Chapter 2 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) ," Technical Memorandum UCB/ERL M04/27, July 29, 2004, University of California, Berkeley, CA USA 94720.

2

Using Vergil

Authors: *Edward A. Lee*
Steve Neuendorffer

2.1 Introduction

There are many ways to use Ptolemy II. It can be used as a framework for assembling software components, as a modeling and simulation tool, as a block-diagram editor, as a system-level rapid prototyping application, as a toolkit supporting research in component-based design, or as a toolkit for building Java applications. This chapter introduces its use as a modeling and simulation tool.

In this chapter, we describe how to graphically construct models using Vergil, a graphical user interface (GUI) for Ptolemy II. Figure 2.1 shows a simple Ptolemy II model in Vergil, showing the graph editor, one of several editors available in Vergil. Keep in mind as you read this document that graphical entry of models is only one of several possible entry mechanisms available in Ptolemy II. For example, you can define models in Java, as shown in figure 1.5, or in XML, as shown in figure 1.3 of the previous chapter. Moreover, only some of the execution engines (called *domains*) are described here. A major emphasis of Ptolemy II is to provide a framework for the construction of modeling and design tools, so the specific modeling and design tools described here should be viewed as representative of our efforts.

2.2 Quick Start

This section shows how to start Vergil, how to execute and explore pre-built models, and how to construct your own models.

2.2.1 Starting Vergil

First start Vergil. From the command line, enter “vergil”, or select Ptolemy II and Vergil in the Start menu[1], or click on a Web Start link on a web page supporting the web edition. You should see an

initial welcome window that looks like the one in figure 2.2. Feel free to explore the links in this window. The “Quick tour” link takes you to the page shown in figure 2.3.

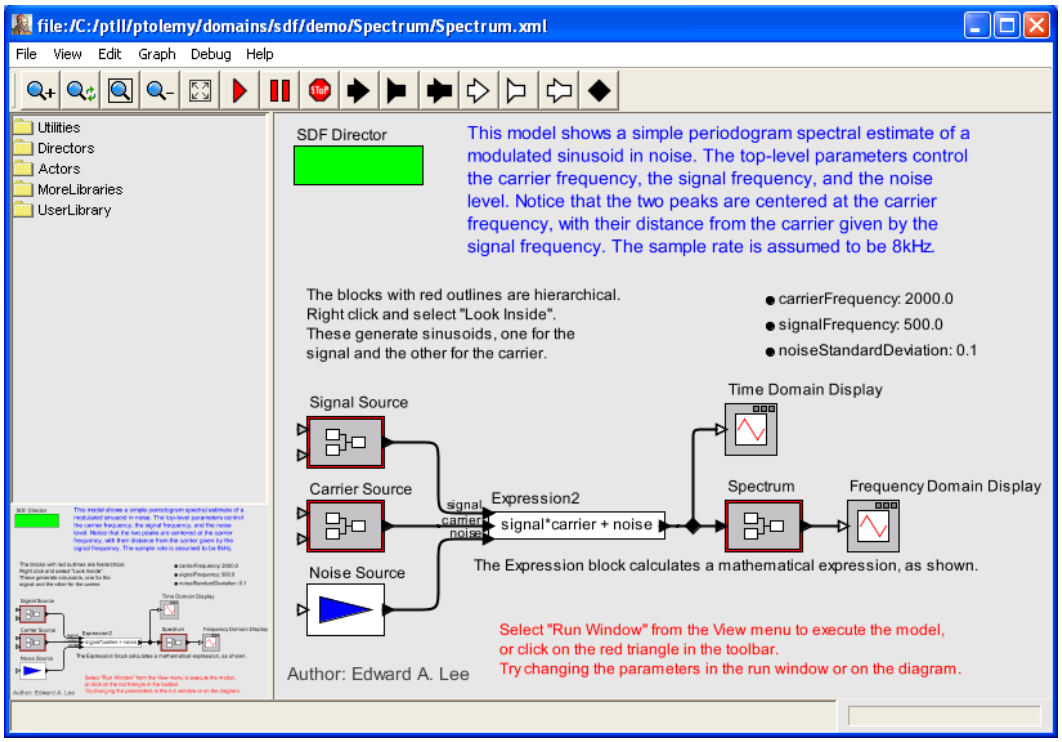

FIGURE 2.1. Example of a Vergil window.

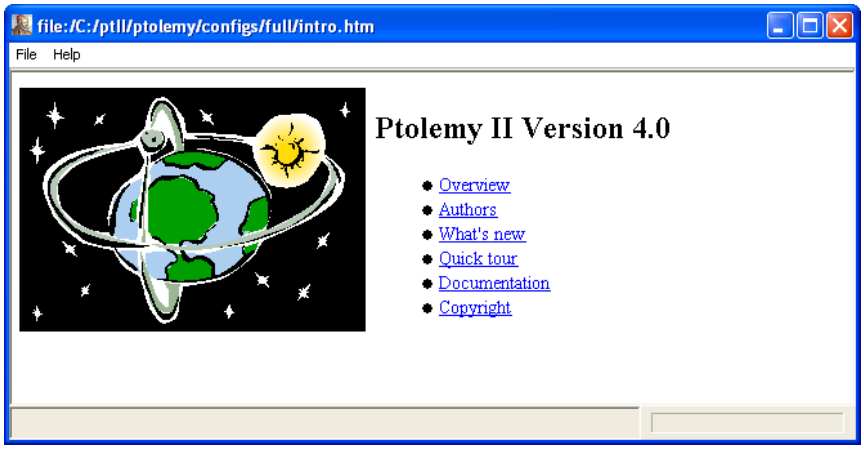

FIGURE 2.2. Initial welcome window.

1. Depending on your installation, you could have several versions of Vergil available in the Start menu. This document assumes you select “Vergil - Full.” There are separate tutorial documents for “Vergil - HyVisual” (which is specialized for modeling hybrid systems) and “Vergil - VisualSense” (which is specialized for modeling wireless and sensor network systems).

2.2.2 Executing a Pre-Built Model: A Signal Processing Example

The very first example on the quick tour page is the model shown in figure 2.1. It creates a sinusoidal signal, multiplies it by a sinusoidal carrier, adds noise, and then estimates the power spectrum. You can execute this model in either of two ways. First, you can select Run Window in the View menu, and then click on Go. The result is shown in figure 2.4. The upper plot shows the spectrum of the time-domain signal shown in the lower plot. Note the four peaks, which indicate the modulated sinusoid. In the run window you can adjust the frequencies of the signal and the carrier as well as the amount of noise. These can also be adjusted in the block diagram in figure 2.1 by double clicking on the bulleted parameters near the upper right of the window.

The second alternative for running the model is to click on the run button in the toolbar, which is indicated by a red triangle pointing to the right. If you use this alternative, then the two signal plots are displayed in their own windows.

You can study the way the model is constructed in figure 2.1. Note the Expression actor in the middle, whose icon indicates the expression being calculated: "`signal*carrier + noise`". The identifiers in this expression, `signal`, `carrier`, and `noise` refer to the input ports by name. The names of

FIGURE 2.3. The quick-tour page.

these ports are shown in the diagram. The Expression actor is a very flexible actor in the Ptolemy II actor library. It can have any number of input ports, with arbitrary names, and uses a rich and expressive expression language to specify the value of the output as a function of the inputs (and parameters of the containing model, if desired).

Three of the actors in figure 2.1 are *composite actors*, which means that their implementation is itself given as a block diagram. Composite actors are indicated visually by the red outline. You can look inside to reveal the implementation, as shown in figure 2.5, which shows the implementation of the Signal Source in figure 2.1. It is evident from the block diagram how a sinusoidal signal is generated.

2.2.3 Executing a Pre-Built Model: A Continuous-Time Example

A key principle of the Ptolemy II system is that the model of computation that defines the meaning of a block diagram is not built-in, but is rather specified by the *director* component that is included in the model. The box labeled "SDF Director" in figures 2.1 and 2.5 specifies that these block diagrams have *synchronous dataflow* semantics, which is explained further below. The second example in the quick tour of figure 2.3, by contrast, has continuous-time semantics (the one labeled "Continuous-Time Modeling"). The example is the well-known *Lorenz attractor*, a non-linear feedback system that exhibits chaotic behavior.

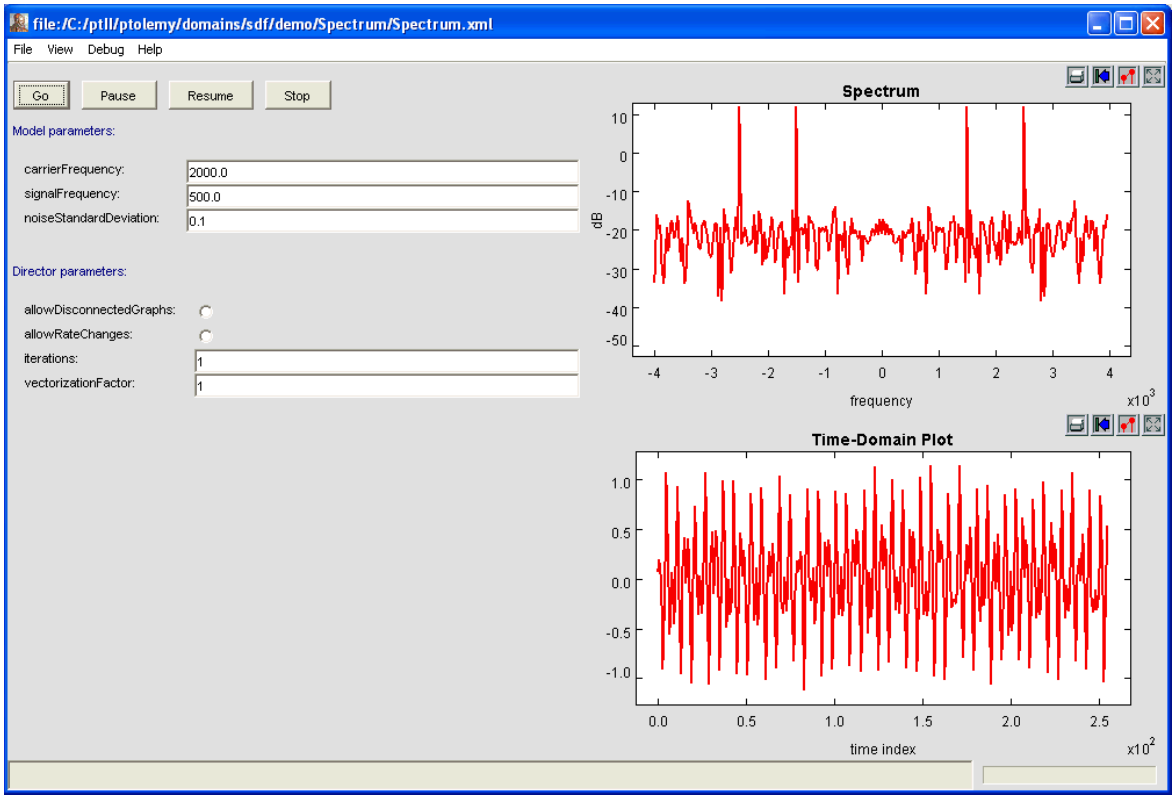

FIGURE 2.4. The run window for the model shown in figure 2.1.

The Lorenz attractor model, shown in figure 2.6, is a block diagram representation of a set of non-linear 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,

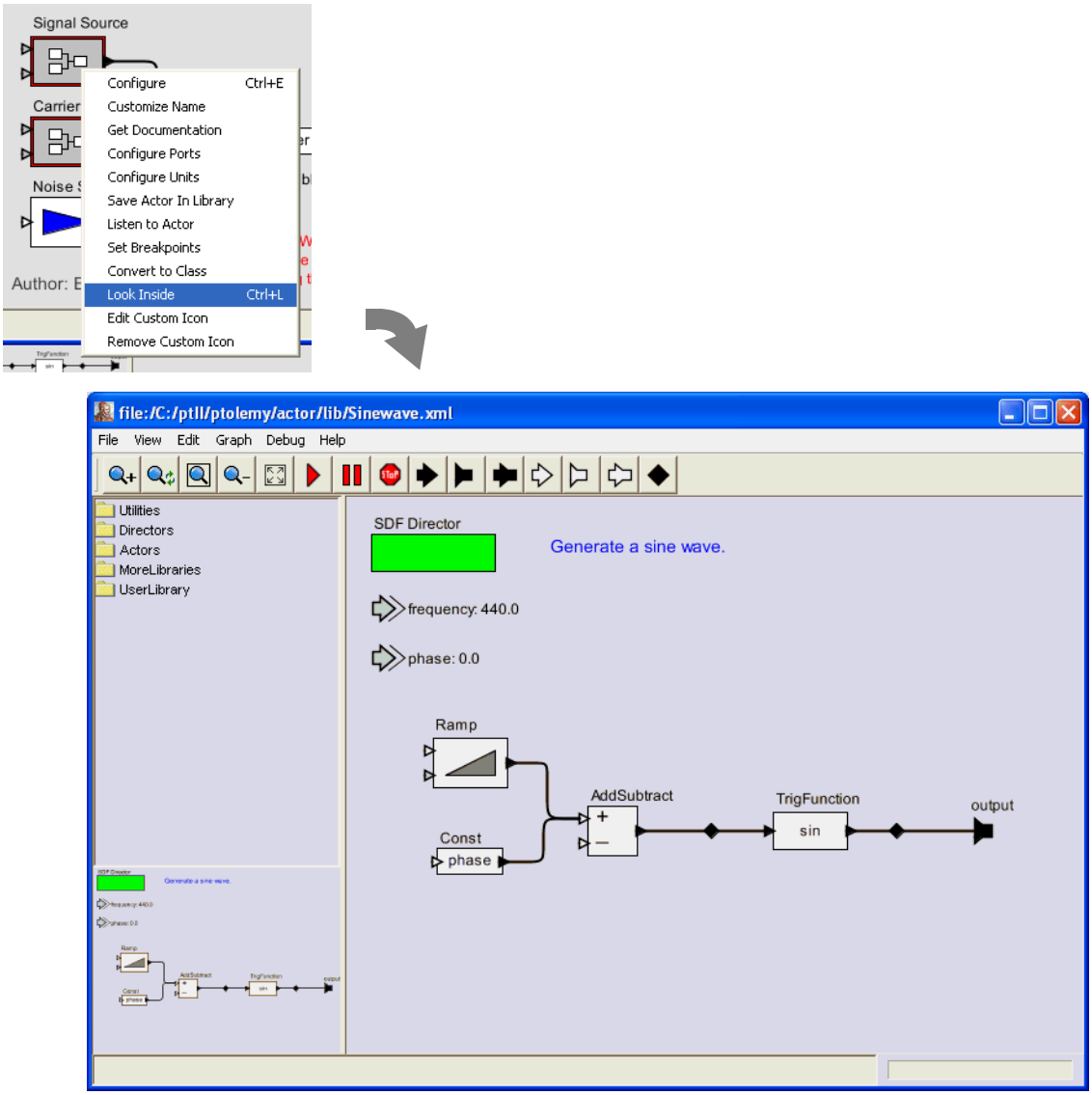

FIGURE 2.5. Look inside composite actors to reveal their implementation.

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

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

Let the output of the top integrator in figure 2.6 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 2.6 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.

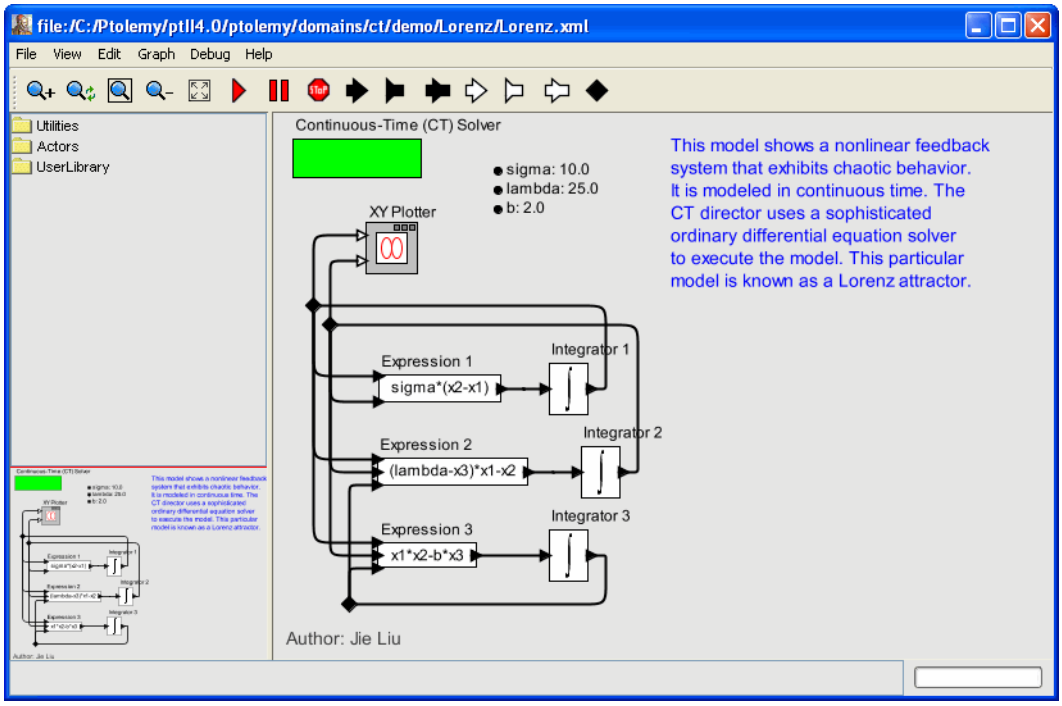

FIGURE 2.6. A block diagram representation of a set of nonlinear ordinary differential equations.

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 the solver box, which results in the dialog shown in figure 2.7. 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 2.8. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

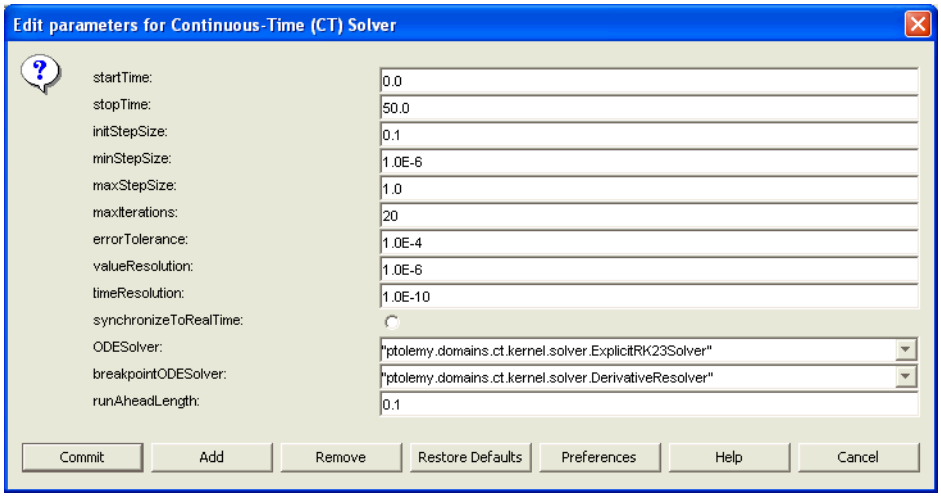

FIGURE 2.7. Dialog box showing solver parameters for the model in figure 2.6.

