The Ptiny Ptolemy Manual

The Ptolemy Pteam
University of California, Berkeley

October 12, 2010

The Ptiny Ptolemy Manual

Heterogeneous Concurrent Modeling and Design in Java

Edited by: Christopher X. Brooks

Authors: Christopher X. Brooks, Thomas Huining Feng, Shanna-Shaye Forbes, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Neil Smyth and Yuhong Xiong

Sponsors: This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MySyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Copyright ©1998-2010 The Regents of the University of California. All rights reserved.

Permission is hereby granted, without written agreement and without license or royalty fees, to use, copy, modify, and distribute this software and its documentation for any purpose, provided that the above copyright notice and the following two paragraphs appear in all copies of this software.

N NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN ”AS IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

All trademarks are property of their respective owners.

Ptolemy II [8] is an open-source software framework supporting experimentation with actor-oriented design. Actors are software components that execute concurrently and communicate through messages sent via interconnected ports. A model is a hierarchical interconnection of actors. In Ptolemy II, the semantics of a model is not determined by the framework, but rather by a software component in the model called a director, which implements a model of computation. The Ptolemy Project has developed directors supporting process networks (PN), discrete-events (DE), dataflow (SDF)[11], synchronous/reactive(SR), rendezvous-based models, 3-D visualization, and continuous-time models. Each level of the hierarchy in a model can have its own director, and distinct directors can be composed hierarchically. A major emphasis of the project has been on understanding the heterogeneous combinations of models of computation realized by these directors. Directors can be combined hierarchically with state machines to make modal models[9]. A hierarchical combination of continuous-time models with state machines yields hybrid systems [13]; a combination of synchronous/reactive with state machines yields StateCharts[14] (the Ptolemy II variant is close to SyncCharts).

Ptolemy II has been under development since 1996; it is a successor to Ptolemy Classic, which was developed since 1990. The core of Ptolemy II is a collection of Java classes and packages, layered to provide increasingly specific capabilities. The kernel supports an abstract syntax, a hierarchical structure of entities with ports and interconnections. A graphical editor called Vergil supports visual editing of this abstract syntax. An XML concrete syntax called MoML provides a persistent file format for the models. Various specialized tools have been created from this framework, including HyVisual[3] (for hybrid systems modeling), Kepler[16] (for scientific workflows), VisualSense[1][2] (for modeling and simulation of wireless networks), Viptos[7] (for sensor network design), and some commercial products. Key parts of the infrastructure include an actor abstract semantics, which enables the interoperability of distinct models of computation with a well-defined semantics; a model of time (specifically, super-dense time, which enables interaction of continuous dynamics and imperative logic); and a sophisticated type system supporting type checking, type inference, and polymorphism. The type system has recently been extended to support user-defined ontologies[15]. Various experiments with synthesis of implementation code and abstractions for verification are included in the project.

This volume updates Volume 1[4] of the Ptolemy II design document and describes how to construct Ptolemy II models for web-based modeling or building applications. The first chapter is a tutorial on building models using Vergil, a graphical user interface where models are built pictorially. The second chapter discusses the Ptolemy II expression language, which is used to set parameter values. The third chapter describes the Ptolemy coding style. See volume 1 for an overview of Ptolemy and information about creating actors.

Volume 2[5] describes the software architecture of Ptolemy II, and volume 3[6] describes the domains, each of which implements a model of computation.

This document is the work of many people who have updated the chapters since 1999.

Contents

Chapter 1

Using Vergil

Authors: Edward A. Lee, Steve Neuendorffer
Contributors: Christopher Brooks, Shanna-Shaye Forbes

1.1 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 1.1 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, or in XML. Moreover, only some of the models of computations (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.

1.2 Quick Start

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

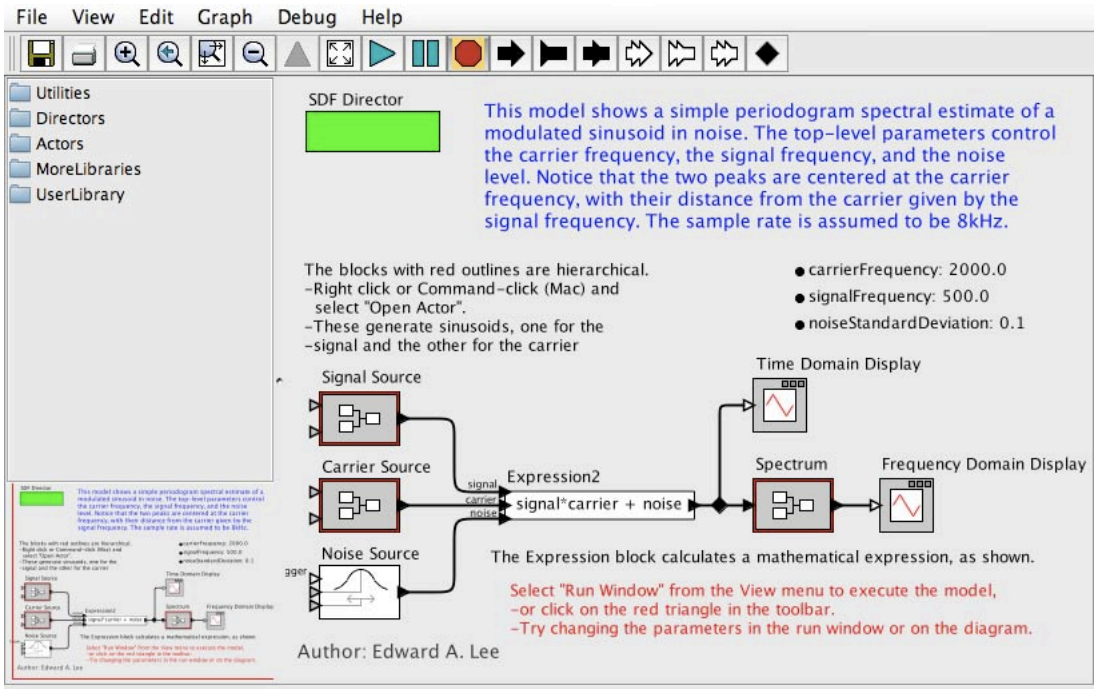

Figure 1.1: Example of a Vergil window.

1.2.1 Starting Vergil

First, go to http://ptolemy.org/ptolemyII/ptIIlatest and download and install Ptolemy II. Then, start Vergil. Choose one of the methods below:

- From the command line, enter `vergil`, or, if your path is not set `$PTII/bin/vergil`.
- Under Windows, click "Start" | "Ptolemy II" | *your version of Ptolemy* | "Vergil"[1]
- Under Mac OS X, click "Applications" | "Ptolemy" | *your version of Ptolemy* | "bin" | "Vergil.app"
- Click on a Web Start link on a web page supporting the web edition. See http://ptolemy.org/ptolemyII/ptIIlatest/webstart.htm

You should see an initial welcome window that looks like the one in figure 1.2. Feel free to explore the links in this window. The "Tour of Ptolemy II" link takes you to the page shown in figure 1.3.

[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).

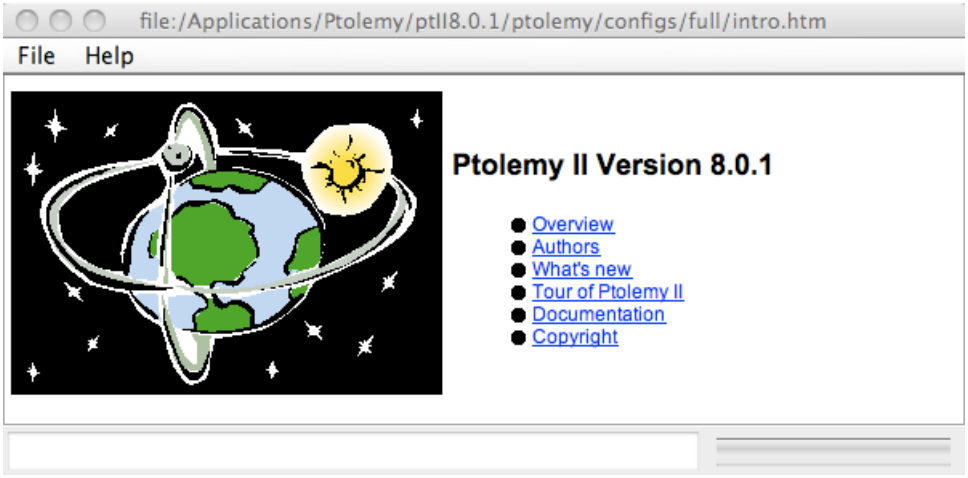

Figure 1.2: Initial welcome window.

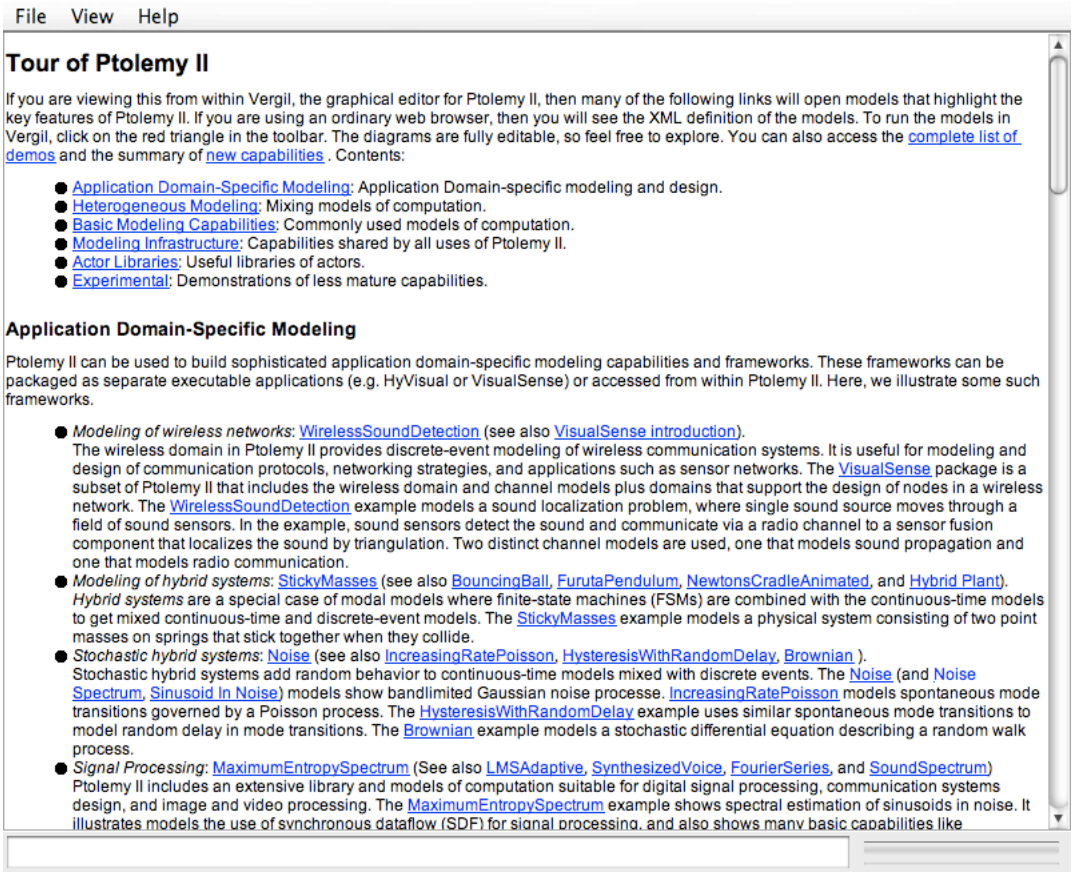

Figure 1.3: The tour of Ptolemy II page.

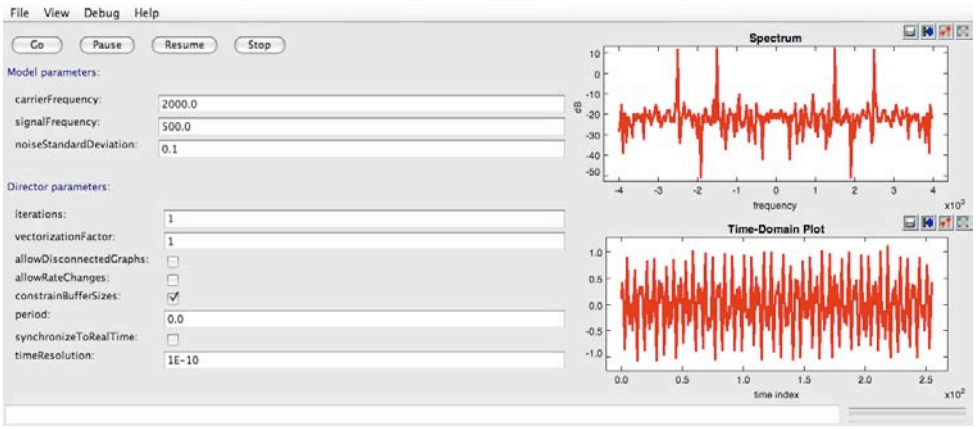

Figure 1.4: The Run Window for the model shown in figure 1.1.

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

The first example (Spectrum) listed under "Basic Modeling Capabilities" on the tour page is the model shown in figure 1.1. 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 1.4. 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 1.1 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 1.1. 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 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 1.1 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 invoke the "Open Actor" context menu choice to reveal the implementation, as shown in figure 1.5, which shows the implementation of the Signal Source in figure 1.1. It is evident from the block diagram how a sinusoidal signal is generated.

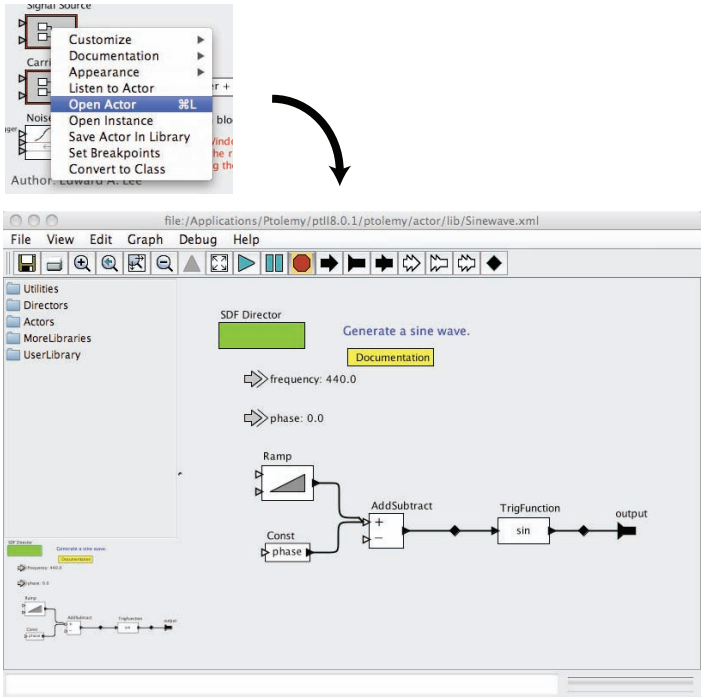

Figure 1.5: Invoke Open Actor on composite actors to reveal their implementation.

1.2.3 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 1.1 and 1.5 specifies that these block diagrams have *synchronous dataflow semantics* [11], which is explained further below. The second example under "Basic Modeling Capabilities" in the Tour of Ptolemy II (figure 1.3), 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.

The Lorenz attractor model, shown in figure 1.6, is a block diagram representation of a set of nonlinear ordinary differential equations. The blocks with integration signs in their icons are integrators. At any given time t, their output is given by

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau) d\tau, \tag{1.1}$$

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

$$\dot{x}(t) = \frac{d}{dt} x(t). \tag{1.2}$$

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 1.6 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 1.6 are

$$\begin{aligned} \dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t) \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t) \end{aligned} \tag{1.3}$$

For each equation, the expression on the right is implemented by an *Expression* actor, whose icon shows the expression. Each expression refers to parameters (such as *lambda* for λ and *sigma* for σ) and input ports of the actor (such as x1 for x_1 and x2 for x_2). The names of the input ports are not shown in the diagram, but if you linger over them with the mouse cursor, the name will pop up in a tooltip. The expression in each Expression actor can be edited by double clicking on the actor, and the parameter values can be edited by double clicking on the parameters, which are shown next to bullets on the right.

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

The *Continuous Director*, shown at the upper left, 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 1.7. 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 1.8. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

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.

1.2.4 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 1.9. 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.

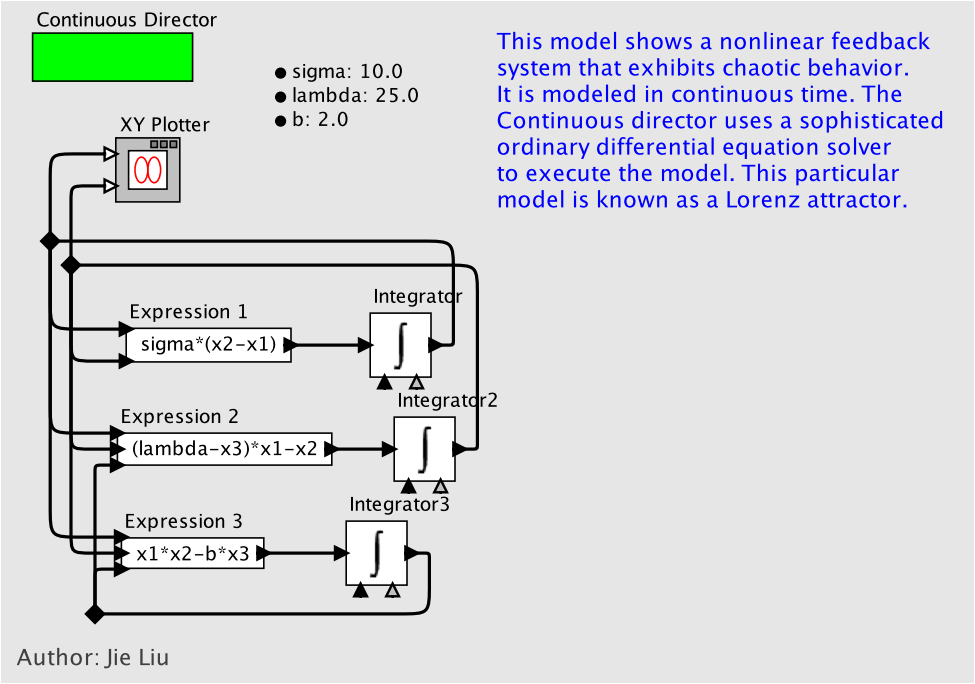

Figure 1.6: A block diagram representation of a set of nonlinear ordinary differential equations.

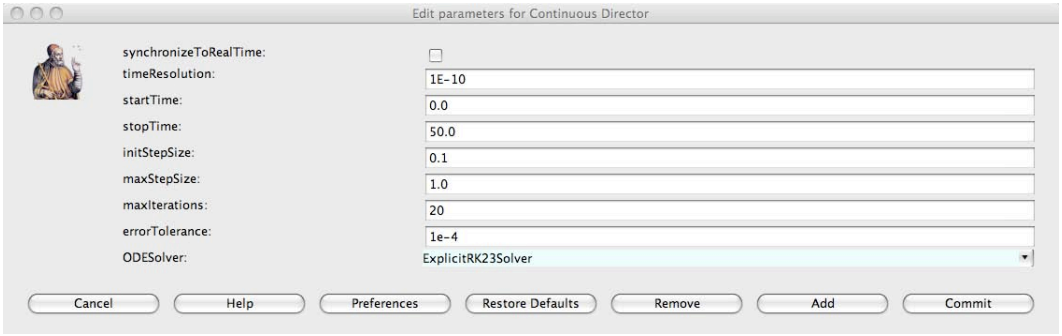

Figure 1.7: Dialog box showing solver parameters for the model in figure 1.6.

