HYVISUAL: A HYBRID SYSTEM VISUAL MODELER

Authors: *Edward A. Lee*
Jie Liu
Xiaojun Liu

Version 2.2-beta
UCB ERL Memorandum Number FIXME
January 15, 2003

1. Introduction

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

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

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

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

1.1 Quick Start

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

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

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

```
vergil -hybrid
```

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

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

2. Continuous-Time Dynamical Systems

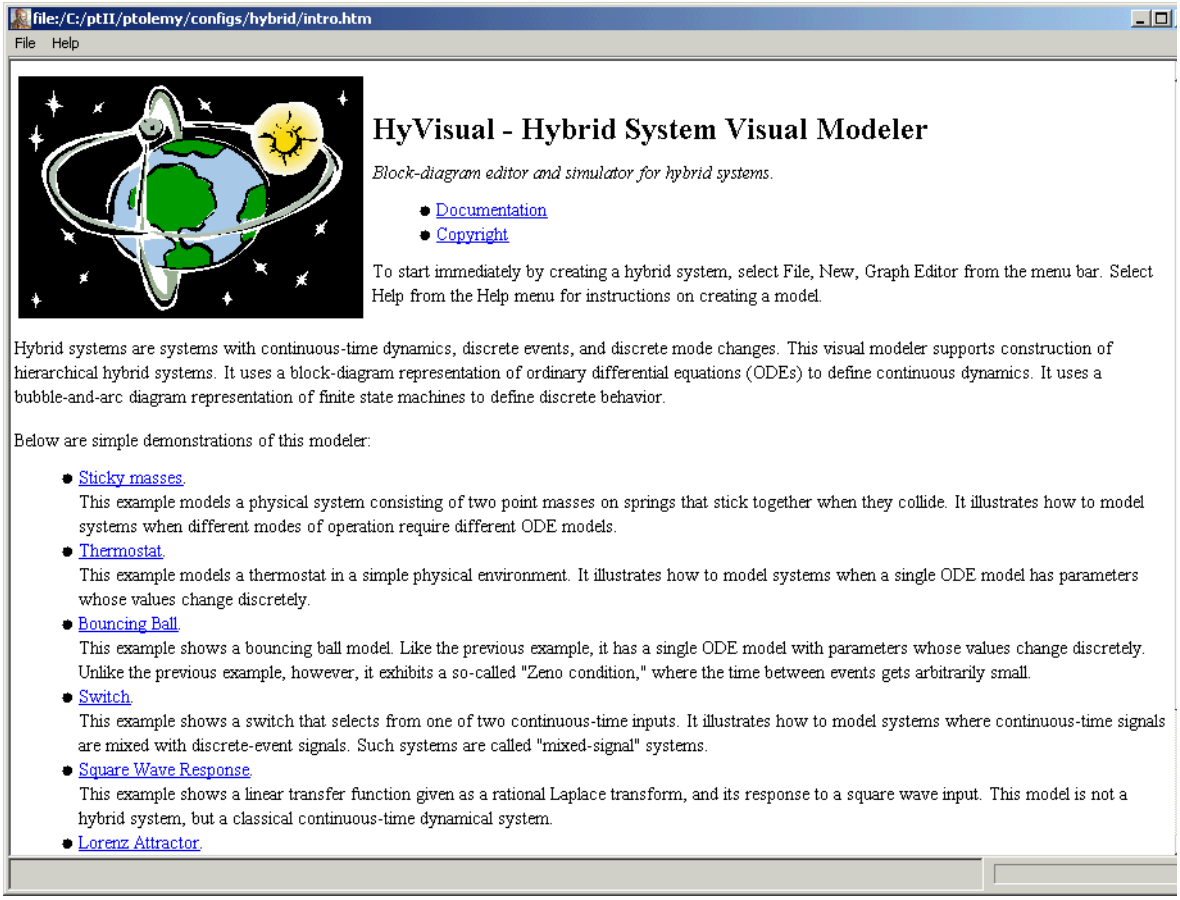

FIGURE 1. Initial welcome window.

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

2.1 Executing a Pre-Built Model

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

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

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

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

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

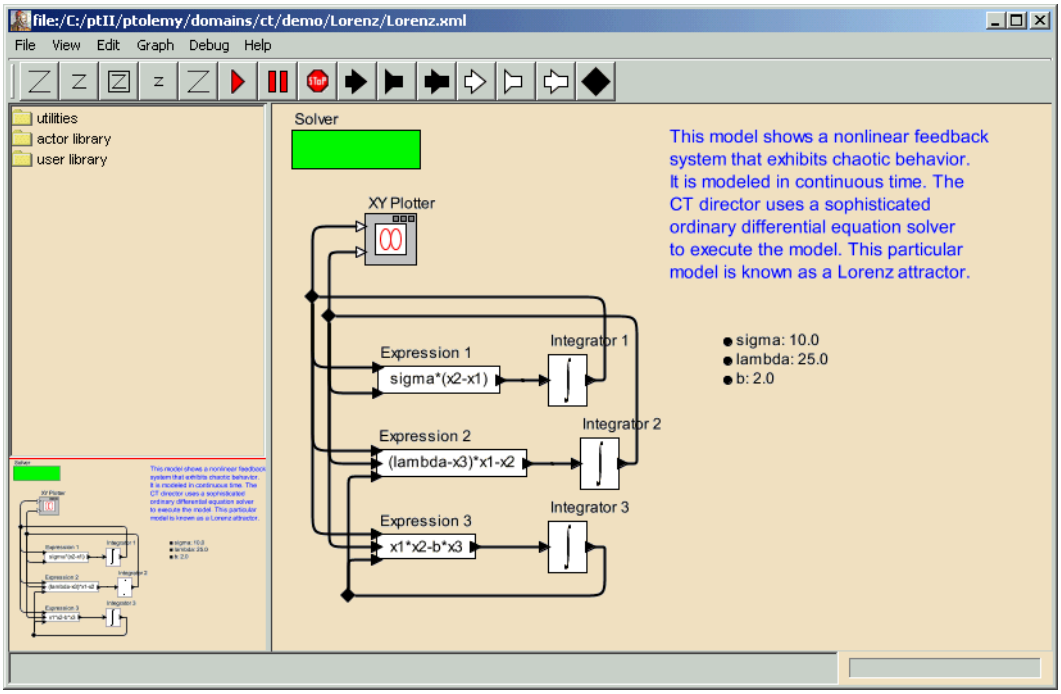

FIGURE 2. A block diagram representation of a set of nonlinear ordinary differential equations.

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 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 values of t in between *startTime* and *stopTime*. The Run Window obtained via the View menu is shown in figure 5.

Like the Lorenz model, a typical continuous-time model contains integrators in feedback loops, or more elaborate blocks that realize linear and non-linear dynamical systems given abstract mathemati-

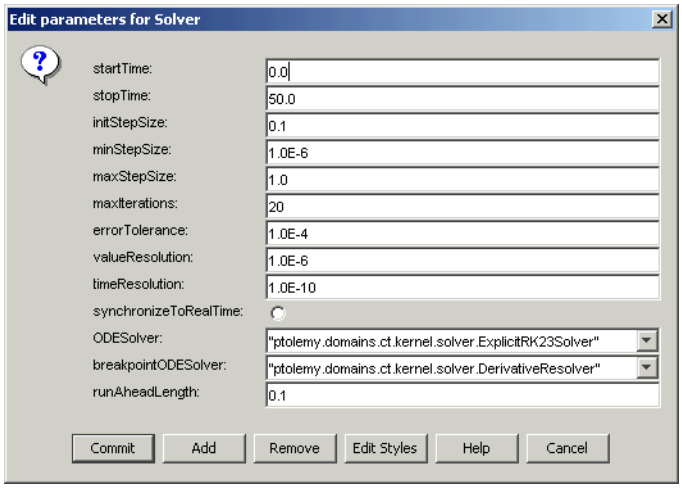

FIGURE 3. Dialog box showing solver parameters for the model in figure 2.

cal representations of them (such as Laplace transforms). In the next section, we will explore how to build a model from scratch.

2.2 Creating a New Model

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

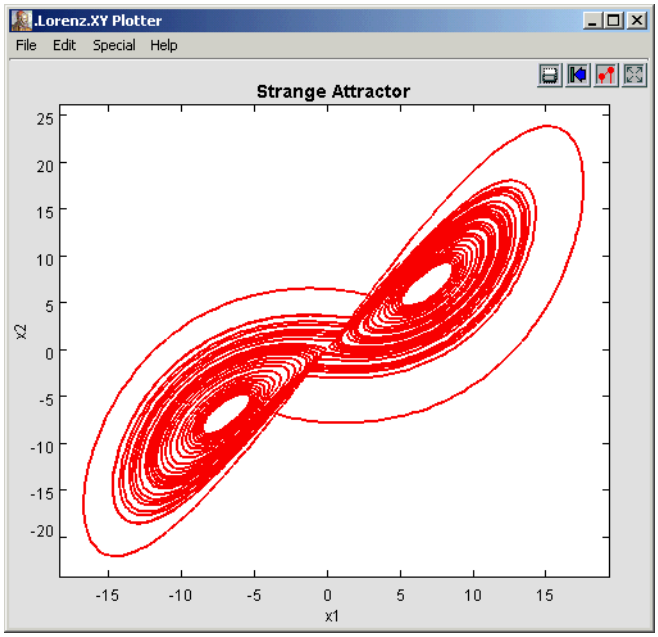

FIGURE 4. Result of running the Lorenz model using the run button in the toolbar.

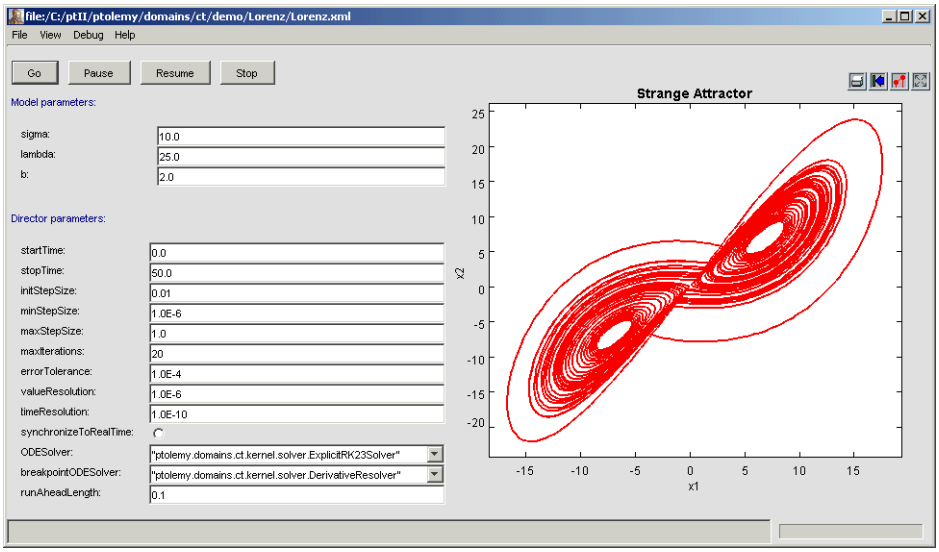

FIGURE 5. Run Window, obtained via the View menu, for the Lorenz model shown in figure 2.

