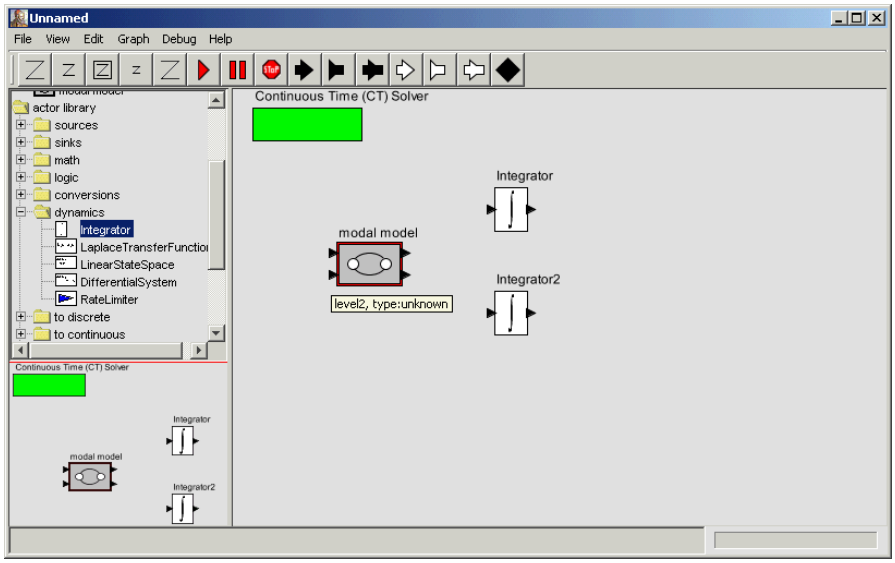


FIGURE 55. Modal model with ports.

Note that if you linger with the mouse over any of the ports, the tooltip will give the name of the port and its data type (which starts out unknown, but will later be inferred from the connections that we will

make). You can also optionally right click on the ports and select Customize Name and then Show name, in which case the ports will be labeled in the block diagram, as follows:

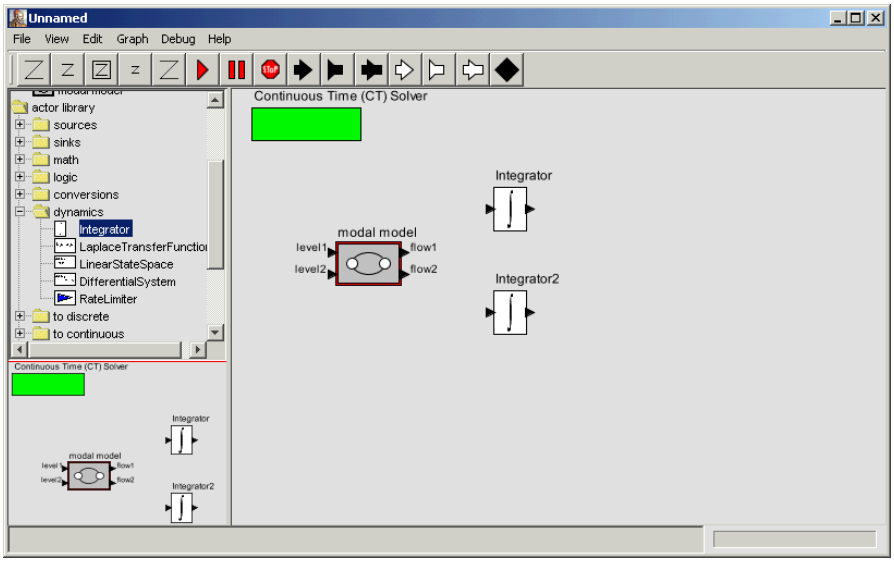


FIGURE 56. Labeled ports.

The final component that we need is a plotter to monitor the tank levels. This can be found in the *sinks* library, as shown below:

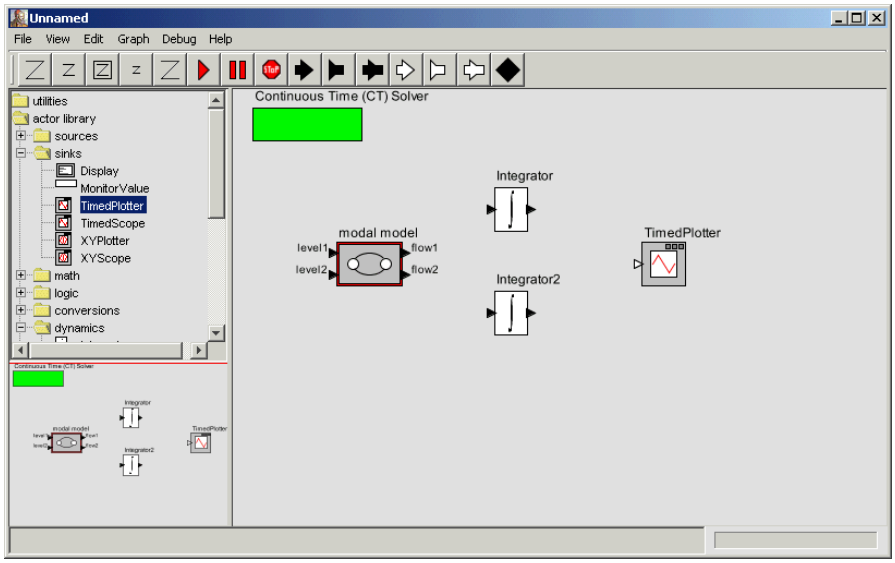


FIGURE 57. Model with a plotter.

The plotter is called a *TimedPlotter* because it plots signals as a function of time.

Next we need to wire the model together as follows:

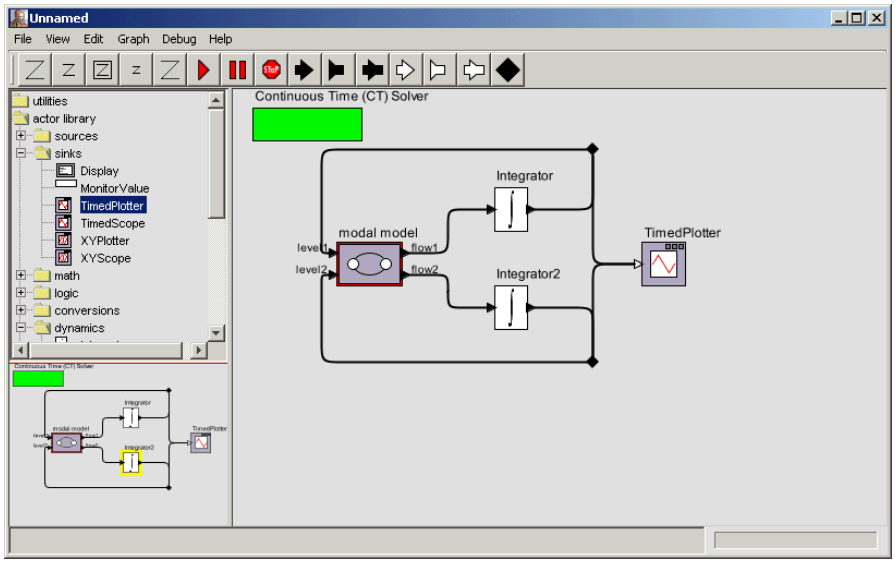


FIGURE 58. Top level tanks model, fully wired.