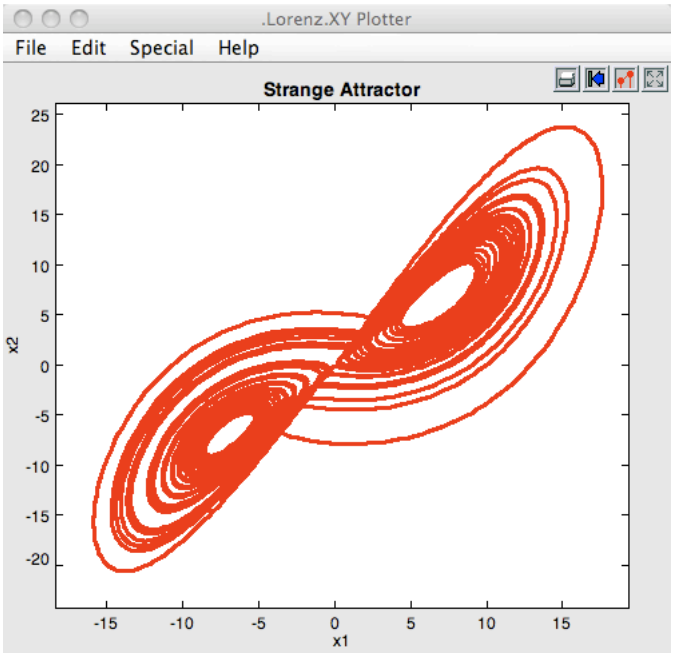

Figure 1.8: Result of running the Lorenz model using the run button in the toolbar.

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 1.10. 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 1.10: 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[2] on the *Const* actor icon and select "Customize" |"Configure". You should see the dialog box in figure 1.11. 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]On a Macintosh, which typically has only one mouse button, instead of right clicking, hold the control key and click the one button.

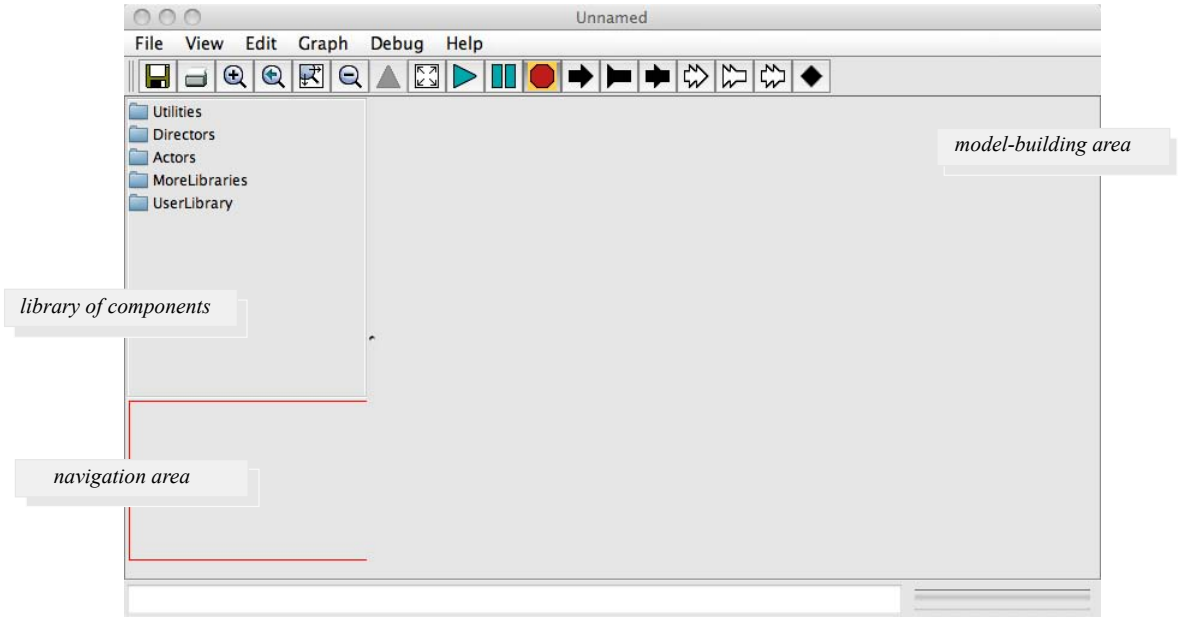

Figure 1.9: An empty Vergil Graph Editor.

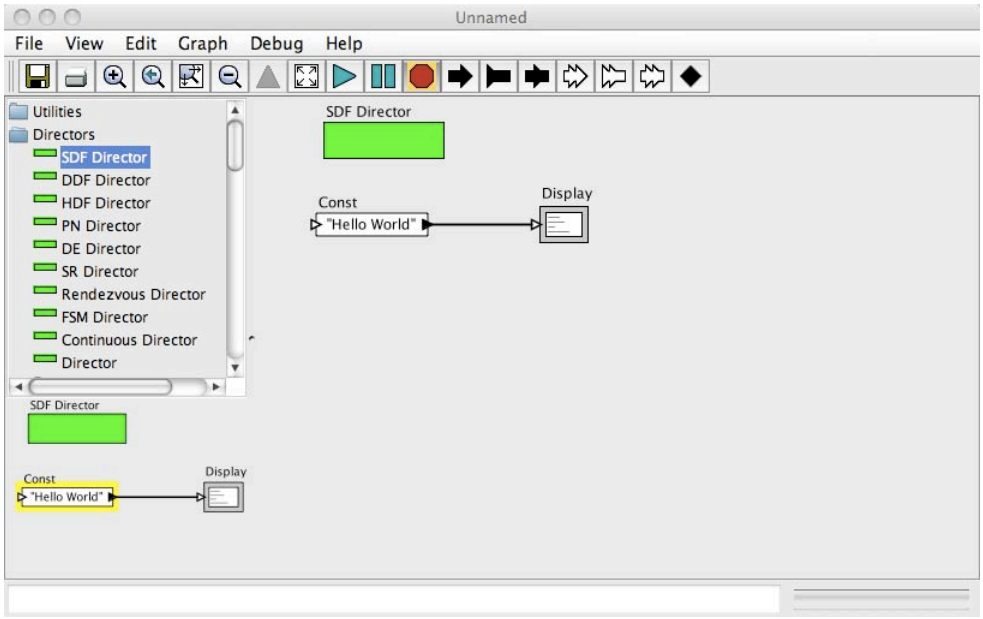

Figure 1.10: The Hello World example.

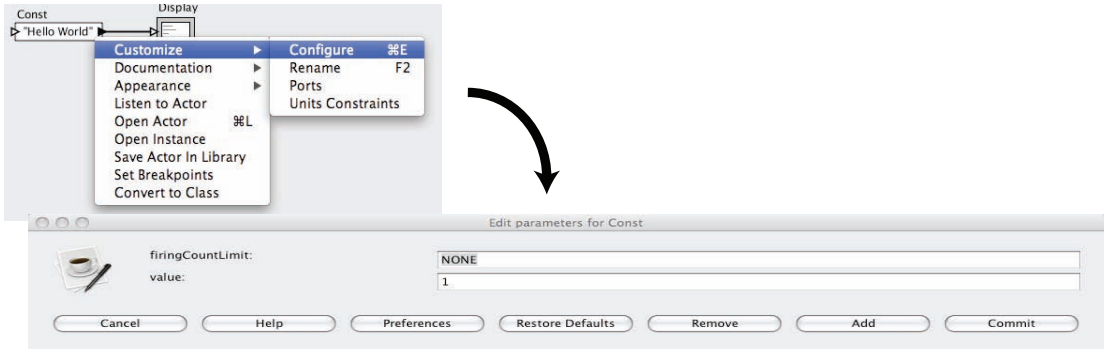

Figure 1.11: The Const parameter editor.

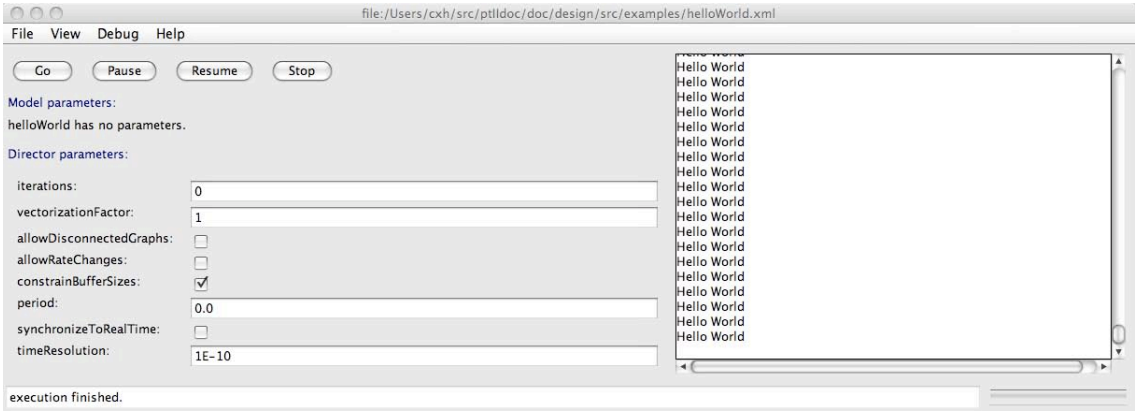

Figure 1.12: Execution of the Hello World example.

1.2.5 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 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 1.12.

1.2.6 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*

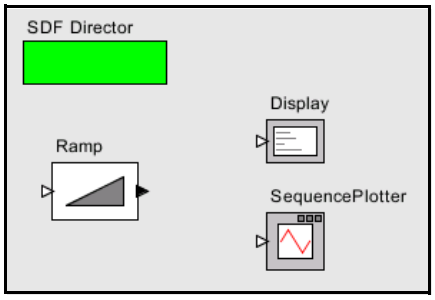

Figure 1.13: Three unconnected actors in a model.

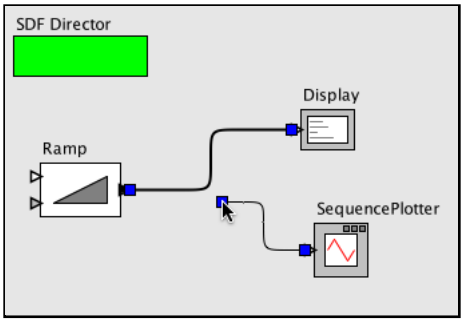

Figure 1.14: Three actors connected using a relation.

library, a *Display* actor, and a *SequencePlotter* actor, found in the *Sinks* library, as shown in figure 1.13. Suppose we wish to route the output of the *Ramp* to both the *Display* and the *SequencePlotter*.

1. Click on the output port of the *Ramp* actor and drag the mouse to the input port of the *Display* actor. Note that the order does not matter, clicking first on the input of the *Display* and then dragging to the output of the *Ramp* actor will also work.

2. Click on the input of the *SequencePlotter* actor and drag to the link between the *Ramp* and the *Display* actor.

3. A *relation*, signified by a black diamond will appear and all three actors will be connected.

What happened is that the user interface detected that a relation was needed to make the connection. 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.

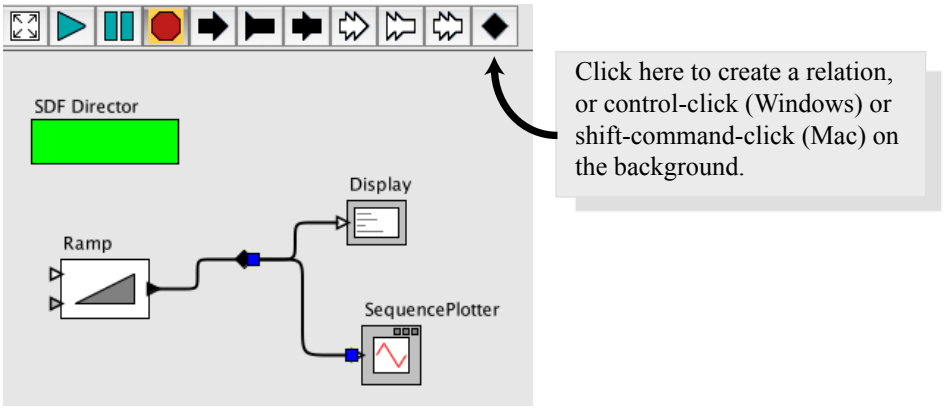

Figure 1.15: A relation can be used to broadcast an output from a single port.

A relation is represented in the diagram by a black diamond, as shown in figure 1.15. It can be created by any one of the ways below:

- Dragging the endpoint of a link into the middle of an existing ling
- Control-clicking[3] 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 key while clicking and dragging on the relation.[4]

In the model shown in figure 1.15, 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. Relations may be linked to other relations. Any two relations that are linked are said to be members of the same *relation group*. Semantically, a relation group has the same meaning as a single relation. In a relation group, there is no significance to the order in which relations are linked.

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.

[3] On a Macintosh, shift-command-click.
[4] On a Macintosh, hold the command key rather than the control key.

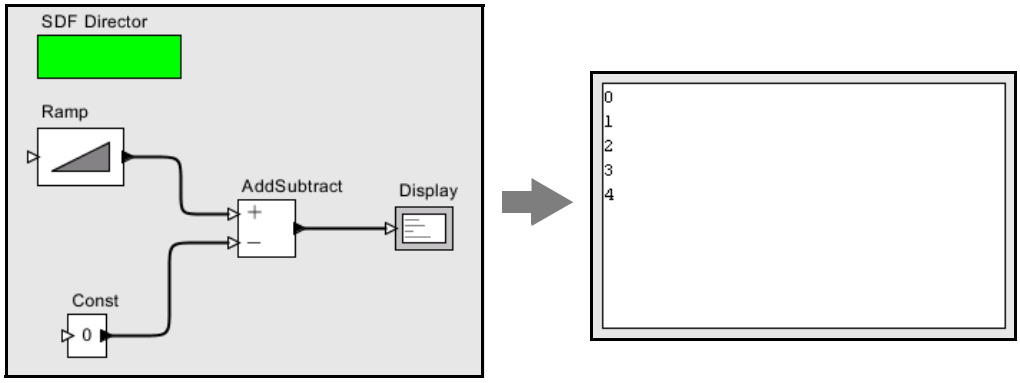

Figure 1.16: Another example, used to explore data types in Ptolemy II.

1.3 Tokens and Data Types

In the example of figure 1.10, 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 2, ”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*[18], 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 1.16. The *Ramp* actor is listed under the *Sources* library, *SequenceSources* sublibrary, and the *AddSubtract* actor is listed under the *Math* sublibrary. Set the *value* parameter of the *Const* 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

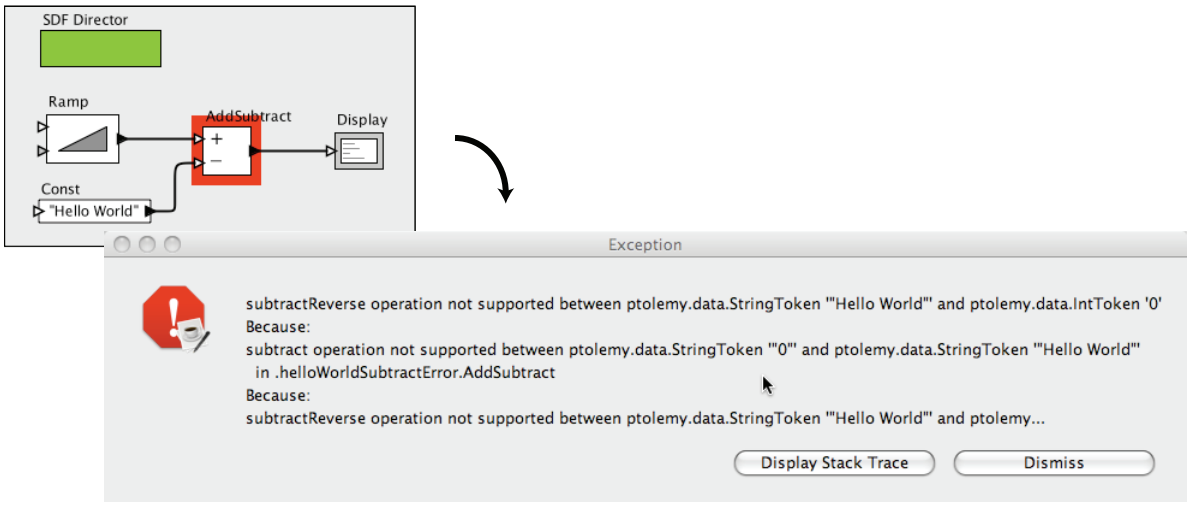

Figure 1.17: An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.

the model, you should see an exception window, as shown in figure 1.17. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a string value from an integer value, which doesn't make much sense at all (following Java, adding strings is allowed). This is an example of a type error.

The actor that caused the exception is highlighted and the name of the actor, ".helloWorldSubtractError.AddSubtract" is mentioned in the exception. In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".helloWorldSubtractError.AddSubtract" is an object named "AddSubtract" contained by a object named ".helloWorldAddSubtractError". This assumes that the model was saved as a file called `helloWorldAddSubtract.xml`.

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 1.17. You should see the stack trace shown in figure 1.18.

This window displays the execution sequence that resulted in the exception. For example, if you scroll towards the bottom, you will see a line like

```
at ptolemy.data.StringToken._subtract(StringToken.java:359)
```

that indicates that the exception occurred within the subtract() method of the class ptolemy.data.StringToken, at line 359 of the source file StringToken.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.

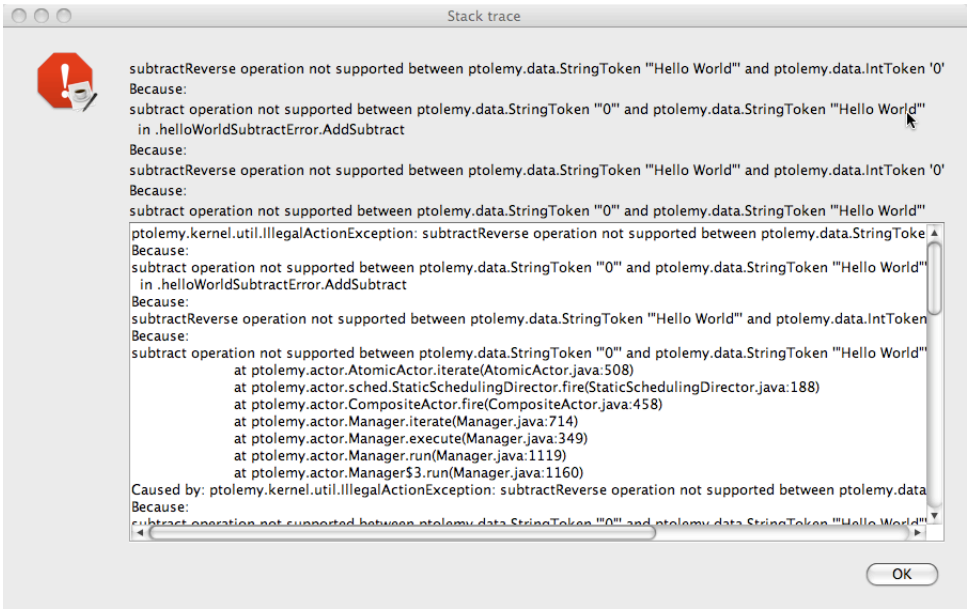

Figure 1.18: Stack trace for the exception shown in figure 1.17

.