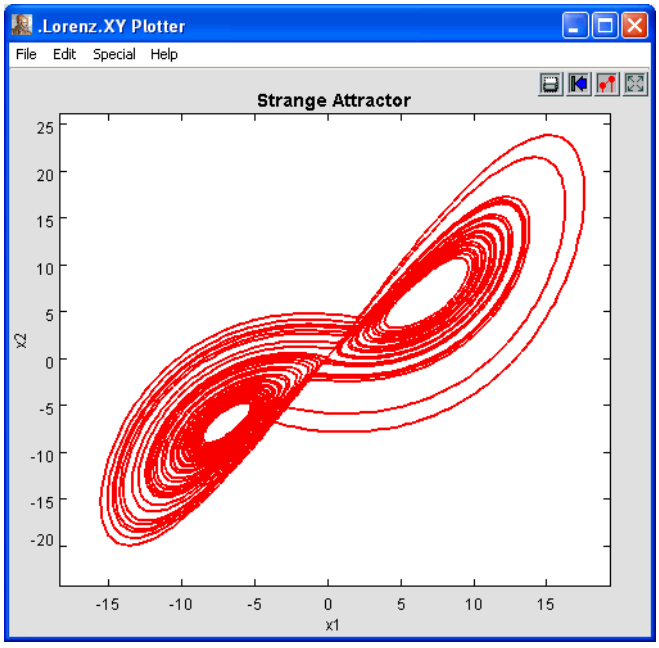

FIGURE 2.8. Result of running the Lorenz model using the run button in the toolbar.

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.

2.2.4 Creating a New Model

Create a new model by selecting File->New->Graph Editor in the welcome window. You should see something like the window shown in figure 2.9. Ignoring the menus and toolbar for a moment, on the left is a palette of objects that can be dragged onto the page on the right. To begin with, the page on the right is blank. Open the *Actors* library in the palette, and go into the *Sources* library. Find the *Const* actor under *GenericSources* and drag an instance over onto the blank page. Then go into the *Sinks* library (*GenericSinks* sublibrary) and drag a *Display* actor onto the page. Each of these actors can be dragged around on the page. However, we would like to connect one to the other. To do this, drag a connection from the output port on the right of the *Const* actor to the input port of the *Display* actor. Lastly, open the *Directors* library and drag an *SDFDirector* onto the page. The director gives a meaning (semantics) to the graph, but for now we don't have to be concerned about exactly what that is.

Now you should have something that looks like figure 2.10. The *Const* actor is going to create our string, and the *Display* actor is going to print it out for us. We need to take care of one small detail to make it look like figure 2.10: we need to tell the *Const* actor that we want the string "Hello World". To do this we need to edit one of the parameters of the *Const*. To do this, either double click on the *Const* actor icon, or right click[1] on the *Const* actor icon and select "Configure". You should see the dialog

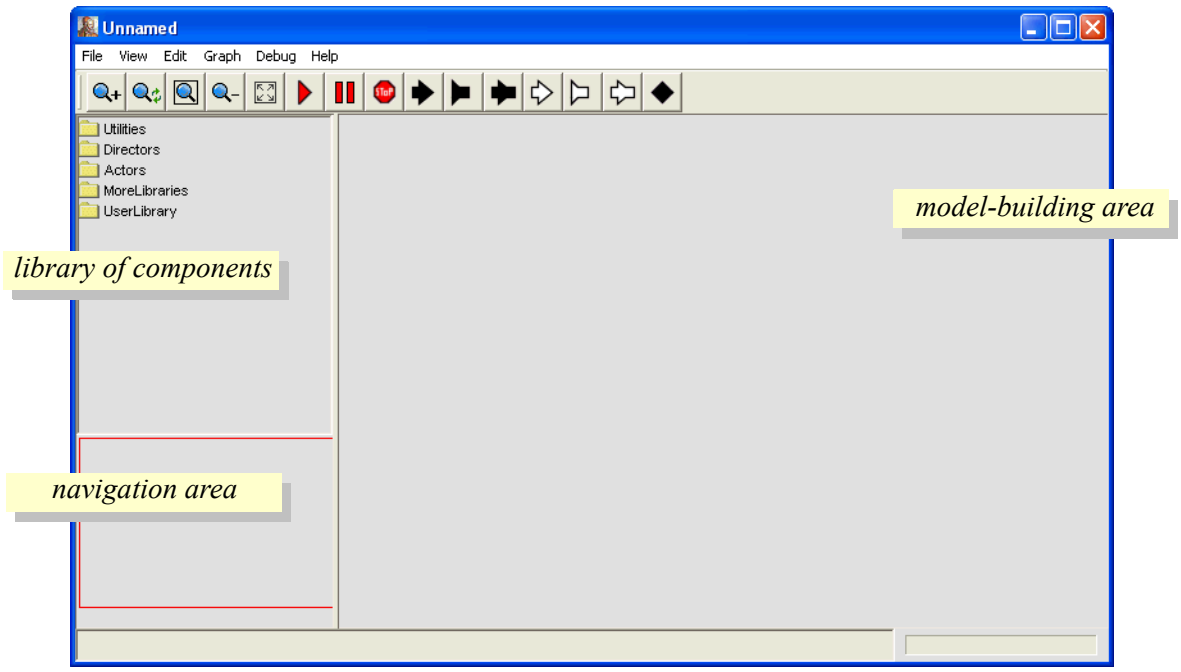

FIGURE 2.9. An empty Vergil Graph Editor.

1. On a Macintosh, which typically has only one mouse button, instead of right clicking, hold the control key and click the one button.

box in figure 2.11. Enter the string "Hello World" for the value parameter and click the Commit button. Be sure to include the double quotes, so that the expression is interpreted as a string.

You may wish to save your model, using 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.

2.2.5 Running the Model

To run the example, go to the View menu and select the Run Window. If you click the "Go" button, you will see a large number of strings in the display at the right. To stop the execution, click the "Stop" button. To see only one string, change the *iterations* parameter of the SDF Director to 1, which can be

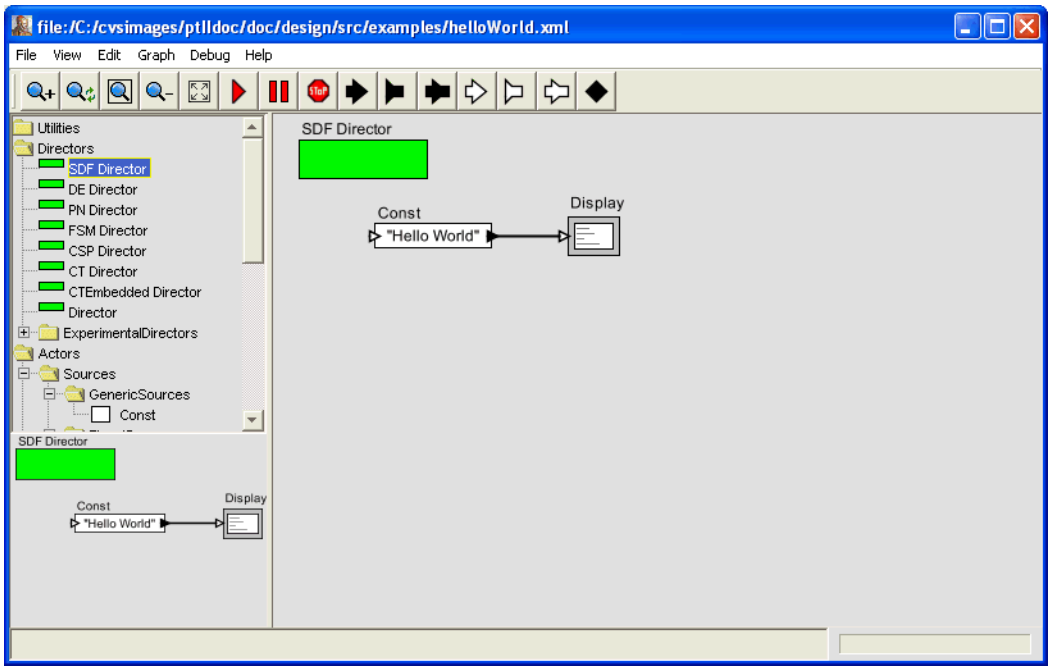

FIGURE 2.10. The Hello World example.

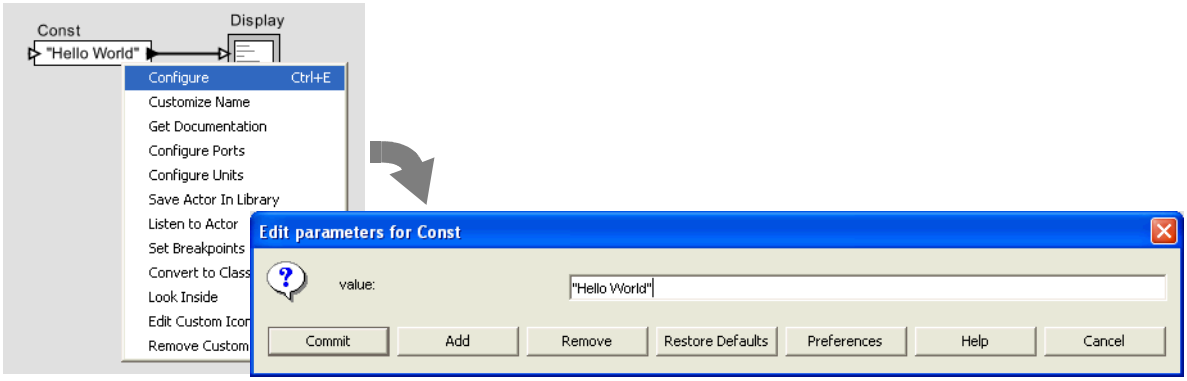

FIGURE 2.11. The Const parameter editor.

done in the run window, or in the graph editor in the same way you edited the parameter of the *Const* actor before. The run window is shown in figure 2.12.

2.2.6 Making Connections

The model constructed above contained only two actors and one connection between them. If you move either actor (by clicking and dragging), you will see that the connection is routed automatically. We can now explore how to create and manipulate more complicated connections.

First create a model in a new graph editor that includes an *SDFDirector*, a *Ramp* actor (found in the *Sources*) library, a *Display* actor, and a *SequencePlotter* actor, found in the *Sinks* library, as shown in figure 2.13. Suppose we wish to route the output of the *Ramp* to both the *Display* and the *SequencePlotter*. If we simply attempt to make the connections, we get the exception shown in figure 2.13.

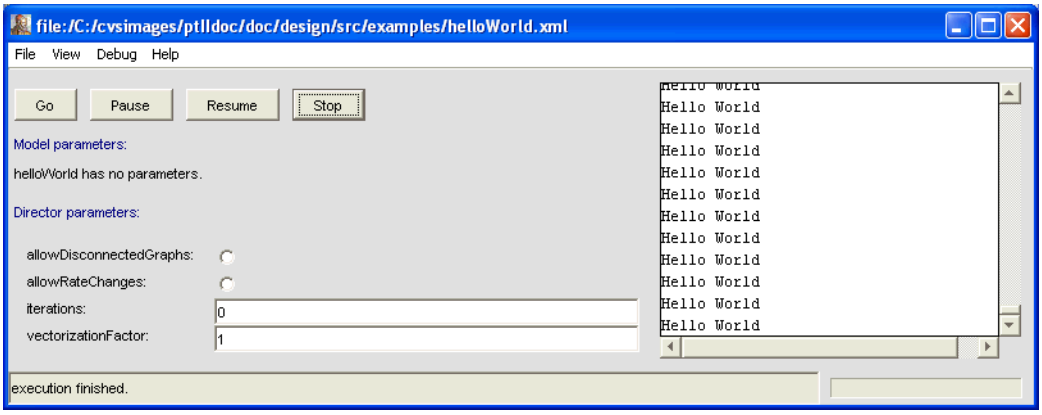

FIGURE 2.12. Execution of the Hello World example.

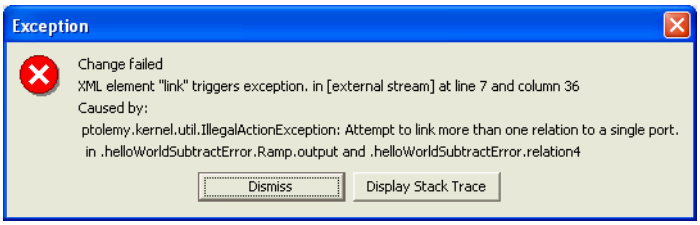

FIGURE 2.13. Exception that occurs if you attempt to simply wire the output of the *Ramp* in figure 2.14 to the inputs of the other two actors.

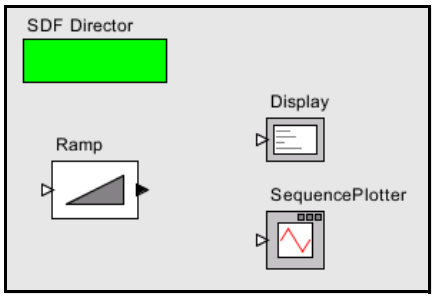

FIGURE 2.14. Three unconnected actors in a model.

Don't panic! Exceptions are normal and common. The key information in this exception report is the text:

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

The last line gives the names of the objects involved, which in this case are:

```
in .broadcastRelations.Ramp.output and .broadcastRelations.relation2
```

(This assumes the model has been saved under the name "broadcastRelations.") In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".<Unnamed Object>.Ramp.output" is an object named "output" contained by an object named "Ramp", which is contained by an unnamed object (the model itself). The model has no name because we have not assigned one (it acquires a name when we save it).

Why did this exception occur? Ptolemy II supports two distinct flavors of ports, indicated in the diagrams by a filled triangle or an unfilled triangle. The output port of the *Ramp* actor is a *single port*, indicated by a filled triangle, which means that it can only support a single connection. The input port of the *Display* and *SequencePlotter* actors are *multiports*, indicated by unfilled triangles, which means that they can support multiple connections. Each connection is treated as a separate *channel*, which 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 *Ramp* 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 2.15. 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.[1]

In the model shown in figure 2.15, 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. However, as of the 4.0 release of

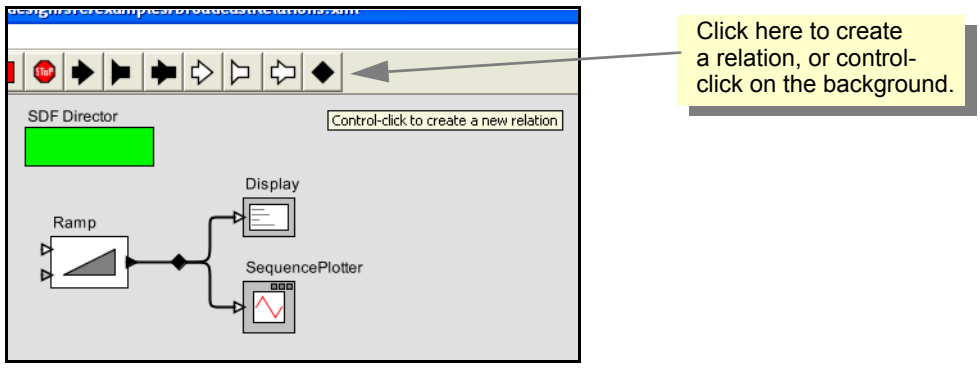

FIGURE 2.15. A relation can be used to broadcast an output from a single port.

1. On a Macintosh, hold the command key rather than the control key.

Ptolemy II, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

To explore multiports, try putting some other signal source in the diagram and connecting it to the *SequencePlotter* or to the *Display*. If you explore this fully, you will discover that the *SequencePlotter* can only accept inputs of type *double*, or some type that can be losslessly converted to *double*, such as *int*. These data type issues are explored next.

2.3 Tokens and Data Types

In the example of figure 2.10, 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 2.16. The *Ramp* actor is listed under *Sources*, *SequenceSources*, 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 2.17. Do not worry; excep-

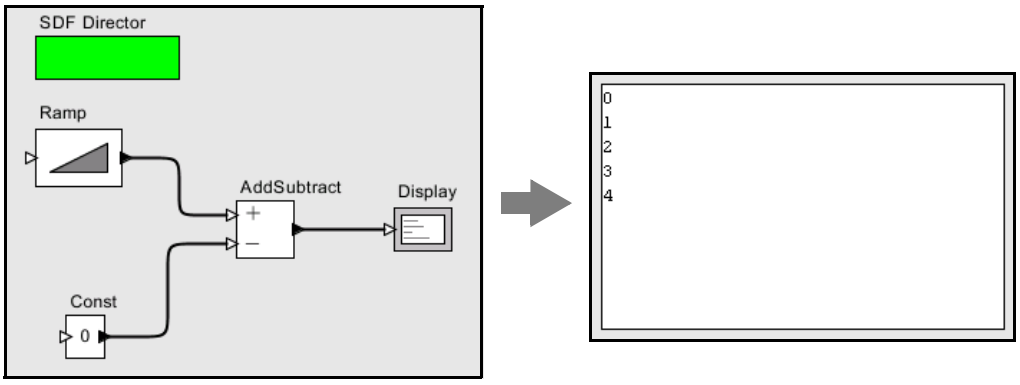

FIGURE 2.16. Another example, used to explore data types in Ptolemy II.

tions 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 2.17. You should see the stack trace shown in figure 2.18. This window displays the execution sequence that resulted in the exception. For example, the line

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

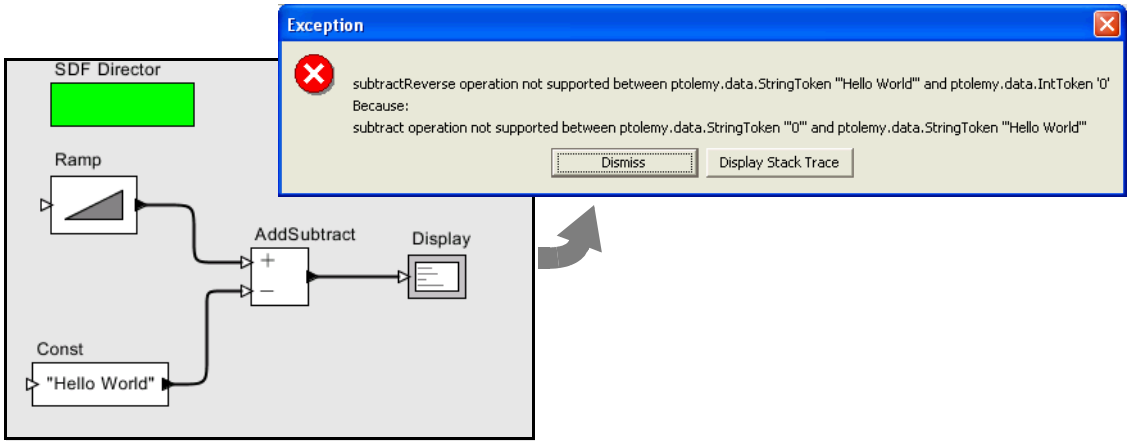

FIGURE 2.17. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.