The black diamonds (which are called *relations*) are necessary in order to route the outputs of the integrators to both the plotter and the modal model. They can be created by clicking on the black diamond in the toolbar or by control-clicking in the diagram.

The integrators need initial values. Double click on the integrators to specify an initial value of 10.0, as follows:

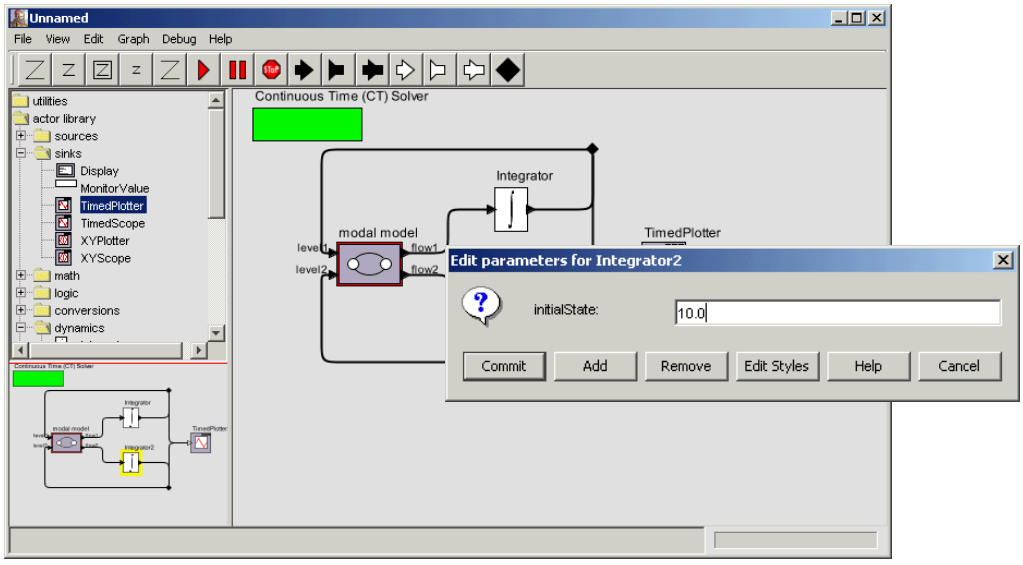


FIGURE 59. Top level tanks model, fully wired.

You may wish to save your model before proceeding.

As yet, our model has no interesting behavior because the modal model itself is empty. We will populate it with a state machine that will alternate between filling one water tank and filling the other. To populate it, simply look inside the modal model. You will see this:

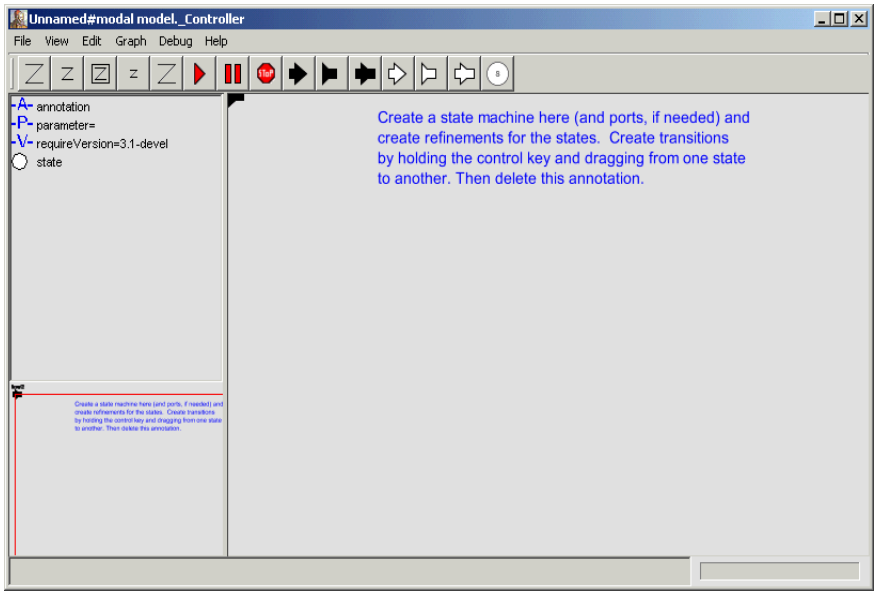


FIGURE 60. Inside a newly created modal model

Unfortunately, as of this writing, the four ports for the model are all placed on top of one another at the upper left of the diagram. Drag them to more reasonable locations as follows:

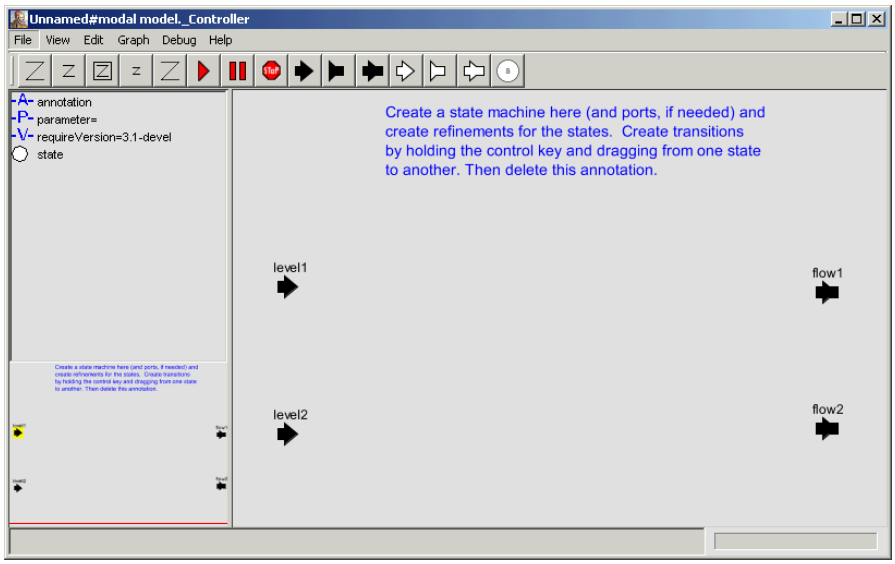


FIGURE 61. Modal model with more reasonable placement of ports.