To find the file StringToken.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/StringToken.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 1.19. 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

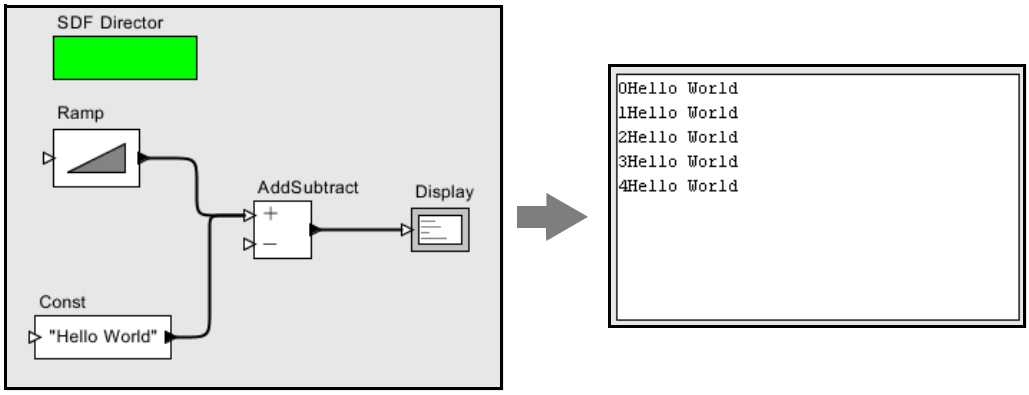

Figure 1.19: Addition of a string to an integer.

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 of the full design doc [5], 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.

1.4 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.

1.4.1 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[5] over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in figure 1.20. (Note that you can alternatively give a display name, which is arbitrary text that will be displayed instead of the name of the actor.) Then, using the context

[5] On a Macintosh, control-click.

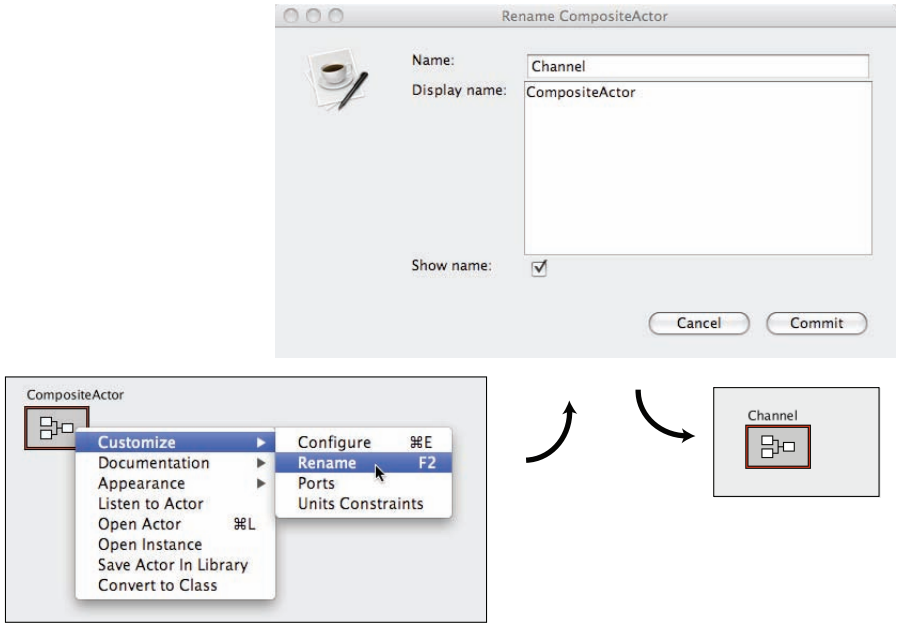

Figure 1.20: Changing the name of an actor

menu again, select "Open Actor" on the actor. You should get a blank graph editor, as shown in figure 1.21. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

The toolbar contains an upward pointing triangle[6] that is used to open the container of a composite entity.

1.4.2 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 1.22. Create an input port and an output port and rename them *input* and *output* by right clicking on the ports and selecting "Customize Name". Note that, as shown in figure 1.23, you can also right click[7] on the background of the composite actor and select "Configure" |"Customize" |"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 determine the type from the connections. You can also specify the *direction* of a port (where it appears on the

[6]The "open container" button is based on an idea by Captain Robbins of the USAF.
[7]On a Macintosh, control-click.

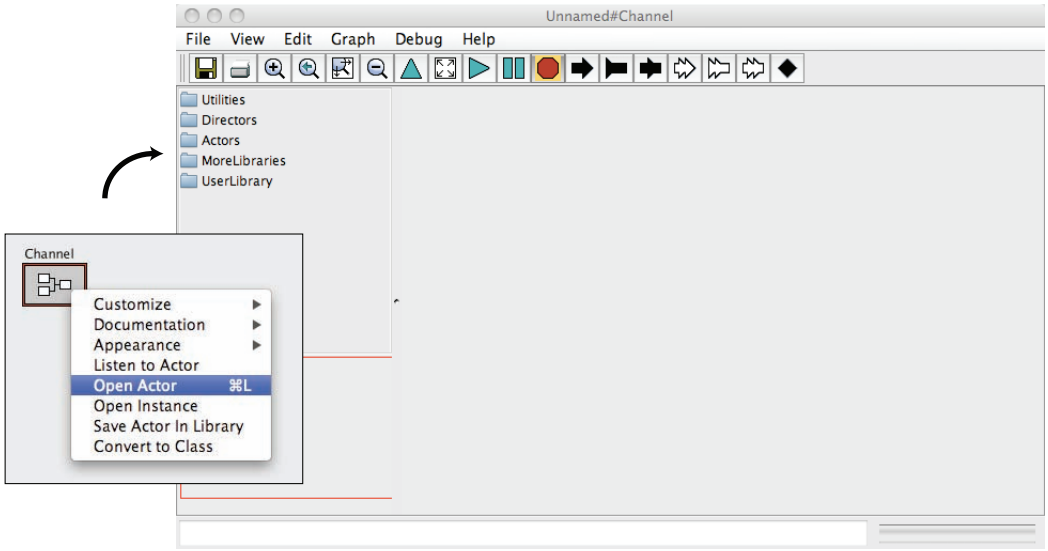

Figure 1.21: Opening a new composite actor which shows the blank inner actor.

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 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.

Then, using these ports, create the diagram shown in figure 1.24[8]. 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

[8]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.

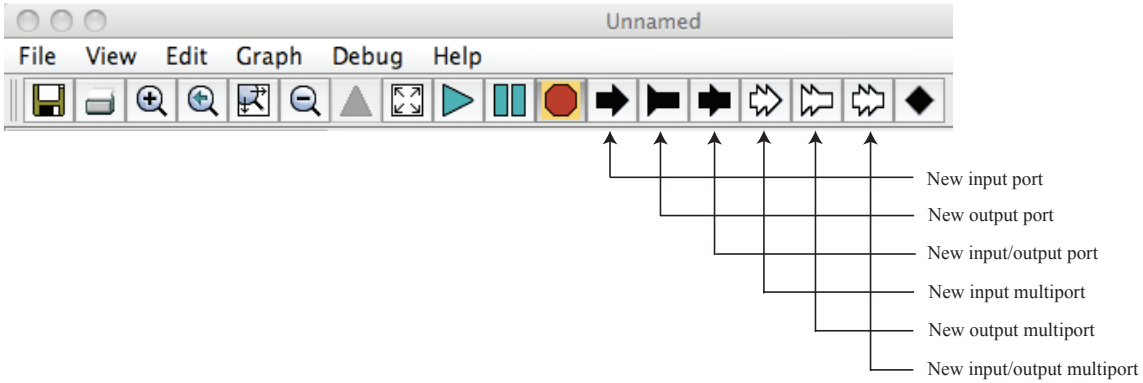

Figure 1.22: Summary of toolbar buttons for creating new ports.

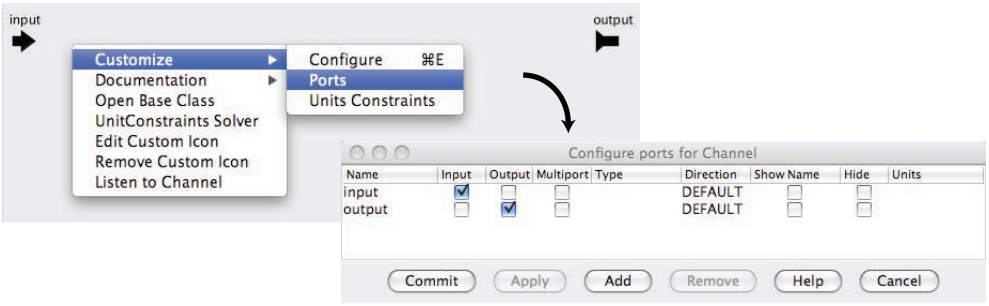

Figure 1.23: Right clicking on the background brings up a dialog that can be used to configure ports.

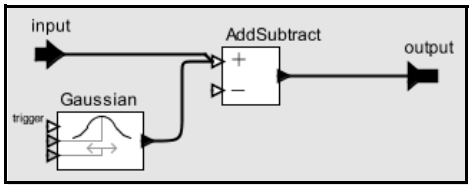

Figure 1.24: A simple channel model defined as a composite actor.

figure 1.25. The *Sinewave* actor is listed under the *Sources* library, and the *SequencePlotter* actor is found under the *Sinks* library. 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 1.26.

1.4.3 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 1.23 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*,

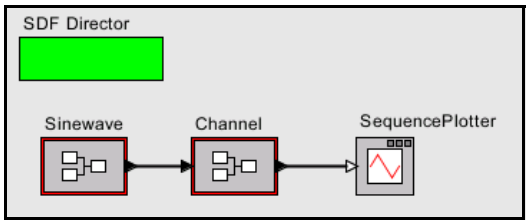

Figure 1.25: A simple signal processing example that adds noise to a sinusoidal signal.

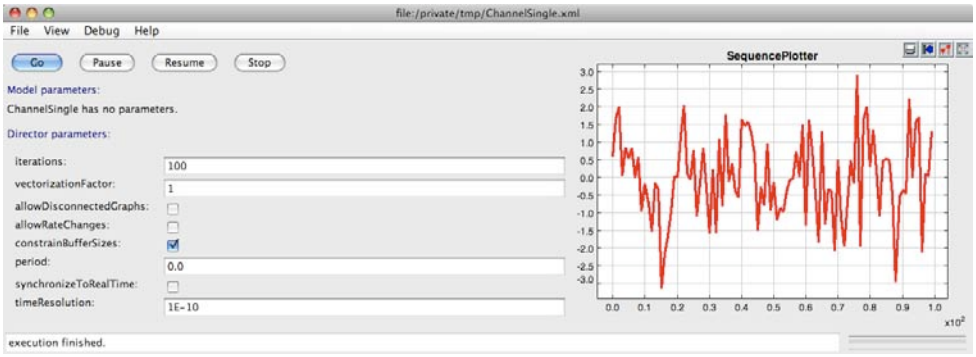

Figure 1.26: The output of the simple signal processing model in figure 1.25.