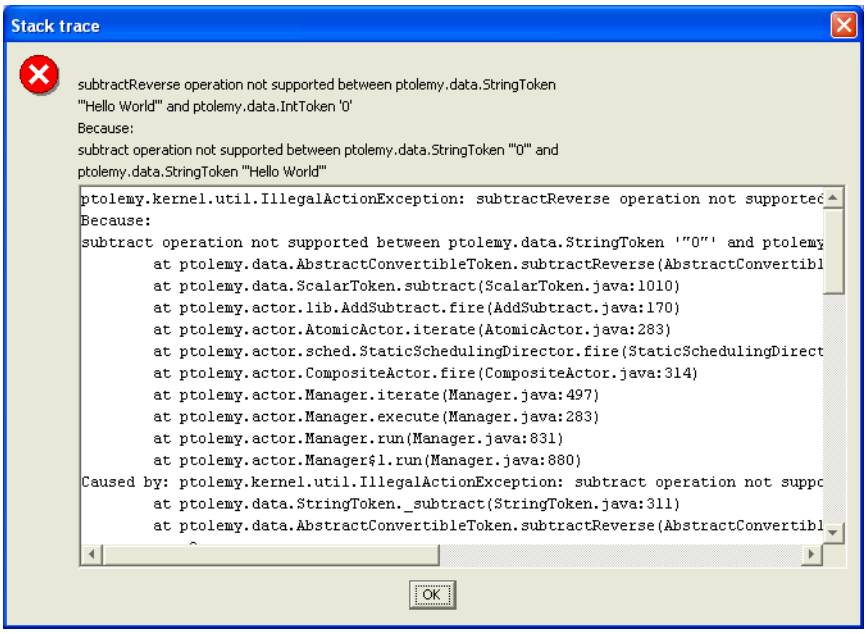

FIGURE 2.18. Stack trace for the exception shown in figure 2.17.

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 (most installation mechanisms at least offer the option of installing the source), 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 $PTII, then the location of this file is given by the full class name, but with the periods replaced by slashes; in this case, it is at `$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 2.19. 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 of Volume 2, 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.

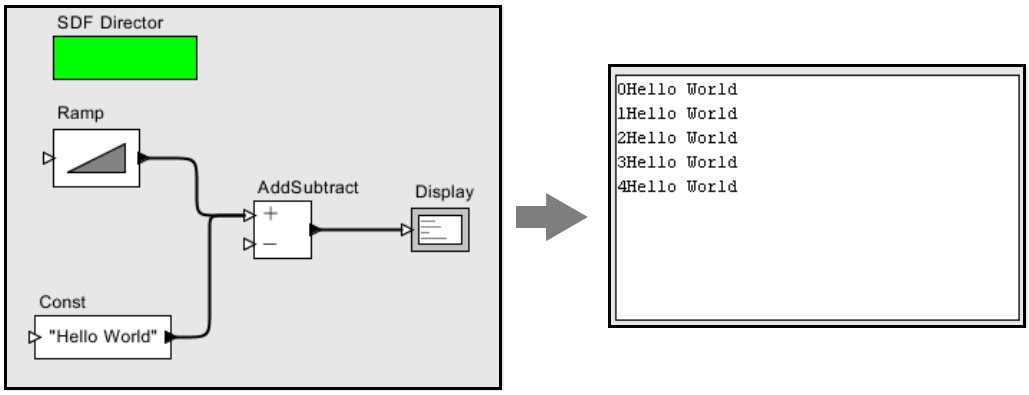

FIGURE 2.19. Addition of a string to an integer.

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.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *CompositeActor* from the *Utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking[1] over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in figure 2.20. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in figure 2.21. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

First we have to add some ports to the composite actor. There are several ways to do this, but clicking on the port buttons in the toolbar is probably the easiest. You can explore the ports in the toolbar by lingering with the mouse over each button in the toolbar. A tool tip pops up that explains the button. The buttons are summarized in figure 2.22. 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 2.23, you can also right click[2] 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. You can also specify the *direction* of a port (where it appears on the icon; by default inputs appear on the left, outputs on the right, and ports that are both inputs and outputs appear on the bottom of the icon). You can also control

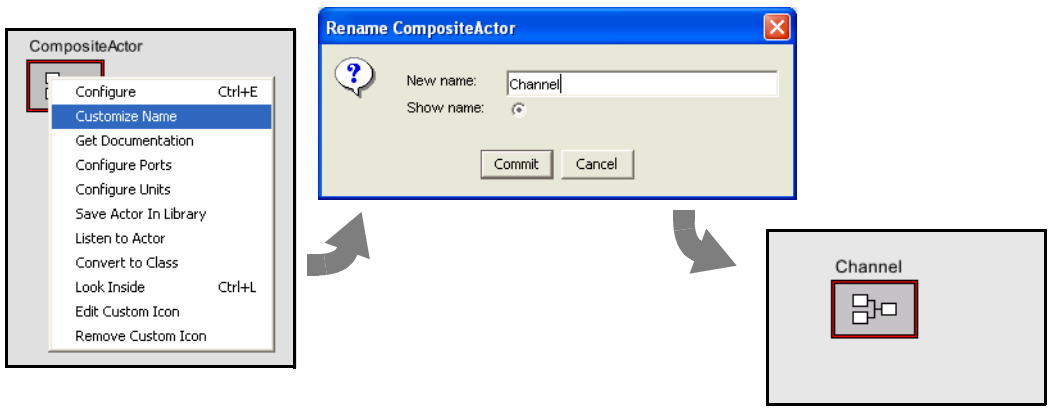

FIGURE 2.20. Changing the name of an actor.

1. On a Macintosh, control-click.
2. On a Macintosh, control-click.

whether the name of the port is shown outside the icon (by default it is not), and even whether the port is shown at all. The “Units” column will be discussed further below.

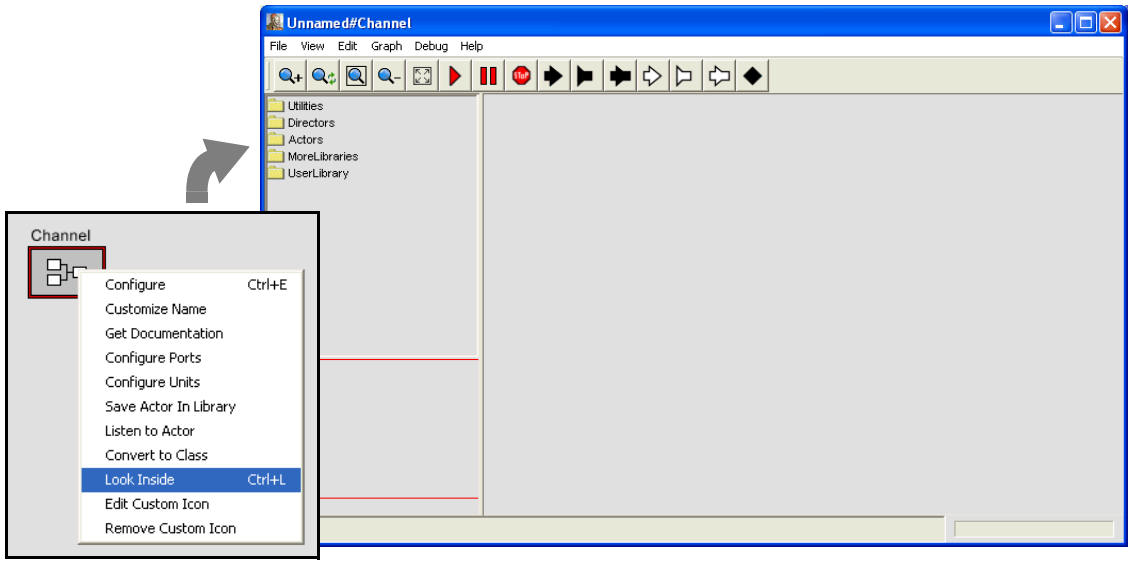

FIGURE 2.21. Looking inside a composite actor.

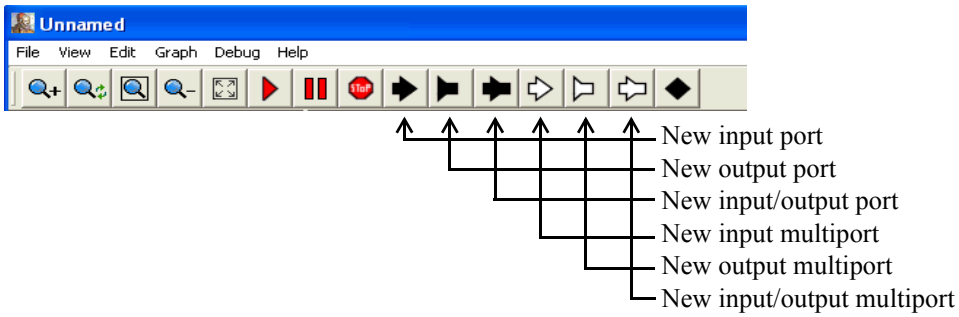

FIGURE 2.22. Summary of toolbar buttons for creating new ports.

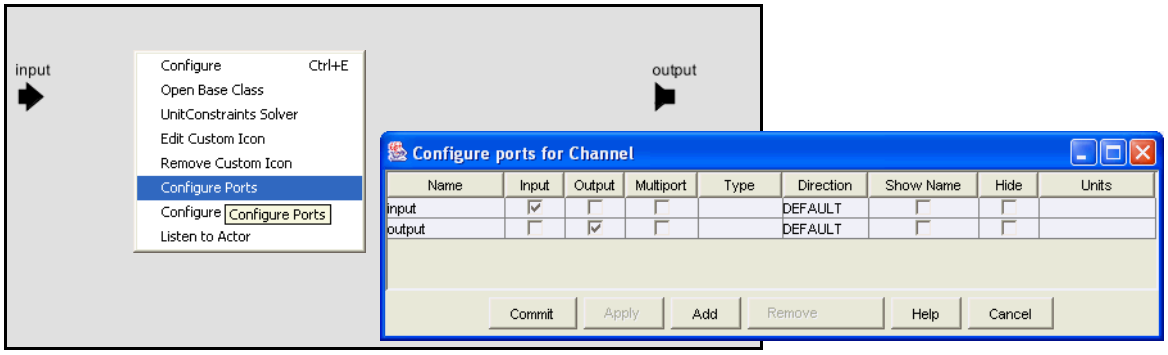

FIGURE 2.23. Right clicking on the background brings up a dialog that can be used to configure ports.

Then using these ports, create the diagram shown in figure 2.24[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 2.25. 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 2.26.

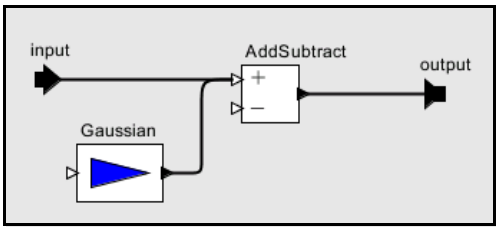

FIGURE 2.24. A simple channel model defined as a composite actor.

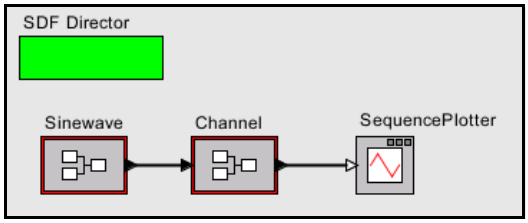

FIGURE 2.25. A simple signal processing example that adds noise to a sinusoidal signal.

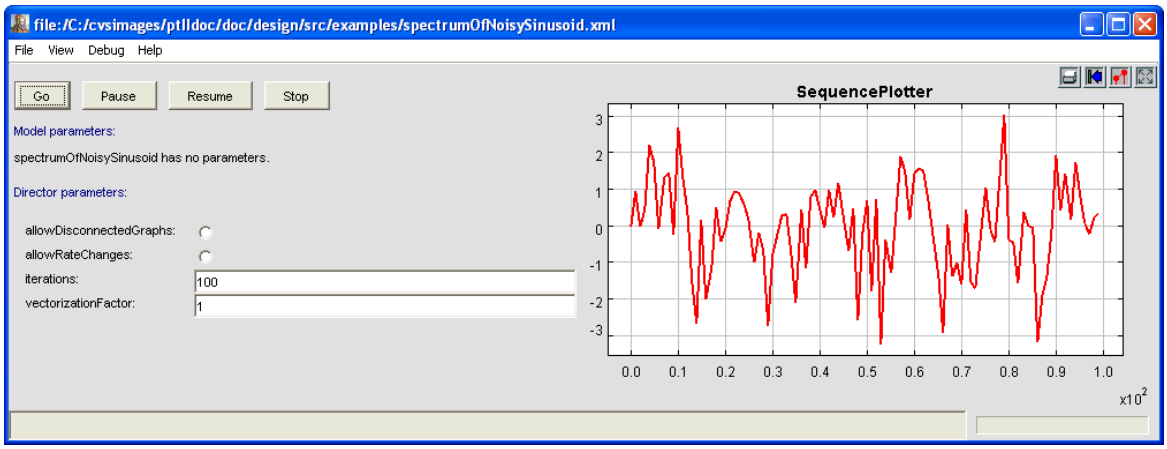

FIGURE 2.26. The output of the simple signal processing model in figure 2.25.