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 *TimedSinewave* actor from the *timed sources* library and the *TimedPlotter* from the *timed sinks* library. Connect the output of the *TimedSinewave* to the input of the *TimedPlotter* by dragging from one port to the other. The result should look something like figure 7.

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

If we zoom in, 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 actually pretty 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*

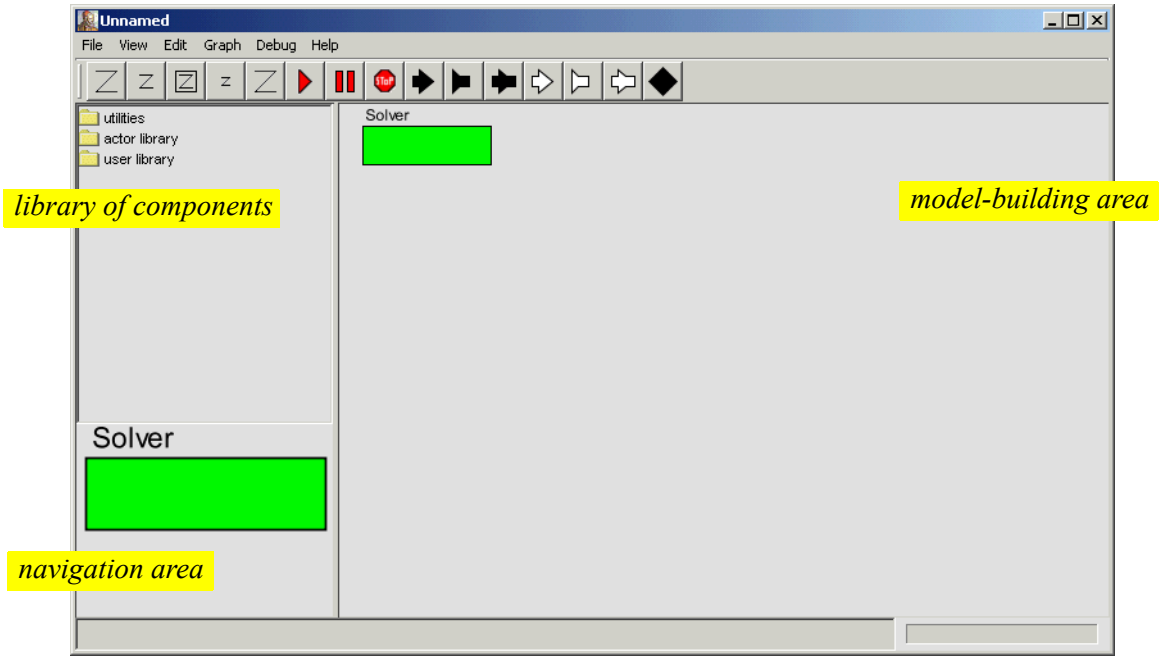

FIGURE 6. A blank model, obtained via File, New, and Graph Editor in the menus.

discrete library. For example, another way to get the sine wave to be sampled with a sampling interval of 0.01 is shown in figure 11. The *PeriodicSampler* block has a parameter *samplePeriod* that you can set to 0.01 (by double clicking on the block). This will result in the same plot as shown in figure 10, irrespective of the *initStepSize* parameter of the solver.

The models shown in figures 7 and 11 have no blocks from the *dynamics* library, and hence do not immediately represent an ordinary differential equation. When blocks from the *dynamics* library are used, then the solver uses sophisticated techniques to determine the spacing between samples. The initial step size is given by *initStepSize*, but the solver may adjust it to any value between *minStepSize* and *maxStepSize*.

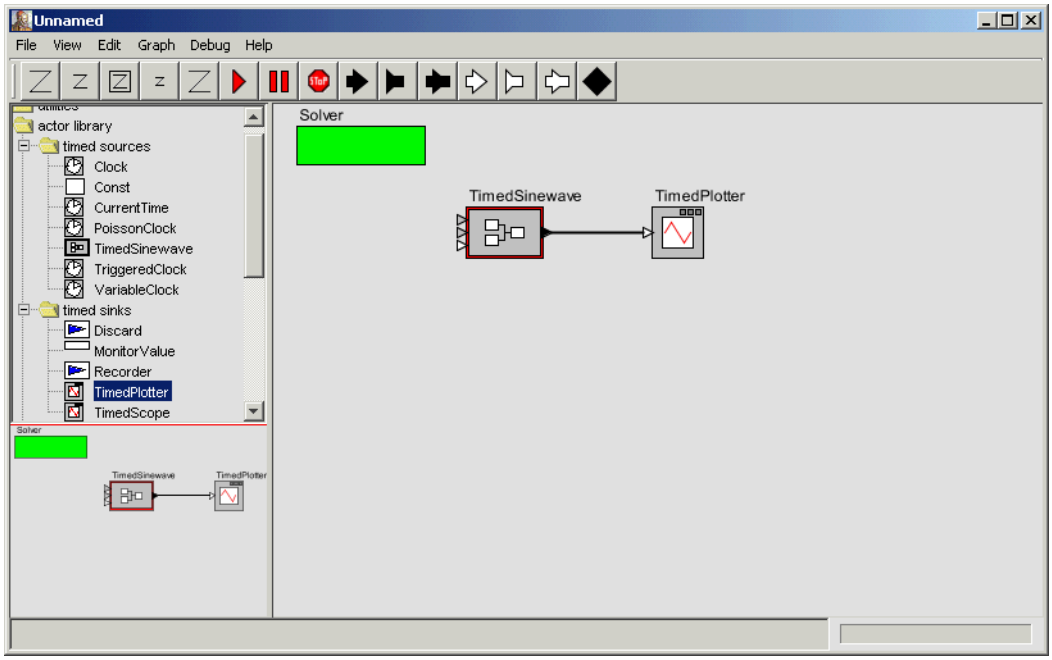

FIGURE 7. A model populated with two actors.

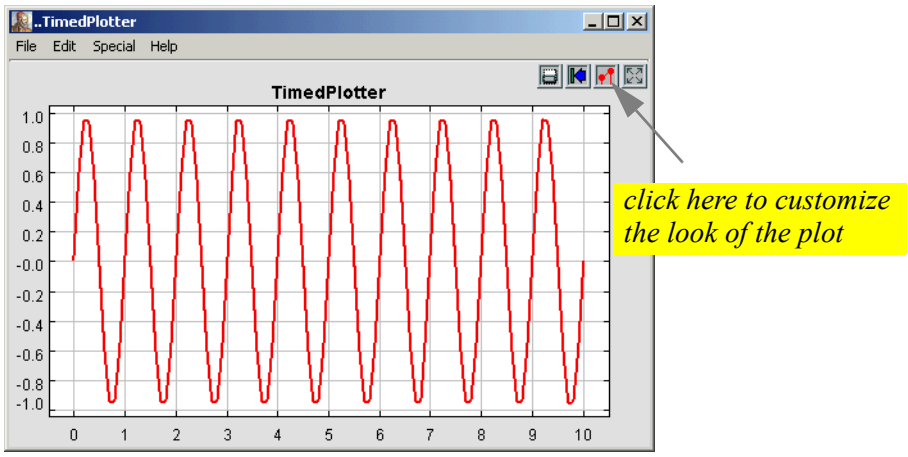

FIGURE 8. Execution of the sine wave example in figure 7, where all parameter values have default.values.

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:

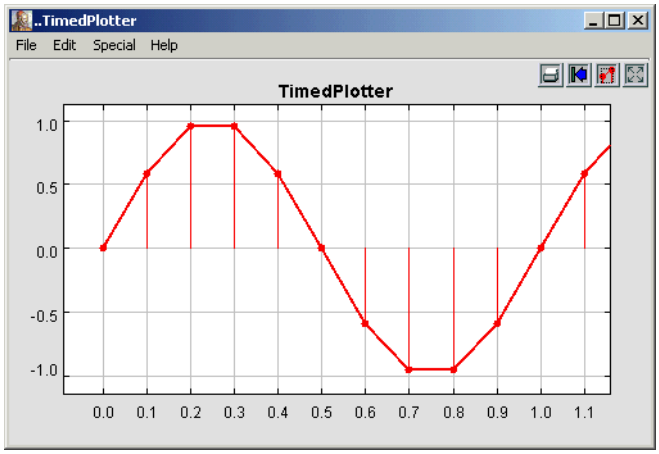

FIGURE 9. Zoomed version of the plot in figure 8, with "dots" and "stems" turned on.

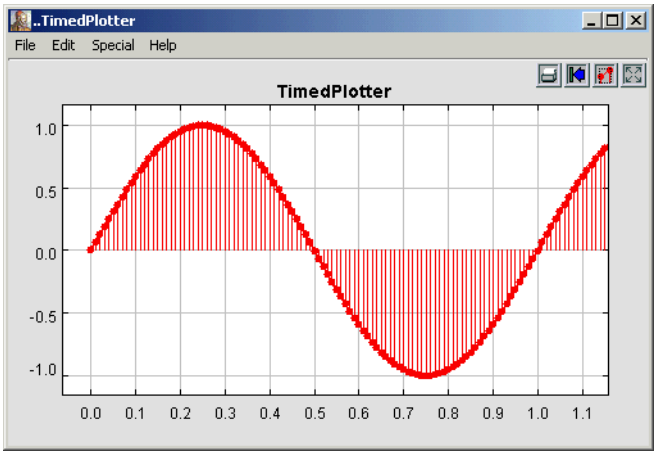

FIGURE 10. The result of running the model in figure 7 with the *initStepSize* parameter of the solver being 0.01.

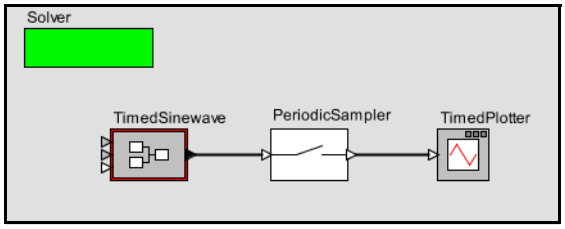

FIGURE 11. Another way to control the step size is to insert a sampler.