In this window, we will create a state machine that represents the water source. Create two states named *fill1* and *fill2*, to correspond to filling one tank or the other. To make the states, drag a state out from the library on the left into the model, or click on the circle in the toolbar. To name a state, right

click it and select Customize Name. When you have given it the name, select Commit. After adding and naming the states, your model should look like this:

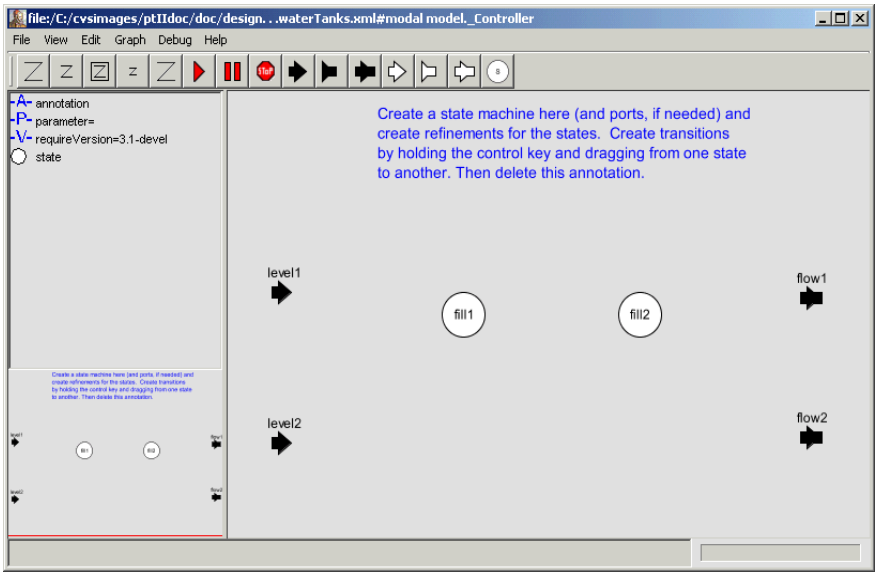


FIGURE 62. A modal model with two states

You will now need to draw arcs to represent possible transitions between the states. In this case, we assume you start in the *fill1* state, where tank 1 is being filled. To specify this, right click on the background, select Configure, and specify the initial state as follows:

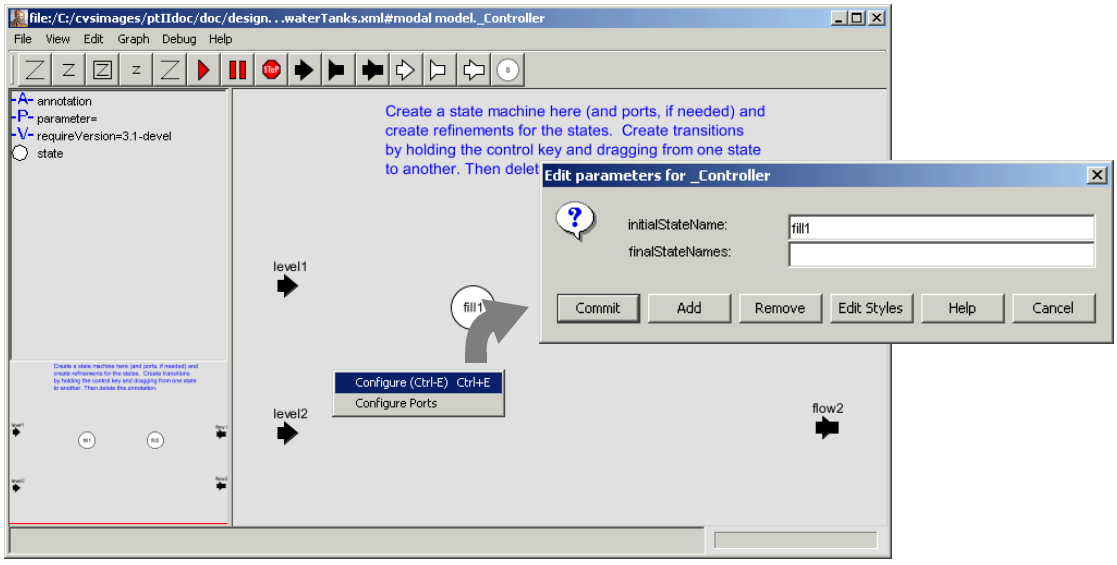


FIGURE 63. Specifying the initial state.

We draw arcs back and forth between *fill1* and *fill2* to represent the switching that happens every time the hose is moved from one tank to the other. To draw an arc hold the Control key while dragging the cursor from one state to the other. Repeat this for the other arc to get:

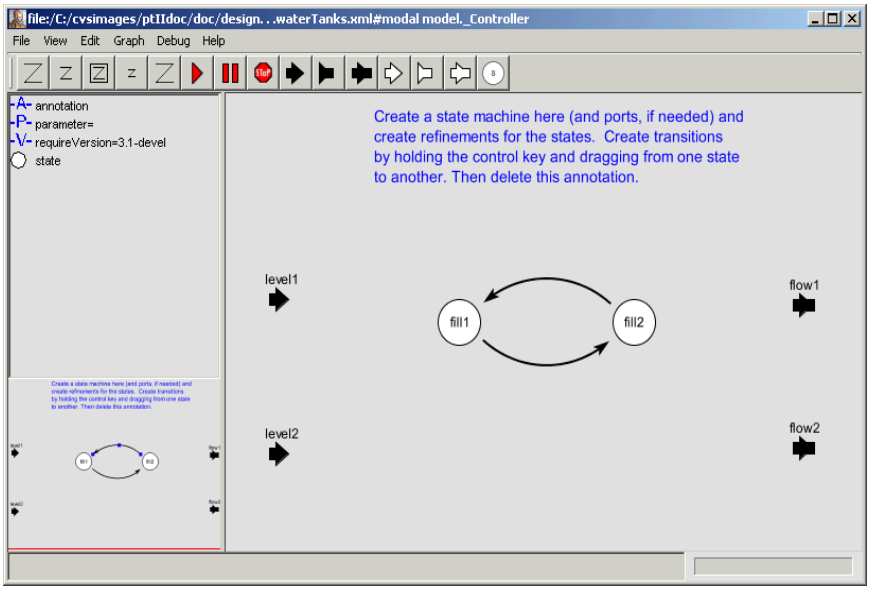


FIGURE 64. The modal model with the states connected

Next we need to specify the conditions under which a transition from one state to the other is taken. Such a condition is called a *guard*. To do this, double click on the arc from *fill1* to *fill2* and fill in the resulting dialog as follows:

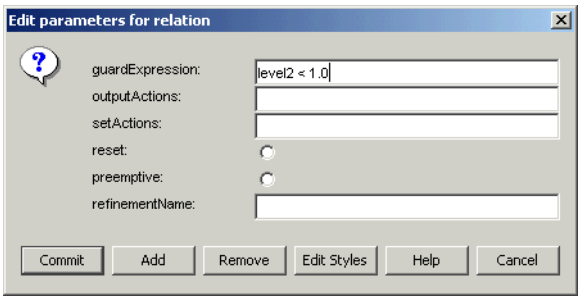


FIGURE 65. Specifying a guard for the transition from *fill1* to *fill2*.