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

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

```
[double]
```

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

```
{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 Annotations and Parameterization

In this section, we will enhance the model in figure 2.25 in a number of ways.

2.5.1 Parameters in Hierarchical Models

First, notice from figure 2.26 that the noise overwhelms the sinusoid, making it barely visible. A useful channel model would have a parameter that sets the level of the noise. Look inside the channel model, and add a parameter by dragging one in from the *Utilities* library, *Parameters* sublibrary, as shown in figure 2.27. Right click[1] on the parameter to change its name to "noisePower". (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 0.1.

Now we can use this parameter. First, let's use it to set the amount of noise. The *Gaussian* actor has a parameter called *standardDeviation*. In this case, the power of the noise is equal to the variance of the Gaussian, not the standard deviation. If you recall from basic statistics, the standard deviation is equal to the square root of the variance. Change the *standardDeviation* parameter of the *Gaussian* actor so its value is "sqrt(noisePower)", as shown in figure 2.28. This is an expression that references the *noisePower* parameter. We will explain the expression language in the next chapter. But first, let check our improved model. Return to the top-level model, and edit the parameters of the *Channel* actor (by either double clicking or right clicking and selecting "Configure"). Change the noise power from

1. On a Macintosh, control-click.

the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in figure 2.29.

Note that you can also add parameters to a composite actor without dragging from the *Utilities* library by clicking on the "Add" button in the edit parameters dialog for the *Channel* composite. This dialog can be obtained by either double clicking on the *Channel* icon, or by right clicking and selecting

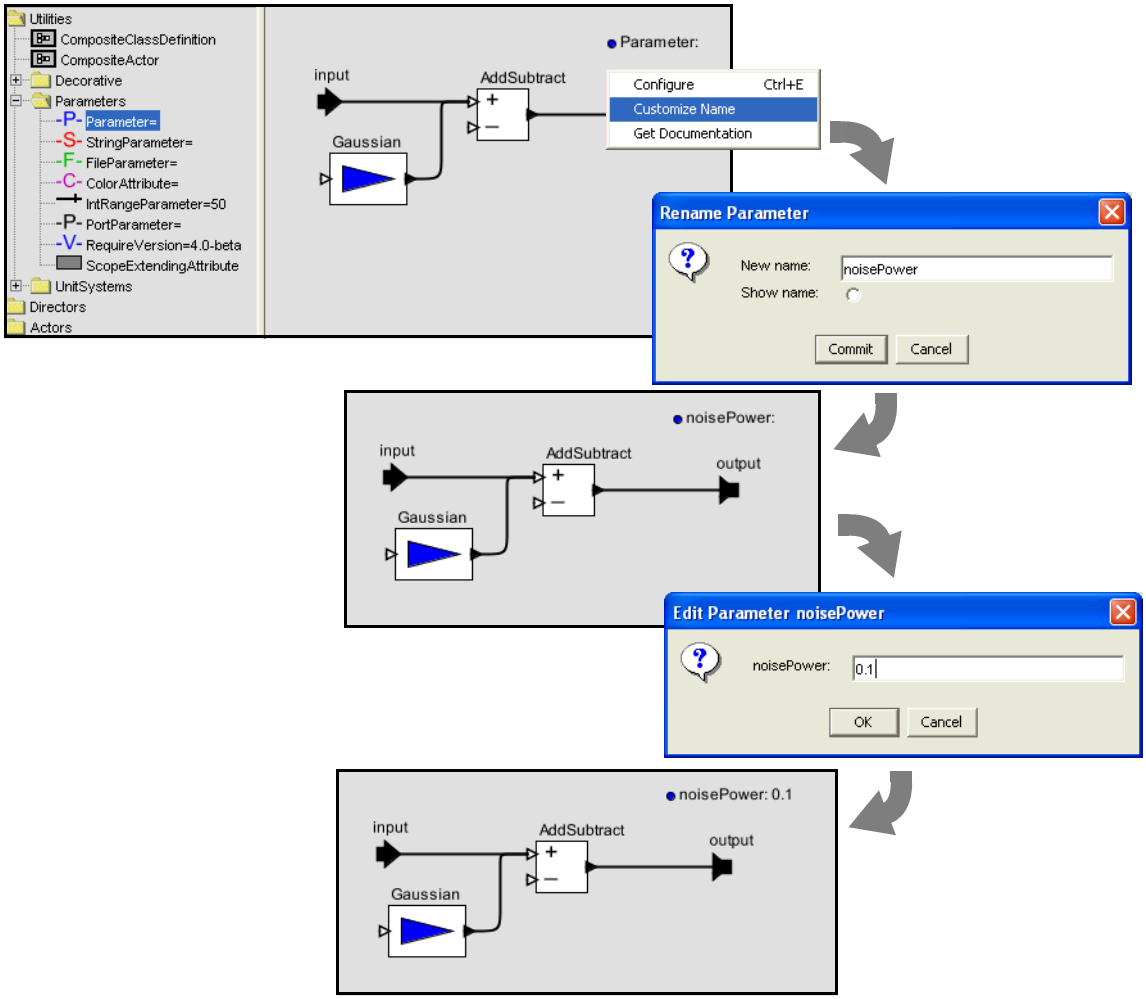

FIGURE 2.27. Adding a parameter to the channel model.

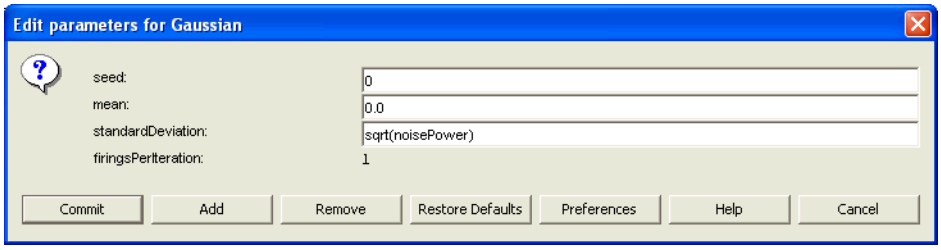

FIGURE 2.28. The standard deviation of the *Gaussian* actor is set to the square root of the noise power.

"Configure", or by right clicking on the background inside the composite and selecting "Edit Parameters". However, parameters that are added this way will not be visible in the diagram when you look inside the Channel actor. Instead, you would have to right click on the background and select Configure to see the parameter.

2.5.2 Decorative Elements

There are several other useful enhancements you could make to this model. Try dragging an *Annotation* from the *Utilities* library, *Decorative* sublibrary, and creating a title on the diagram. A limited number of other decorative elements like geometric shapes can also be added to the diagram from this same library.

2.5.3 Creating Custom Icons

A (rather primitive) icon editor is also provided with Vergil. To create a custom icon, right click on the icon and select "Edit Custom Icon," as shown in figure 2.30. The box in the middle of the icon editor displays the size of the default icon, for reference. Try creating an icon like the one shown in figure 2.31. Hint: The fill color of the rectangle is set to "none" and the fill color of the trapezoid is first selected using the color selector, then modified to have an *alpha* (transparency) of 0.5. Finally, since the icon itself has the actor name in it, the Customize Name dialog is used to deselect "show name."

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 2.33. 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

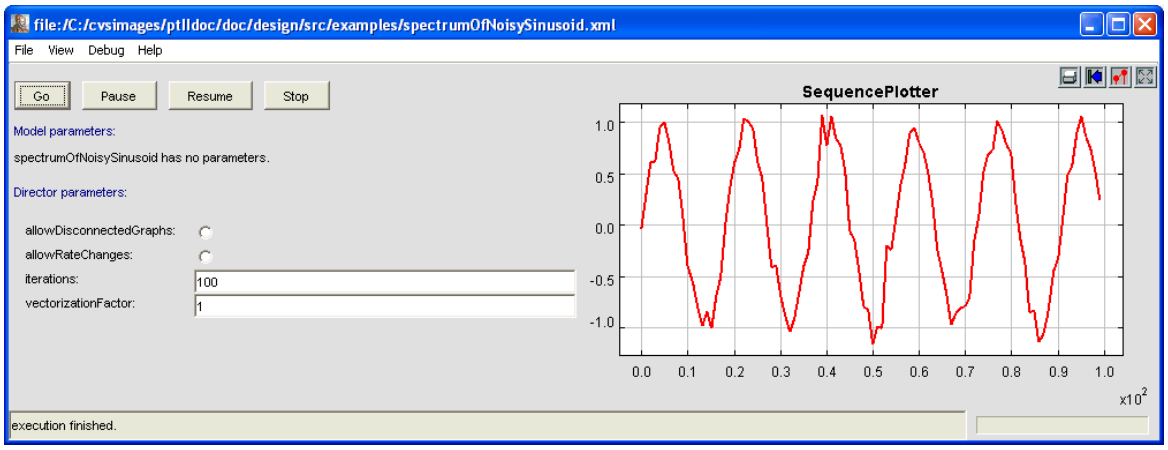

FIGURE 2.29. The output of the simple signal processing model in figure 2.25 with noise power = 0.01

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.

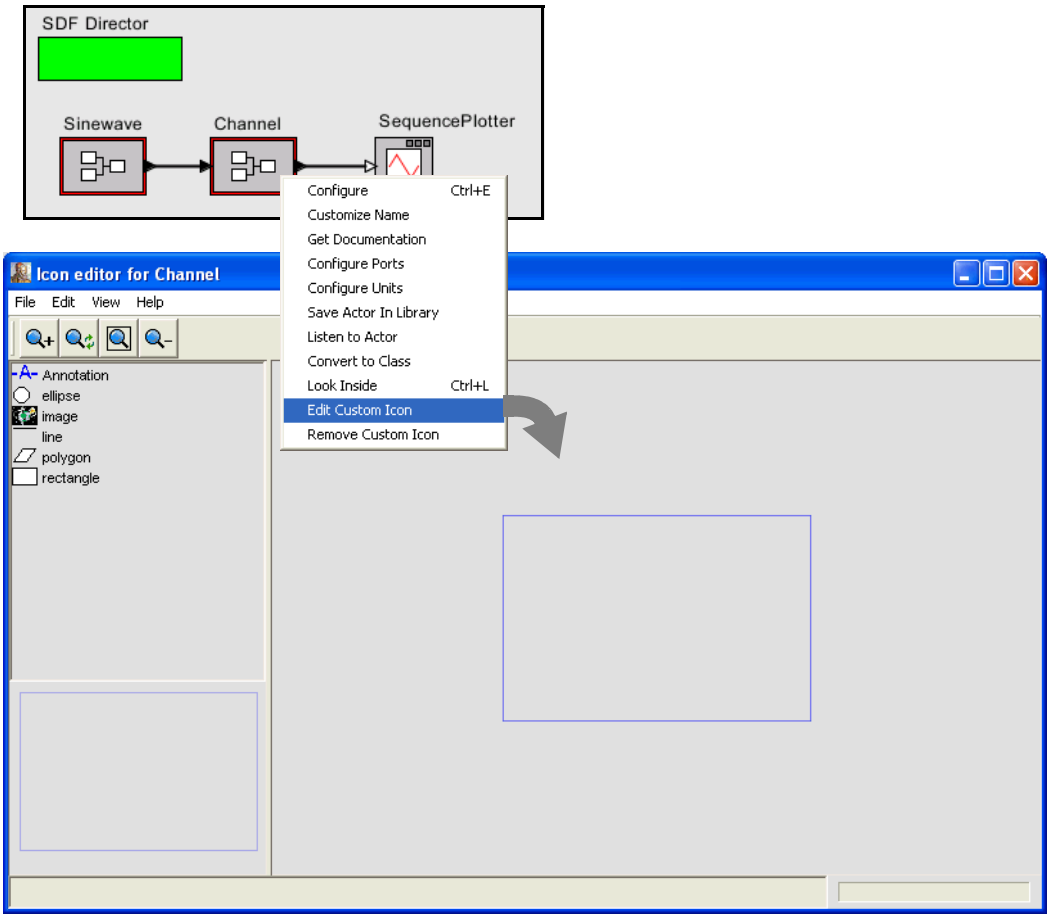

FIGURE 2.30. Custom icon editor for the Channel actor.

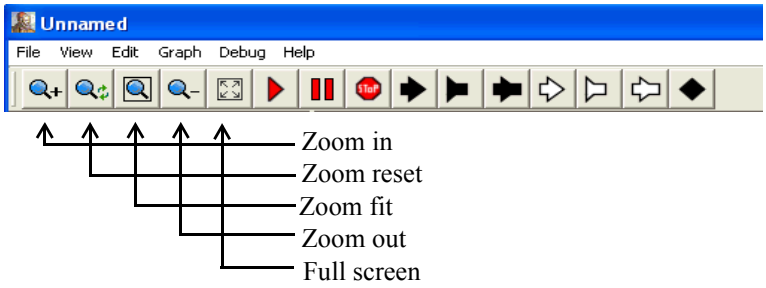

FIGURE 2.32. Summary of toolbar buttons for zooming and fitting.

2.7 Classes and Inheritance

One of the major new capabilities introduced with version 4.0 of Ptolemy II is the ability to define *actor-oriented classes* with instances and subclasses with inheritance. The key idea is that you can specify that a component definition is a *class*, in which case all instances and subclasses inherit its structure. This improves modularity in designs. We will illustrate this capability with an example.

2.7.1 Creating and Using Actor-Oriented Classes

Consider the model that we developed in section 2.4, shown for reference in figure 2.34. Suppose that we wish to create multiple instances of the channel, as shown in figure 2.35. In that figure, the sinewave signal passes through five distinct channels (note the use of a relation to broadcast the same signal to each of the five channels). The outputs of the channels are added together and plotted. The result is a significantly cleaner sine wave than the one that results from one channel alone[1]. However, this is

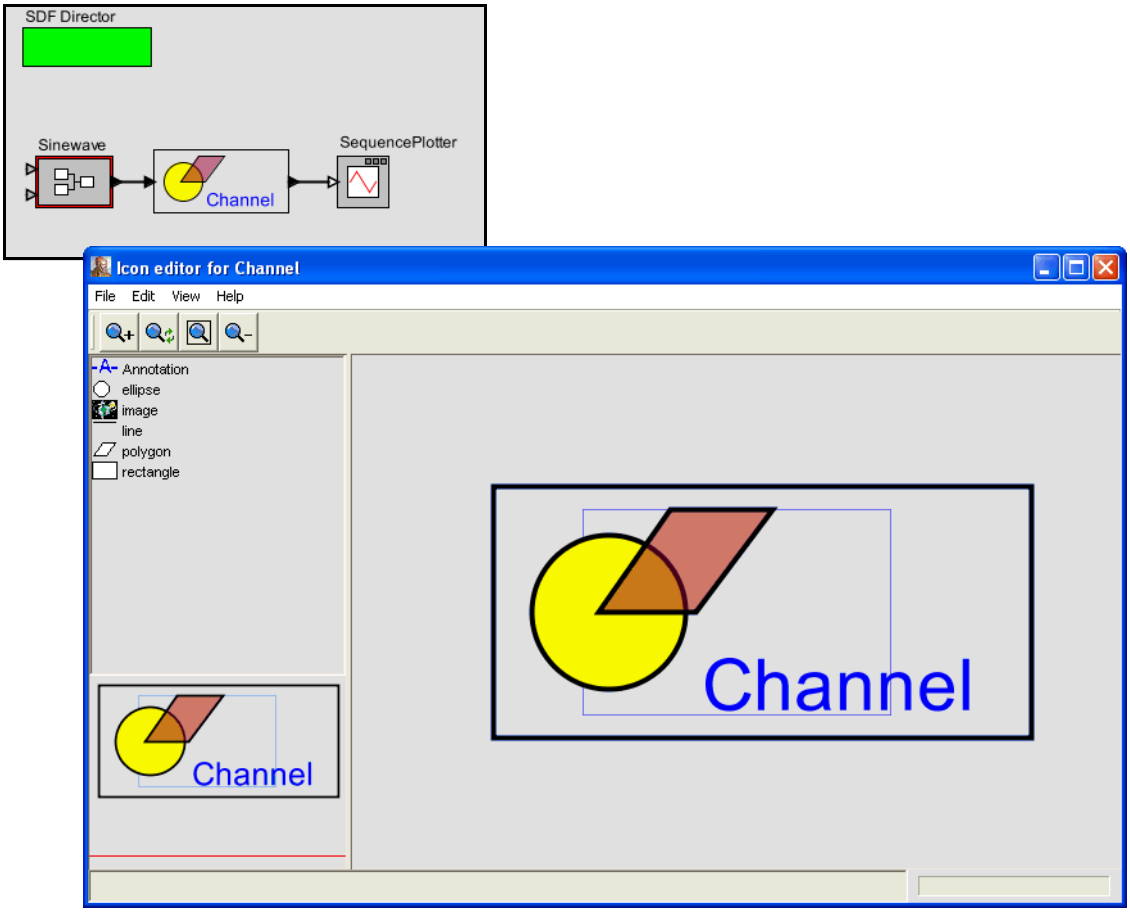

FIGURE 2.31. Custom icon for the Channel actor.

1. In communication systems, this technique is known as *diversity*, where multiple channels with independent noise are used to achieve more reliable communication.

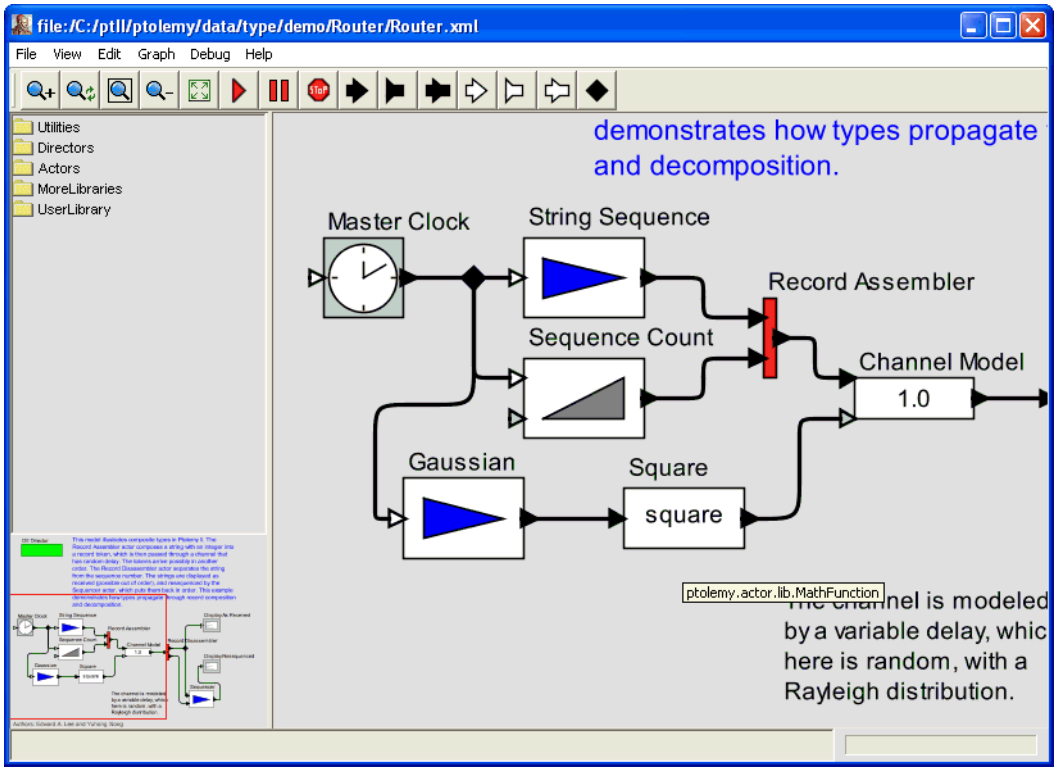

FIGURE 2.33. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

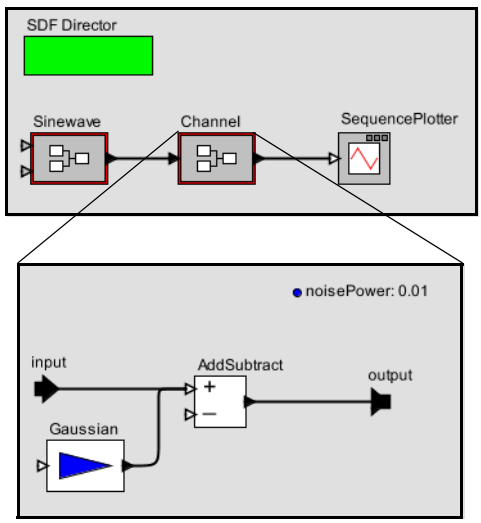

FIGURE 2.34. Hierarchical model that we will modify to use classes.

a poor design, for two reasons. First, the number of channels is hardwired into the diagram. We will deal with that problem in the next section. Second, each of the channels is a *copy* of the composite actor in figure 2.34. This results in a far less maintainable or scalable model than we would like. Consider, for example, what it would take to change the design of the channel. Each of the five copies would have to be changed individually.

A better solution is to define a channel class. To do this, begin with the design in figure 2.34, and remove the connections to the channel, as shown in figure 2.36. Then right click and select "Convert to Class." (Note that if you fail to first remove the connections, you will get an error message when you try to convert to class. A class is not permitted to have connections.) The actor icon acquires a blue halo, which serves as a visual indication that it is a class, rather than an ordinary actor (which is an instance). Classes play no role in the execution of the model, and merely serve as definitions of components that must then be instantiated. By convention, we put classes at the top of the model, near the director, since they function as declarations.

Once you have a class, you can create an instance by right clicking and selecting "Create Instance" or typing Control-N. Do this five times to create five instances of the class, as shown in figure 2.36. Although this looks similar to the design in figure 2.35, it is, in fact, a much better design. To verify this, try making a change to the class, for example by creating a custom icon for it, as shown in figure 2.37. Note that the changes propagate to each of the instances of the class. A more subtle advantage is that the XML file representation of the model is much smaller, since the design of the class is given only once rather than five times.