- 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* 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

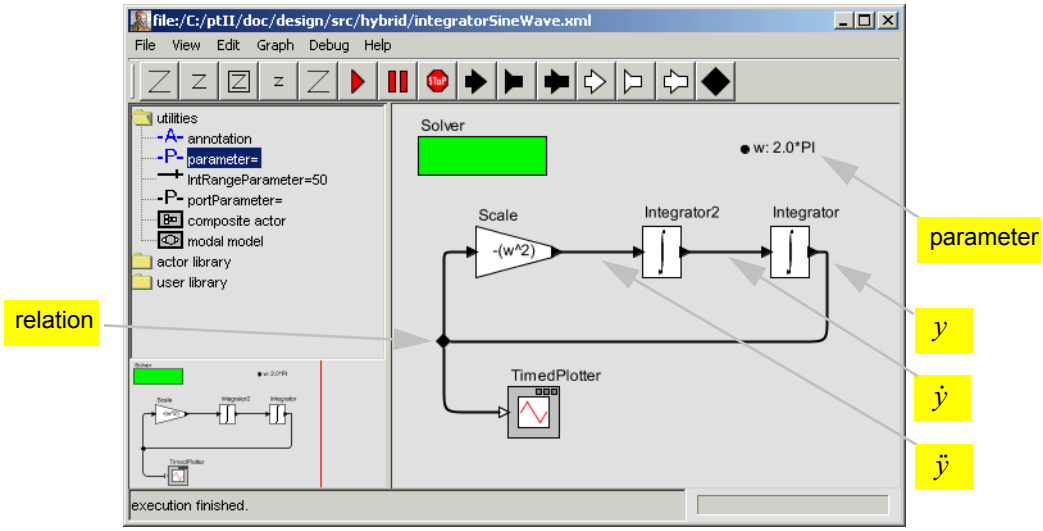

FIGURE 12. Model that generates a sine wave if the itegrators have a non-zero initial condition.

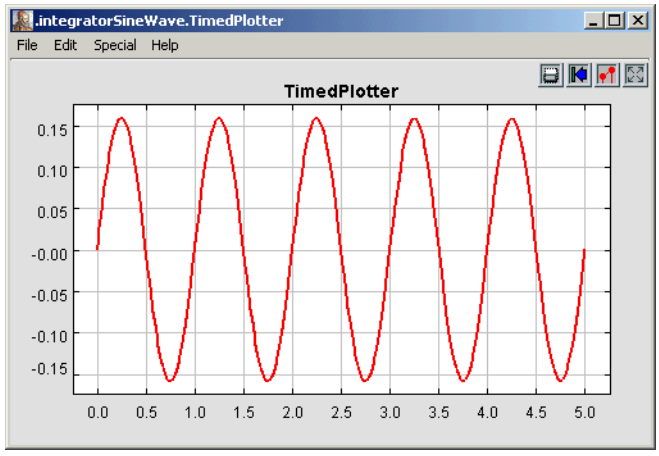

FIGURE 13. Result of running the model in figure 12 with the *initialState* of the left integrator set to 1.0.

```
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? HyVisual supports 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 *multiports*, 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.

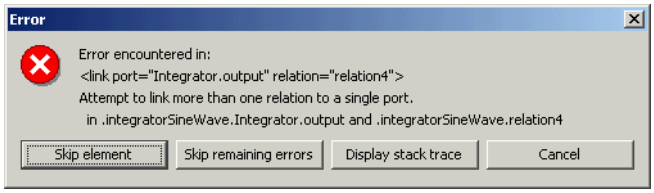

FIGURE 14. An exception that results from attempting to make a multi-way connection without a relation.

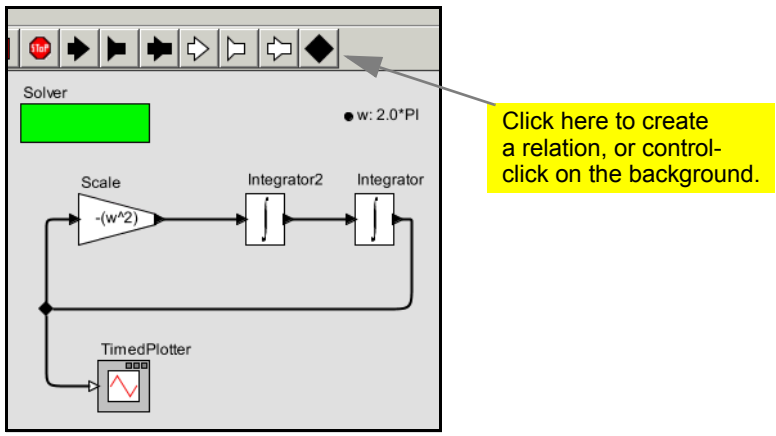

FIGURE 15. A relation can be used to broadcast an output from a single port.

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

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

2.2.4 Parameters

Figure 12 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 FIXME.

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.

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 12, for instance, the *factor* parameter of the *Scale* actor has the value "–(w^2)," which references the parameter *w*.

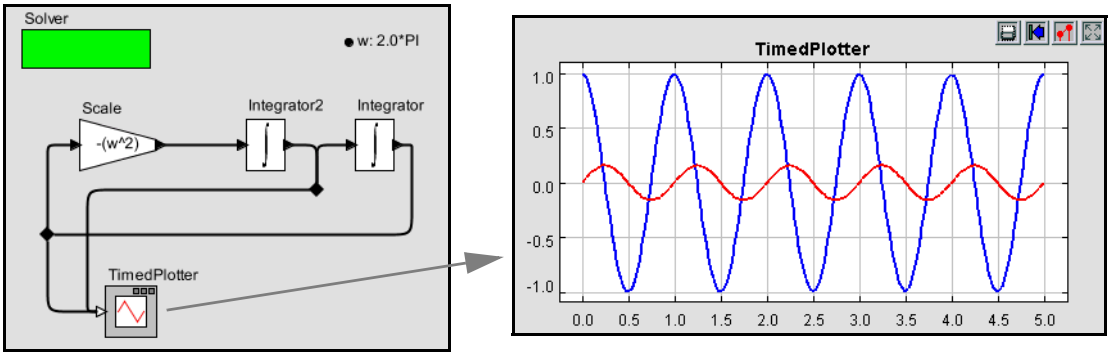

FIGURE 16. Multiple signals can be sent to a multiport, shown with the unfilled triangle on the *TimedPlotter*. In this case, two signals are plotted.

2.2.5 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 produces the tool tip “Set the plot format” and bring up the format control window.

FIXME: Restore the input and get an impulse response.

FIXME: Use a Laplace Transform actor.

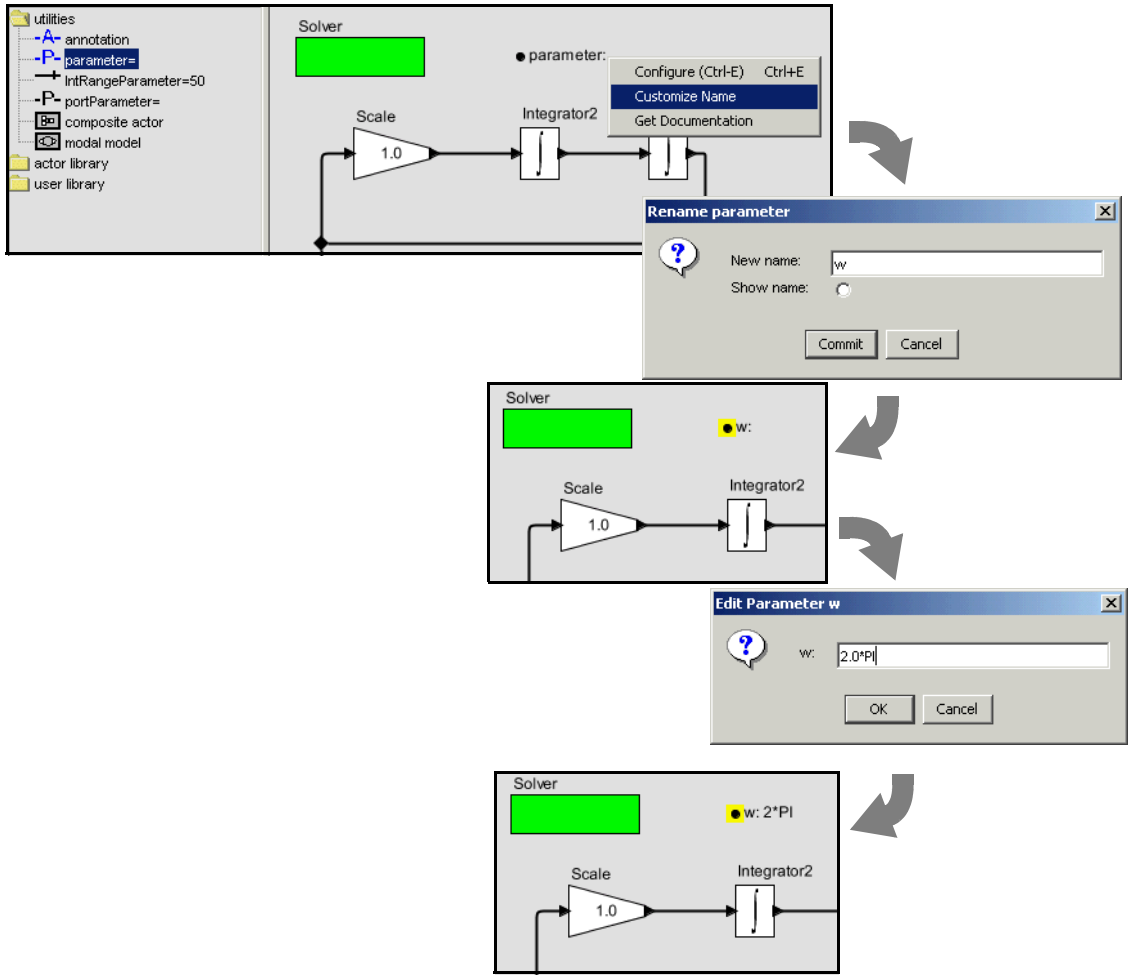

FIGURE 17. Adding a parameter to the channel model.

2.3 Tokens and Data Types

In the example of Figure 7, the *Const* actor creates a sequence of values on its output port. The values are encapsulated as *tokens*, and sent to the *Display* actor, which consumes them and displays them in the run window.

The tokens produced by the *Const* actor can have any value that can be expressed in the Ptolemy II *expression language*. We will say more about the expression language in chapter 3, ”Expressions”, but for now, try giving the value 1 (the integer with value one), or 1.0 (the floating-point number with value one), or {1.0} (An array containing a one), or {value=1, name="one"} (A record with two elements: an integer named “value” and a string named “name”), or even [1,0;0,1] (the two-by-two identity matrix). These are all expressions.

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

To explore data types a bit further, try creating the model in Figure 18. The *Ramp* actor is listed under *sources* and the *AddSubtract* actor is listed under *math*. Set the *value* parameter of the constant to be 0 and the *iterations* parameter of the director to 5. Running the model should result in 5 numbers between 0 and 4, as shown in the figure. These are the values produced by the *Ramp*, which are having the value of the *Const* actor subtracted from them. Experiment with changing the value of the *Const* actor and see how it changes the 5 numbers at the output.

Now for the real test: change the value of the *Const* actor back to "Hello World". When you execute the model, you should see an exception window, as shown in Figure 19. 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 integer value, which doesn’t make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

Exceptions can be a very useful debugging tool, particularly if you are developing your own components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of Figure 19. You should see the stack trace shown in Figure 20. This window displays the execution sequence that resulted in the exception. For example, the line

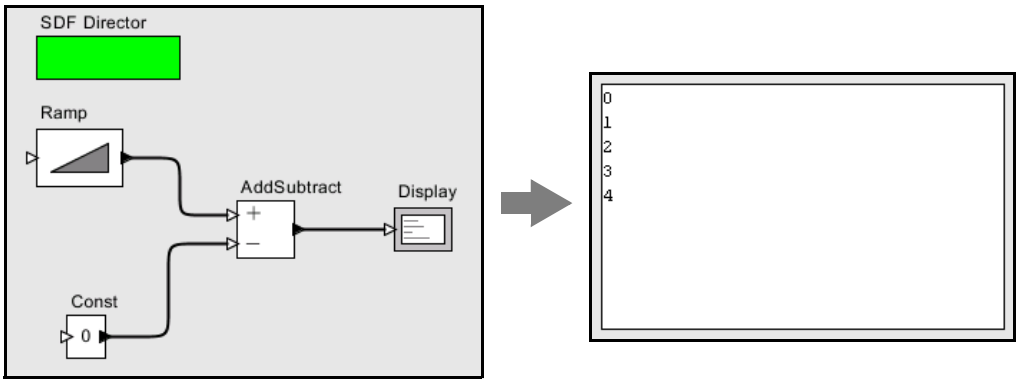

FIGURE 18. Another example, used to explore data types in Ptolemy II.