This specifies that the transition from *fill1* to *fill2* should be taken when the level of tank 2 is less than 1.0 (in whatever units this model uses). Note that the input port *level2* is simply referenced by name in this expression. Any boolean-valued expression can be given as a guard. Double click on the second

transition to specify that it should be taken when the level of tank 1 is less than 1.0. The diagram should now look like this:

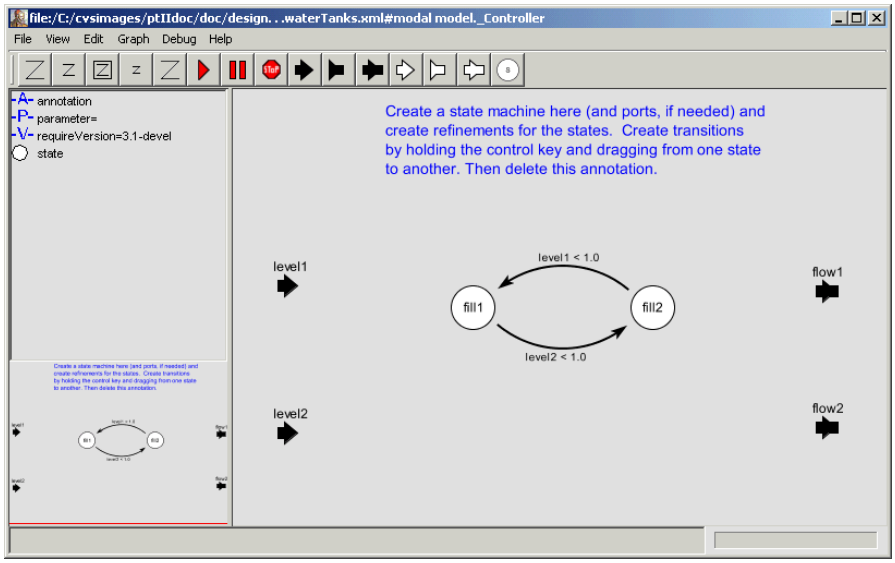


FIGURE 66. State machine with guards on the transitions.

For each of the two states, *fill1* and *fill2*, we can now specify a *refinement*, which is a model that defines the values of the output signals, possibly as a function of the input signals. To add the refinement for *fill1*, right click on the *fill1* state, and select Add Refinement. A refinement window will pop up that looks like this:

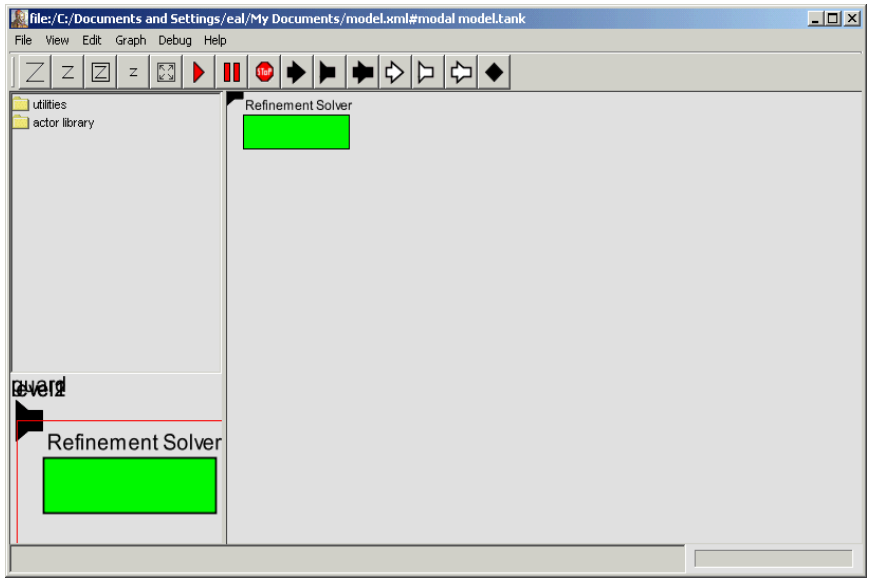


FIGURE 67. Inside the *fill1* refinement

All four ports have been automatically placed in the upper left corner of the screen, so you will need to drag them to get a more reasonable placement, for example as shown here:

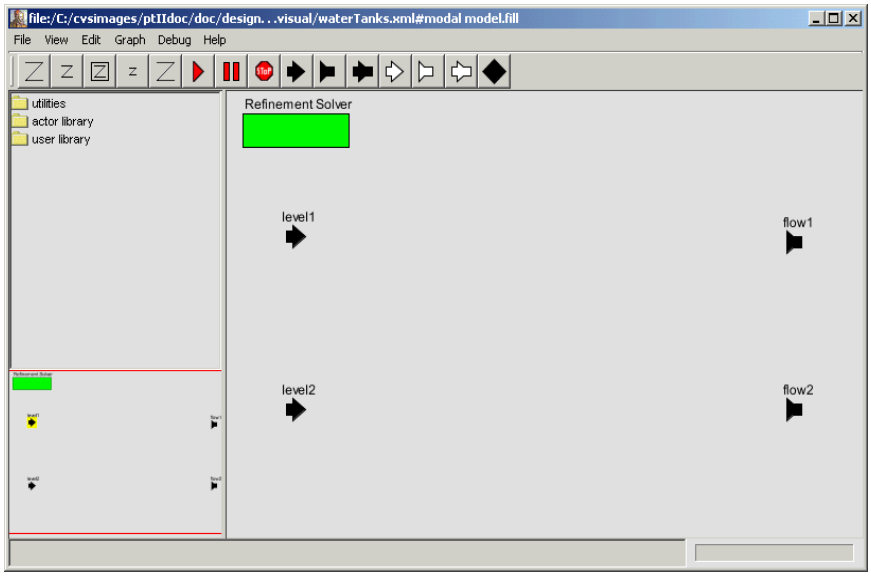


FIGURE 68. The *fill1* refinement after moving the ports.

We will initially assume a constant outflow rate for each of the two tanks. Assume in mode *fill1* that tank 1 gets a net in flow of 5.0 and that tank 2 gets a net outflow of 7.0. This can be specified by dragging in two instances of the Const actor from the sources library and double clicking on them to specify the values, as shown below:

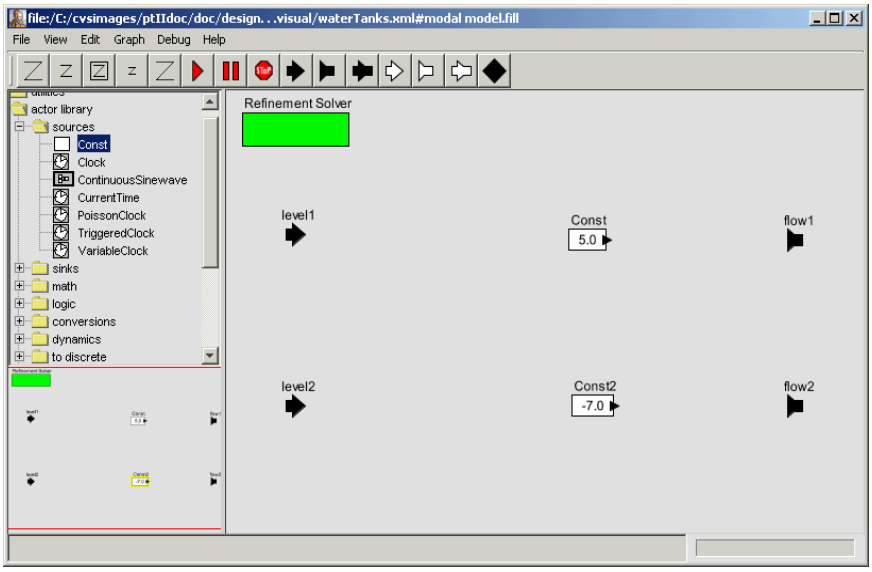


FIGURE 69. Flows added.