If you look inside any of the instances (or the class) in figure 2.37, you will see the same channel model. In fact, you will see the class definition. Any change you make inside this hierarchical model will be automatically propagated to all the instances. Try changing the value of the noisePower parameter, for example.

2.7.2 Overriding Parameter Values in Instances

By default, all instances of Channel in figure 2.37 have the same icon and the same parameter values. However, each instance can be customized by overriding these values. In figure 2.38, for example,

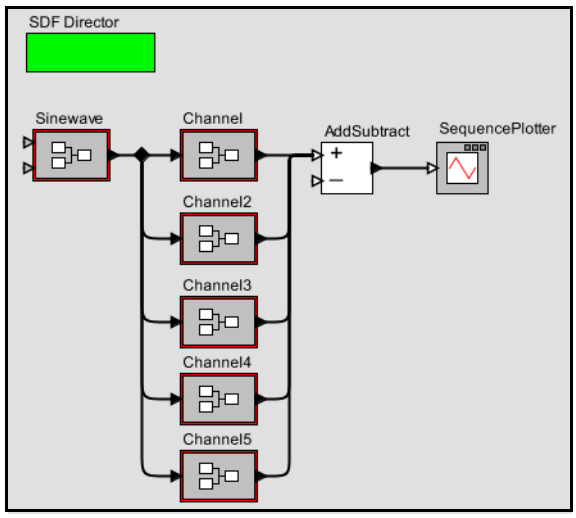

FIGURE 2.35. A poor design of a diversity communication system, which has multiple copies of the channel as defined in figure 2.34.

we have modified the custom icons so that each has a different color, and the fifth one has an extra graphical element. To do this, just right click on the icon of the instance and select "Edit Custom Icon."

2.7.3 Subclassing and Inheritance

Suppose now that we wish to modify some of the channels to add interference in the form of another sinewave. A good way to do this is to create a subclass of the Channel class, as shown in figure 2.39. A subclass is created by right clicking on the class icon and selecting "Create Subclass." The

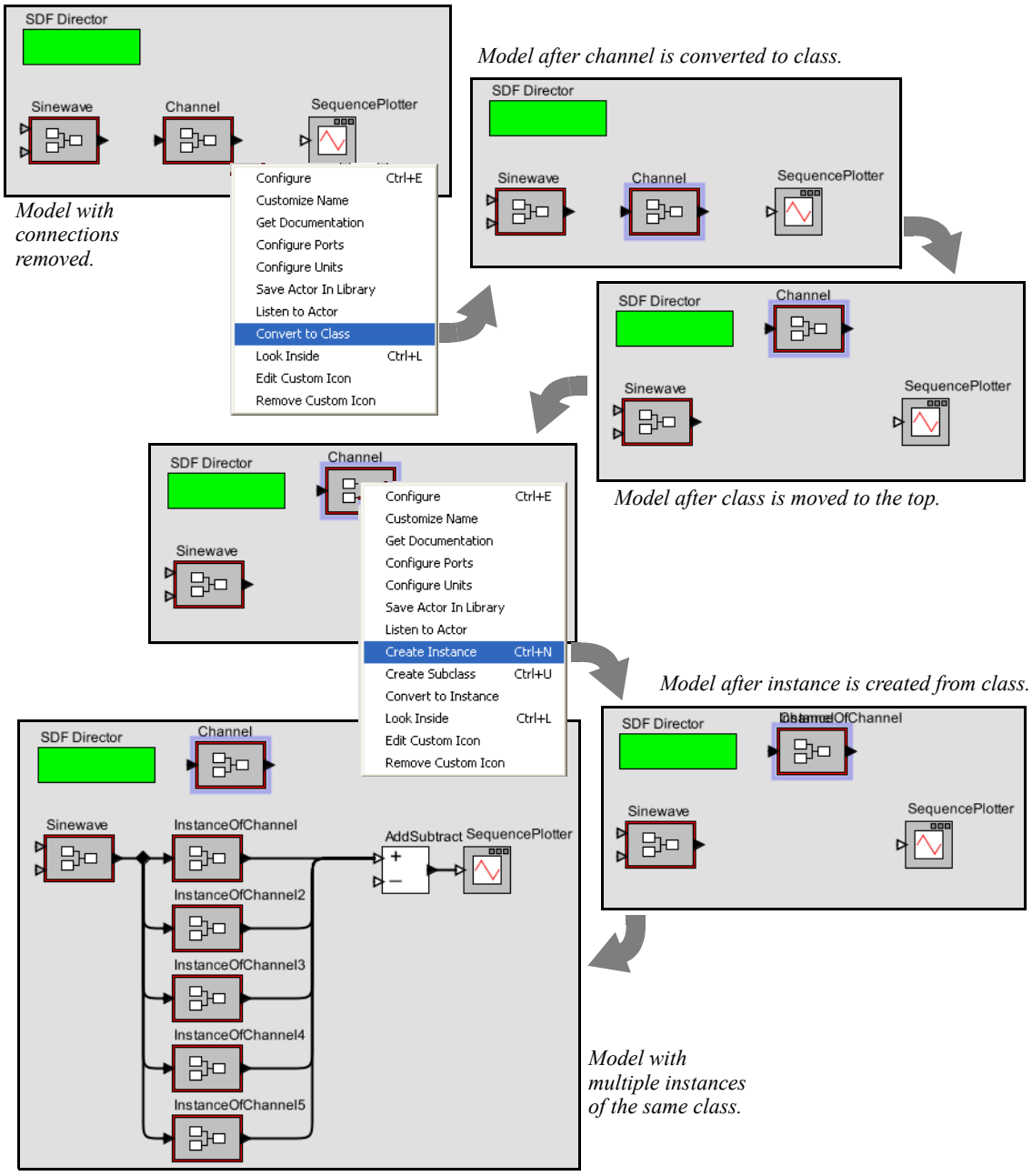

FIGURE 2.36. Creating and using a channel class.

resulting icon for the subclass appears right on top of the icon for the class, so it needs to be moved over, as shown in the figure.

Looking inside the subclass reveals that it contains all the elements of the class, but with their icons now surrounded by a dashed pink outline. These elements are *inherited*. They cannot be removed from the subclass (try to do so, and you will get an error message). You can, however, change their parameter values and add additional elements. Consider the design shown in figure 2.40, which adds an additional pair of parameters named "interferenceAmplitude" and "interferenceFrequency" and an additional pair of actors implementing the interference. A model that replaces the last channel with an instance of the subclass is shown in figure 2.41, along with a plot where you can see the sinusoidal interference.

An instance of a class may be created anywhere in a hierarchical model that is either in the same composite as the class or in a composite contained by that composite. To put an instance into a sub-

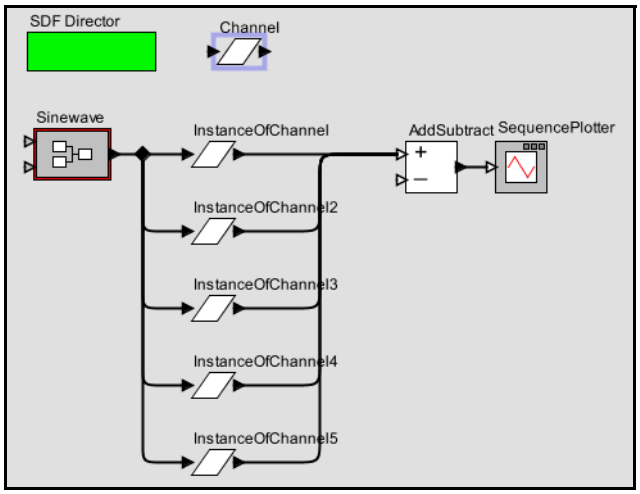

FIGURE 2.37. The model from figure 2.36 with the icon changed for the class. Note that changes to the base class propagate to the instances.

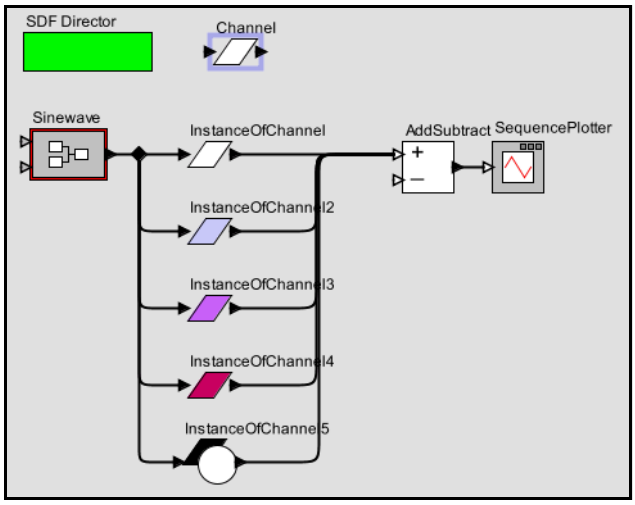

FIGURE 2.38. The model from figure 2.37 with the icons of the instance changed to override parameter values in the class.

model, simply copy (or cut) an instance from the composite where the class is, and then paste that instance into the composite.

2.7.4 Sharing Classes Across Models

A class may be shared across multiple models by saving the class definition in its own file. We will illustrate how to do that with the Channel class. First, look inside the Channel class, and then select Save As from the File menu. The dialog that appears is shown in figure 2.42. The checkbox at the right, labeled "Save submodel only" is by default unchecked, and if left unchecked, what will be saved will be the entire model. In our case, we wish to save the Channel submodel only, so we must check the box.

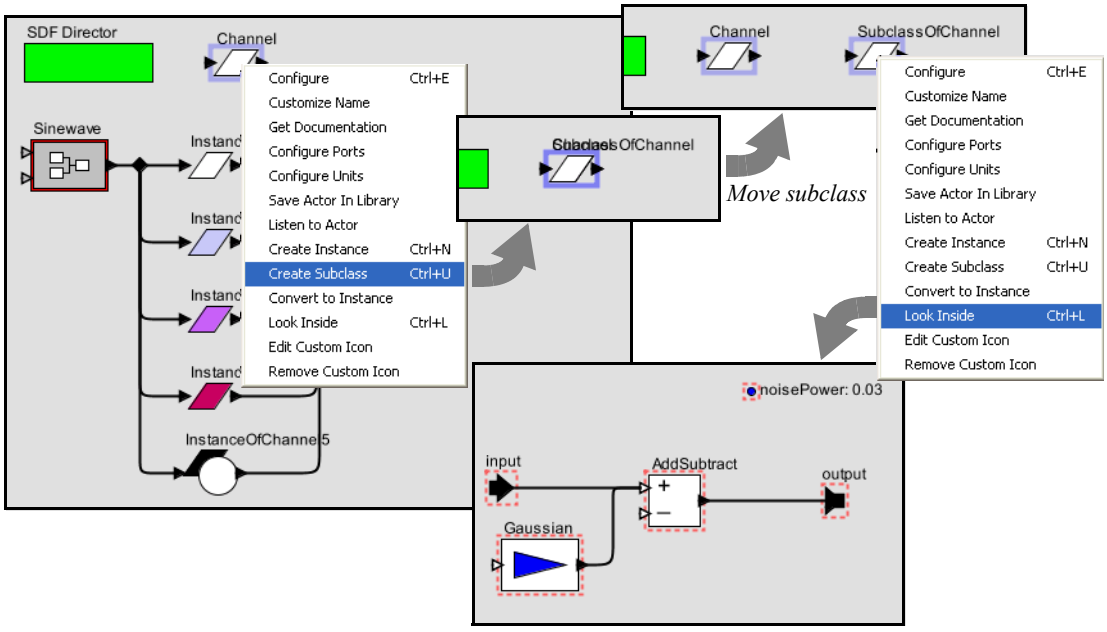

FIGURE 2.39. The model from figure 2.38 with a subclass of the Channel with no overrides (yet).

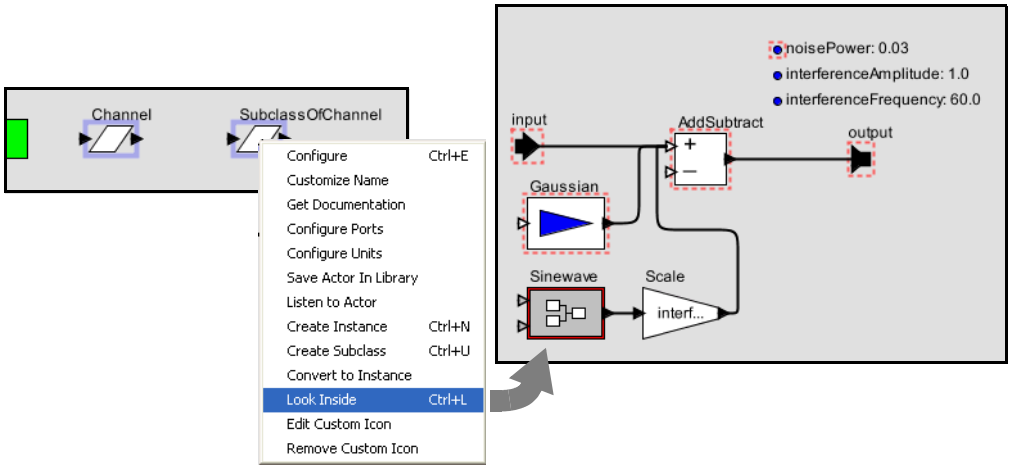

FIGURE 2.40. The subclass from figure 2.39 with overrides that add sinusoidal interference.

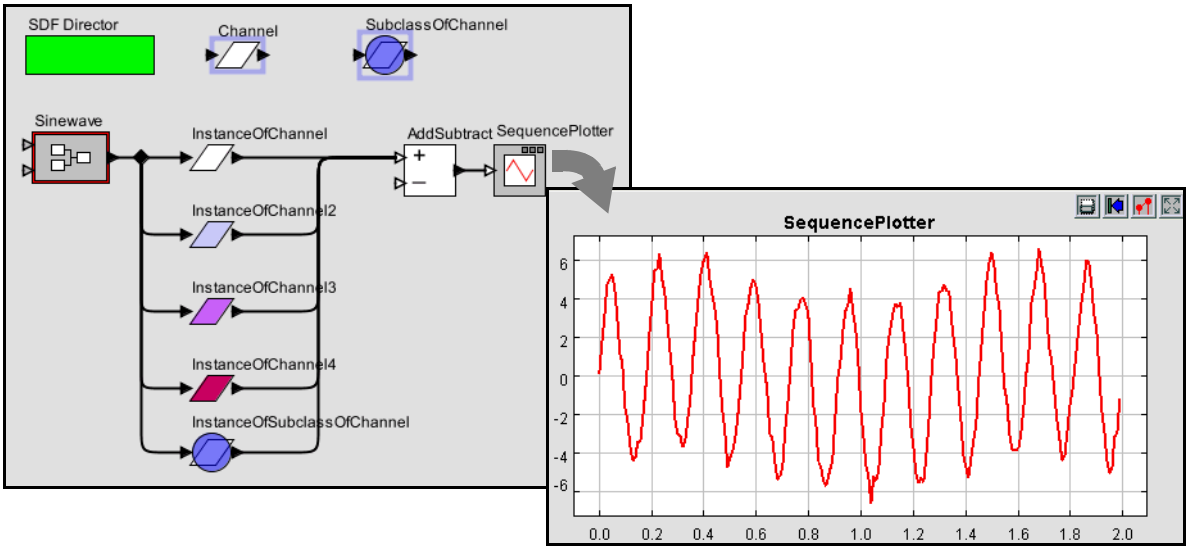

FIGURE 2.41. A model using the subclass from figure 2.40 and a plot of an execution.

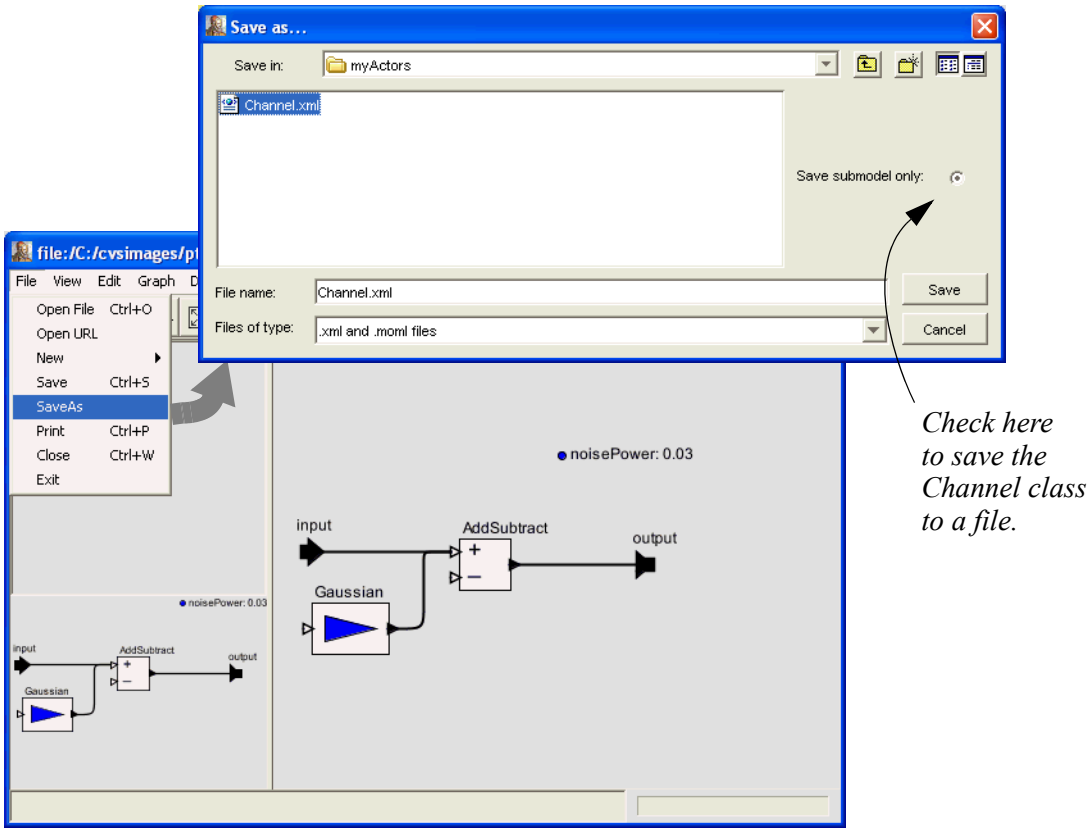

FIGURE 2.42. A class can be saved in a separate file to then be shared among multiple models.

A key issue is to decide where to save the file. As always with files, there is an issue that models that use a class defined in an external file have to be able to find that file. In general Ptolemy II searches for class definitions relative to the *classpath*, which is given by an environment variable called CLASSPATH. In principle, you can set this environment variable to include any particular directory that you would like searched. In practice, changing the CLASSPATH variable often causes problems with programs, so we recommend, when possible, simply storing the file in a directory within the Ptolemy II installation directory.[1]

In figure 2.42, the Channel class is saved to a file called Channel.xml in the directory $PTII/myActors, where $PTII is the location of the Ptolemy II installation. This class definition can now be used in any model as follows. Open the model, and select "Instantiate Entity" in the Graph menu, as shown in figure 2.43. Simply enter the fully qualified class name relative to the $PTII entry in the classpath, which in this case is "myActors.Channel".

Once you have an instance of the Channel class that is defined in its own file, you can add it to the UserLibrary that appears in the library browser to the left in Vergil windows, as shown in figure 2.44. To do this, right click on the instance and select "Save Actor in Library." As shown in the figure, this causes another window to open, which is actually the user library. The user library is a Ptolemy II model like any other, stored in an XML file. If you now save that library model, then the class instance will be available in the UserLibrary henceforth in any Vergil window.

One subtle point is that it would not accomplish the same objective if the class definition itself (vs. an instance of the class) were to be saved in the user library. If you were to do that, then the user library would provide a new class definition rather than an instance of the class when you drag from it.

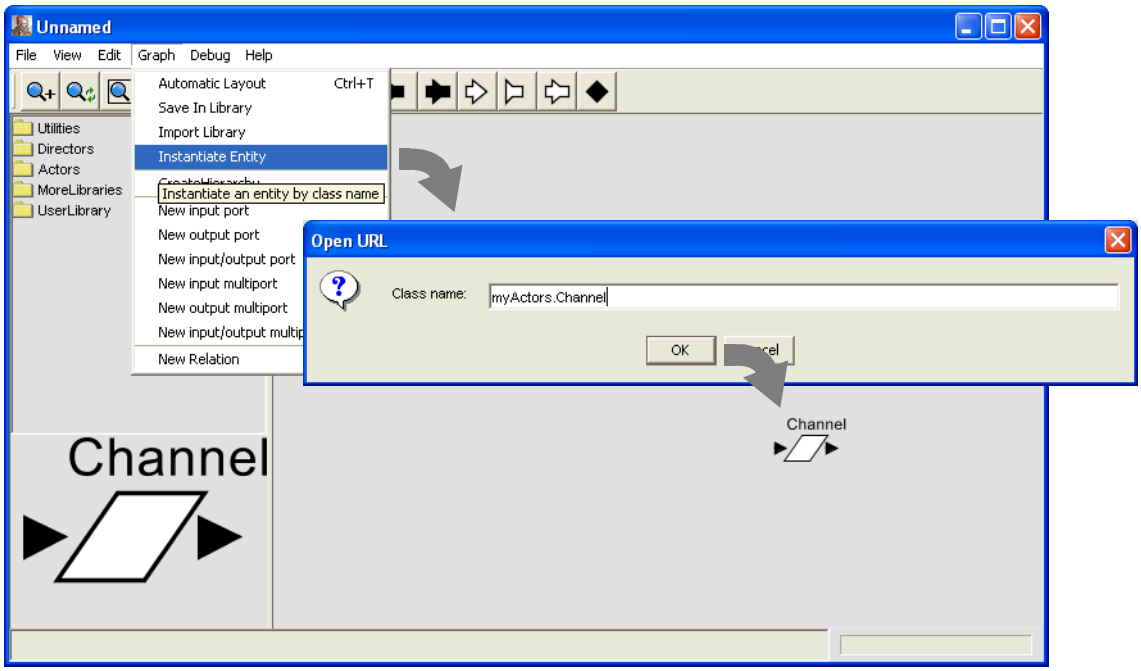

FIGURE 2.43. An instance of a class defined in a file can be created using Instantiate Entity in the Graph menu.

1. If you don't know where Ptolemy II is installed on your system, you can find out by invoking File, New, Expression Evaluator and typing PTII followed by Enter.

2.8 Higher-Order Components

Ptolemy II includes a number of *higher-order components*, which are actors that operate on the structure of the model rather than on data. This notion of higher-order components appeared in Ptolemy Classic and is described in [81], but the realization in Ptolemy II is more flexible than that in Ptolemy Classic. These higher-order components help significantly in building large designs where the model structure does not depend on the scale of the problem. In this section, we describe a few of these components, all of which are found in the HigherOrderActors library. The ModalModel actor is described below in section 2.10, after explaining some of the domains that can make effective use of it.

2.8.1 MultiInstance Composite