```
at ptolemy.data.IntToken.subtract(IntToken.java:547)
```

indicates that the exception occurred within the subtract() method of the class ptolemy.data.IntToken, at line 547 of the source file IntToken.java. Since Ptolemy II is distributed with source code (except in the Windows installer version and the Web Start Web Edition), this can be very useful information. For type errors, you probably do not need to see the stack trace, but if you have extended the system with your own Java code, or you encounter a subtle error that you do not understand, then looking at the stack trace can be very illuminating.

To find the file IntToken.java referred to above, find the Ptolemy II installation directory. If that directory is $PTII, then the location of this file is given by the full class name, but with the periods

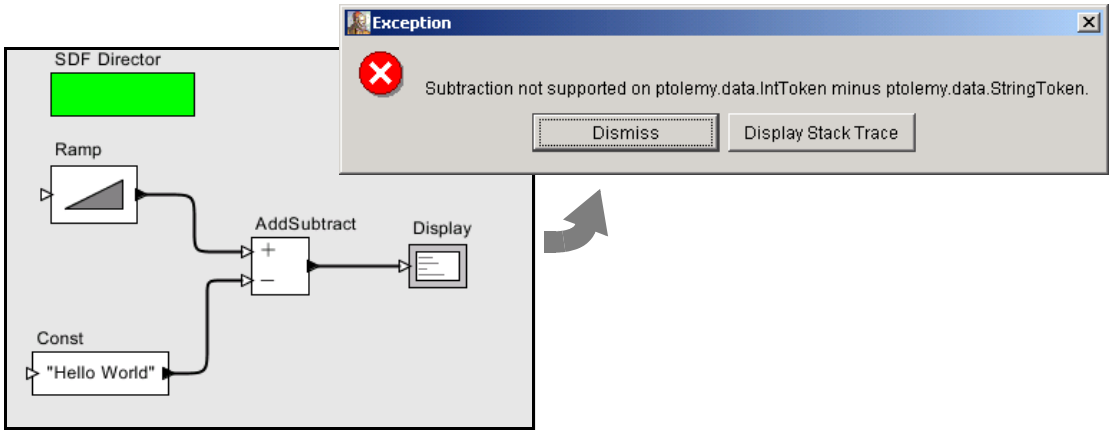

FIGURE 19. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.

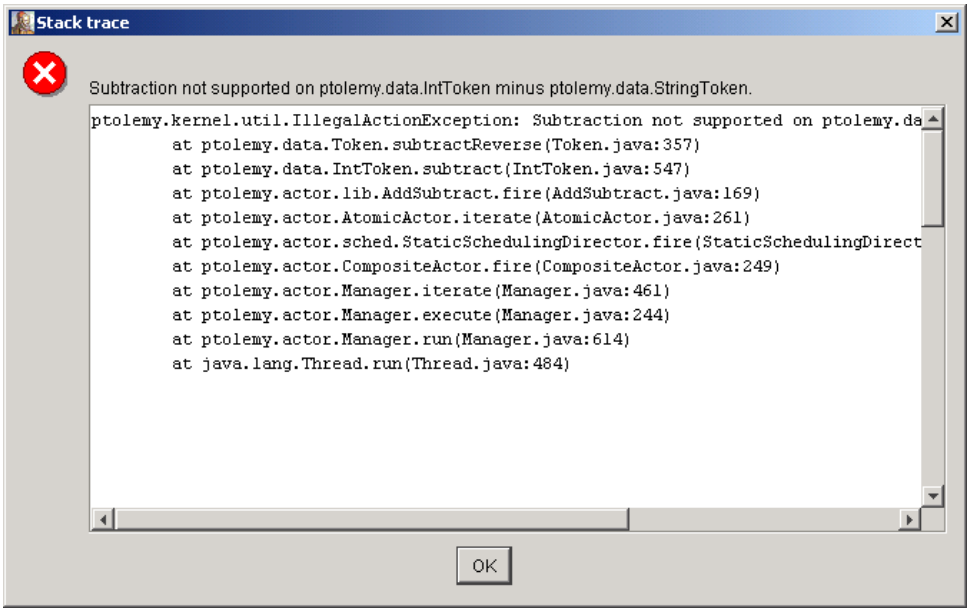

FIGURE 20. Stack trace for the exception shown in Figure 19.

replaced by slashes; in this case, it is at $PTII/ptolemy/data/IntToken.java (the slashes might be backslashes under Windows).

Let's try a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in Figure 21. You can do this by selecting the connection and deleting it (using the delete key), then adding a new connection, or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like "0HelloWorld", as shown in the figure.

There are two interesting things going on here. The first is that, as in Java, strings are added by concatenating them. The second is that the integers from the *Ramp* are converted to strings and concatenated with the string "Hello World". All the connections to a multiport must have the same type. In this case, the multiport has a sequence of integers coming in (from the *Ramp*) and a sequence of strings (from the *Const*).

Ptolemy II automatically converts the integers to strings when integers are provided to an actor that requires strings. But in this case, why does the *AddSubtract* actor require strings? Because it would not work to require integers; the string "Hello World" would have to be converted to an integer. As a rough guideline, Ptolemy II 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. The details are explained in the Data chapter, but many users will not need to understand the full sophistication of the system. You should find that most of the time it will just do what you expect.

To further explore data types, try modifying the *Ramp* so that its parameters have different types. For example, try making *init* and *step* strings.

2.4 Hierarchy

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

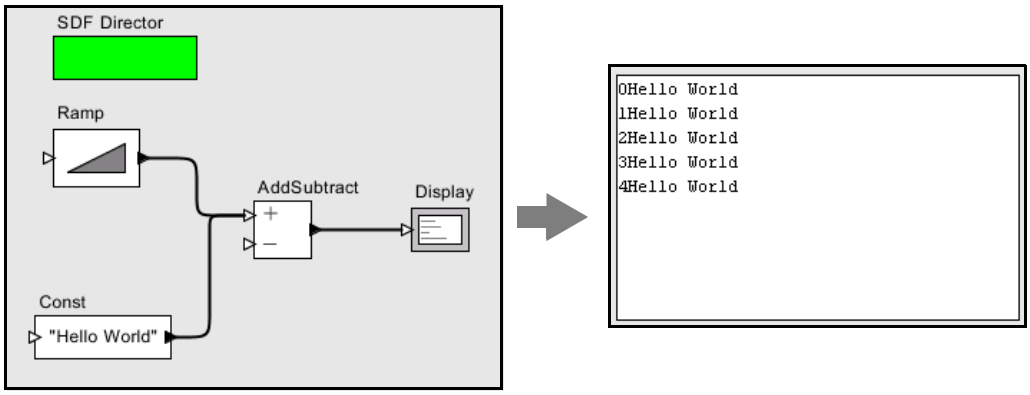

FIGURE 21. Addition of a string to an integer.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in Figure 22. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in Figure 23. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

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

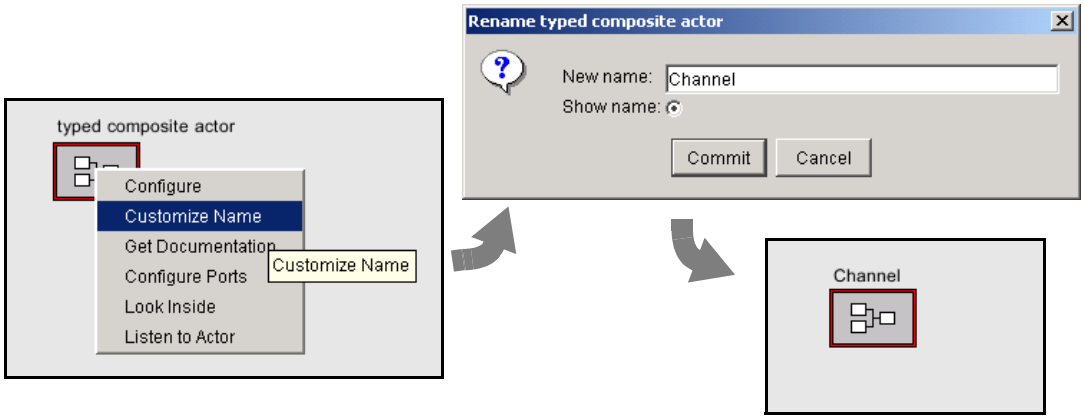

FIGURE 22. Changing the name of an actor.

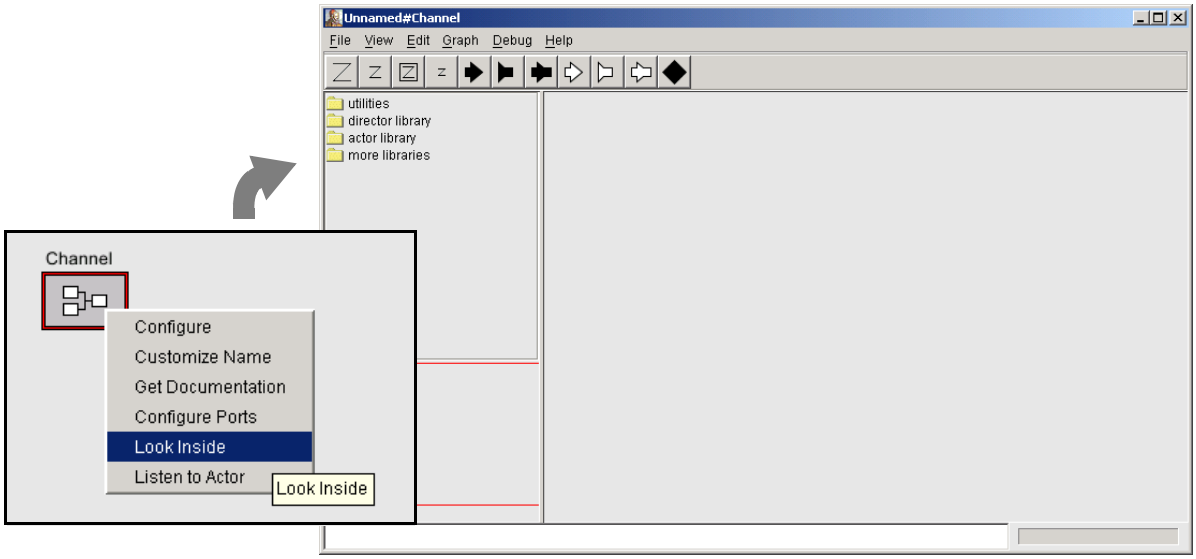

FIGURE 23. Looking inside a composite actor.

The buttons are summarized in Figure 24. 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 25, 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 26[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close this

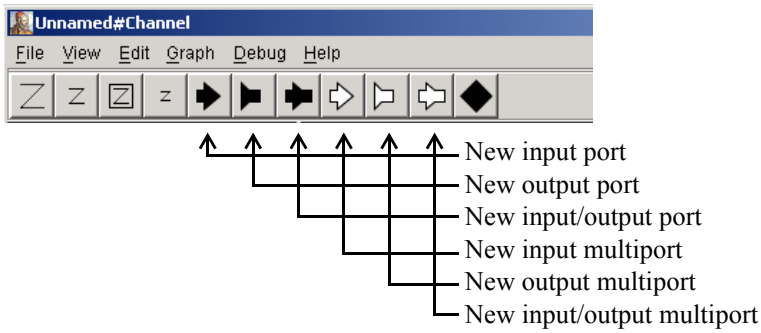

FIGURE 24. Summary of toolbar buttons for creating new ports.

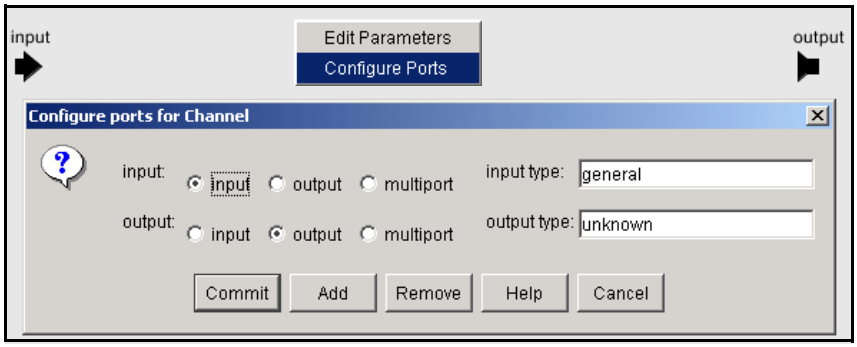

FIGURE 25. Right clicking on the background brings up a dialog that can be used to configure ports.

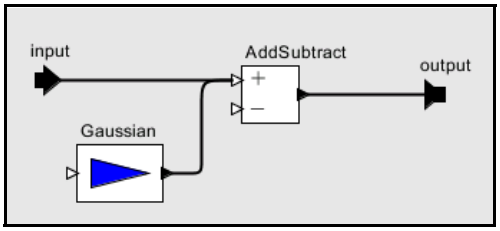

FIGURE 26. A simple channel model defined as a composite actor.

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

editor and return to the previous one, you should be able to easily create the model shown in Figure 27. The *Sinewave* actor is listed under *sources*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline (try looking inside). If you execute this model (you will probably want to set the iterations to something reasonable, like 100), you should see something like Figure 28.

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 25 that there is a position in the dialog box that configures ports for specifying the type. Thus, to specify that a port has type *boolean*, you could enter *boolean* into the dialog box. There are other commonly used types: *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. Let's take a more complicated case. How would you specify that the type of a port is a double matrix? Easy:

```
[double]
```

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

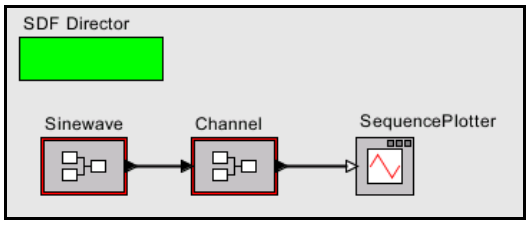

FIGURE 27. A simple signal processing example that adds noise to a sinusoidal signal.

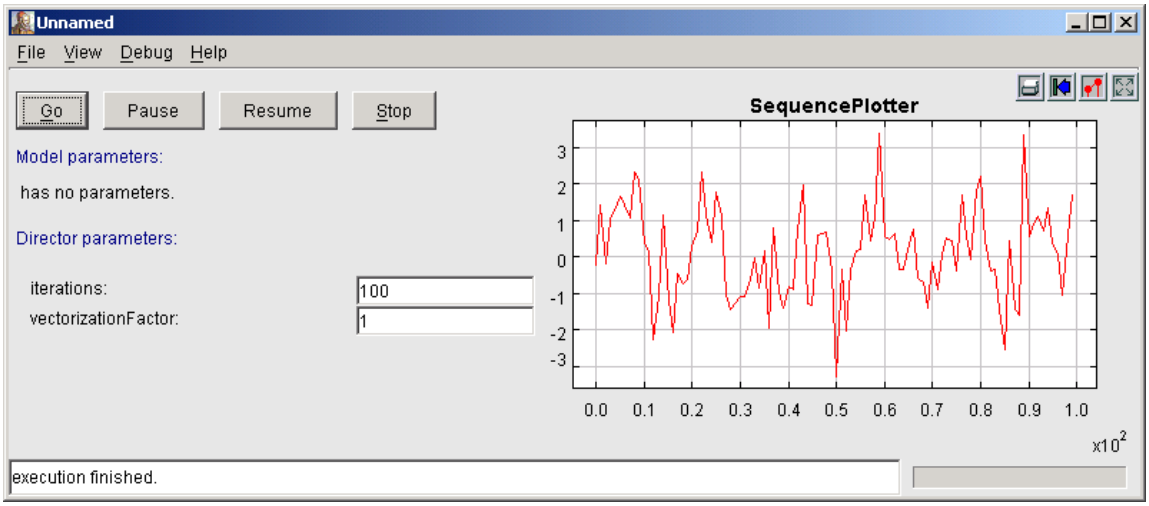

FIGURE 28. The output of the simple signal processing model in Figure 27.

plex numbers as