Notice that a net outflow is given as a negative number.

Wire the two Const actors to the output ports as follows:

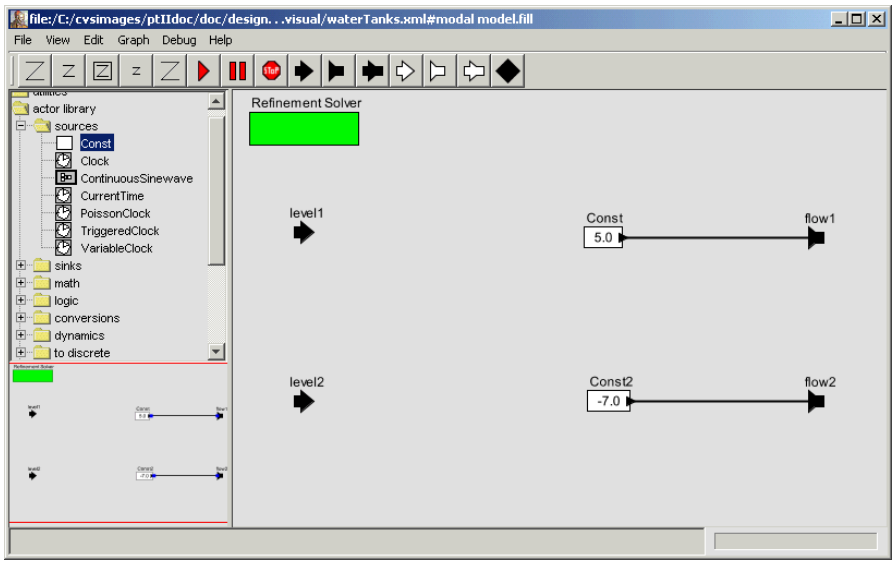


FIGURE 70. Refinement for mode *fill1*, fully wired.

To wire from one of the ports of the refinement to one of the ports of an actor it contains, hold the Control key while dragging the mouse from one to the other.

Create a similar refinement for the *fill2* mode, but this time, set the *flow2* output to 5.0 and the *flow1* output to -6.0, as shown below:

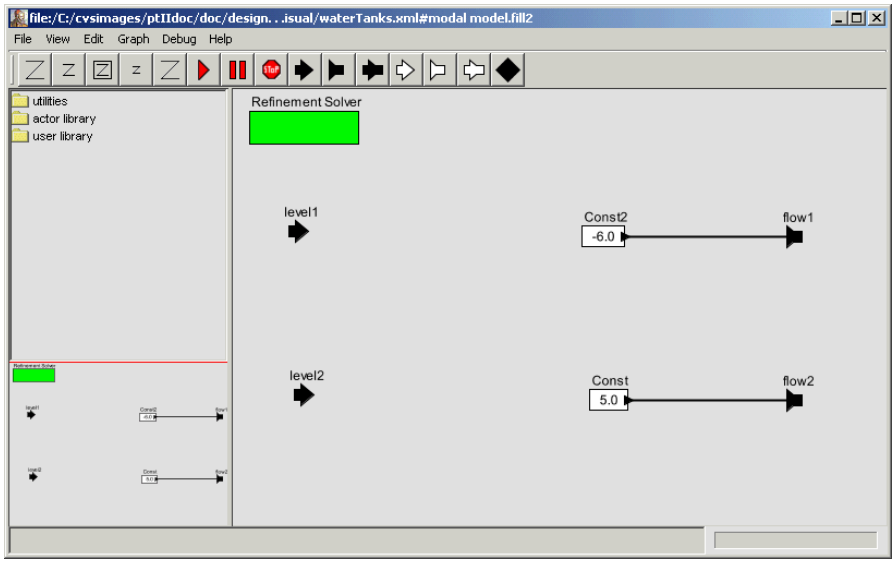


FIGURE 71. Refinement for mode *fill1*, fully wired.

You should probably save again at this point. You can use Ctrl-S as a shortcut to save. To run the model, press the Play button on the menu bar (red triangle), and you should see this:

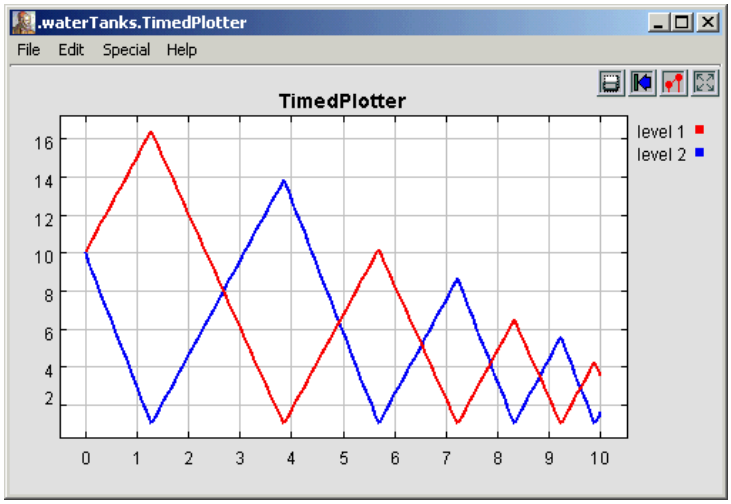


FIGURE 72. Levels of the two water tanks plotted as a function of time.

To get the legend as shown on the upper right, double click on the TimedPlotter actor and fill in its parameters as shown here:

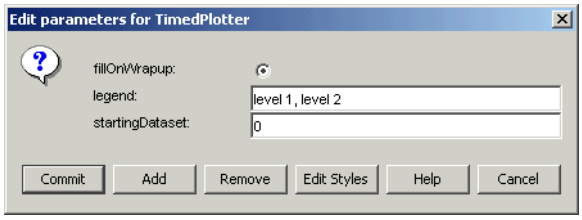


FIGURE 73. Parameters of the TimedPlotter to get a legend.

Other attributes of the plot (such as the title) can be set by clicking on the second button from the right at the upper right of the plot.