Consider model in figure 2.37, which has five instances of the Channel class wired in parallel. This model has the unfortunate feature that the number of instances is hardwired into the diagram. It is awkward, therefore, to change this number, and particularly awkward to create a larger number of

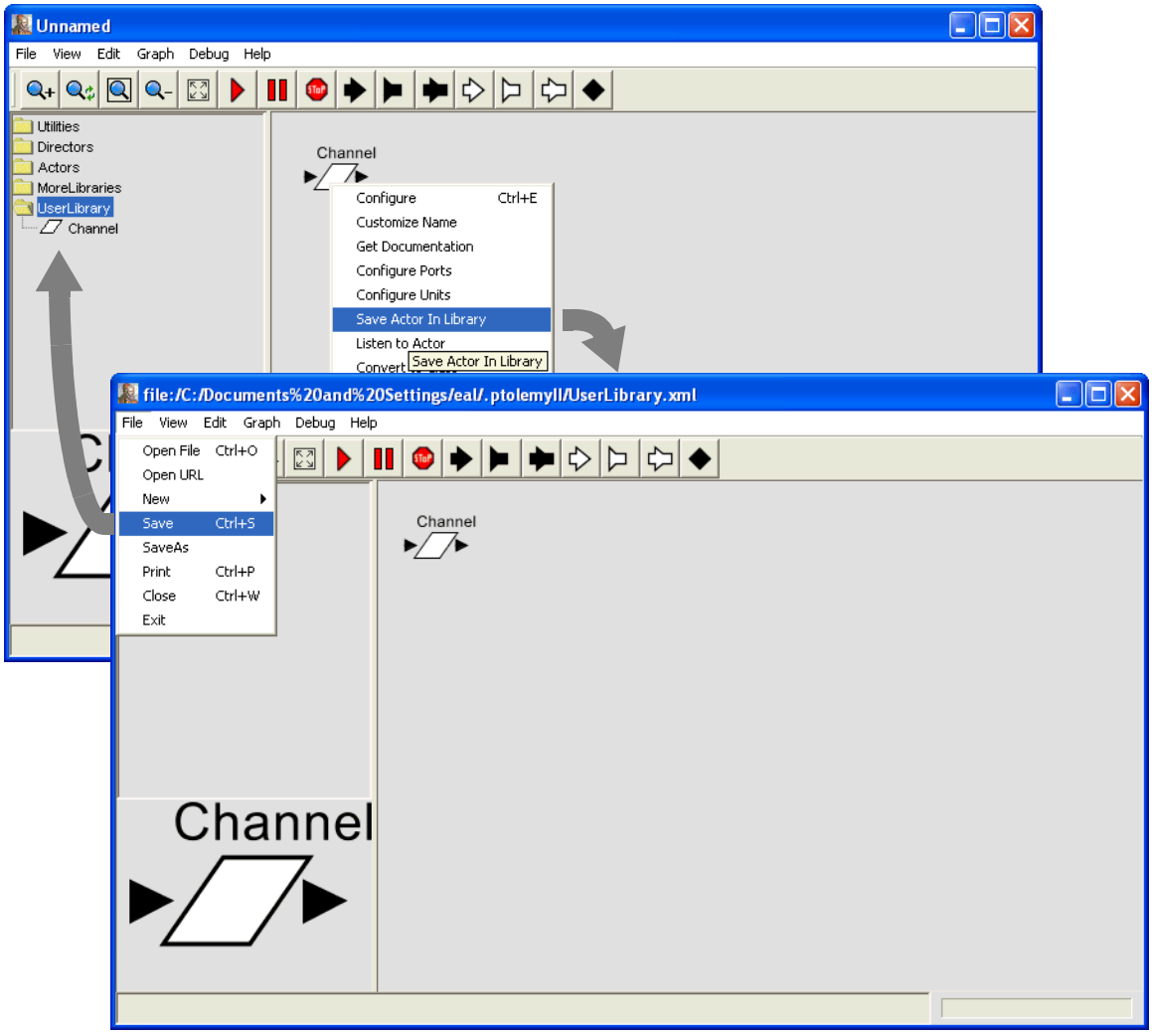

FIGURE 2.44. Instances of a class that is defined in its own file can be made available in the UserLibrary.

instances. This problem is solved by the MultiInstanceComposite actor[1]. A model equivalent to that of figure 2.37 but using the MultiInstanceComposite actor is shown in figure 2.45. The MultiInstance-Composite is a composite actor into which we have inserted a single instance of the Channel (this is inserted by creating an instance of the of Channel, then copying and pasting it into the composite).

The MultiInstanceComposite actor has two parameters, *nInstances* and *instance*, shown in figure 2.46. The first of these specifies the number of instances to create. At run time, this actor replicates itself this number of times, connecting the inputs and outputs to the same sources and destinations as the first (prototype) instance. In figure 2.45, notice that the input of the MultiInstanceComposite is connected to a relation (the black diamond), and the output is connected directly to a multiport input of the AddSubtract actor. As a consequence, the multiple instances will be wired in a manner similar to figure 2.37, where the same input value is broadcast to all instances, but distinct output values are supplied to the AddSubtract actor.

The model of figure 2.45 is better than that of figure 2.37 because now we can change the number of instances by changing one parameter value. The instances can also be customized on a per-instance basis by expressing their parameter values in terms of the *instance* parameter of the MultiInstance-Composite. Try, for example, making the *noisePower* parameter of the InstanceOfChannel actor in fig-

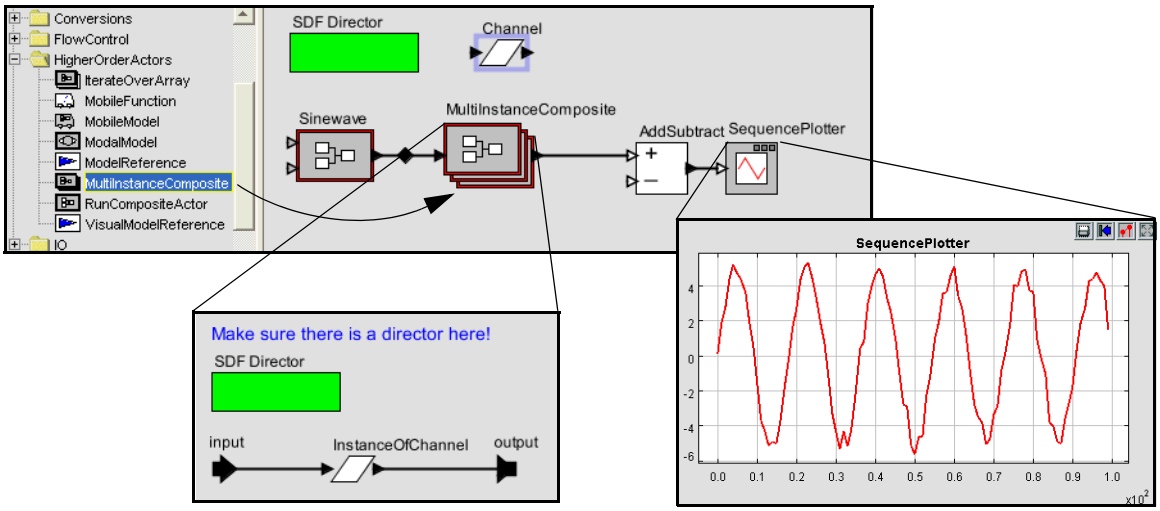

FIGURE 2.45. A model that is equivalent to that of figure 2.37, but using a MultiInstanceComposite, which permits the number of instances of the channel to change by simply changing one parameter value.

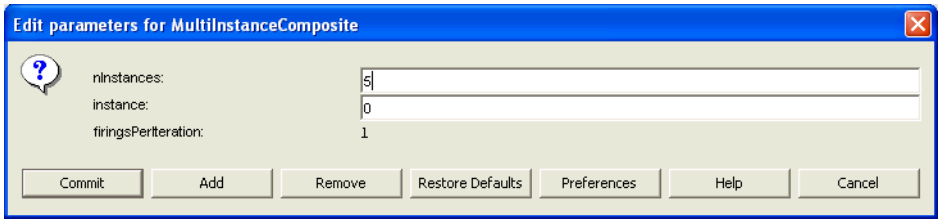

FIGURE 2.46. The first parameter of the MultiInstanceComposite specifies the number of instances. The second parameter is available to the model builder to identify individual instances.

1. The MultiInstanceComposite actor was contributed to the Ptolemy II code base by Zoltan Kemenczy and Sean Simmons, of Research In Motion Limited.

ure 2.45 depend on *instance*. E.g., set it to "`instance * 0.1`" and then set *nInstances* to 1. You will see a clean sine wave when you run the model.

2.8.2 IterateOverArray

The implementation of the Channel class, which is shown in figure 2.42, happens to not have any state, meaning that an invocation of the Channel model does not depend on data calculated in a previous invocation. As a consequence, it is not really necessary to use *n* distinct instances of the Channel class to realize a diversity communication system. A single instance could be invoked *n* times on *n* copies of the data. We can do this using the IterateOverArray higher-order actor.

The IterateOverArray actor can be used in a manner similar to how we used the MultiInstanceComposite in the previous section. That is, we can populate it with an instance of the Channel class, similar to figure 2.45. Just like the MultiInstanceComposite, the IterateOverArray actor requires a director inside. An implementation is shown in figure 2.47. Notice that in the top-level model, instead of using a relation to broadcast the input to multiple instances of the channel, we create an array with multiple copies of the channel input. This is done using a combination of the Repeat actor (found in the FlowControl library, SequenceControl sublibrary) and the SequenceToArray actor (found in the Array library). The Repeat actor has a single parameter, *numberOfTimes*, which in figure 2.47 we have set equal to the value of the *diversity* parameter that we have added to the model. The SequenceToArray actor has a parameter *arrayLength* that we have also set equal to *diversity* (this parameter, interestingly, can also be set via the *arrayLength* port, which is filled in gray to indicate that it is both parameter and a port). The output is sent to an ArrayAverage actor, also found in the Array library.

The execution of the model in figure 2.47 is similar to that of the model in figure 2.45, except that the scale of the output is different, reflecting the fact that the output is an average rather than a sum.

The IterateOverArray actor also supports dropping into it an actor by dropping the actor onto its icon. The actor can be either an atomic library actor or a composite actor (although if it is composite actor, it is required to have a director). This mechanism is illustrated in figure 2.48. When an actor is dragged from the library, when it is dragged over the IterateOverArray actor, the icon acquires a white halo, suggesting that if the actor is dropped, it will be dropped into the actor under the cursor, rather

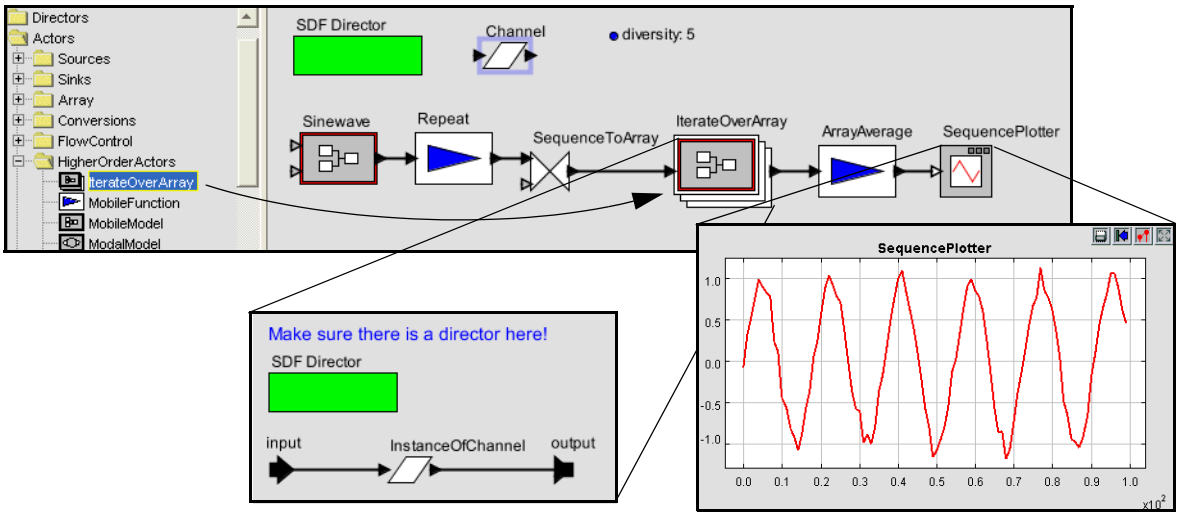

FIGURE 2.47. The IterateOverArray actor can be used to accomplish the same diversity channel model as in figure 2.45, but without creating multiple instances of the channel model. This works because the channel model has no state.

than onto the model containing that actor. When you look inside the IterateOverArray actor after doing this, you will see the class definition. Add an SDFDirector to it before executing it.

2.8.3 Mobile Code

A pair of (still experimental) actors in Ptolemy II support mobile code in two forms. The MobileFunction actor accepts a function in the expression language (see the Expression Language chapter) at one input port and applies that function to data that arrives at the other input port. The MobileModel actor accepts a MoML description of a Ptolemy II model at an input port and then executes that model, streaming data from the other input port through it.

A use of the MobileFunction actor is shown in figure 2.49. In that model, two functions are provided to the MobileFunction in an alternating fashion, one that computes x^2 and the other that computes 2^x. These two functions are provided by two instances of the Const actor, found in Sources, GenericSources. The functions are interleaved by the Commutator actor, from FlowControl, Aggregators.

2.8.4 Lifecycle Management Actors

A few actors in the *HigherOrderActors* library provide in a single firing the entire execution of another Ptolemy II model. The RunCompositeActor actor executes the contained model. The ModelReference actor executes a model that is defined elsewhere in its own file or URL. The VisualModelReference actor opens a Vergil view of a referenced model when it executes a referenced model. These actors generally associate ports (that the user of the actor creates) with parameters of the referenced or contained model. They can be used, for example, to create models that repeatedly run other models with varying parameter values. See the documentation of the actors and the demonstrations in the quick tour for more details.

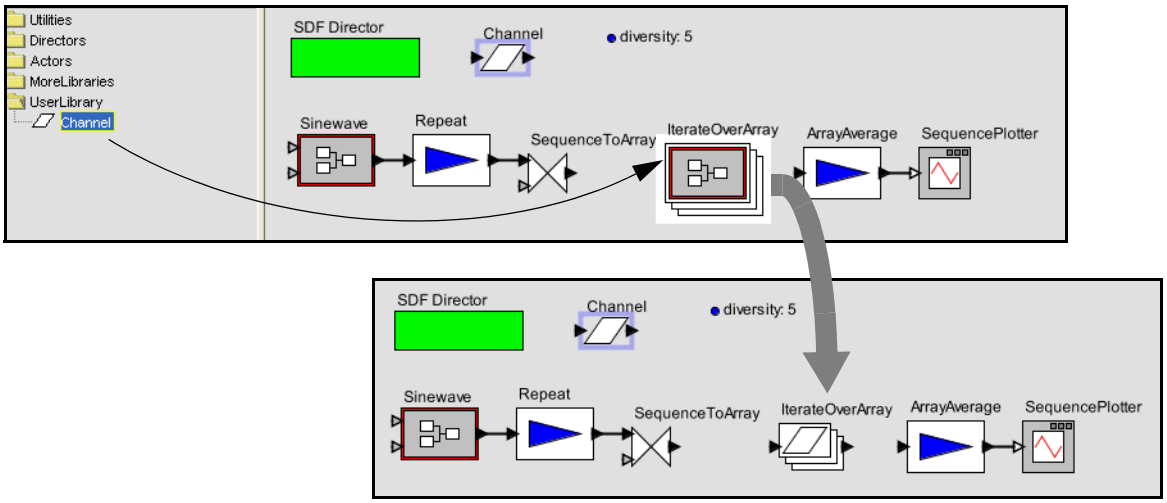