```
{complex}
```

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

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

2.5 Navigating Larger Models

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

In addition, it is possible to pan over a model. Consider the window shown in Figure 30. Here, we have zoomed in so that icons are larger than the default. The *pan window* at the lower left shows the entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the “Zoom fit” button in the toolbar results in the editor area showing the entire model, just as the pan window does.

2.6 Domains

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

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

But there are other models of computation available in Ptolemy II. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most

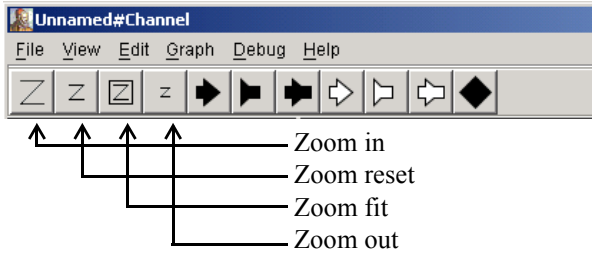

FIGURE 29. Summary of toolbar buttons for zooming and fitting.

actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are using. Here, we give a very brief introduction to some of the domains. But we begin first by explaining some of the subtleties in SDF.

2.6.1 SDF and Multirate Systems

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

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

One such actor is a spectral estimation actor. Figure 31 shows a system that computes the spectrum of the same noisy sine wave that we constructed in Figure 27. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 32 shows the output of the model with *order* set to 8 and the number of *iterations* set to 1. **Note that there are 256 output**

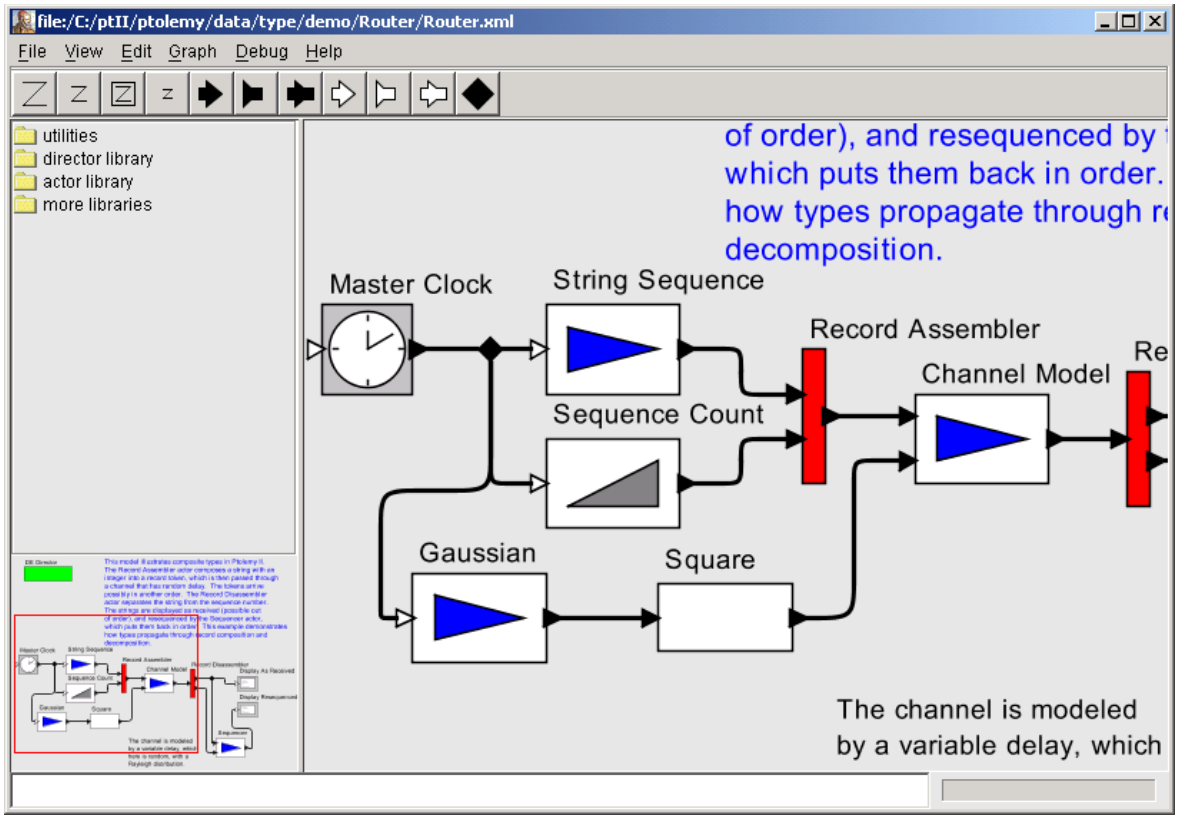

FIGURE 30. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

samples output from the *Spectrum* actor. This is because the *Spectrum* actor requires 2^8, or 256 input samples to fire, and produces 2^8, or 256 output samples when it fires. Thus, one iteration of the model produces 256 samples. The *Spectrum* actor makes this a *multirate* model, because the firing rates of the actors are not all identical.

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

A second subtlety with SDF models is that if there is a feedback loop, as in Figure 33, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *flow control* library). Without this actor, the loop will deadlock. The *SampleDelay* actor produces initial tokens on its output, before the model begins firing. The initial tokens produced are given by a the *initialOutputs* parameter, which specifies an array of tokens. These initial tokens enable downstream actors and break the circular dependencies that would result otherwise from a feedback loop.

A final issue to consider with the SDF domain is time. Notice that in all the examples above we have suggested using the *SequencePlotter* actor, not the *TimedPlotter* actor, which is in the same *sinks* library. This is because the SDF domain does not include in its semantics a notion of time. Time does

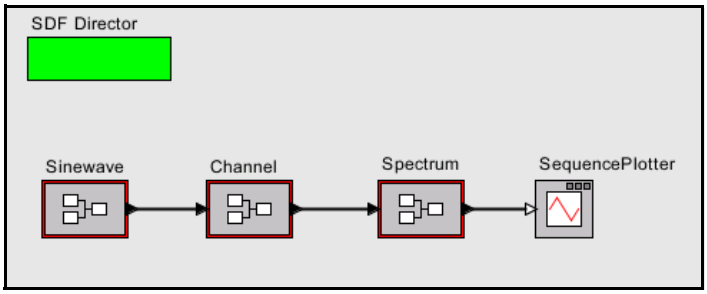

FIGURE 31. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave, Channel,* and *SequencePlotter*, and one firing of *Spectrum*.

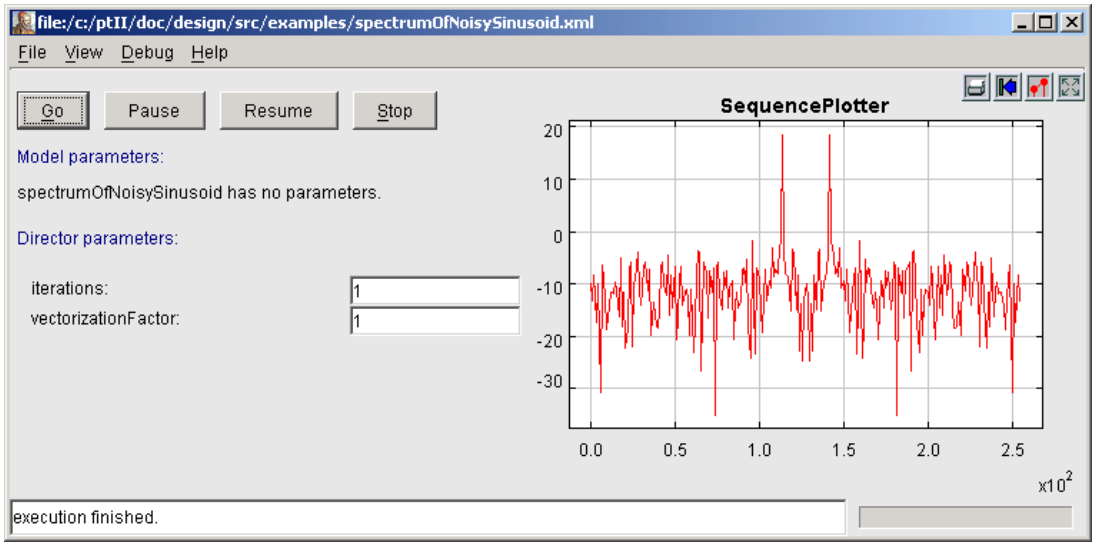

FIGURE 32. A single iteration of the SDF model in Figure 31 produces 256 output tokens.

not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.6.2 Discrete-Event Systems

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

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

Consider the DE model shown in Figure 34. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead, these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the time stamp of the input). The *WallClockTime* actor an event with the same time stamp as the input, but whose value is the current real time, in seconds since initialization of the model.

The plot in Figure 34 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

A significant subtlety in using the DE domain is in how simultaneous events are handled. Simultaneous events are simply events with the same time stamp. We have stated that events are processed in

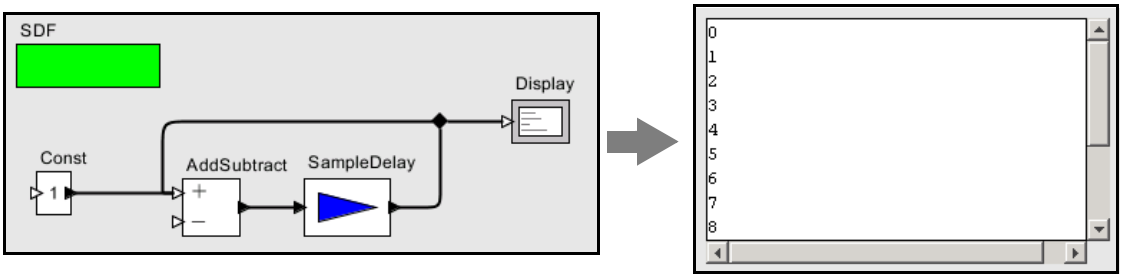

FIGURE 33. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

chronological order, but if two events have the same time stamp, then there is some ambiguity. Which one should be processed first? If the two events are on the same signal, then the answer is simple: process first the one that was produced first. However, if the two events are on different signals, then the answer is not so clear.