Note in figure 72 that the time between switching appears to get shorter. Indeed, you can run the model longer, **but be careful**. This model exhibits a condition called the Zeno condition, and in fact, time cannot progress beyond a certain point because the time between switches becomes vanishingly

small. To witness this phenomenon, double click on the director, which is the green box named Continuous Time (CT) Solver, and change its *stopTime* parameter to 20.0, as shown below:

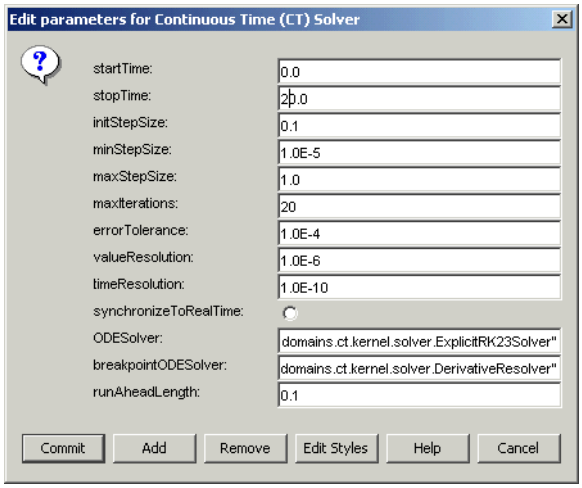


FIGURE 74. Solver parameters to get a longer run.

Then click on the run button, and note that the execution appears to freeze after a short time. **You must now click on the stop button,** or else the run will continue to produce an extremely large number of events, and attempt to plot them. The plotter can get very sluggish when asked to plot a large number of events, so you should stop the model shortly after it appears to freeze. At this point, you should see something like this:

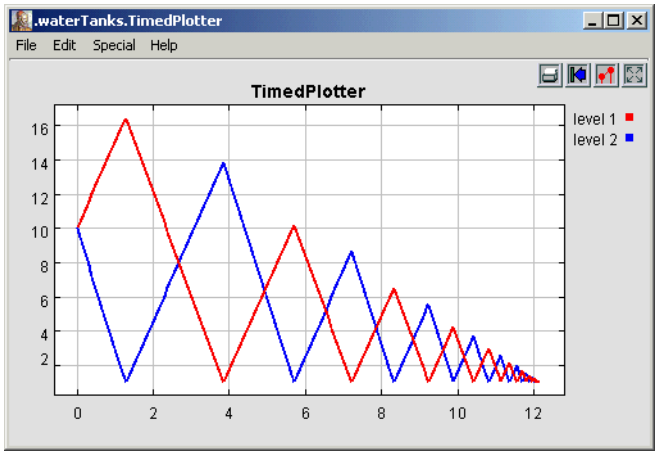


FIGURE 75. Levels of the two water tanks plotted as a function of time.

You can verify that the time between events is getting very small by dragging a box around the tail of this plot to zoom in.

A more interesting model would close the feedback loop in the state refinements. For example, you can replace the *fill2* refinement as follows so that the rate at which water drains from tank 1 is proportional to the level in tank 1:

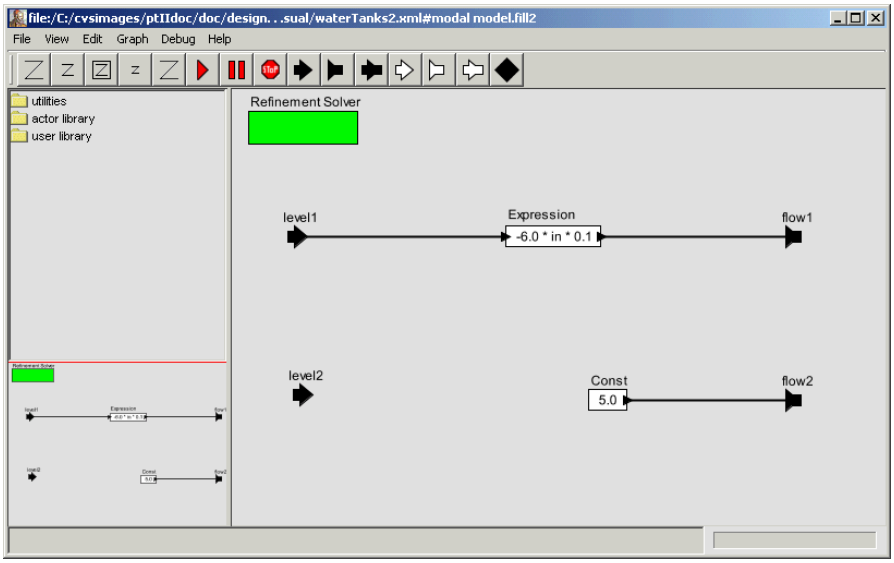


FIGURE 76. Variant of *fill2* refinement that makes the rate at which water drains from tank 1 proportional to the level in the tank..

Running this variant yields the following plot:

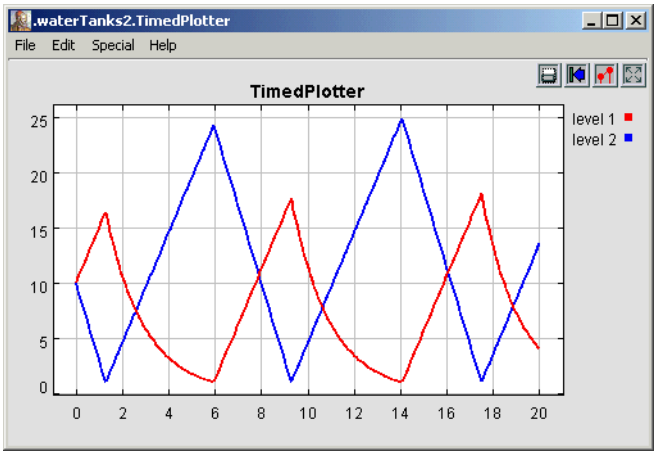


FIGURE 77. Plot with the variant of figure 76.

Note that the Zeno condition is gone.

Appendix B: Tables of Functions

In this appendix, we tabulate the functions available in the expression language. Further explanation of many of these functions is given in section section 5.7 above.

B.1 Trigonometric Functions

TABLE 3: Trigonometric functions.

function	argument type(s)	return type	description
acos	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range [0.0, pi] or NaN if out of range or *complex*	arc cosine *complex* case: $\mathrm{acos}(z) = -i\log(z + i\,\mathrm{sqrt}(1 - z^2))$
asin	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range [-pi/2, pi/2] or NaN if out of range or *complex*	arc sine *complex* case: $\mathrm{asin}(z) = -i\log(iz + \mathrm{sqrt}(1 - z^2))$
atan	*double* or *complex*	*double* in the range [-pi/2, pi/2] or *complex*	arc tangent *complex* case: $\mathrm{atan}(z) = -\frac{i}{2}\log\left(\frac{i - z}{i + z}\right)$
atan2	*double*, *double*	*double* in the range [-pi, pi]	angle of a vector (note: the arguments are (*y*,*x*), not (*x*,*y*) as one might expect).
acosh	*double* greater than 1 or *complex*	*double* or *complex*	hyperbolic arc cosine, defined for both *double* and *complex* case by: $\mathrm{acosh}(z) = \log(z + \mathrm{sqrt}(z^2 - 1))$
asinh	*double* or *complex*	*double* or *complex*	hyperbolic arc sine *complex* case: $\mathrm{asinh}(z) = \log(z + \mathrm{sqrt}(z^2 + 1))$
cos	*double* or *complex*	*double* in the range $[-1, 1]$, or *complex*	cosine *complex* case: $\cos(z) = \frac{(\exp(iz) + \exp(-iz))}{2}$
cosh	*double* or *complex*	*double* or *complex*	hyperbolic cosine, defined for *double* or *complex* by: $\cosh(z) = \frac{(\exp(z) + \exp(-z))}{2}$
sin	*double* or *complex*	*double* or *complex*	sine function *complex* case: $\sin(z) = \frac{(\exp(iz) - \exp(-iz))}{2i}$
sinh	*double* or *complex*	*double* or *complex*	hyperbolic sine, defined for *double* or *complex* by: $\sinh(z) = \frac{(\exp(z) - \exp(-z))}{2}$
tan	*double* or *complex*	*double* or *complex*	tangent function, defined for *double* or *complex* by: $\tan(z) = \frac{\sin(z)}{\cos(z)}$
tanh	*double* or *complex*	*double* or *complex*	hyperbolic tangent, defined for *double* or *complex* by: $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