FIGURE 2.48. The IterateOverArray actor supports dropping an actor onto it. When you do this, it transforms to mimic the icon of the actor you dropped onto it, as shown. Here we are using the Channel class that we saved to the UserLibrary as shown in figure 2.44.

2.9 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 *SDF Director* 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. And the system is extensible. You can invent your own. This richness has a downside, however. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are using. Here, we give a very brief introduction to some of the domains. We begin first by explaining some of the subtleties in SDF.

2.9.1 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

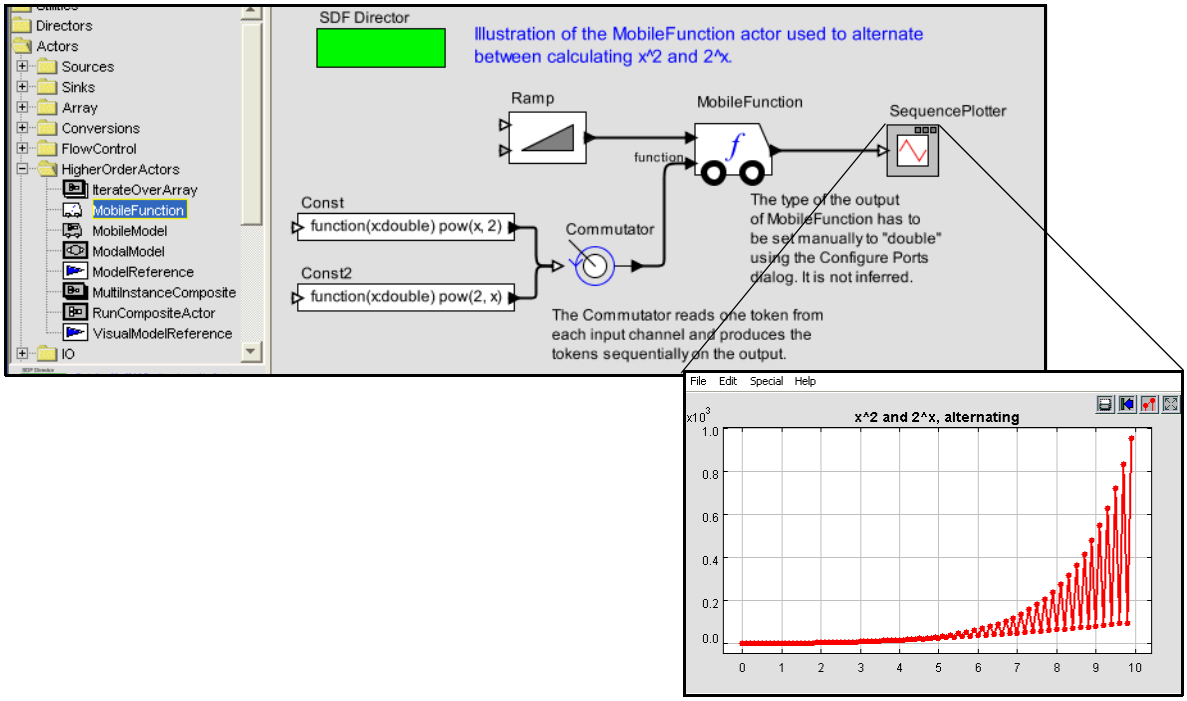

FIGURE 2.49. The MobileFunction actor accepts a function definition at one port and applies it to data that arrives at the other port.

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 2.50 shows a system that computes the spectrum of the same noisy sine wave that we constructed in figure 2.25. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 2.51 shows the output of the model with *order* set to 8 and the number of *iterations* set to 1. **Note that there are 256 output samples output from the *Spectrum* actor**. This is because the *Spectrum* actor requires 2^8, or 256 input samples to fire, and produces 2^8, or 256 output samples when it fires. Thus, one iteration of the 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 in volume 3 for details.

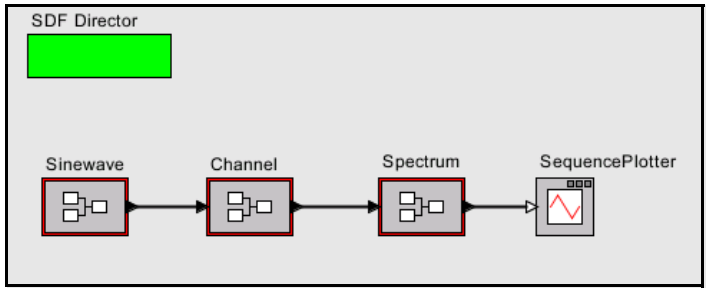

FIGURE 2.50. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave, Channel,* and *SequencePlotter*, and one firing of *Spectrum*.

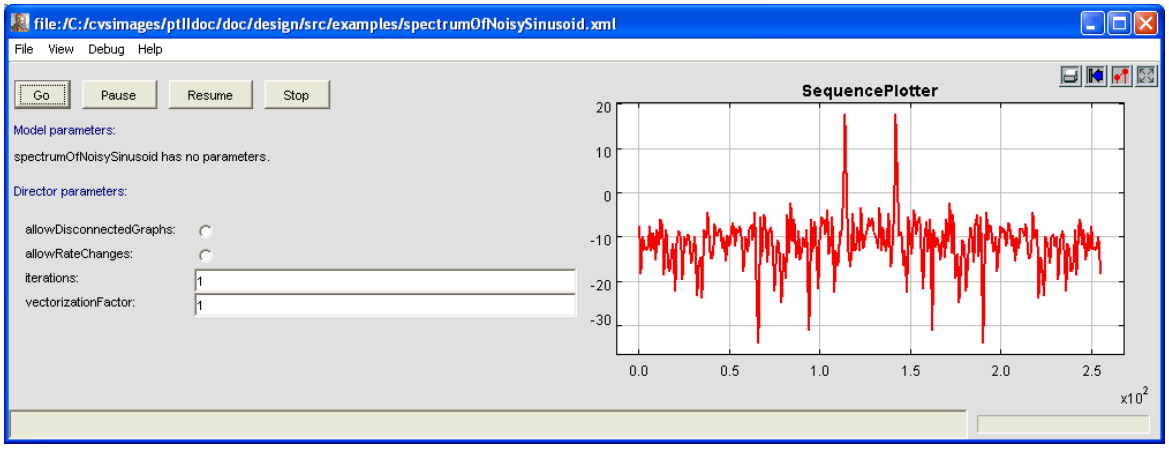

FIGURE 2.51. A single iteration of the SDF model in figure 2.50 produces 256 output tokens.

A second subtlety with SDF models is that if there is a feedback loop, as in figure 2.52, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *FlowControl* library, *SequenceControl* sublibrary). 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 *Sinks* library, *TimedSinks* sublibrary. This is because the SDF domain does not include in its semantics a notion of time. Time does not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.9.2 Data-Dependent Rates

Several domains generalize SDF to support data-dependent rates. The most mature of these is the process networks domain (PN), which associates with each actor its own thread of control. PSDF (parameterized SDF) and HDF (heterochronous dataflow) are more experimental, but are possibly more efficient and formally analyzable than PN. See volume 3 for details about domains.

2.9.3 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.

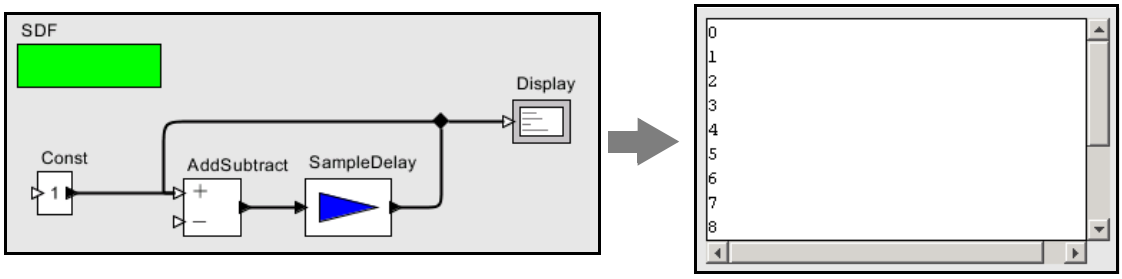

FIGURE 2.52. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

Consider the DE model shown in figure 2.53. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *Sources* library, *TimedSources* sublibrary. 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 produces 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 2.53 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

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

Consider the model shown in figure 2.54, 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

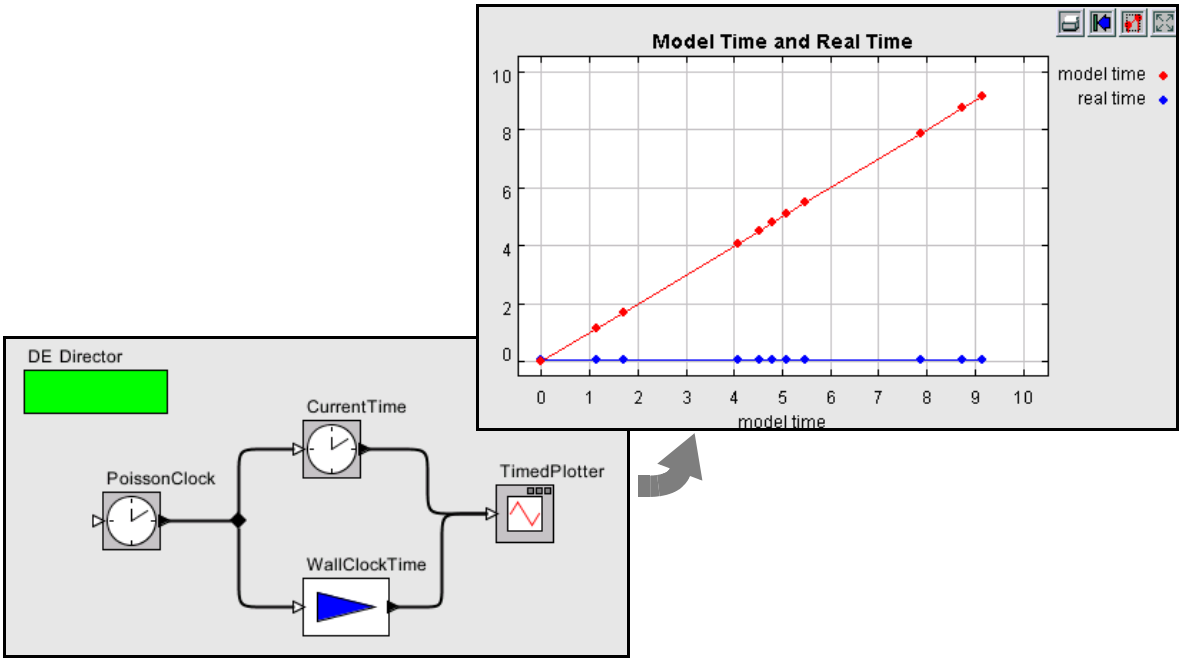

FIGURE 2.53. Model time vs. real time (wall clock time).

(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 at most one token from each channel of the *plus* port, and subtracts at most one token from each channel of 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 2.54, there is only one allowable topological sort:

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

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

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

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *DomainSpecific* library under *DiscreteEvent* (this library is shown on the left in figure 2.55). Consider for example the model shown in figure 2.55. That model has a *Clock* actor, which is set to produce events every 1.0 time units. Those

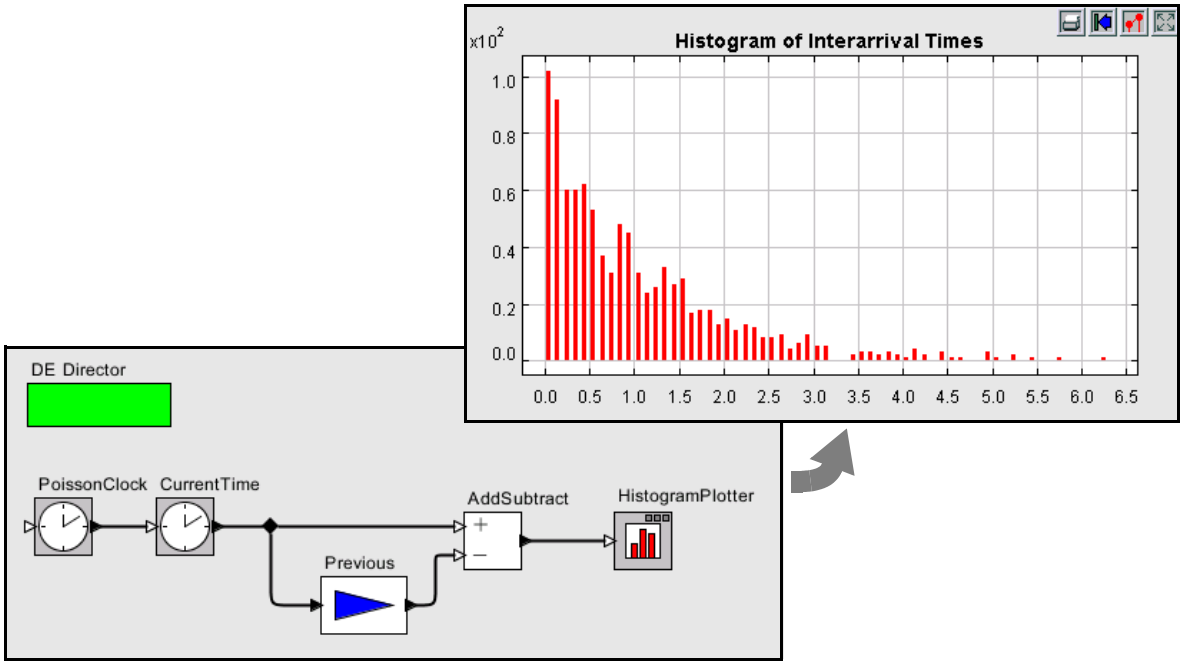

FIGURE 2.54. Histogram of interarrival times, illustrating handling of simultaneous events.

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.

2.9.4 Wireless and Sensor Network Systems

The wireless domain builds on the discrete event domain to support modeling of wireless and sensor network systems. In the wireless domain, channel models mediate communication between actors, and the visual syntax does not require wiring between components. See [10] and [11] for details.

2.9.5 Continuous-Time Systems

The continuous-time domain (CT) is another relatively mature domain with semantics considerably different from either DE or SDF. In CT, the signals sent along connections between actors are usu-

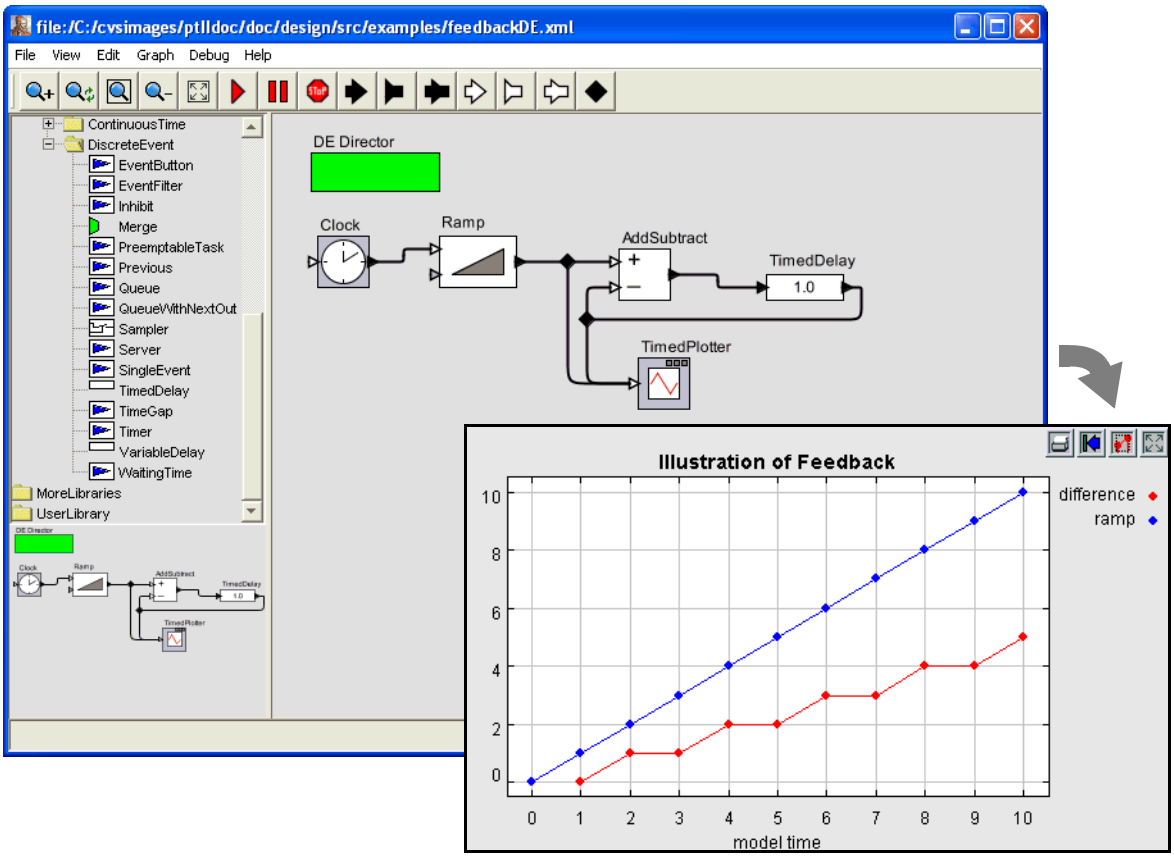

FIGURE 2.55. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

ally continuous-time signals. A CT example is described above in section 2.2.3.

The CT domain can also handle discrete events. These events are usually related to a continuous-time signal, for example representing a zero-crossing of the continuous-time signal. The CT director is quite sophisticated in its handling of such mixed signal systems.

2.10 Hybrid Systems and Modal Models

Hybrid systems are models that combine continuous dynamics with discrete mode changes. They are created in Ptolemy II by creating a *ModalModel*, found in the *HigherOrderActors* library. We start by examining a pre-built modal model, and conclude by illustrating how to construct one. Modal models can be constructed with other domains besides CT, but this section will concentrate on CT. Feel free to examine other examples of modal models given in the quick tour, figure 2.3.

2.10.1 Examining a Pre-Built Model

Consider the bouncing ball example, which can be found under "Bouncing Ball" in figure 2.3 (in the "Hybrid Systems" entry). The top-level contents of this model is shown in figure 2.56. It contains a *Ball Model*, a *TimedPlotter*, and *PeriodicSampler*, and an *Animate Ball* composite actor. The *Ball Model* is an instance of the *ModalModel* found in the *HigherOrderActors* library, but renamed. If you execute the model, you should see a plot like that in the figure and a 3-D animation that is constructed using the GR (graphics) domain. 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 2.57. Figure 2.57 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the left. The circles in figure 2.57 are states, and the arcs between circles are *transitions* between states. A

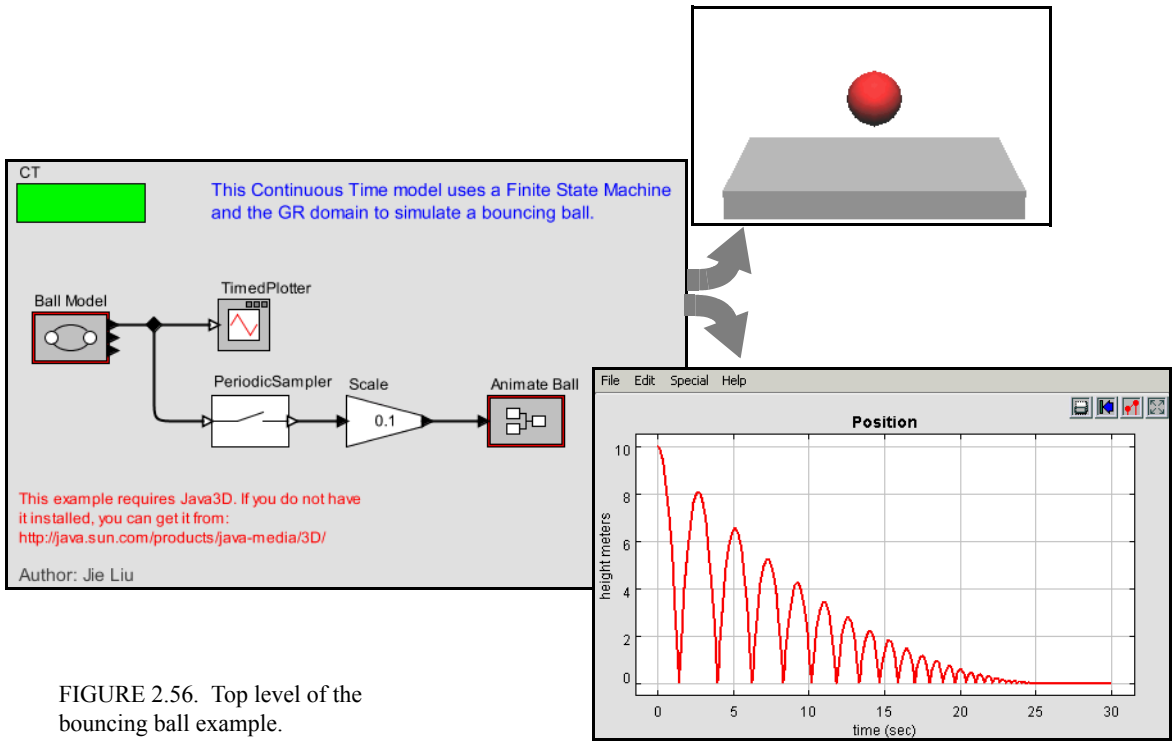

FIGURE 2.56. Top level of the bouncing ball example.

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 2.57 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 2.58. 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.

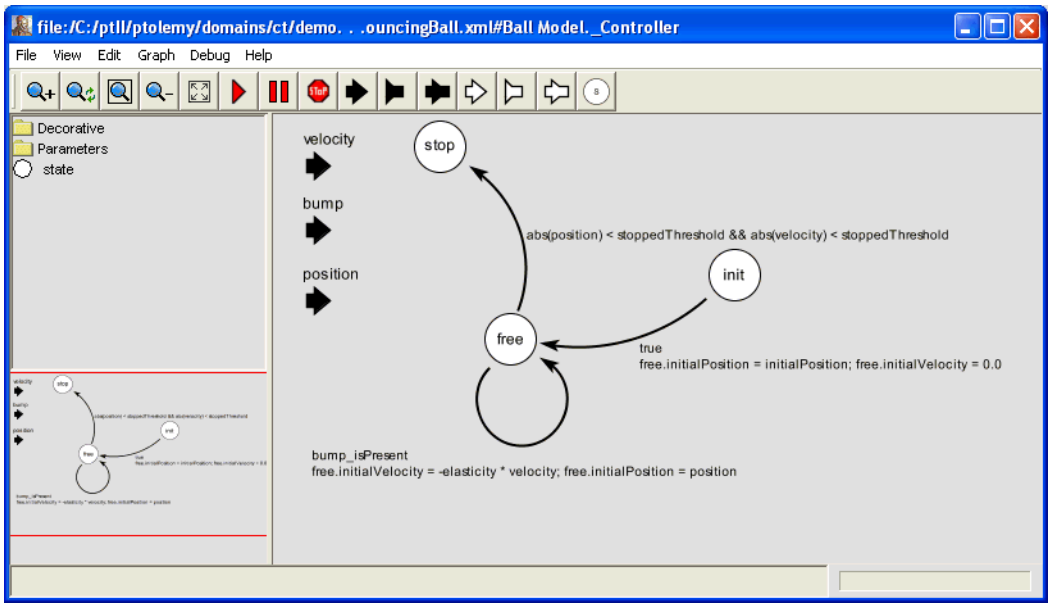

FIGURE 2.57. Inside the *Ball Model* of figure 2.56.

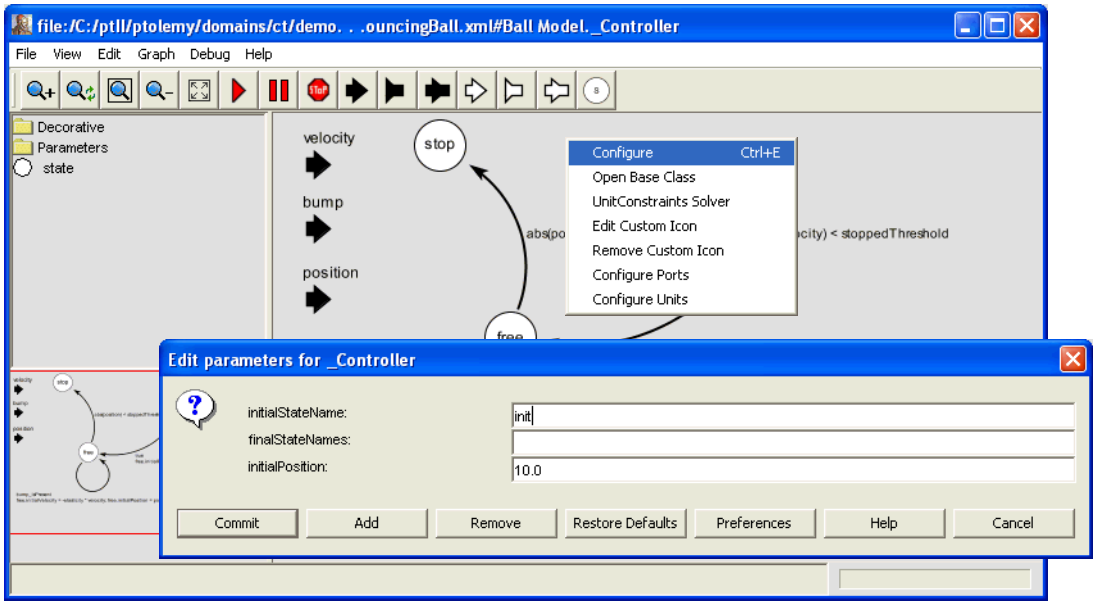

FIGURE 2.58. The initial state of a state machine is set by right clicking on the background and specifying the state name.

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 2.59, the *init* state has no refinement.

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