Consider the model shown in Figure 35, which produces a histogram of the interarrival times of events from the *PoissonClock* actor. In this model, we calculate the difference between the current event time and the previous event time, resulting in the plot that is shown in the figure. The *Previous* actor is a *zero-delay* actor, meaning that it produces an output with the same time stamp as the input (except on the first firing, where in this case it produces no output). Thus, when the *PoissonClock* actor produces an output, there will be two simultaneous events, one at the input to the *plus* port of the *AddSubtract* actor, and one at the input of the *Previous* actor. Should the director fire the *AddSubtract* actor or the *Previous* actor? Either seems OK if it is to respect chronological order, but it seems intuitive that the *Previous* actor should be fired first.

It is helpful to know how the *AddSubtract* actor works. When it fires, it adds all available tokens on the *plus* port, and subtracts all available tokens on the *minus* port. If the *AddSubtract* actor fires before the *Previous* actor, then the only available token will be the one on the *plus* port, and the expected subtraction will not occur. Intuitively, we would expect the director to invoke the *Previous* actor before the *AddSubtract* actor so that the subtraction occurs.

How does the director deliver on the intuition that the *Previous* actor should be fired first? Before executing the model, the DE director constructs a *topological sort* of the model. A topological sort is simply a list of the actors in data-precedence order. For the model in Figure 35, there is only one allowable topological sort:

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

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

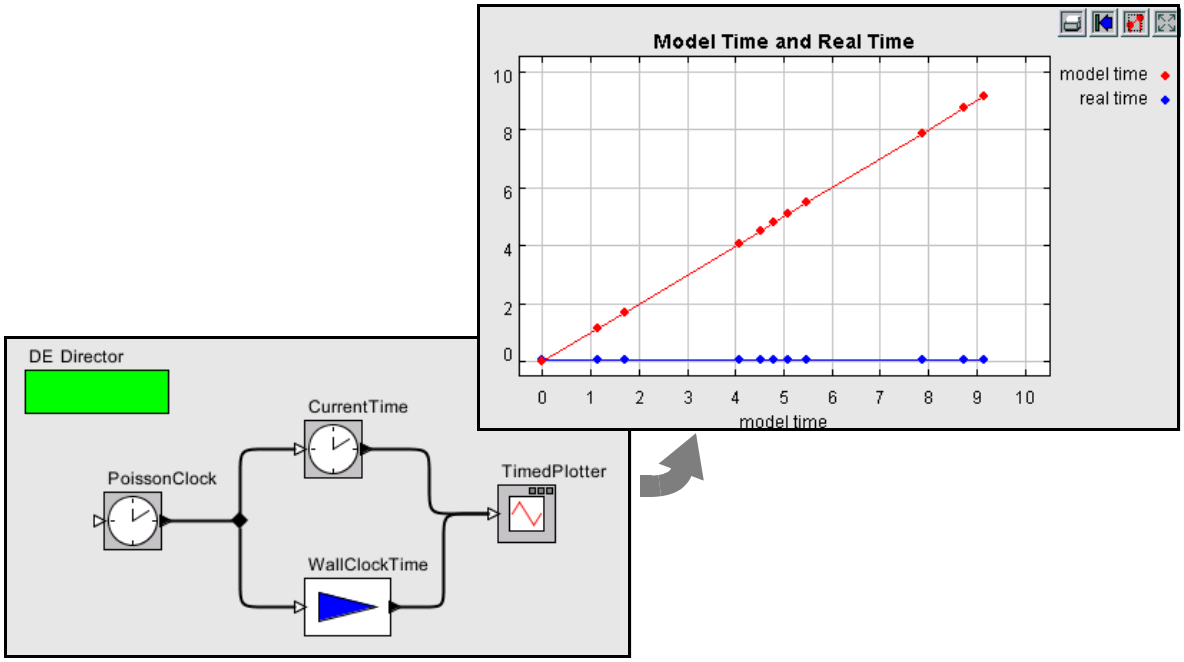

FIGURE 34. Model time vs. real time (wall clock time).

fires *Previous* first.

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

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *domain specific* library under *discrete-event* (this library is shown on the left in Figure 36). Consider for example the model shown in Figure 36. That model has a *Clock* actor, which is set to produce events every 1.0 time units. Those events trigger the *Ramp* actor, which produces outputs that start at 0 and increase by 1 on each firing. In this model, the output of the *Ramp* goes into an *AddSubtract* actor, which subtracts from the *Ramp* output its own prior output delayed by one time unit. The result is shown in the plot in the figure.

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

2.6.3 FSM and Modal Models

The finite-state machine domain (FSM) in Ptolemy II is a relatively less mature domain (but mature enough to be useful) with semantics very different from the domains covered so far. An FSM model looks different in Vergil. An example is shown in Figure 37. Notice that the component library on the left and the toolbar at the top are different for this model. We will explain how to construct this model.

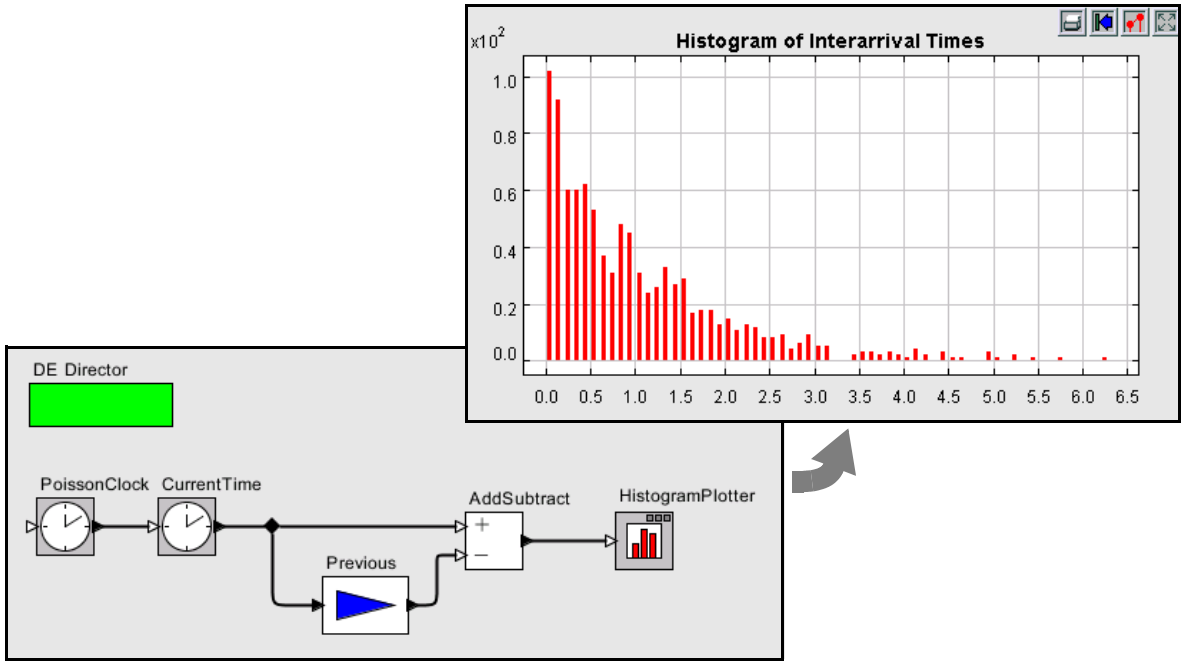

FIGURE 35. Histogram of interarrival times, illustrating handling of simultaneous events.

First, the FSM domain is almost always used in combination with other domains in Ptolemy II to create *modal models*. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a *state* in a finite-state machine. The circles in Figure 37 are states, and the arcs between circles are *transitions* between states.

A modal model is typically a component in a larger model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will probably need to add ports. Figure 38 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. It represents a bouncing ball. Three output ports have been added, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, and then look inside, you will get an FSM editor like that in Figure 37, except that it will be almost blank. The only items in it will be the ports you have added, and possibly some text that you can delete once you no longer need it. You may want to move the ports to reasonable locations.

To create a finite-state machine like that in Figure 37, drag in states (white circles). You can rename these states by right clicking on them and selecting “Customize Name”. Choose names that are pertinent to your application. In Figure 37, there is an *init* state for initialization, a *free* state for when

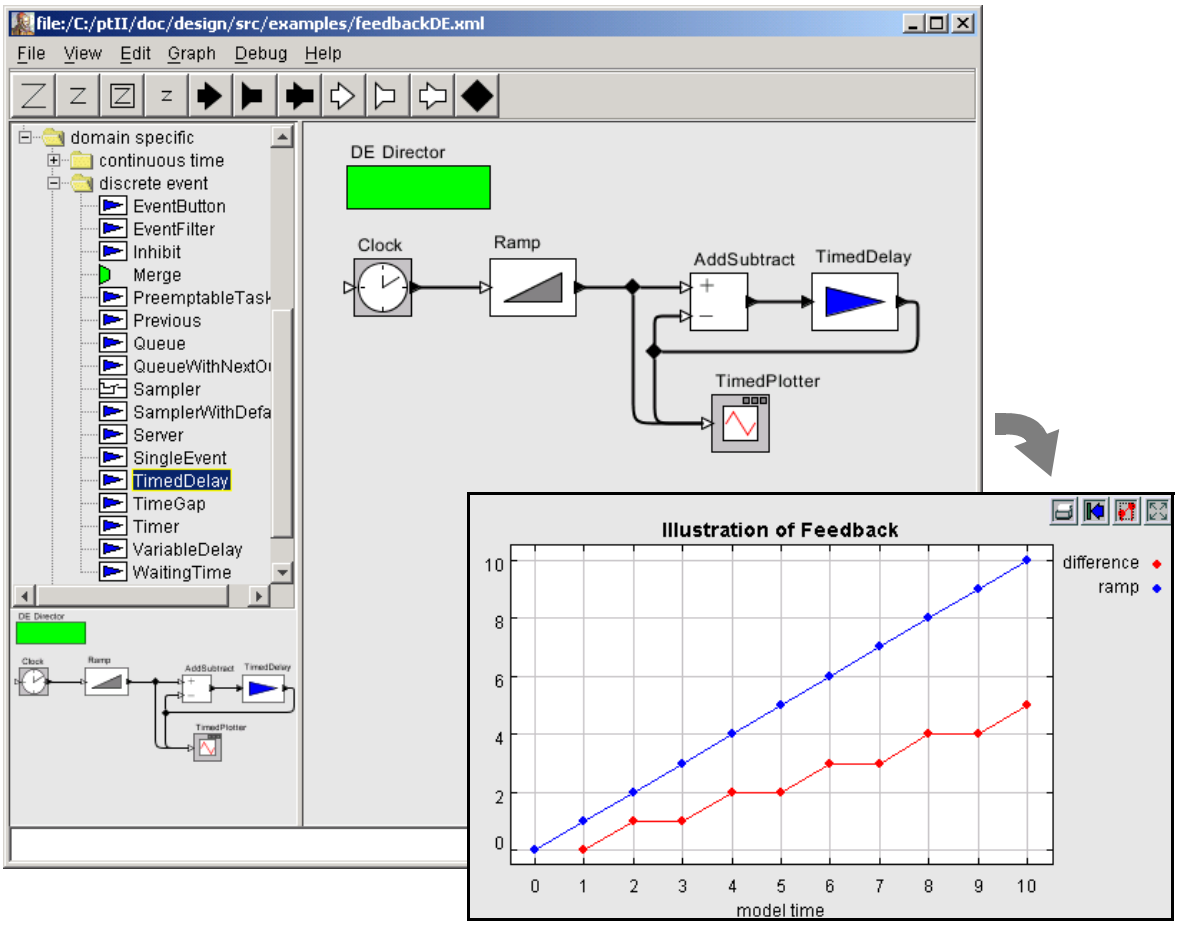

FIGURE 36. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name. In this example, the initial state is *init*.

To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or right clicking and selecting "Configure") allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in Figure 39. In that dialog, we see the following:

- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.
- The output actions are empty, meaning that when this transition is taken, no output is specified.

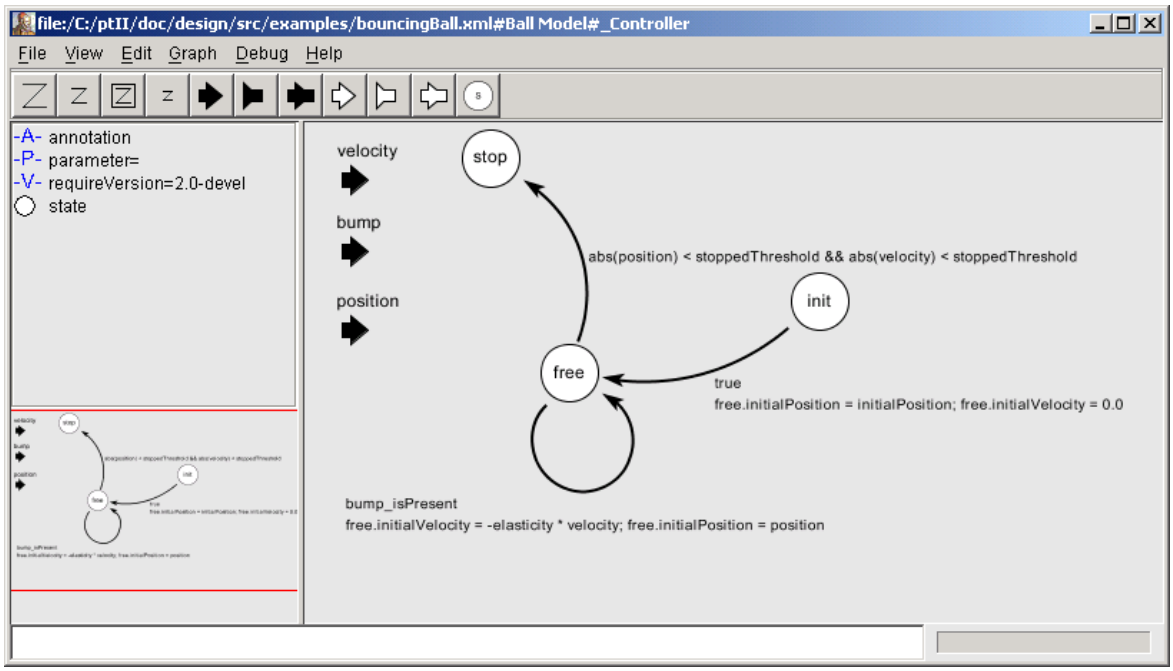

FIGURE 37. Finite-state machine model used in the bouncing ball example.

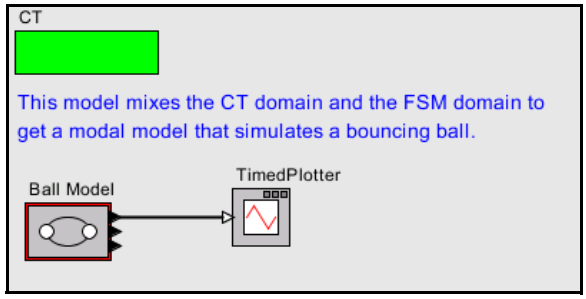