B.2 Basic Mathematical Functions

TABLE 4: Basic mathematical functions

function	argument type(s)	return type	description
abs	*double* or *int* or *long* or *complex*	*double* or *int* or *long* (*complex* returns double)	absolute value complex case: $\text{abs}(a+ib) = \|z\| = \sqrt{a^2+b^2}$
angle	*complex*	*double* in the range [-pi, pi]	angle or argument of the complex number: $\angle z$
ceil	*double*	*double*	ceiling function, which returns the smallest (closest to negative infinity) *double* value that is not less than the argument and is an integer.
compare	*double*, *double*	*int*	compare two numbers, returning -1, 0, or 1 if the first argument is less than, equal to, or greater than the second.
conjugate	*complex*	*complex*	complex conjugate
exp	*double* or *complex*	*double* in the range [0.0, infinity] or complex	exponential function (e^argument) complex case: $e^{a+ib} = e^a(\cos(b)+i\sin(b))$
floor	*double*	*double*	floor function, which is the largest (closest to positive infinity) value not greater than the argument that is an integer.
gaussian	*double*, *double* or *double*, *double*, *int*, or *double*, *double*, *int*, *int*	*double* or {*double*} or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see page 55).
imag	*complex*	*double*	imaginary part
isInfinite	*double*	*boolean*	return true if the argument is infinite
isNaN	*double*	*boolean*	return true if the argument is “not a number”
log	*double* or *complex*	*double* or *complex*	natural logarithm complex case: $\log(z) = \log(\text{abs}(z)+i\text{angle}(z))$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2
max	*double*, *double* or *int*, *int* or *long*, *long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	maximum
min	*double*, *double* or *int*, *int* or *long*, *long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	minimum

TABLE 4: Basic mathematical functions

function	argument type(s)	return type	description
neighborhood	*type, type, double*	*boolean*	return true if the first argument is in the neighborhood of the second, meaning that the distance is less than or equal to the third argument. The first two arguments can be any type for which such a distance is defined. For composite types, arrays, records, and matrices, then return true if the first two arguments have the same structure, and each corresponding element is in the neighborhood.
pow	*double*, *double* or *complex*, *complex*	double or complex	first argument to the power of the second
random	no arguments or *int* or *int, int*	*double* or {*double*} or [*double*]	one or more random numbers between 0.0 and 1.0 (see page 55)
real	*complex*	*double*	real part
remainder	*double*, *double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see page 56).
round	*double*	*long*	round to the nearest *long*, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0L. If the argument is out of range, the result is either MaxLong or MinLong, depending on the sign.
roundToInt	*double*	*int*	round to the nearest *int*, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0. If the argument is out of range, the result is either MaxInt or MinInt, depending on the sign.
sgn	*double*	*int*	-1 if the argument is negative, 1 otherwise
sqrt	*double* or *complex*	*double* or *complex*	square root. If the argument is *double* with value less than zero, then the result is NaN. complex case: $\mathrm{sqrt}(z) = \sqrt{\lvert z\rvert}\left(\cos\left(\frac{\angle z}{2}\right) + i\sin\left(\frac{\angle z}{2}\right)\right)$
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians

5.9 Matrix, Array, and Record Functions.

TABLE 5: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
arrayToMatrix	{*type*}, int, int	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type, type,* int	{*type*}	Create an array with values starting with the first argument, incremented by the second argument, of length given by the third argument.
crop	[*int*], *int*, *int*, *int*, *int* or [*double*], *int*, *int*, *int*, *int* or [*complex*], *int*, *int*, *int*, *int* or [*long*], *int*, *int*, *int*, *int* or	[*int*] or [*double*] or [*complex*] or [*long*] or	Given a matrix of any *type*, return a submatrix starting at the specified row and column with the specified number of rows and columns.
determinant	[*double*] or [*complex*]	*double* or *complex*	Return the determinant of the specified matrix.
diag	{*type*}	[*type*]	Return a diagonal matrix with the values along the diagonal given by the specified array.
divideElements	[*type*], [*type*]	[*type*]	Return the element-by-element division of two matrices
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = 1/(i+j+1)$. A Hilbert matrix is nearly, but not quite singular.
identityMatrixComplex	*int*	[*complex*]	Return an identity matrix with the specified dimension.
identityMatrixDouble	*int*	[*double*]	Return an identity matrix with the specified dimension.
identityMatrixInt	*int*	[*int*]	Return an identity matrix with the specified dimension.
identityMatrixLong	*int*	[*long*]	Return an identity matrix with the specified dimension.
intersect	*record, record*	*record*	Return a record that contains only fields that are present in both arguments, where the value of the field is taken from the first record.
inverse	[*double*] or [*complex*]	[*double*] or [*complex*]	Return the inverse of the specified matrix, or throw an exception if it is singular.
matrixToArray	[*type*]	{*type*}	Create an array containing the values in the matrix
merge	*record*, *record*	*record*	Merge two records, giving priority to the first one when they have matching record labels.
multiplyElements	[*type*], [*type*]	[*type*]	Multiply elementwise the two specified matrices.
orthogonalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal columns.
orthogonalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal rows.
orthonormalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal columns.
orthonormalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal rows.
repeat	*int*, *type*	{*type*}	Create an array by repeating the specified token the specified number of times.
sum	{*type*} or [*type*]	*type*	Sum the elements of the specified array or matrix. This throws an exception if the elements do not support addition or if the array is empty (an empty matrix will return zero).
trace	[*type*]	*type*	Return the trace of the specified matrix.

TABLE 5: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
zeroMatrixComplex	*int*, *int*	[*complex*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixDouble	*int*, *int*	[*double*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixInt	*int*, *int*	[*int*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixLong	*int*, *int*	[*long*]	Return a zero matrix with the specified number of rows and columns.

B.3 Functions for Evaluating Expressions

TABLE 6: Utility functions for evaluating expressions

function	argument type(s)	return type	description
eval	*string*	any type	evaluate the specified expression (see page 55).
parseInt	*string* or *string*, *int*	*int*	return an *int* read from a *string*, using the given radix if a second argument is provided.
parseLong	*string* or *string*, *int*	*int*	return a *long* read from a *string*, using the given radix if a second argument is provided.
toBinaryString	*int* or *long*	*string*	return a binary representation of the argument
toOctalString	*int* or *long*	string	return an octal representation of the argument
toString	*double* or *int* or *int*, *int* or *long* or *long*, *int*	*string*	return a string representation of the argument, using the given radix if a second argument is provided.
traceEvaluation	*string*	*string*	evaluate the specified expression and report details on how it was evaluated (see page 55).