fixedpoint, *float*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *short*, *string*, *unknown*, *unsignedByte*, *xmlToken*, *arrayType(int)*, *arrayType(int, 5)*, *[double]* and { *x=double, y=double*}, 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}
```

1.5 Annotations and Parameterization

In this section, we will enhance the model in figure 1.25 in a number of ways.

1.5.1 Parameters in Hierarchical Models

First, notice from figure 1.26 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. Open the channel

model by right clicking on the channel model, and select "Open Actor". In the channel model, add a parameter by dragging one in from the *Utilities* library, *Parameters* sublibrary, as shown in figure 1.27. Right click [9] 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 1.28. 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 "Customize" |"Configure"). Change the *noisePower* from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in figure 1.29.

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 "Customize" |"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 invoke "Open Actor" on the *Channel* actor. Instead, you would have to right click on the background and select "Customize" |"Configure" to see the parameter.

1.5.2 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.

1.5.3 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 1.30. 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 1.31. 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,

[9] On a Macintosh, control-click.

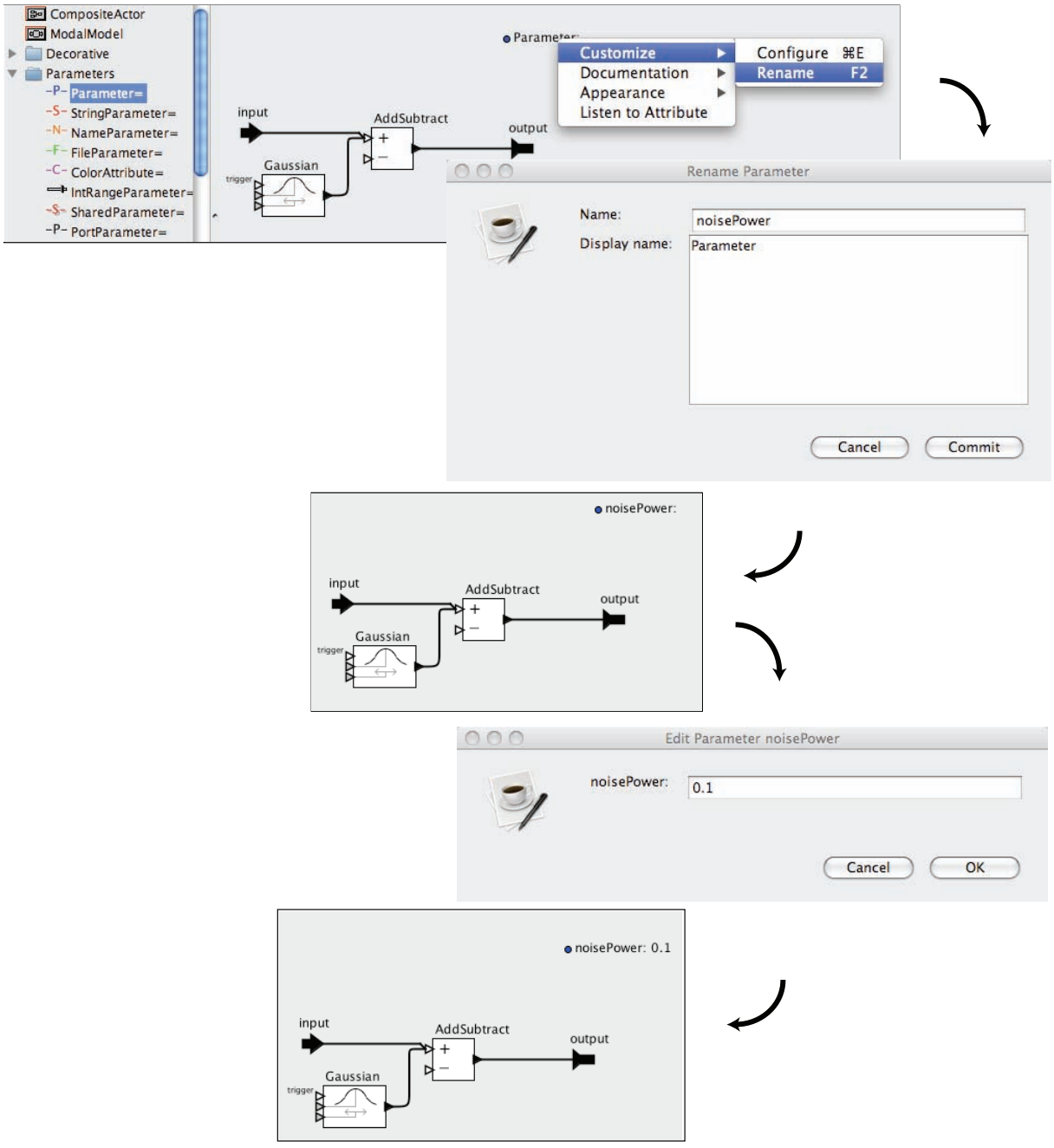

Figure 1.27: Adding a parameter to the channel model.

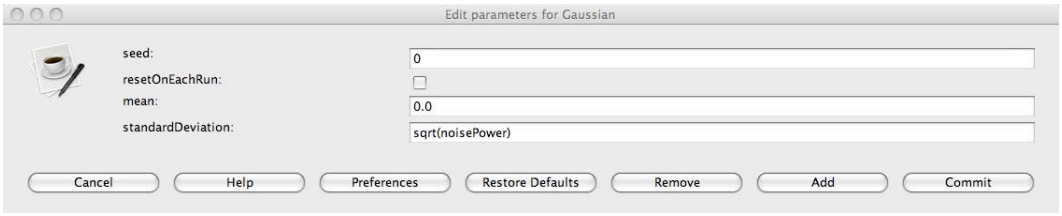

Figure 1.28: The standard deviation of the Gaussian actor is set to the square root of the noise power.

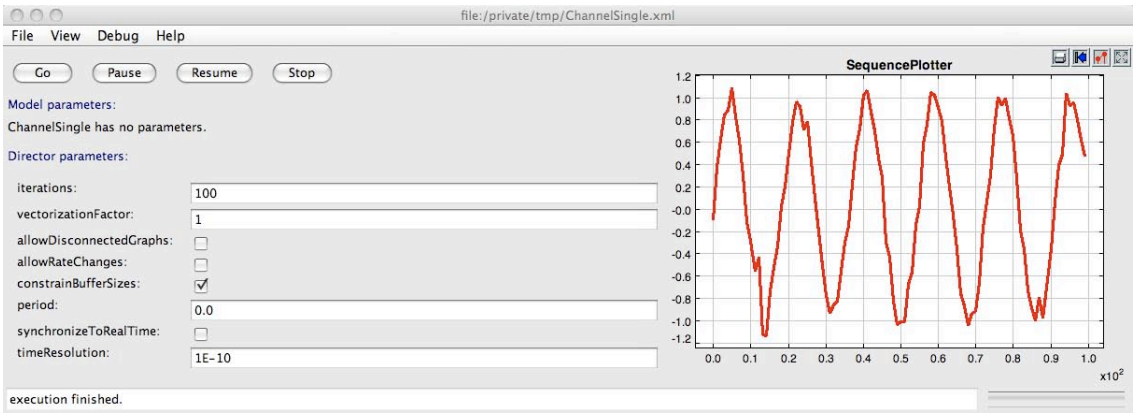

Figure 1.29: The output of the simple signal processing model in figure 1.25 with noise power = 0.01

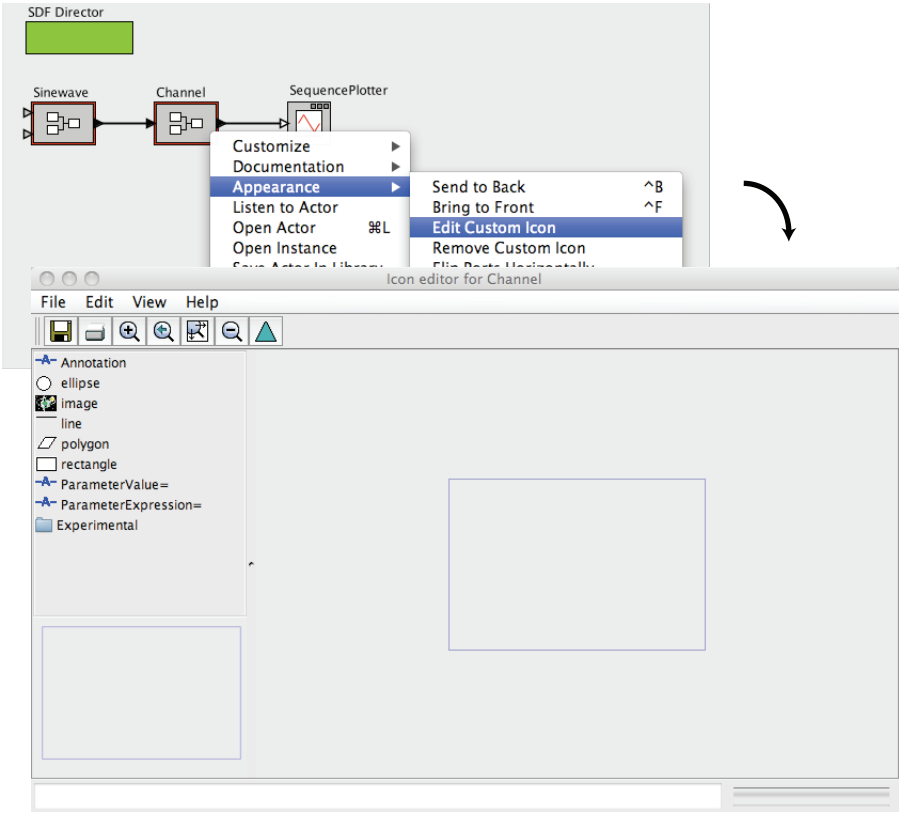

Figure 1.30: Custom icon editor for the Channel actor.

since the icon itself has the actor name in it, the Customize Name dialog is used to deselect "show name."

1.6 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 1.32 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 1.33. 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 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.

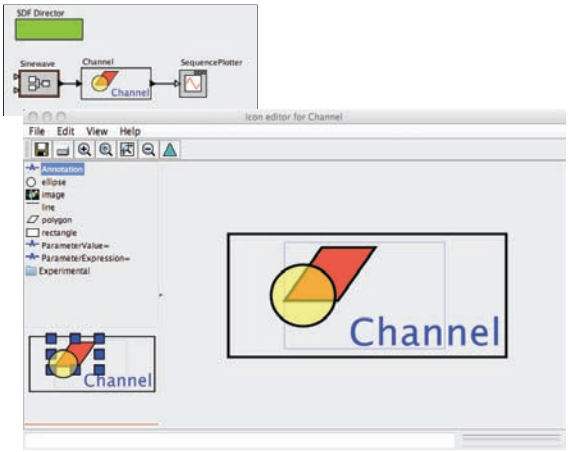

Figure 1.31: Custom icon for the Channel actor.

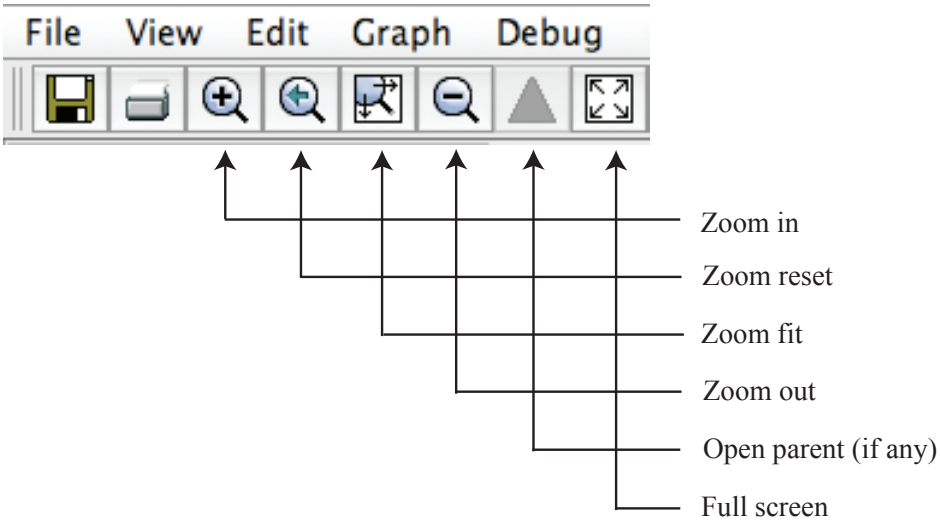

Figure 1.32: Summary of toolbar buttons for zooming and fitting.

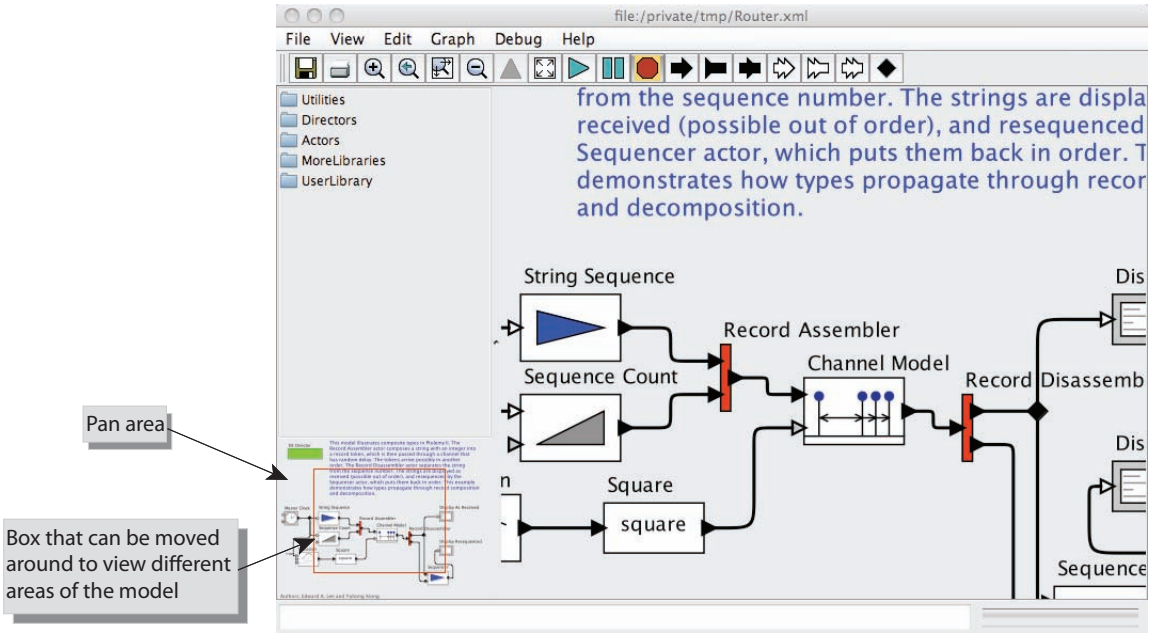

Figure 1.33: The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

1.7 Classes and Inheritance

Ptolemy II includes the ability to define *actor-oriented classes*[10] 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.

1.7.1 Creating and Using Actor-Oriented Classes

Consider the model that we developed in section 1.4, shown for reference in figure 1.34. Suppose that we wish to create multiple instances of the channel, as shown in figure 1.35. 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[10]. However, this is 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 1.34. This results in a far less maintainable or scalable

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

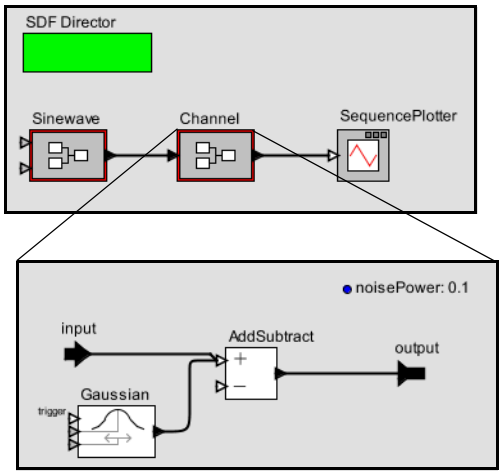

Figure 1.34: Hierarchical model that we will modify to use classes.

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 1.34, and remove the connections to the channel, as shown in figure 1.36. 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 1.36. Although this looks similar to the design in figure 1.35, 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 1.37. 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 invoke "Open Actor" any of the instances (or the class) in figure 1.37, 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.

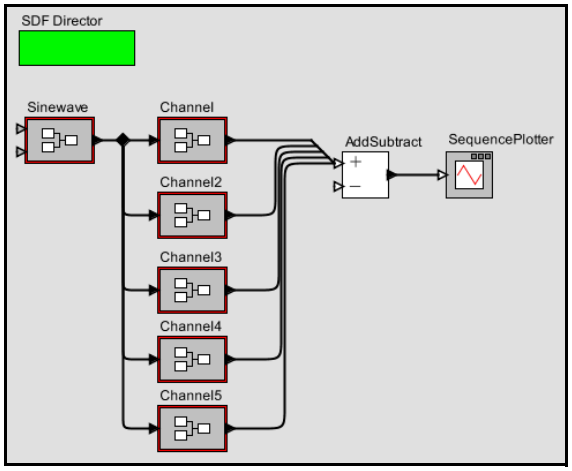

Figure 1.35: A poor design of a diversity communication system, which has multiple copies of the channel as defined in figure 1.34.

1.7.2 Overriding Parameter Values in Instances

By default, all instances of Channel in figure 1.37 have the same icon and the same parameter values. However, each instance can be customized by overriding these values. In figure 1.38, for example, 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."

1.7.3 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 1.39. A subclass is created by right clicking on the class icon and selecting "Create Subclass." The 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 1.40, 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 1.41, along with a plot where you can see the sinusoidal interference.

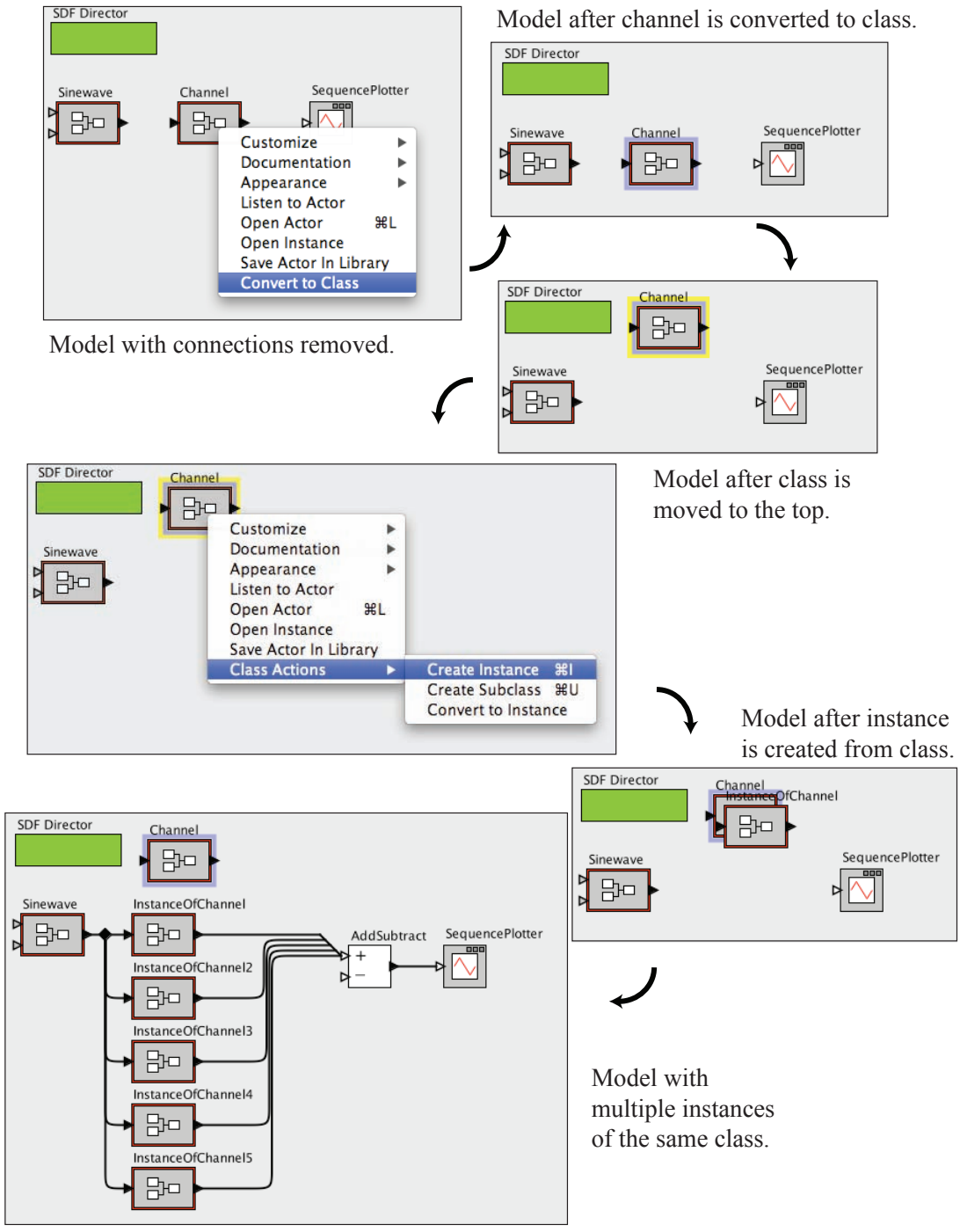

Figure 1.36: Creating and using a channel class.

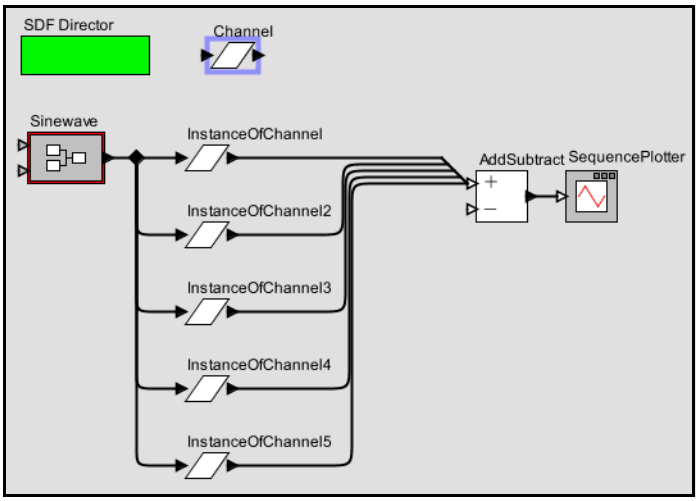

Figure 1.37: The model from figure 1.36 with the icon changed for the class. Note that changes to the base class propagate to the instances.

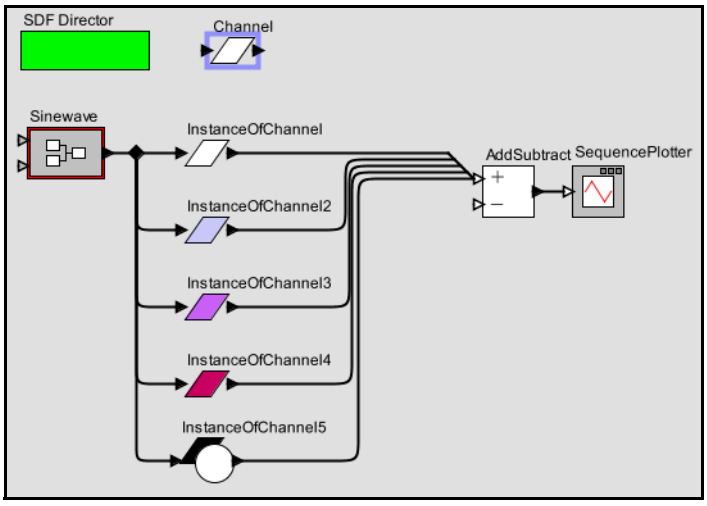

Figure 1.38: The model from figure 1.37 with the icons of the instance changed to override parameter values in the class.

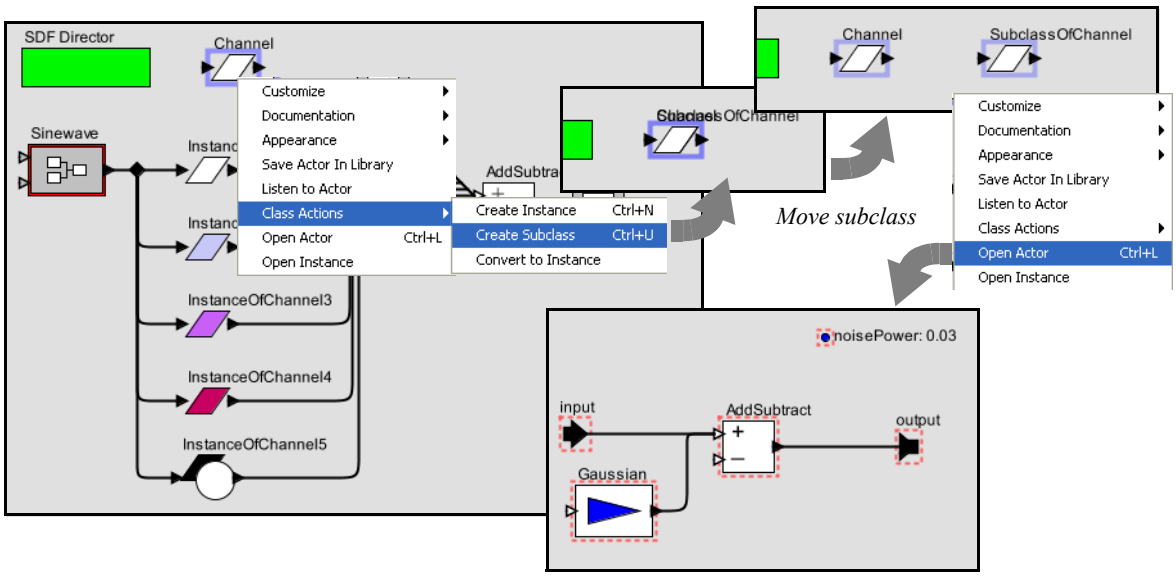

Figure 1.39: The model from figure 1.38 with a subclass of the Channel with no overrides (yet).

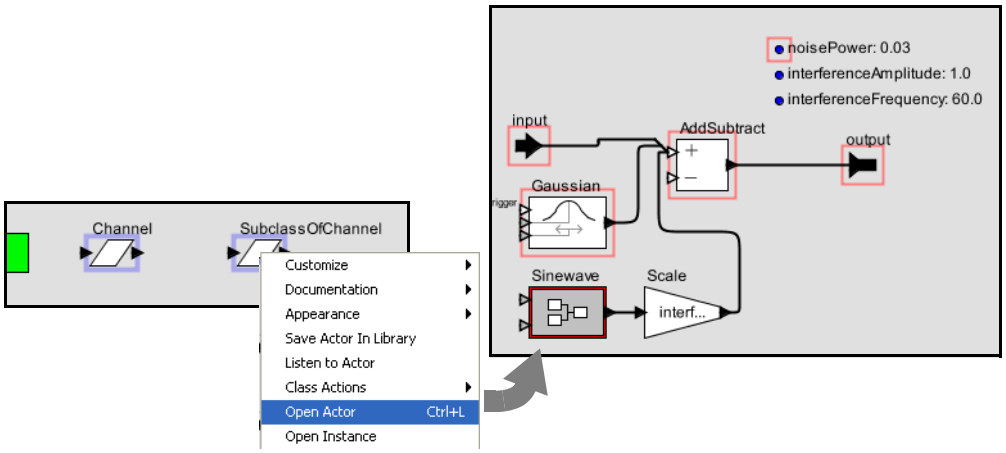

Figure 1.40: The subclass from figure 1.39 with overrides that add sinusoidal interference.

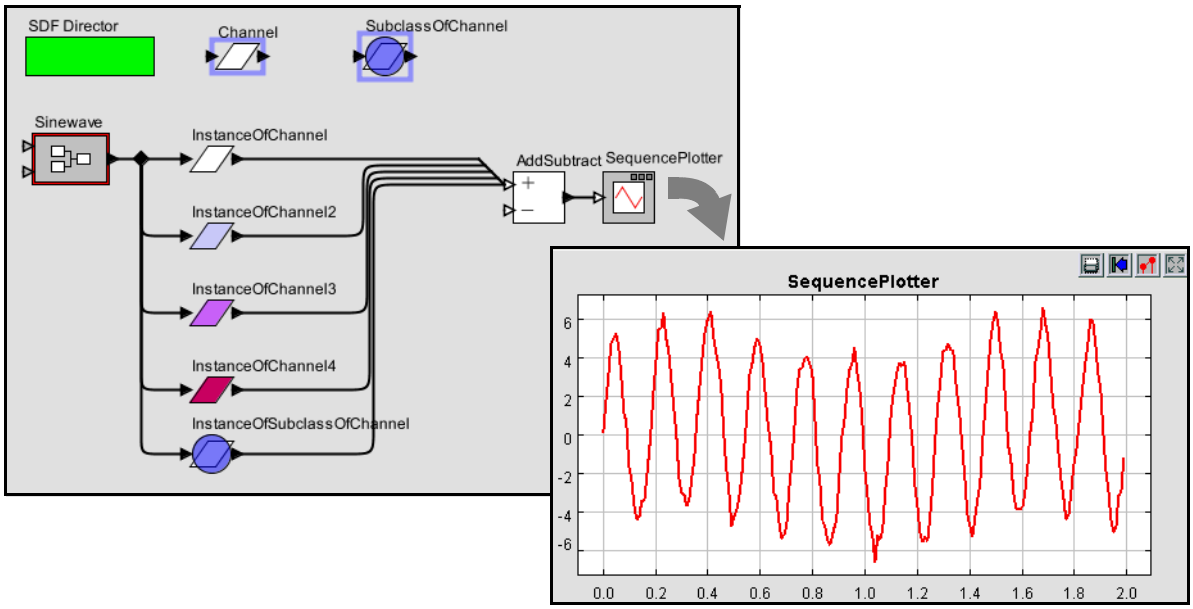

Figure 1.41: A model using the subclass from figure 1.40 and a plot of an execution.

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

1.7.4 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, right click and invoke "Open Actor" on the Channel class, and then select "Save As" from the File menu. The dialog that appears is shown in figure 1.42. 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.

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

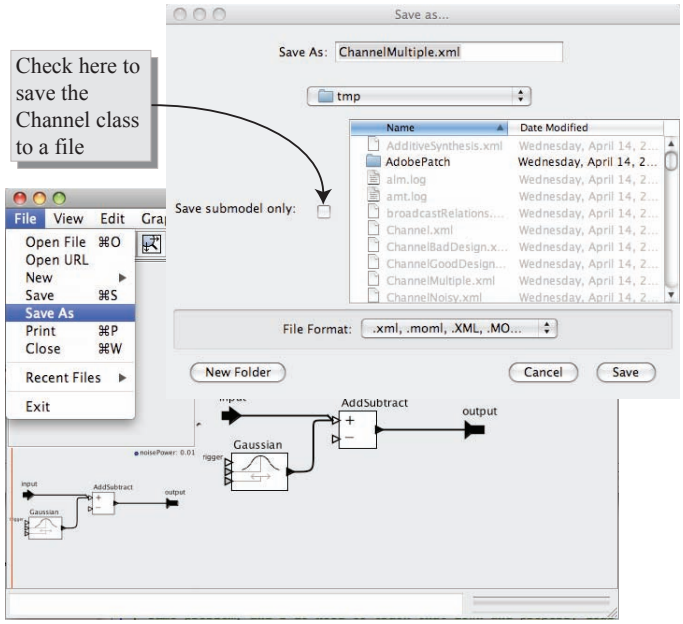

Figure 1.42: A class can be saved in a separate file to then be shared among multiple models.

within the Ptolemy II installation directory.[11] In figure 1.42, 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 1.43. 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 1.44. 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.

[11] 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. Or, in a Graph editor, select "View" |"JVM Properties" and look for the ptolemy.ptII.dir property.

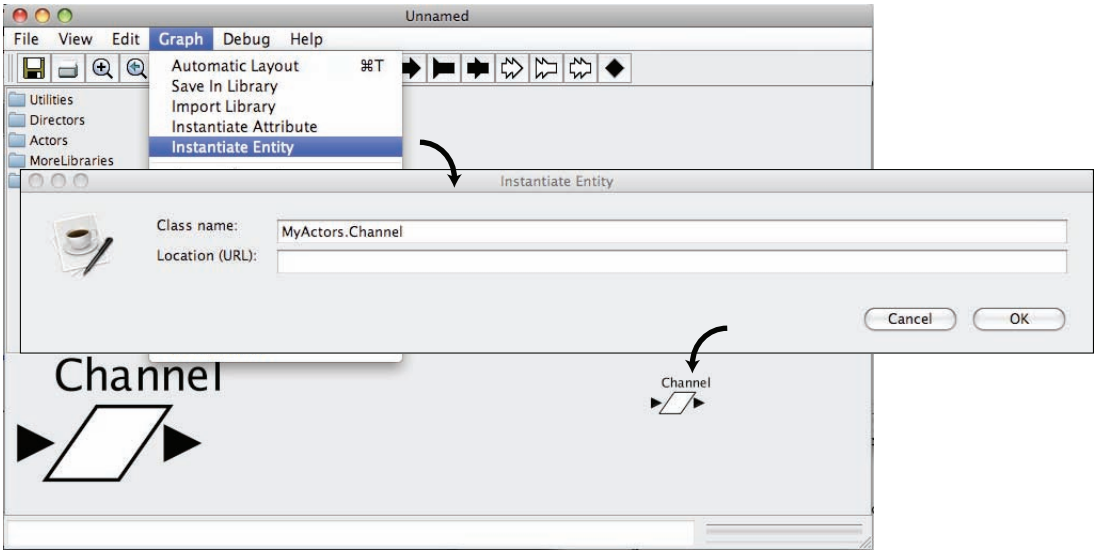

Figure 1.43: An instance of a class defined in a file can be created using Instantiate Entity in the Graph menu.

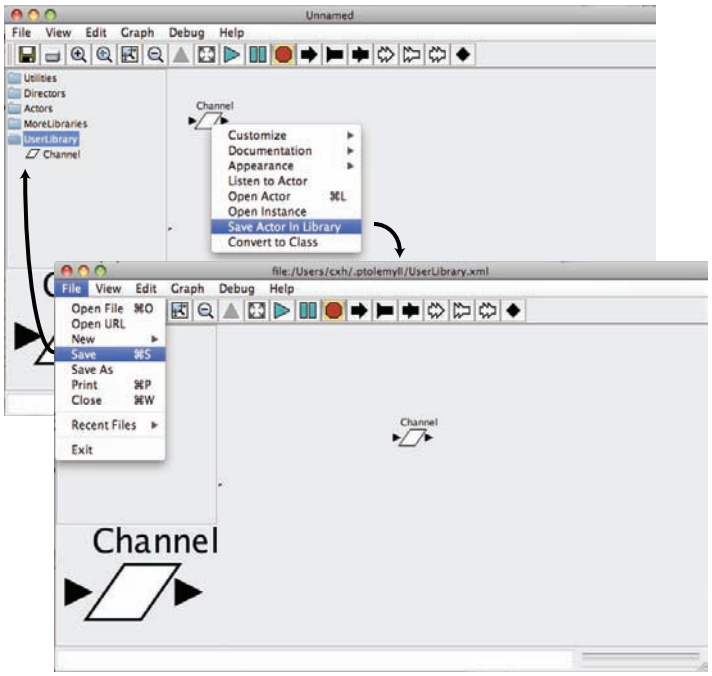

Figure 1.44: Instances of a class that is defined in its own file can be made available in the UserLibrary.

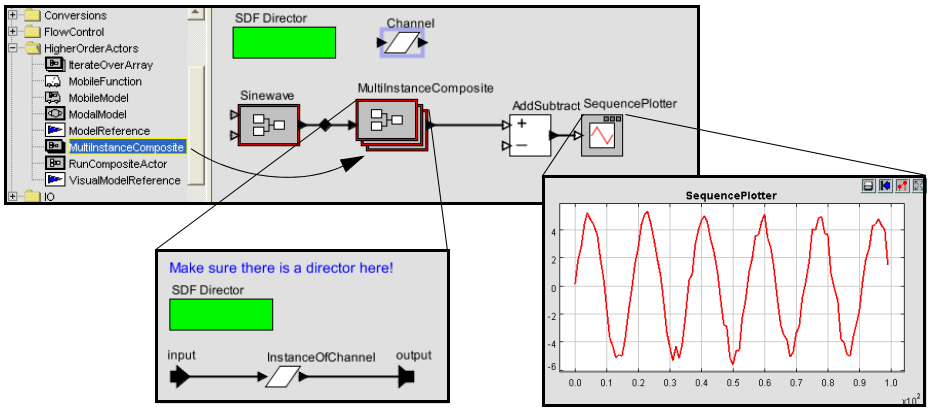

Figure 1.45: A model that is equivalent to that of figure 1.37, but using a MultiInstanceComposite, which permits the number of instances of the channel to change by simply changing one parameter value.

1.8 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 [12], 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 1.10, after explaining some of the domains that can make effective use of it.

1.8.1 MultiInstance Composite

Consider the model in figure 1.37, 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 instances. This problem is solved by the *MultiInstanceComposite* actor[12]. A model equivalent to that of figure 1.37 but using the *MultiInstanceComposite* actor is shown in figure 1.45. The *MultiInstanceComposite* 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). *MultiInstanceComposite* **must be opaque** (have a director), so that its Actor interface methods (preinitialize(), ..., wrapup()) are invoked during model initialization.

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

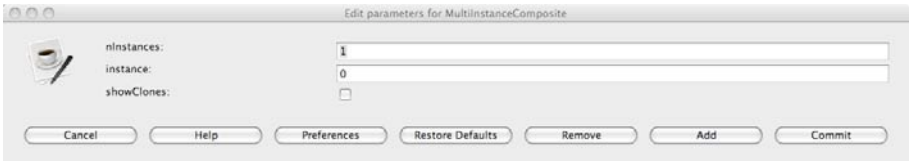

Figure 1.46: The first parameter of the MultiInstanceComposite specifies the number of instances. The second parameter is available to the model builder to identify individual instances. The third parameter controls whether the instances are rendered on the screen.

The *MultiInstanceComposite* actor has three parameters, *nInstances*, *instance* and *showClones* shown in figure 1.46. 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 1.45, 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 1.37, where the same input value is broadcast to all instances, but distinct output values are supplied to the *AddSubtract* actor.

The model of figure 1.45 is better than that of figure 1.37 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 *MultiInstanceComposite*. Try, for example, making the *noisePower* parameter of the *InstanceOfChannel* actor in figure 1.45 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.

1.8.2 IterateOverArray

The implementation of the *Channel* class, which is shown in figure 1.42, 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 1.45. Just like the *MultiInstanceComposite*, the *IterateOverArray* actor requires a director inside. An example is shown in figure 1.47. 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 1.47 we have set equal to the value of the *diversity* parameter that we have added to the model. The

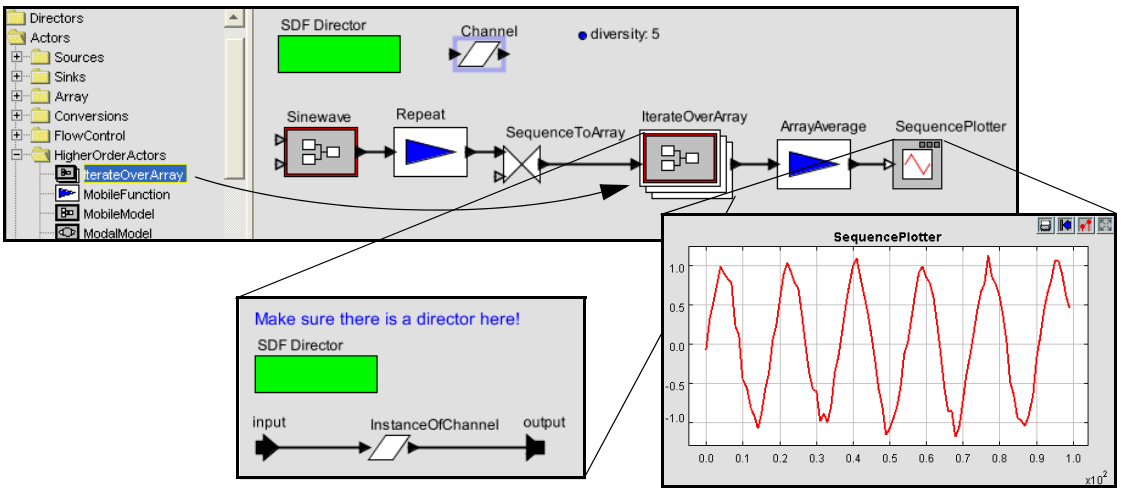

Figure 1.47: The *IterateOverArray* actor can be used to accomplish the same diversity channel model as in figure 1.45, but without creating multiple instances of the channel model. This works because the channel model has no state.

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 1.47 is similar to that of the model in figure 1.45, 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 the *IterateOverArray* 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 1.48. When an actor is dragged from the library, and then 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 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.

1.8.3 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

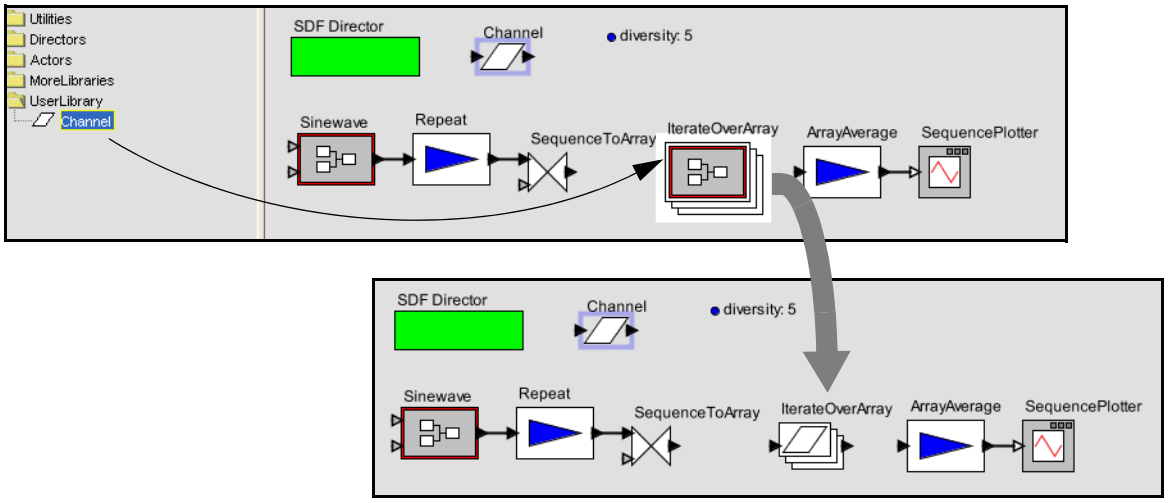

Figure 1.48: The *IterateOverArray* actor supports dropping an actor onto it. When you do this, it transforms to mimic the icon of the actor you dropped onto it, as shown. Here we are using the *Channel* class that we saved to the UserLibrary as shown in figure 1.44.

that model, streaming data from the other input port through it.

A use of the *MobileFunction* actor is shown in figure 1.49. 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 the *Sources* library, *GenericSources* sublibrary. The functions are interleaved by the *Commutator* actor, from *FlowControl* library, *Aggregators* sublibrary.

1.8.4 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 tour for more details.

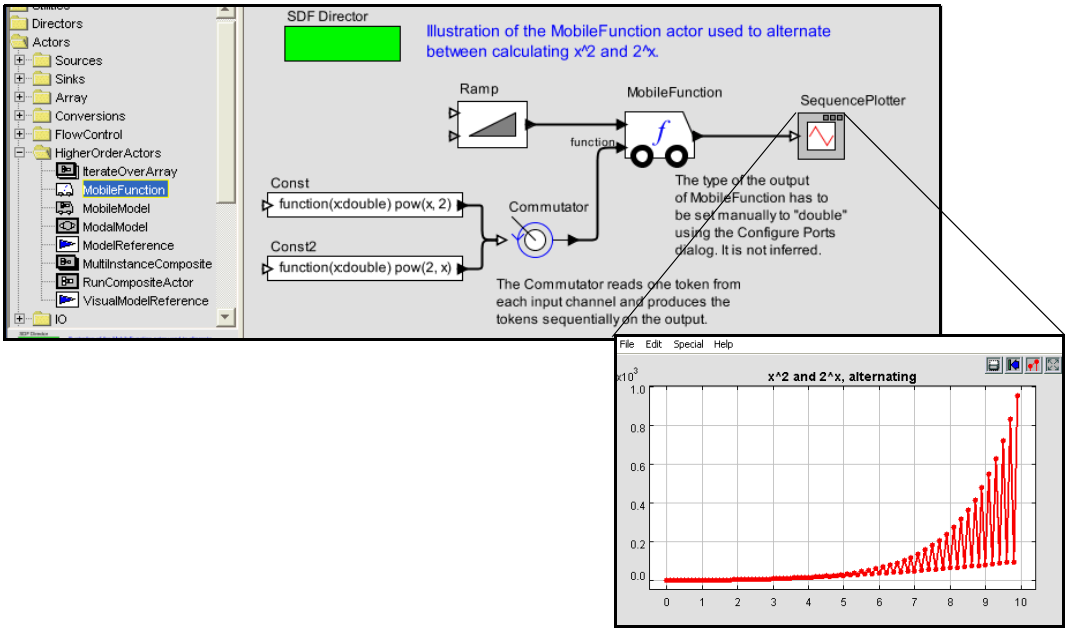

Figure 1.49: The MobileFunction actor accepts a function definition at one port and applies it to data that arrives at the other port.

1.9 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 *SDFDirector* without justifying why. A director in Ptolemy II gives meaning (semantics) to a diagram. It specifies what a connection means, and how the diagram should be executed. In Ptolemy II terminology, the director realizes a *domain*. Thus, when you construct a model with an SDF director, you have constructed a model "in the SDF domain."

The SDF director is fairly easy to understand. "SDF" stands for "synchronous dataflow."[11] In dataflow models, actors are invoked (fired) when their input data is available. SDF is a 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

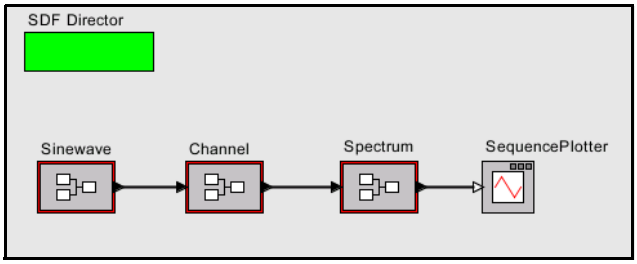

Figure 1.50: A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave*, *Channel*, and *SequencePlotter*, and one firing of *Spectrum*.

explaining some of the subtleties in SDF.

1.9.1 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 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 1.50 shows a system that computes the spectrum of the same noisy sine wave that we constructed in figure 1.25. The *Spectrum* actor has a single parameter, which gives the order of the Fast Fourier Transform (FFT) used to calculate the spectrum. Figure 1.51 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[6] of the design document for details.

A second subtlety with SDF models is that if there is a feedback loop, as in figure 1.52, 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

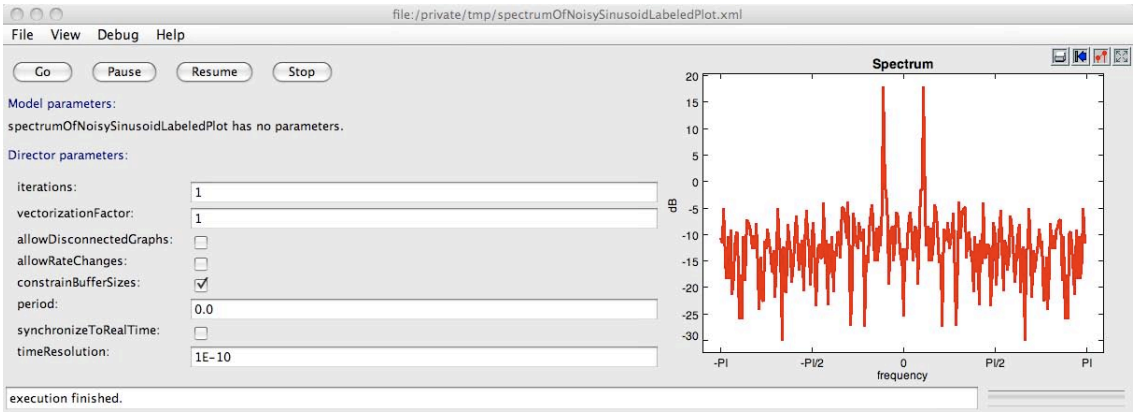

Figure 1.51: A single iteration of the SDF model in figure 1.50 produces 256 output tokens.

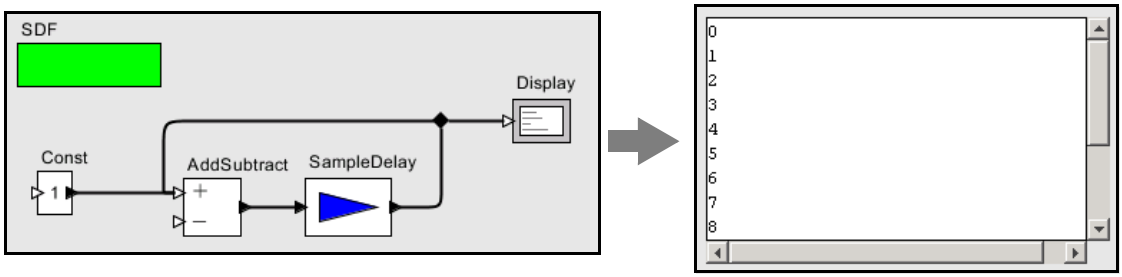

Figure 1.52: An SDF model with a feedback loop must have at least one instance of the SampleDelay actor in it.