FIGURE 38. Top-level of the bouncing ball example. The *Ball Model* actor is an instance of *modal model* from the *utilities* library. It has been renamed.

This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.

- The set actions contain the following statements:

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

The "free" in these expressions refers to the mode refinement in the "free" state. Thus, "free.initialPosition" is a parameter of that mode refinement. Here, its value is assigned to the value of the parameter "initialPosition". The parameter "free.initialVelocity" is set to zero.

- The *reset* parameter is set to *true*, meaning that the destination mode should be initialized when the transition is taken.
- The *preemptive* parameter is set to *false*. In this case, it makes no difference, since the *init* state has no refinement. Normally, if a transition out of a state is enabled and *preemptive* is *true*, then the transition will be taken without first firing the refinement.

Refinements. Both states and transitions can have *refinements.* To create a refinement, right click on the state or transition, and select "Add Refinement". You will see a dialog like that in Figure 43. You can specify the class name for the refinement, but for now, it is best to accept the default. Once you have created a refinement, you can look inside a state or transition. For the bouncing ball example, the refinement of the *free* state is shown in Figure 41. This model exhibits certain key properties of refinements:

- Refinements must contain directors. In this case, the CTEmbeddedDirector is used. When a continuous-time model is used inside a mode, this director must be used instead of the default

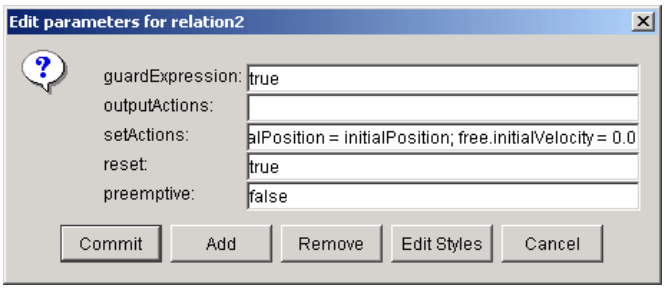

FIGURE 39. Transition dialog for the transition from *init* to *free* in Figure 37.

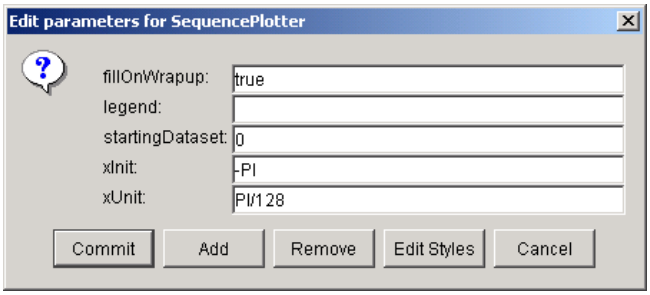

FIGURE 40. Parameters of the SequencePlotter actor.

CTDirector (see the CT chapter for details).

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

As you can see from Figure 37, when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop*, which has no refinement. This results in the model producing no further output. The result of an execution is shown in Figure 42. Notice that the ball bounces until it stops, after which there are no further outputs.

This model illustrates an interesting property of the CT domain. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each

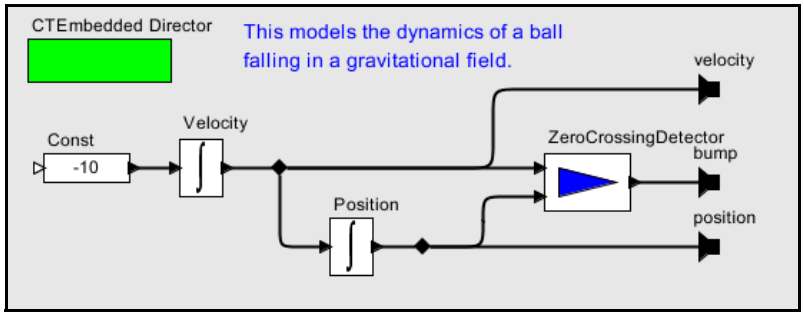

FIGURE 41. Refinement of the *free* state of the modal model in Figure 37.

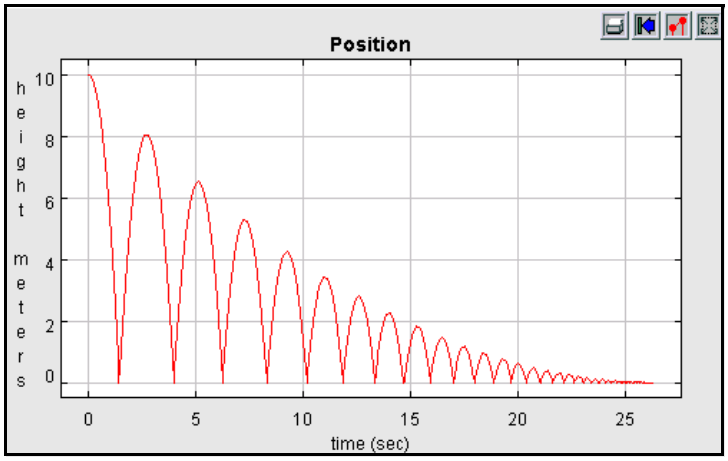

FIGURE 42. Result of execution of the bouncing ball model.

bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. Try removing the *stop* state from the FSM, and re-run the model. What happens? Why?

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

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

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

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

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

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

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

   In the postfire phase:

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

2.7 Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in Figure 32 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces 256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

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

The *SequencePlotter* actor has some pertinent parameters, shown in Figure 40. The *xInit* parameter specifies the value to use on the horizontal axis for the first token. The *xUnit* parameter specifies the value to increment this by for each subsequent token. Setting these to "-PI" and "PI/128" respectively results in the plot shown in Figure 44.

This plot is better, but still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in Figure 45, filled in with values that

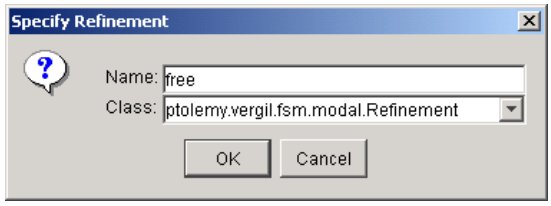

FIGURE 43. Dialog for creating a refinement of a state.

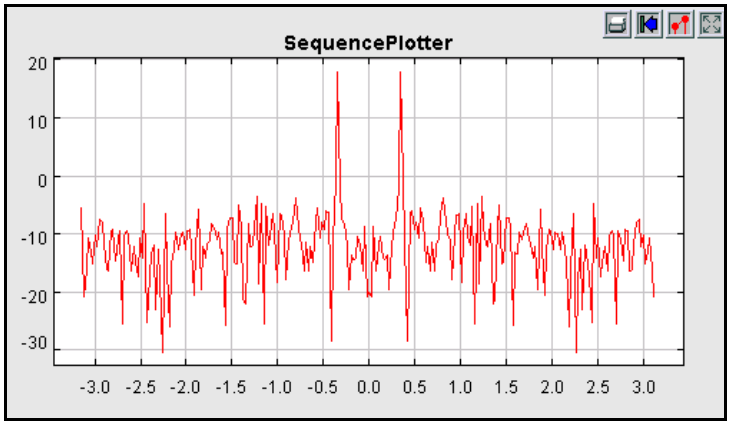

FIGURE 44. Better labeled plot, where the horizontal axis now properly represents the frequency values.

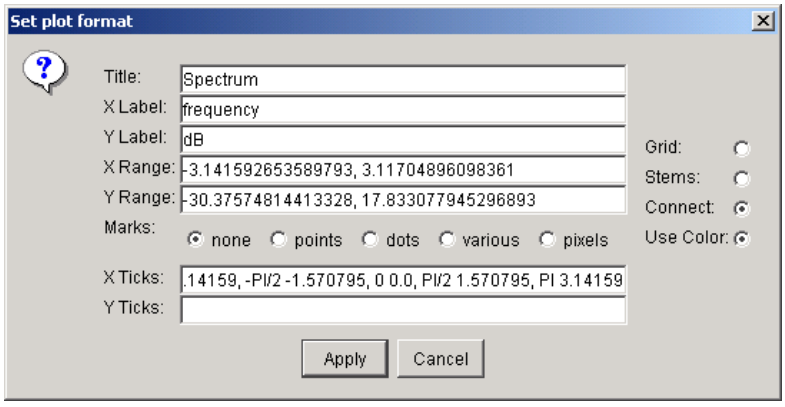

FIGURE 45. Format control window for a plot.

result in the plot shown in Figure 46. 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”
- Individual tokens can be shown by clicking on “dots”
- Connecting lines can be eliminated by deselecting “connect”
- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

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

  The syntax in general is:

  label value, label value, ...

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

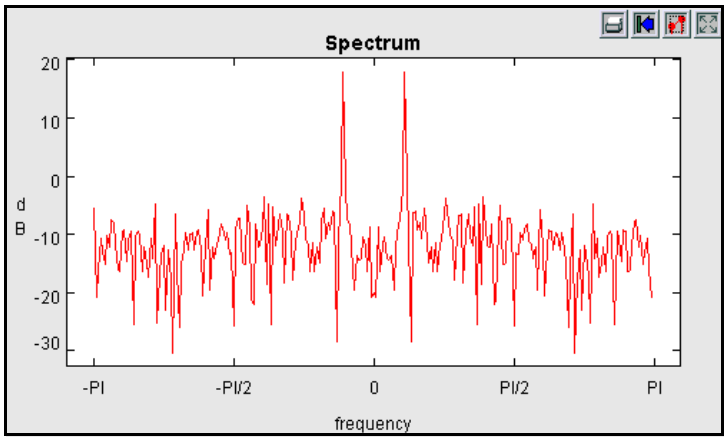

FIGURE 46. Still better labeled plot.