B.4 Signal Processing Functions

TABLE 7: Functions performing signal processing operations

function	argument type(s)	return type	description
convolve	{*double*}, {*double*} or {*complex*}, {*complex*}	{*double*} or {*complex*}	Convolve two arrays and return an array whose length is sum of the lengths of the two arguments minus one. Convolution of two arrays is the same as polynomial multiplication.
DCT	{*double*} or {*double*}, *int* or {*double*}, *int, int*	{*double*}	Return the discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 56).
downsample	{*double*}, *int* or {*double*}, *int, int*	{*double*}	Return a new array with every n -th element of the argument array, where n is the second argument. If a third argument is given, then it must be between 0 and $n-1$, and it specifies an offset into the array (by giving the index of the first output).
FFT	{*double*} or {*complex*} or {*double*}, *int* {*complex*}, *int*	{*complex*}	Return the fast Fourier transform of the specified array. If the second argument is given with value n , then the length of the transform is 2^n . Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
generateBartlettWindow	*int*	{*double*}	Return a Bartlett (rectangular) window with the specified length. The end points have value 0.0, and if the length is odd, the center point has value 1.0. For length M + 1, the formula is: $w(n) = \begin{cases} 2\frac{n}{M}; & \text{if } 0 \le n \le \frac{M}{2} \\ 2-2\frac{n}{M}; & \text{if } \frac{M}{2} \le n \le M \end{cases}$
generateBlackmanWindow	*int*	{*double*}	Return a Blackman window with the specified length. For length M + 1, the formula is: $w(n) = 0.42 + 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M)$
generateBlackmanHarrisWindow	*int*	{*double*}	Return a Blackman-Harris window with the specified length. For length M + 1, the formula is: $w(n) = 0.35875 + 0.48829\cos(2\pi n/M) + 0.14128\cos(4\pi n/M) + 0.01168\cos(6\pi n/M)$
generateGaussianCurve	*double, double, int*	{*double*}	Return a Gaussian curve with the specified standard deviation, extent, and length. The extent is a multiple of the standard deviation. For instance, to get 100 samples of a Gaussian curve with standard deviation 1.0 out to four standard deviations, use generateGaussianCurve(1.0, 4.0, 100).
generateHammingWindow	*int*	{*double*}	Return a Hamming window with the specified length. For length M + 1, the formula is: $w(n) = 0.54 - 0.46\cos(2\pi n/M)$
generateHanningWindow	*int*	{*double*}	Return a Hanning window with the specified length. For length M + 1, the formula is: $w(n) = 0.5 - 0.5\cos(2\pi n/M)$

TABLE 7: Functions performing signal processing operations

function	argument type(s)	return type	description
generatePolynomialCurve	{*double*}, *double*, *double*, *int*	{*double*}	Return samples of a curve specified by a polynomial. The first argument is an array with the polynomial coefficients, beginning with the constant term, the linear term, the squared term, etc. The second argument is the value of the polynomial variable at which to begin, and the third argument is the increment on this variable for each successive sample. The final argument is the length of the returned array.
generateRaisedCosinePulse	*double*, *double*, *int*	{*double*}	Return an array containing a symmetric raised-cosine pulse. This pulse is widely used in communication systems, and is called a "raised cosine pulse" because the magnitude its Fourier transform has a shape that ranges from rectangular (if the excess bandwidth is zero) to a cosine curved that has been raised to be non-negative (for excess bandwidth of 1.0). The elements of the returned array are samples of the function: $$h(t) = \frac{\sin(\pi t/T)}{\pi t/T} \times \frac{\cos(x\pi t/T)}{1-(2xt/T)^2},$$ where *x* is the excess bandwidth (the first argument) and *T* is the number of samples from the center of the pulse to the first zero crossing (the second argument). The samples are taken with a sampling interval of 1.0, and the returned array is symmetric and has a length equal to the third argument. With an excessBandwidth of 0.0, this pulse is a sinc pulse.
generateRectangularWindow	*int*	{*double*}	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	{*double*} or {*double*}, *int* or {*double*}, *int, int*	{*double*}	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 56).
IFFT	{*double*} or {*complex*} or {*double*}, *int* {*complex*}, *int*	{*complex*}	Return the inverse fast Fourier transform of the specified array. If the second argument is given with value n, then the length of the transform is 2^n. Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroToFrequency	{*complex*}, {*complex*}, *complex*, *int*	{*complex*}	Given an array of pole locations, an array of zero locations, a gain term, and a size, return an array of the specified size representing the frequency response specified by these poles, zeros, and gain. This is calculated by walking around the unit circle and forming the product of the distances to the zeros, dividing by the product of the distances to the poles, and multiplying by the gain.
sinc	*double*	*double*	Return the sinc function, $\sin(x)/x$, where special care is taken to ensure that 1.0 is returned if the argument is 0.0.

TABLE 7: Functions performing signal processing operations

function	argument type(s)	return type	description
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	{*double*}	{*double*}	Modify the specified array to unwrap the angles. That is, if the difference between successive values is greater than π in magnitude, then the second value is modified by multiples of 2π until the difference is less than or equal to π. In addition, the first element is modified so that its difference from zero is less than or equal to π in magnitude.
upsample	{*double*}, *int*	{*double*}	Return a new array that is the result of inserting $n-1$ zeroes between each successive sample in the input array, where n is the second argument. The returned array has length nL, where L is the length of the argument array. It is required that $n > 0$.

I/O Functions and Other Miscellaneous Functions

TABLE 8: Miscellaneous functions.

function	argument type(s)	return type	description
cast	*type1*, *type2*	*type1*	Return the second argument converted to the type of the first, or throw an exception if the conversion is invalid.
constants	none	*record*	Return a record identifying all the globally defined constants in the expression language.
findFile	*string*	*string*	Given a file name relative to the user directory, current directory, or classpath, return the absolute file name of the first match, or return the name unchanged if no match is found.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	*function*, *int*, *type*	{*type*}	Return an array that results from first applying the specified function to the third argument, then applying it to the result of that application, and repeating to get an array whose length is given by the second argument.
map	*function*, {*type*}	{*type*}	Return an array that results from applying the specified function to the elements of the specified array.
property	*string*	*string*	Return a system property with the specified name from the environment, or an empty string if there is none. Some useful properties are java.version, ptolemy.ptII.dir, ptolemy.ptII.dirAsURL, and user.dir.
readFile	*string*	*string*	Get the string text in the specified file, or throw an exception if the file cannot be found. The file can be absolute, or relative to the current working directory (user.dir), the user's home directory (user.home), or the classpath.
readResource	*string*	*string*	Get the string text in the specified resource (which is a file found relative to the classpath), or throw an exception if the file cannot be found.
totalMemory	none	*long*	Return the approximate number of bytes used by current objects plus those available for future object allocation.