```
true
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*.

If you look inside the *free* state, you will see the refinement shown in figure 2.60. 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 2.60, 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 2.57, 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 2.56.

As you can see from figure 2.57, 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.

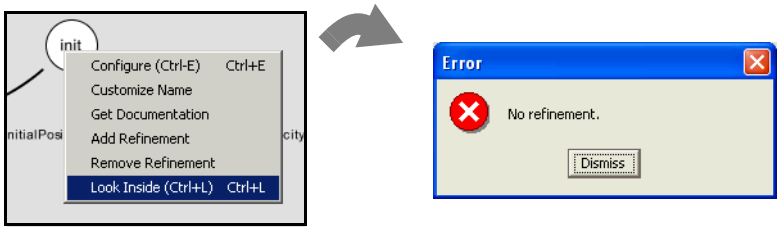

FIGURE 2.59. A state may or may not have a refinement, which specified the behavior of the model while the model is in that state. In this case, *init* has no refinement.

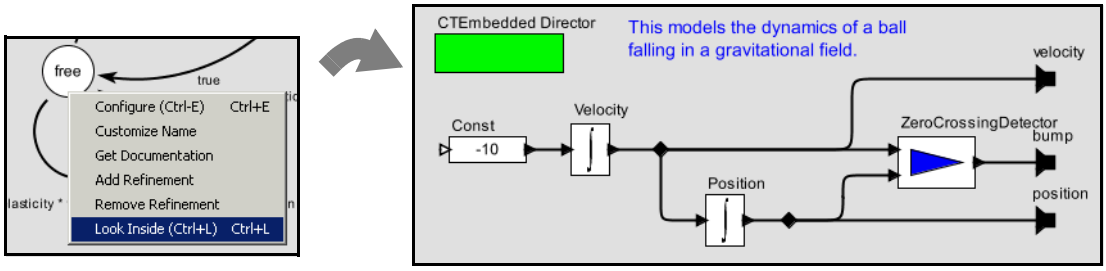

FIGURE 2.60. The refinement of the *free* state, shown here, is a continuous-model representing the laws of gravity.

2.10.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 2.56 and 2.57 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 re-run the model, you get the result shown in figure 2.61. 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.

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.

2.10.3 Constructing Modal Models

A modal model is a component in a larger continuous-time (or other kind of) model. You can create a modal model by dragging one in from the *HigherOrderActors* 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 2.4.2. Figure 2.56 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.

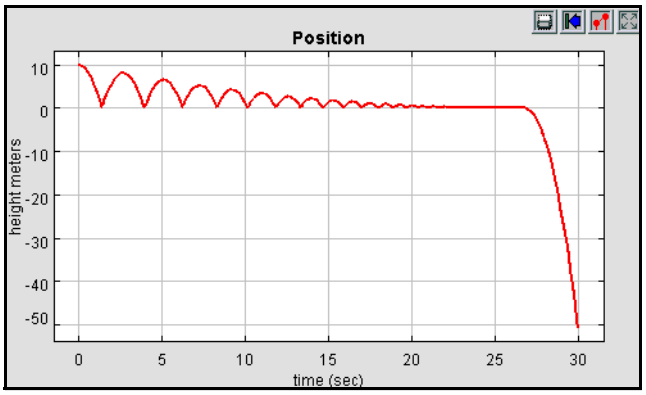

FIGURE 2.61. Result of running the bouncing ball model without the *stop* state.

If you create a new modal model by dragging it in from the *HigherOrderActors* library, create an output port and name it *output*, and then look inside, you will get an FSM editor like that shown in figure 2.62. Note that the output port is (regrettably) located at the upper left, and is only partially visible. The annotation text suggests that you delete it 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 2.57, 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 2.57, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name, as shown in figure 2.58. In that figure, the initial state is named *init*.

Creating Transitions. To create transitions, you must hold the control button[1] 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 2.63. In that dialog, we see the following:

- The guard expression is *true*, so this transition is always enabled. The transition will be taken as

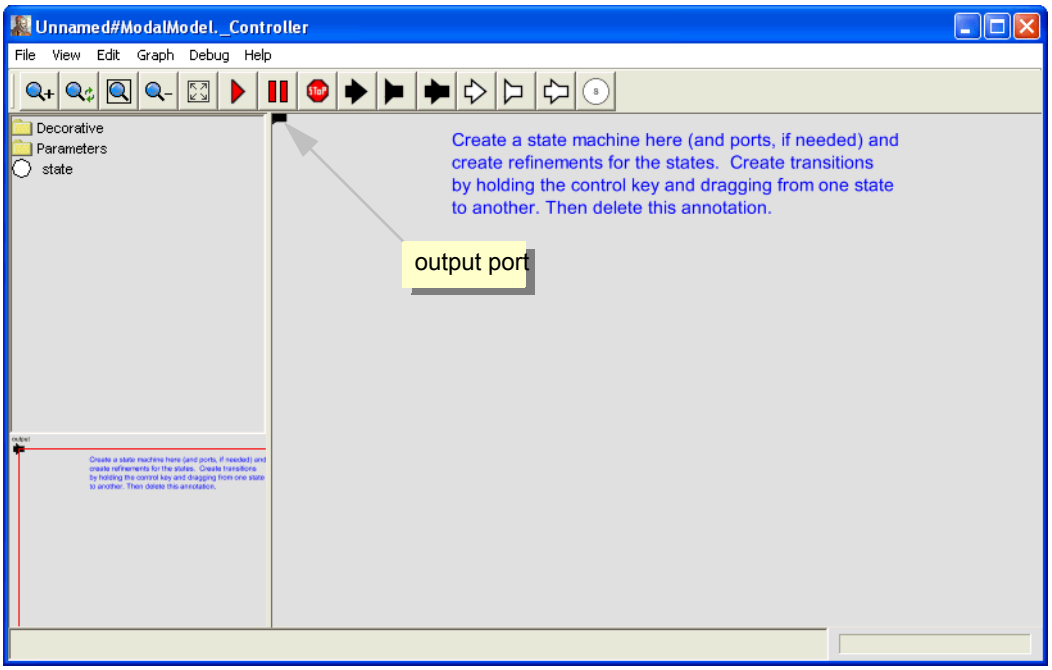

FIGURE 2.62. Inside of a new modal model that has had a single output port added.

1. Or the command button on a Macintosh computer.

soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.

- The output actions are empty, meaning that when this transition is taken, no output is specified. This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.
- The set actions 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, Ptolemy II does not allow nondeterministic state machines, and will throw an exception if it encounters one.

Creating Refinements. Both states and transitions can have *refinements*. To create a refinement, right click[1] on the state or transition, and select "Add Refinement." You will see a dialog like that in figure 2.64. As shown in the figure, you will be offered the alternatives of a "Default Refinement" or a "State Machine Refinement." The first of these provides a block diagram model as the refinement. The second provides another finite state machine as the refinement. In the former case (the default), a 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. You will have to create a director in the refinement. The modal model will not operate without a director in the refinement.

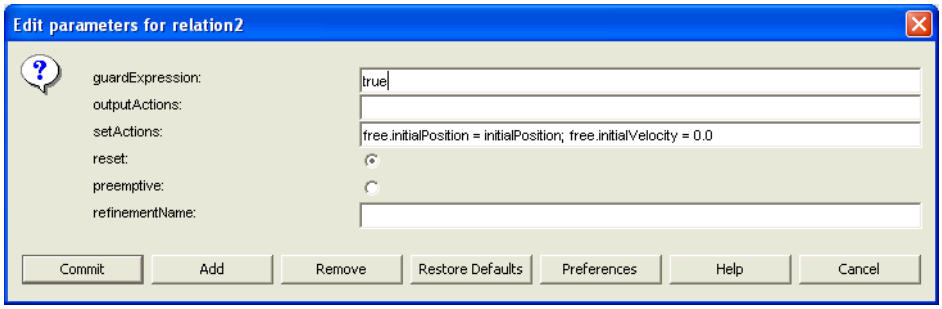

FIGURE 2.63. Transition dialog for the transition from *init* to *free* in figure 2.57.

1. On a Macintosh, control-click.

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, only certain directors make sense in such refinements. The most commonly useful is the SDF director. Such refinements 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 2.60. This model exhibits certain key properties of refinements:

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

2.10.4 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

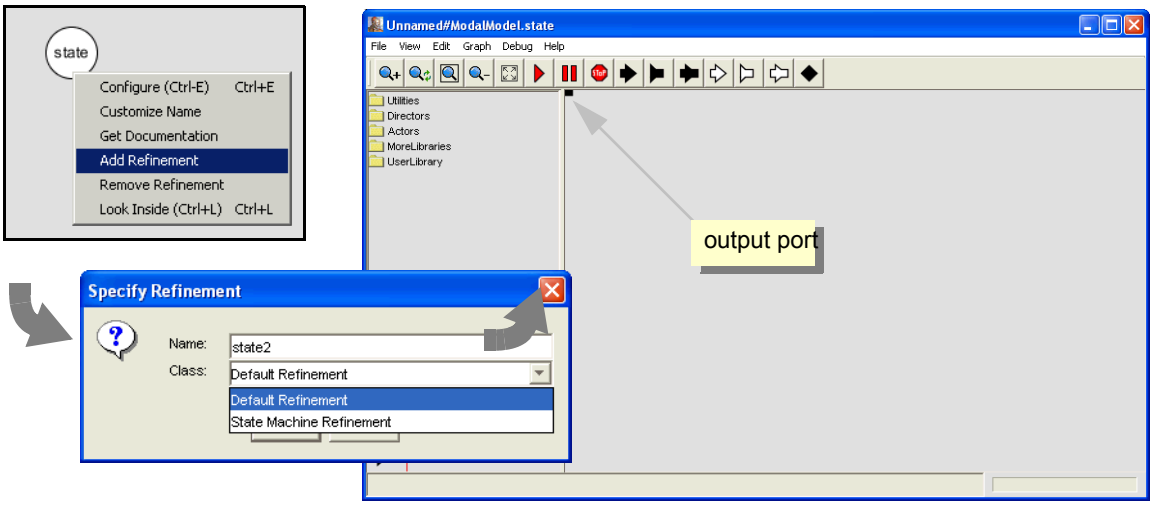

FIGURE 2.64. Adding a refinement to a state.

"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.

2.11 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 2.51 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces 256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

The *SequencePlotter* actor has some pertinent parameters, shown in figure 2.65. 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 2.66.

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 2.67, filled in with values that result in the plot shown in figure 2.68. 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"

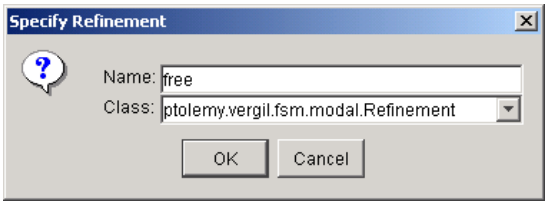

FIGURE 2.65. Dialog for creating a refinement of a state.

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

- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by

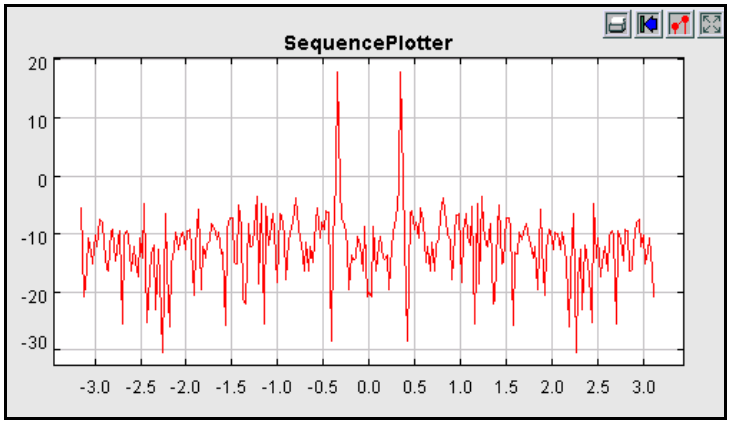

FIGURE 2.66. Better labeled plot, where the horizontal axis now properly represents the frequency values.

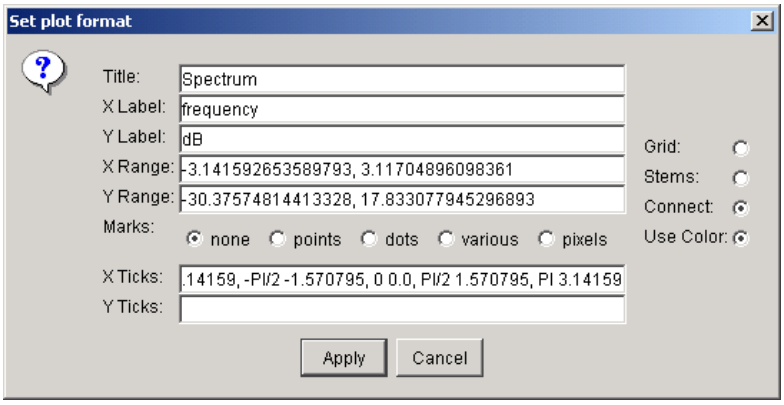

FIGURE 2.67. Format control window for a plot.

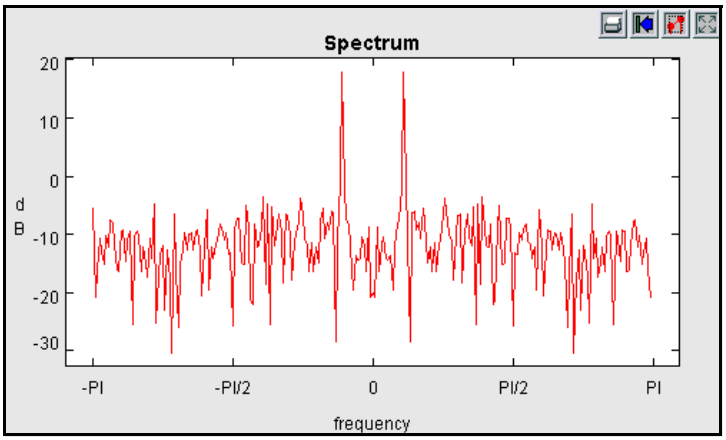

FIGURE 2.68. Still better labeled plot.

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.