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 few actors in the actor library are particularly useful for building SDF models that manipulate token streams in nontrivial ways. These are:

- *ArrayToElements*, *ArrayToSequence*, *ElementsToArray*, and *SequenceToArray*, found in the *Array* sublibrary.
- *Commutator* and *Distributor*, found under *FlowControl* library, *Aggregators* sublibrary
- *Chop* and , found under *FlowControl* library, *Sequencers* sublibrary
- *Downsample*, *UpSample*, and *FIR*, found under *SignalProcessing* library, *Filtering* sublibrary, and

- *Case* and *ModalModel* found in HigherOrderActors (*ModalModel*, which is a very expressive actor, is also explained further below).

The reader is encouraged to explore the documentation for these actors (right click on the actor select "Get Documentation").

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. By default, 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. However, the *SDFDirector* does contain a parameter called *period* that can be used to advance time by a fixed amount on each iteration of the model. The next domain we consider, Discrete Event (DE) includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

1.9.2 Data-Dependent Rates

Several domains generalize SDF to support data-dependent rates. The most mature of these are the process networks domain (PN), which associates with each actor its own thread of control, and the dynamic dataflow domain (DDF), which dynamically schedules actor firings. PSDF (parameterized SDF) and HDF (heterochronous dataflow) are more experimental, but are possibly more efficient and formally analyzable than PN. See volume 3[6] of the design doc for details about domains.

1.9.3 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 an instance of the ptolemy.actor.util.Time class. A Time object has no limit on the magnitude of the time and the resolution is specified by the *timeResolution* parameter of the associated director. The value of the time stamp may be accessed as 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

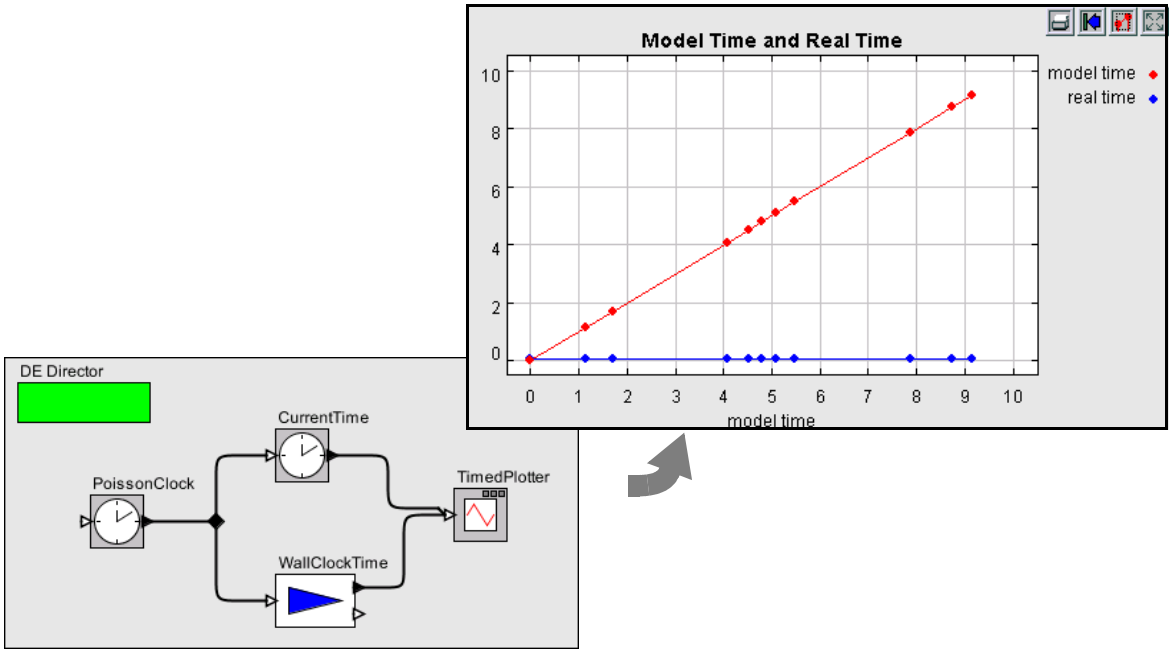

Figure 1.53: Model time vs. real time (wall clock time).

the two notions of time. It does this by delaying execution of the model, if necessary, allowing real time to catch up with model time.

Consider the DE model shown in figure 1.53. This model includes a *PoissonClock* actor, a *CurrentTime* actor, a *WallClockTime* actor, all found in the *TimedSources* sublibrary and *RealTime* sublibrary of the *Sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead, these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the time stamp of the input). The *WallClockTime* actor 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 1.53 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.

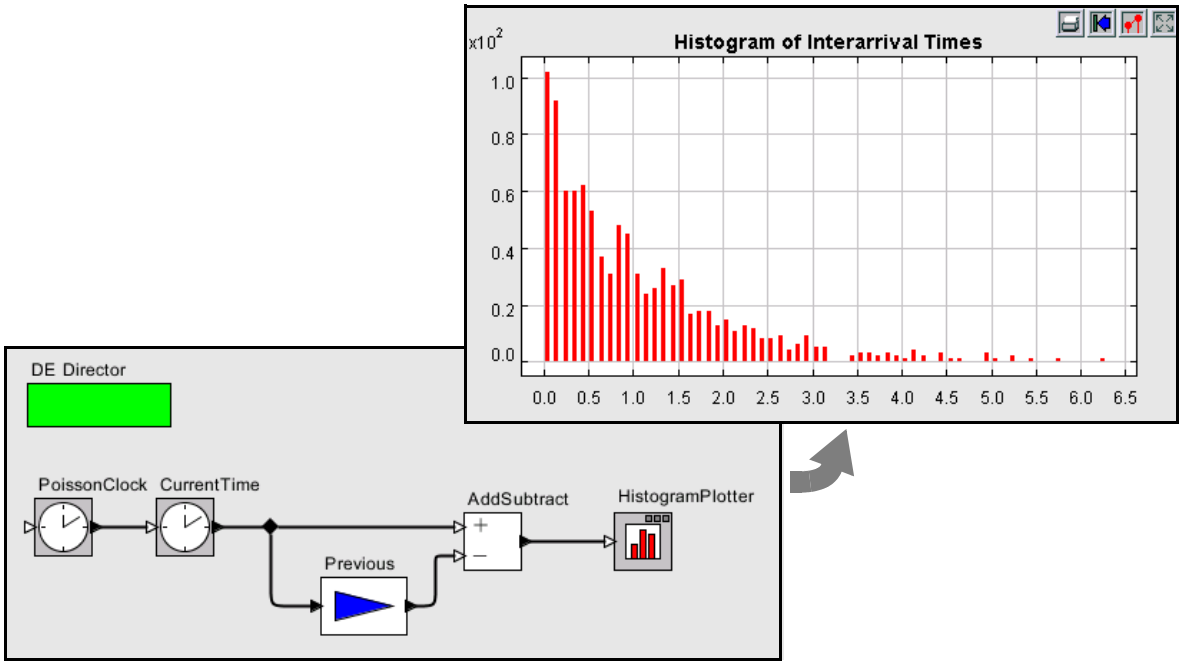

Figure 1.54: Histogram of interarrival times, illustrating handling of simultaneous events.

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 1.54, 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 (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 1.54, there is only one

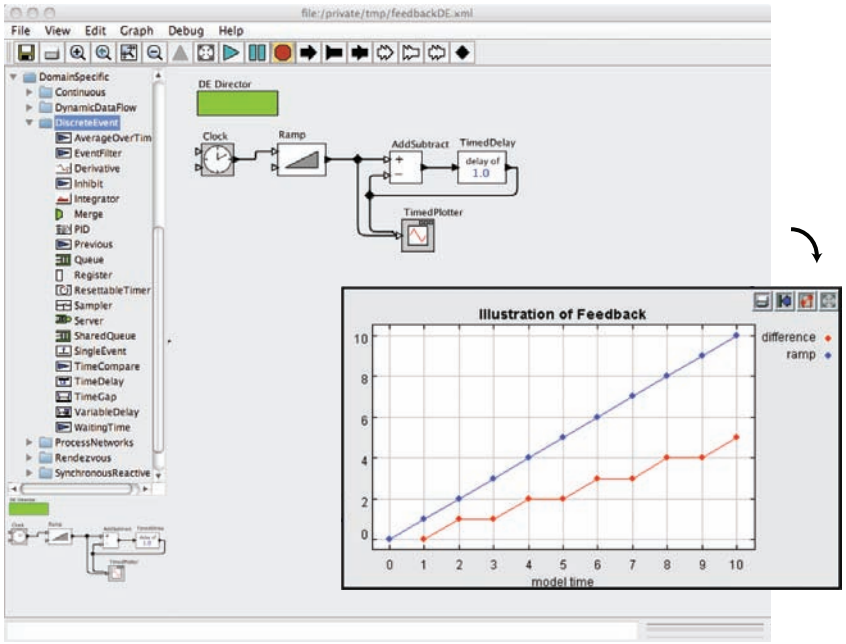

Figure 1.55: Discrete-event model with feedback, which requires a delay actor such as TimedDelay. Notice the library of domain-specific actors at the left.

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 1.55). Consider for example the model shown in figure 1.55. That model has a *Clock* actor, which is set to produce events every 1.0 time units. Those events trigger the *Ramp* actor, which produces outputs that start at 0 and increase by 1 on each firing. In this model, the output of the *Ramp* goes into an *AddSubtract* actor, which subtracts from the *Ramp* output its own prior output delayed by one time unit. The result is shown in the plot in the figure.

Occasionally, you will need to put a *TimedDelay* actor in a feedback loop with a delay of 0.0. This

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

1.9.4 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 [1] and [2] for details.

1.9.5 Continuous Time Systems

Ptolemy II 8.0 includes a new implementation of continuous-time models based on the semantics given in [14], In particular, the Continuous domain cleanly supports continuous-time models (using an ODE solver), discrete-event models, and arbitrary mixtures of the two, including signals that combine continuous-time segments with discrete events. It also interoperates cleanly with most other domains (pretty much all domains except Kahn Process Networks (PN) and Rendezvous, for which there does not appear to be reasonable semantic model of such an interaction). Modal models are also cleanly supported with the Continuous domain, enabling hybrid system modeling with a rigorous semantics. The Continuous domain replaces the older CT domain.

The Continuous domain has semantics considerably different from either DE or SDF. In Continuous, the signals sent along connections between actors are usually continuous-time signals. A Continuous example is described above in section 1.2.3.

The Continuous 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 Continuous director is quite sophisticated in its handling of such mixed signal systems.

1.10 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 Continuous, but this section will concentrate on Continuous. Feel free to examine other examples of modal models given in the tour (figure 1.3). See also [9], which is a detailed discussion of Finite State Machines and Modal Models and also includes hyperlinks to runnable examples.

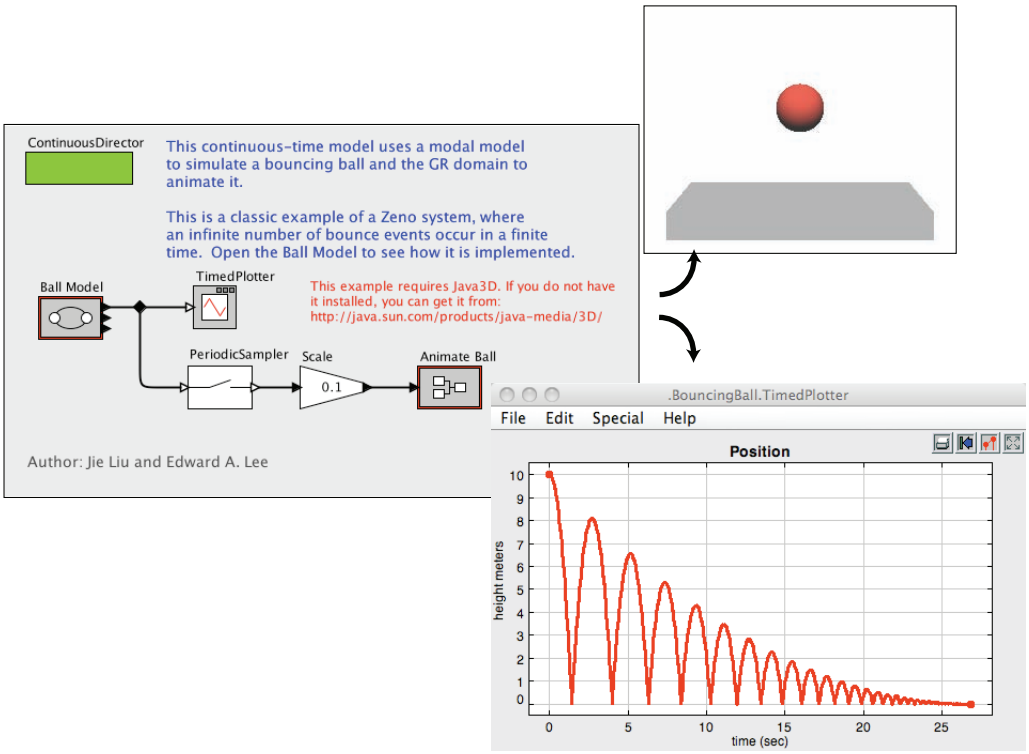

Figure 1.56: Top level of the bouncing ball example.

1.10.1 Examining a Pre-Built Model

Consider the bouncing ball example, which can be found under "Bouncing Ball" in figure 1.3 (in the "Hybrid Systems" entry). The top-level contents of this model is shown in figure 1.56. It contains *Ball Model*, *TimedPlotter*, *PeriodicSampler*, *Scale* actors 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 invoke "Open Actor" on the *Ball Model*, you will see something like figure 1.57. Figure 1.57 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the left. The circles in figure 1.57 are states, and the arcs between circles are transitions between states. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a state in a finite-state machine.

The state machine in figure 1.57 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 1.58 (as indicated by the thicker border). The *free* state

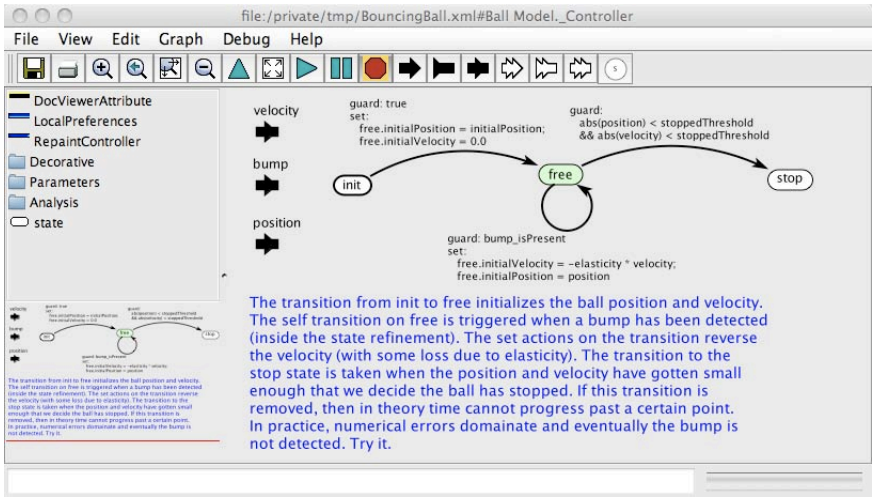

Figure 1.57: Inside the Ball Model of figure 1.56

represents the mode of operation where the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

At any time during the execution of the model, the modal model is in one of these three states. When the model begins executing, it is in the *init* state. During the time a modal model is in a state, the behavior of the modal model is specified by the *refinement* of the state. The refinement can be examined by right clicking on the state and selecting ”Look Inside”. As shown in figure 1.59, the *init* state has no refinement.

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

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

The first line is a *guard*, which is a predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value *true*. Thus, the first thing this model will do is take this transition and change mode to *free*. The second line “set:” indicates that the successive lines are from the set action. The third and fourth lines specify 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 1.60. This model represents the laws of gravity, which state that an object of any mass will have an acceleration of roughly $-10 meters/second^2$. The acceleration is integrated to get the velocity, which is, in turn, integrated to get the vertical position.

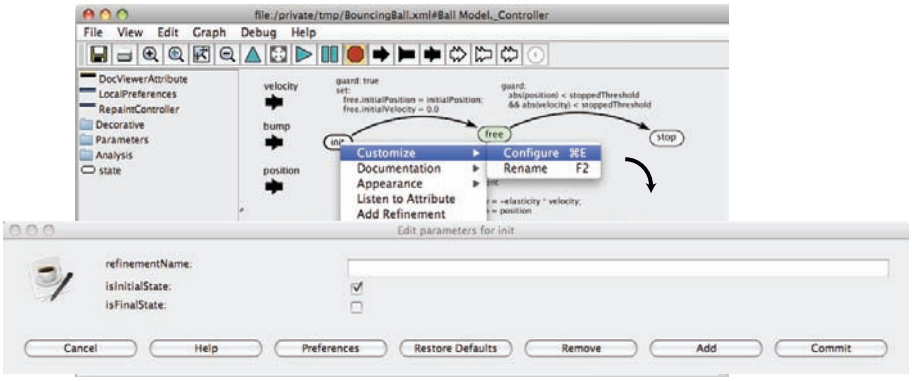

Figure 1.58: The initial state of a state machine is set by right clicking on the background and specifying the state name.

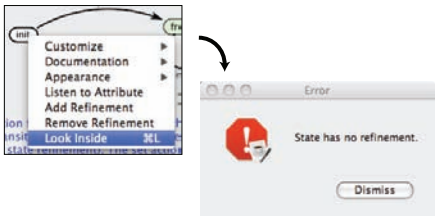

Figure 1.59: A state may or may not have a refinement, which specified the behavior of the model while the model is in that state. In this case, init has no refinement.

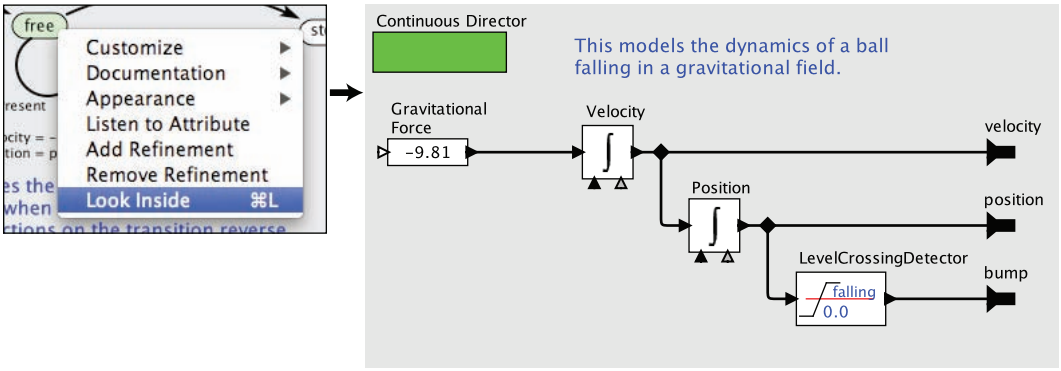

Figure 1.60: The refinement of the free state, shown here, is a continuous-model representing the laws of gravity.

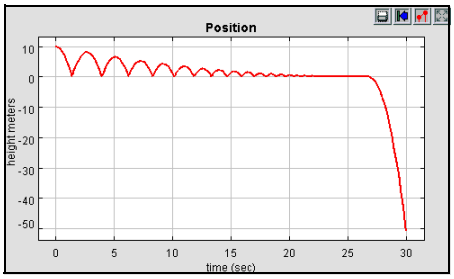

Figure 1.61: Result of running the bouncing ball model without the stop state.

In figure 1.60, 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 1.57 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 1.56.

As you can see from figure 1.57, 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.

1.10.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 1.56 and 1.57 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 1.61. 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.

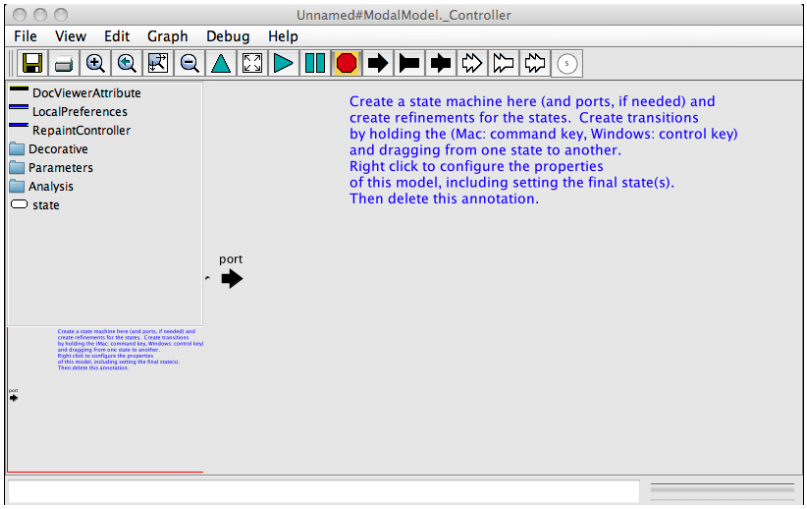

Figure 1.62: Inside of a new modal model that has had a single output port added.

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.

1.10.3 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 1.4.2. Figure 1.56 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.

If you create a new modal model by dragging in a *Modal Model* from the *HigherOrderActors* library, create an output port and name it *output*, and then look inside, you will get a Finite State Machine (FSM) editor like that shown in figure 1.62. The annotation text suggests that you delete it once you no longer need it.

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 1.57, 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 1.57, there is an

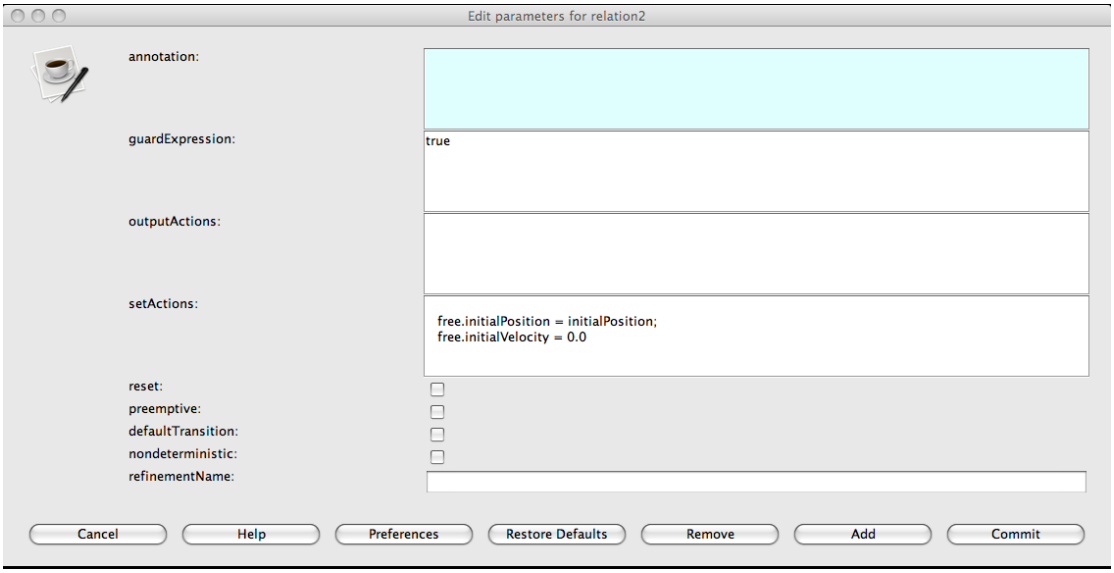

Figure 1.63: Transition dialog for the transition from init to free in figure 1.57.

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 1.58. In that figure, the initial state is named *init*.

Creating Transitions

To create transitions, you must hold the control button[13] 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 "Customize" |"Configure") allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in figure 1.63. In that dialog, we see the following:

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

[13] Or the command button on a Macintosh computer.

- The *setActions* 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 *false*, meaning that the refinement of the destination mode is not 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.

- The *defaultTransition* parameter is set to false. If this parameter was set to true, then this transition would be enabled if and only if no other non-default transition is enabled.

- The *nondeterministic* parameter is set to false meaning that if this transition is enabled, it must be the only enabled transition. This parameter specifies whether this transition is nondeterministic. Here nondeterministic means that this transition may not be the only enabled transition at a time.

- The *refinementName* parameter is empty, indicating that the state is not refined. This parameter specifies the names of refinements. The refinements must be instances of TypedActor and have the same container as the FSMActor containing this state, otherwise an exception will be thrown when getRefinement() is called. Usually, the refinement is the empty string or a single name. However, if a comma-separated list of names is provided, then all the specified refinements will be executed.

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[14] on the state or transition, and select "Add Refinement." You will see a dialog like that in figure 1.64. 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

[14] On a Macintosh, control-click.

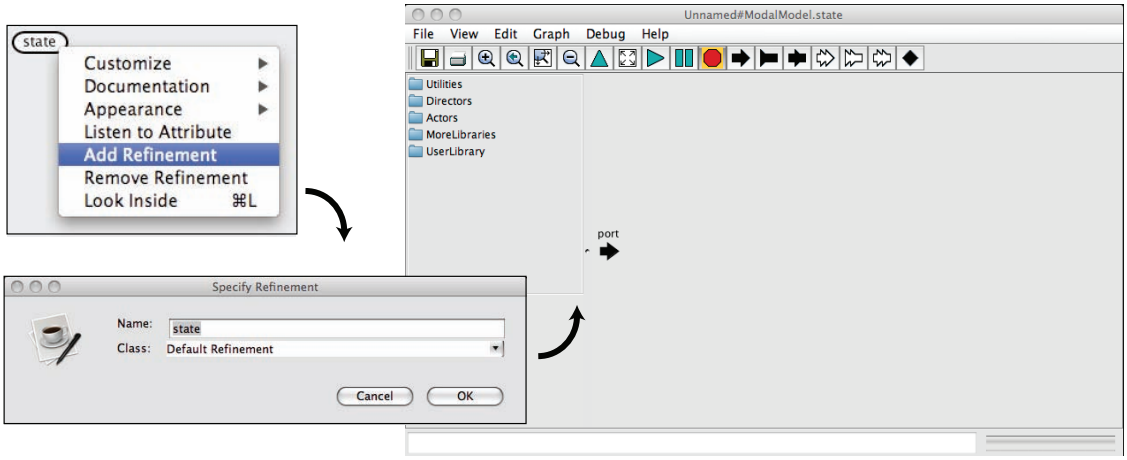

Figure 1.64: Adding a refinement to a state.

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. You will have to create a director in the refinement. The modal model will not operate without a director in the refinement.

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 1.61. This model exhibits certain key properties of refinements:

- Refinements must contain directors. In this case, the *ContinuousDirector* is used.
- The refinement has the same ports as the modal model, and can read input values 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.

1.10.4 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 "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.

1.11 Vergil Command Line Arguments

The vergil script at $PTII/bin/vergil has several command line arguments. Typical use is `$PTII/bin/vergil` *model.xml* where *model.xml* is a Ptolemy II MoML file, for example:

```
$PTII/bin/vergil ptolemy/domains/sdf/demo/Butterfly/Butterfly.xml
```

Not all combinations of command line argments make sense. The best way to see what command line arguments are available is to run `$PTII/bin/vergil -help`. The vergil command line arguments are listed below:

Table 1.1: Vergil command line arguments, part 1.

Command Line Argument	Description
`-debug`	Enable debugging with jdb, see $PTII/doc/coding/debugging.htm
`-help`	Print a help message for the vergil command
`-helpall`	List the Ptolemy II scripts in $PTII/bin that can be invoked
`-jdb`	Run jdb instead of java, see $PTII/doc/coding/debugging.htm
`-profile`	Run under CPU sample profiling
`-q`	Do not echo the command being run
`-policyfile` *policyfile*	Run the model with a specified *policyfile*. For example, to run in a restricted Java sandbox: `$PTII/bin/vergil -policyfile $PTII/bin/sandbox.policy` The `-sandbox` argument is an alias for `-policyfile $PTII/bin/sandbox.policy`. Note that the file browser does not work well in the sandbox. To run a model in the sandbox, specify the model on the command line: `vergil -sandbox $PTII/ptolemy/moml/demo/modulation.xml` The `-sandbox` argument the first argument after the command name: `vergil -sandbox -hyvisual $PTII/ptolemy/moml/demo/modulation.xml`
`-class` *classname*	The `-class` option can be used to specify a Java class to be loaded. The named class must have a constructor that takes a Workspace as an argument. In the example below, $PTII/ptolemy/domains/sdf/demo/Butterfly/Butterfly.java is a class that has a constructor Butterfly(Workspace). `$PTII/bin/vergil -class ptolemy.domains.sdf.demo.Butterfly.Butterfly` Note that `-class` is not very well tested now that we have use MoML for almost all models. The `-class` argument is usually used with $PTII/bin/ptolemy, not $PTII/bin/vergil
-parameterName parameterValue	Set a parameter, where *parameterName* is the name of a parameter relative to the top level of a model or the director of a model. For instance, to set the phase parameter of the signal composite: `vergil $PTII/ptolemy/moml/demo/modulation.xml -signal.phase 0.5`

Table 1.2: Vergil command line arguments, part 2.

Command Line Argument	Description
`-configuration` *configurationURL*	Set the Ptolemy configuration to be used. Ptolemy uses configurations to provide custom actor libraries and splash screens for different tools such as HyVisual, VisualSense and Vergil. The default is `ptolemy/configs/full/configuration.xml`, which uses the vergil configuration. See the table below for predefined configurations.
`-run`	Open the model or models and run them.
`-runThenExit`	Open the model or models, run them, then exit after the models finish.
`-test`	Start up and then exit after two seconds.
`-version`	Print version information.

Vergil has a number of predefined configuration options that include specific sets of actors and splash screens. Only one configuration option or `-configuration` *configurationURL* may be used at a time. The configuration options are defined below, for example `$PTII/bin/vergil -bcvtb` will use the Build Controls Virtual Test Bed configuration.

Table 1.3: Vergil configuration options

Command line argument	Description
`-bcvtb`	Building Controls Virtual Test Bed Configuration which provides interfaces to C programs via sockets. For details, see https://gaia.lbl.gov/bcvtb
`-codegen`	Configuration that includes actors that have C code generation implementations
`-dsp`	Configuration to use for Digital Signal Processing (DSP) applications
`-full`	Configuration that invokes the full version of vergil. This is the default configuration.
`-fullViewer`	Configuration including all domains, used to view, not edit models.
`-hyvisual`	Configuration for use with Hybrid Systems, see http://ptolemy.eecs.berkeley.edu/hyvisual/index.htm
`-jxta`	Experimental configuration for use with JXTA, a peer to peer protocol.
`-luminary`	Experimental Configuration for generating code for the Luminary board.
`-ptiny`	Configuration including only mature domains.
`-ptinyKepler`	Configuration including only domains shipped with Kepler. For details about Kepler, see http://kepler-project.org.
`-ptinyViewer`	Configuration including only mature domains, used to view, not edit models.
`-space`	Configuration used to edit SpaceCadet office space management models.
`-viptos`	Configuration for interface between Ptolemy and TinyOS, see http://ptolemy.eecs.berkeley.edu/viptos.
`-visualsense`	Configuration for use with VisualSense, a wireless tool, see http://ptolemy.berkeley.edu/visualsense/index.htm.

1.12 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 1.51 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[15], 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 1.65. 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 1.66.

[15]Hint: Notice the "$x10^2$" at the bottom right, which indicates that the label "2.5" stands for "250".

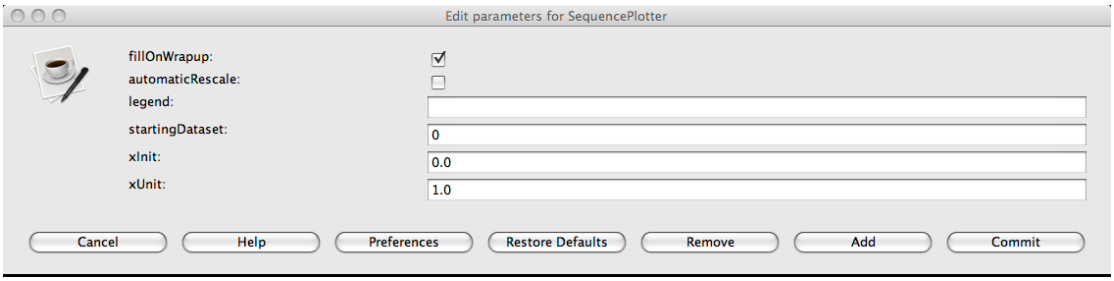

Figure 1.65: Parameters for the *SequencePlotter* actor.

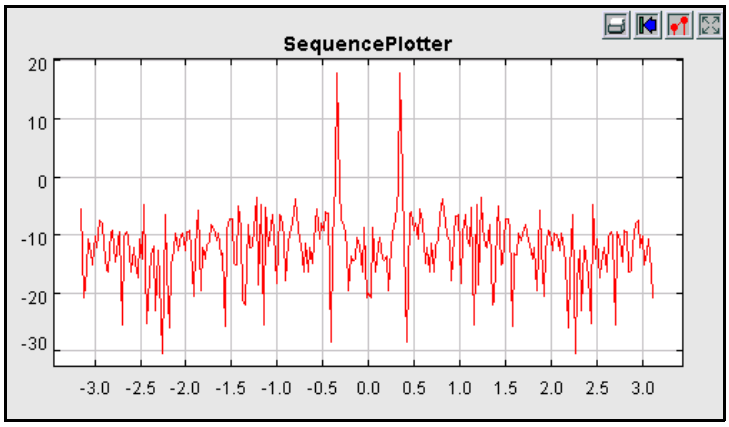

Figure 1.66: Better labeled plot, where the horizontal axis now properly represents the frequency values

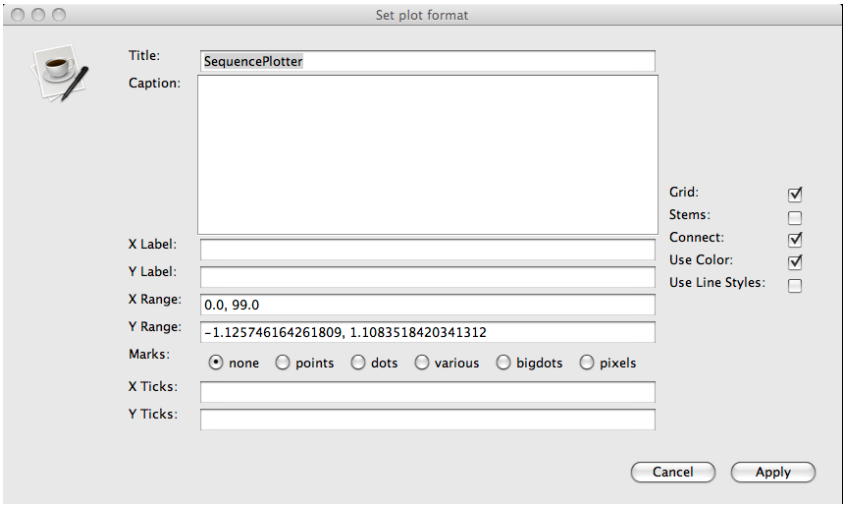

Figure 1.67: Format control window for a plot.

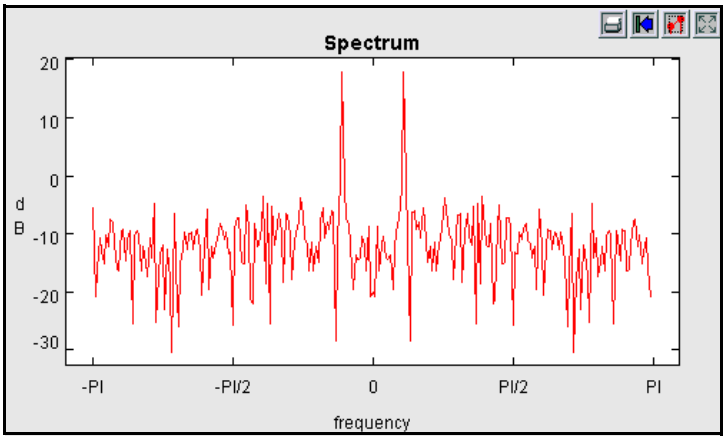

Figure 1.68: Still better labeled plot.

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 1.67, filled in with values that result in the plot shown in figure 1.68. 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”
- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

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

  The syntax in general is:

  label value, label value, ...

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

This page intentionally left mostly blank.

"There are 10 kinds of people in the world,
those that understand binary
and those that don't."

Chapter 2

Expressions

Edward A. Lee, Thomas Huining Feng, Xiaojun Liu,
Steve Neuendorffer, Neil Smyth, Yuhong Xiong, Christopher X. Brooks

2.1 Introduction

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

The Ptolemy II expression language provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the Expression actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, and it can be used by programmers extending the Ptolemy II system. In this chapter, we describe how to use expressions from the perspective of a user rather than a programmer.

2.1.1 Expression Evaluator

Vergil provides an interactive expression evaluator, which is accessed through the "File" |"New" |"Expression Evaluator" menu. This operates like an interactive command shell, and is shown in figure 2.1. It supports a command history. To access the previously entered expression, type the up arrow or Control-P. To go back, type the down arrow or Control-N. The expression evaluator is useful for experimenting with expressions.

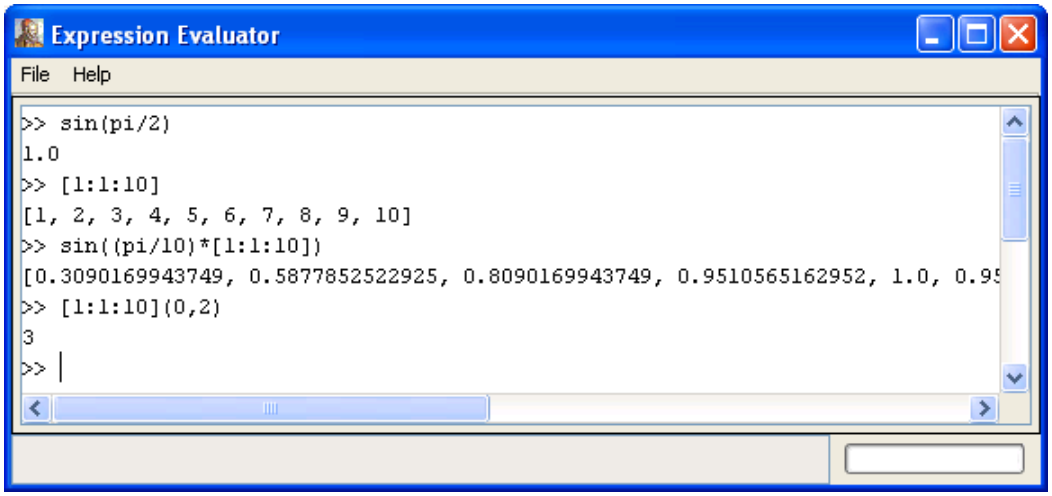

Figure 2.1: The Expression Evaluator

2.2 Simple Arithmetic Expressions

2.2.1 Constants and Literals

The simplest expression is a constant, which can be given either by the symbolic name of the constant, or by a literal. By default, the symbolic names of constants supported are PI, pi, E, e, true, false, i, j, NaN, Infinity, PositiveInfinity, NegativeInfinity, MaxUnsignedByte, MinUnsignedByte, MaxShort, MinShort, MaxInt, MinInt, MaxLong, MinLong, MaxFloat, MinFloat, MaxDouble, MinDouble. For example,

```
PI/2.0
```

is a valid expression that refers to the symbolic name "PI" and the literal "2.0." The constants i and j are the imaginary number with value equal to $\sqrt{-1}$. The constant NaN is "not a number," which for example is the result of dividing 0.0/0.0. The constant Infinity is the result of dividing 1.0/0.0. The constants that start with "Max" and "Min" are the maximum and minimum values for their corresponding types.

Numerical values without decimal points, such as "10" or "-3" are integers (type *int*). Numerical values with decimal points, such as "10.0" or "3.14159" are of type *double*. Numerical values followed by "f" or "F" are of type *float*. Numerical values without decimal points followed by the character "l" (el) or "L" are of type *long*. long. Numerical values without decimal points followed by the character "s" or "S" are of type *short*. Unsigned integers followed by "ub" or "UB" are of type , *unsignedByte*, as in "5ub". An *unsignedByte* has a value between 0 and 255; note that it is not quite the same as the Java byte, which has a value between -128 and 127. Numbers of type *int*, *long*, *short* or *unsignedByte* can be specified in decimal, octal, or hexadecimal. Numbers beginning with

a leading "0" are octal numbers. Numbers beginning with a leading "0x" are hexadecimal numbers. For example, "012" and "0xA" are both equal to the integer 10.

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

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

Table 2.1: String-valued constants defined in the expression language

Variable name	Meaning	Property name	Example under Windows
PTII	The directory in which Ptolemy II is installed	ptolemy.ptII.dir	c:\tmp
HOME	The user home directory	user.home	c:\Documents and Settings\you
CWD	The current working directory	user.dir	c:\ptII
TMPDIR	The temporary directory	java.io.tmpdir	c:\Documents and Settings\you\Local Settings\Temp\

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

The `ptolemy.ptII.dir` property is set automatically when Vergil or any other Ptolemy II executable is started up. You can also set it when you start a Ptolemy II process using the java command by a syntax like the following:

```
java -Dptolemy.ptII.dir=${PTII} classname
```

where classname is the full class name of a Java application. The constants() utility function returns a record with all the globally defined constants. If you open the expression evaluator and invoke this function, you will see that its value is something like:

```
CLASSPATH = "xxxxxxCLASSPATHxxxxxx", CWD = "/Users/Ptolemy/ptII",
E = 2.718281828459, HOME = "/Users/Ptolemy", Infinity = Infinity,
```

```
MaxDouble = 1.7976931348623E308, MaxFloat = 3.4028234663853E38,
MaxInt = 2147483647, MaxLong = 9223372036854775807L,
MaxShort = 32767s, MaxUnsignedByte = 255ub, MinDouble = 4.9E-324,
MinFloat = 1.4012984643248E-45, MinInt = -2147483648,
MinLong = -9223372036854775808L, MinShort = -32768s,
MinUnsignedByte = 0ub, NaN = NaN, NegativeInfinity = -Infinity,
PI = 3.1415926535898, PTII = "/Users/Ptolemy/ptII",
PositiveInfinity = Infinity,
TMPDIR = "/var/folders/7f/7f-o2nyjFgewH67h0keKu++++TI/-Tmp-/",
boolean = false, complex = 0.0 + 0.0i, double = 0.0,
e = 2.718281828459, false = false, fixedpoint = fix(0,2,2),
float = 0.0f, general = present, i = 0.0 + 1.0i, int = 0,
j = 0.0 + 1.0i, long = 0L, matrix = [], nil = nil,
null = object(null), object = object(null), pi = 3.1415926535898,
scalar = present, short = 0s, string = "", true = true,
unknown = present, unsignedByte = 0ub, xmltoken = null
```

2.2.2 Variables

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

```
PI*x/2.0
```

is valid if "x" a variable in scope. In the expression evaluator, the variables that are in scope include the built-in constants plus any assignments that have been previously made. For example,

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

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

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

You can add parameters to actors (composite or not) by right clicking on the actor, selecting "Customize" |"Configure" and then clicking on "Add", or by dragging in a parameter from the utilities

library. Thus, you can add variables to any scope, a capability that serves the same role as the "let" construct in many functional programming languages.

Occasionally, it is desirable to access parameters that are not in scope. The expression language supports a limited syntax that permits access to certain variables out of scope. In particular, if in place of a variable name `x` in an expression you write `A::x`, then instead of looking for `x` in scope, the interpreter looks for a container named `A` in the scope and a parameter named `x` in `A`. This allows reaching down one level in the hierarchy from either the current container or any of its containers.

2.2.3 Operators

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

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

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

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

The magnitude of the result is always less than the magnitude of the divisor (the right argument). Note that when this operator is used on doubles, the result is not the same as that produced by the remainder() function (see Table 2.6). For instance,

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

The remainder() function calculates the IEEE 754 standard remainder operation. It uses a rounding division rather than a truncating division, and hence the sign can be positive or negative, depending on complicated rules (see 2.8). For example, counter intuitively,

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

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

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

Just as *long* cannot be cast to *double*, *int* cannot be cast to *float* and vice versa.

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

- For *double* numbers, overflow results in the corresponding positive or negative infinity. Underflow (i.e. the precision does not suffice to represent the result) will yield zero.
- For integer types and *fixedpoint*, overflow results in wraparound. For instance, while the value of `MaxInt` is 2147483647, the expression `MaxInt + 1` yields –2147483648. Similarly, while `MaxUnsignedByte` has value 255ub, `MaxUnsignedByte + 1ub` has value 0ub. Note, however, that `MaxUnsignedByte + 1` yields 256, which is an *int*, not an *unsignedByte*. This is because `MaxUnsignedByte` can be losslessly converted to an *int*, so the addition is *int* addition, not *unsignedByte* addition.

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

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

```
1 == 1.0
```

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

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

Boolean-valued expressions can be used to give conditional values. The syntax for this is

```
boolean ? value1 : value2
```

If the boolean is true, the value of the expression is `value1`; otherwise, it is `value2`. The logical boolean operators are `&&`, `||`, `!`, `&` and `|`. They operate on type *boolean* and return type *boolean*. The difference between logical `&&` and logical `&` is that `&` evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical `||` and `|`. This approach is borrowed from Java. Thus, for example, the expression `false && x` will evaluate to false irrespective of whether `x` is defined. On the other hand, `false & x` will throw an exception if `x` is undefined.

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

2.2.4 Comments

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

2.3 Uses of Expressions

2.3.1 Parameters

The values of most parameters of actors can be given as expressions[1]. The variables in the expression refer to other parameters that are in scope, which are those contained by the same container or some container above in the hierarchy. They can also reference variables in a *scope-extending attribute*, which includes variables defining units, as explained below in section 2.12. Adding parameters to actors is straightforward, as explained in the previous chapter.

2.3.2 Port Parameters

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

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

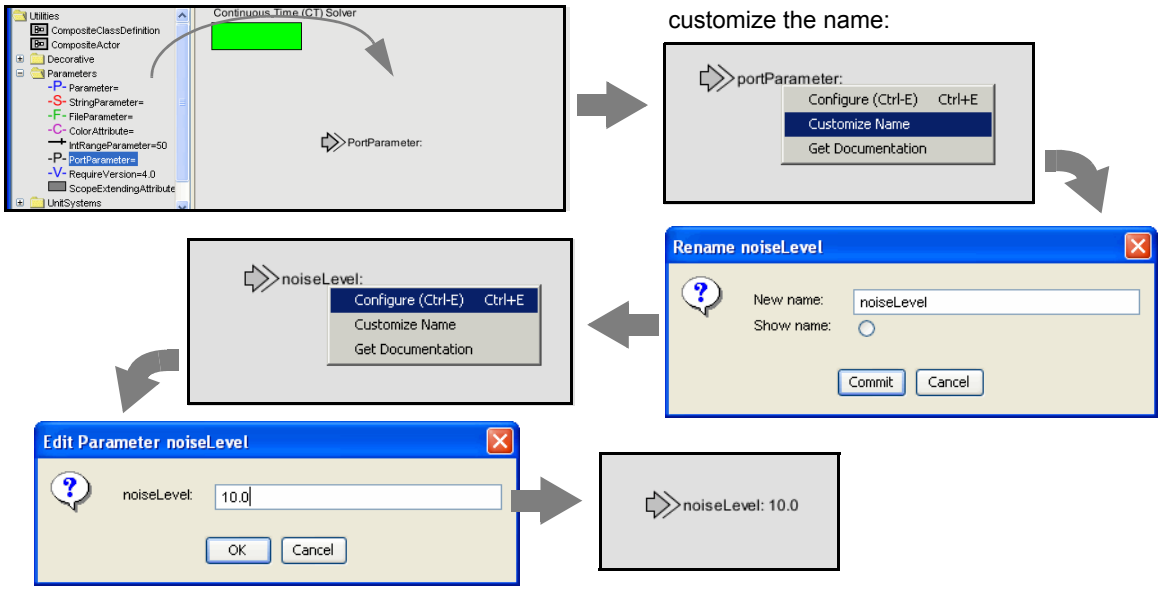

Figure 2.2: A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful and set its default value.

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

A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the utilities library, as shown in figure 2.2. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parameter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

To be useful, a PortParameter has to be given a name (the default name, "portParameter," is not very compelling). To change the name, right click on the icon and select "Customize Name," as shown in figure 2.2. In the figure, the name is set to "noiseLevel." Then set the default value by either double clicking or selecting "Customize" |"Configure' in the figure, the default value is set to 10.0.

An example of a library actor that uses a PortParameter is the *Sinewave* actor, which is found in the *sources* library in Vergil. It is shown in figure 2.3. If you double click on this actor, you can set the default values for *frequency* and *phase*. But both of these values can also be set by the corresponding ports, which are shown with grey fill.

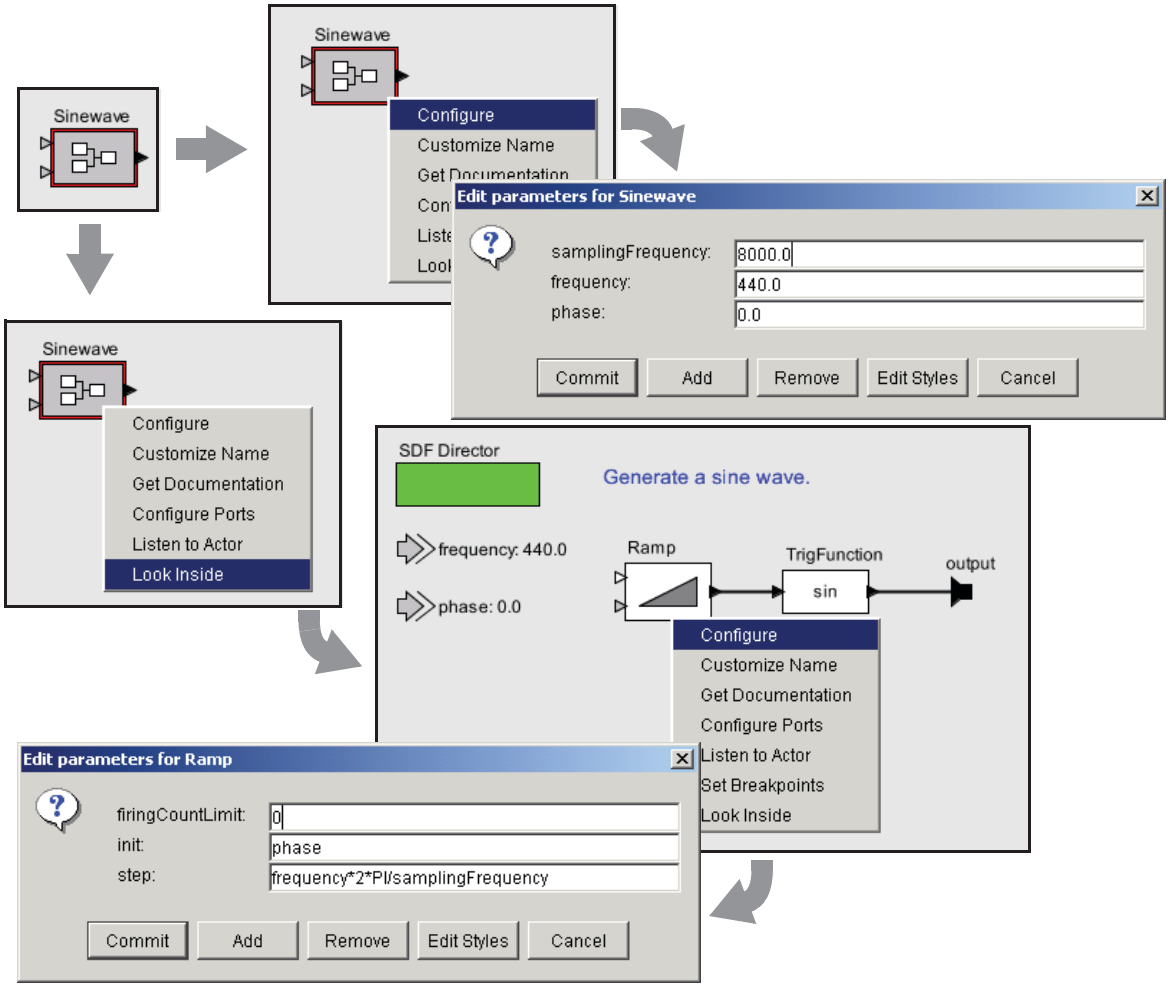

Figure 2.3: Sinewave actor, showing its port parameters, and their use at the lower level of hierarchy

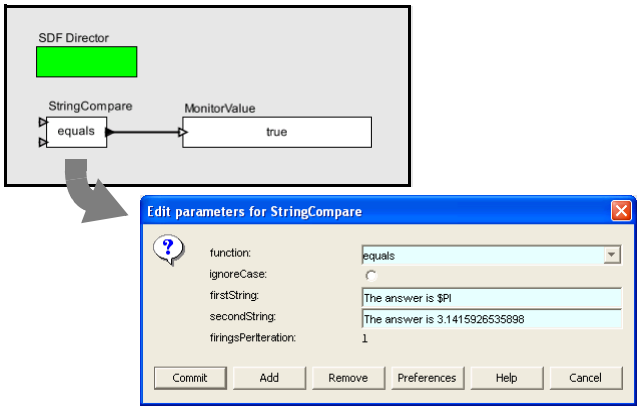

Figure 2.4: String parameters are indicated in the parameter editor boxes by a light blue background. A string parameter can include references to variables in scope with $*name*, where *name* is the name of the variable. In this example, the built-in constant $PI is referenced by name in the first parameter.

2.3.3 String Parameters

Some parameters have values that are always strings of characters. Such parameters support a simple string substitution mechanism where the value of the string can reference other parameters in scope by name using the syntax $*name*, where name is the name of the parameter in scope. For example, the StringCompare actor in figure 2.4 has as the value of *firstString* "The answer is $PI". This references the built-in constant PI. The value of *secondString* is "The answer is 3.1415926535898". As shown in the figure, these two strings are deemed to be equal because $PI is replaced with the value of PI.

2.3.4 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output and no inputs, as shown in Figure 2.5. The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).

2.3.5 State Machines

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

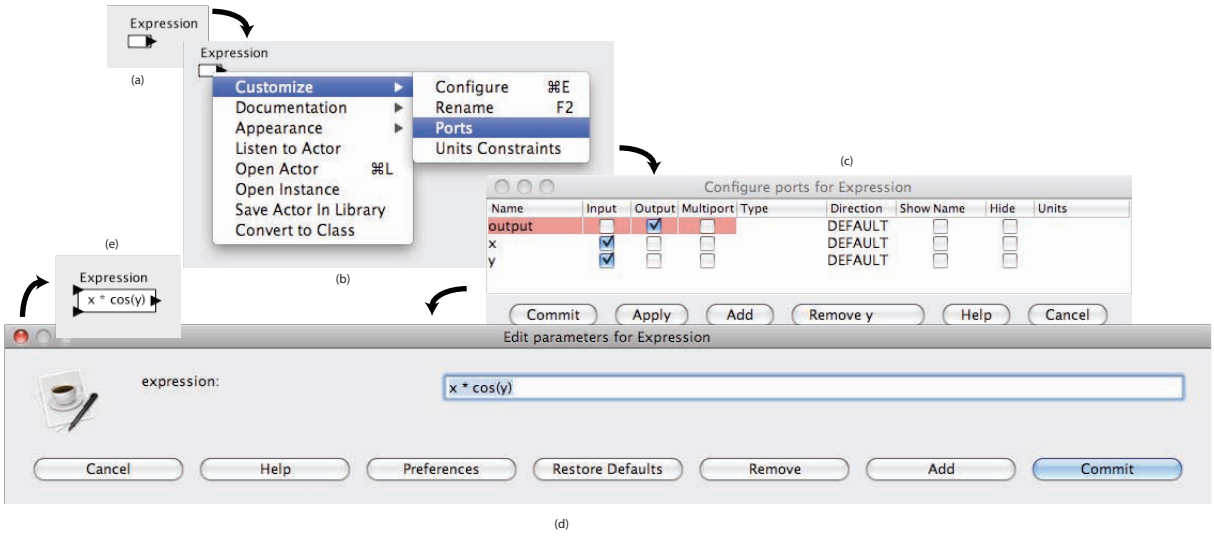

Figure 2.5: Illustration of the Expression actor.

2.4 Composite Data Types

2.4.1 Arrays

Arrays are specified with curly brackets, e.g., "`{1, 2, 3}`" is an array of *int*, while "`{"x", "y", "z"}`" is an array of string. The types are denoted "`arrayType(int,3)`" and "`arrayType(string,3)`" respectively. An array is an ordered list of tokens of any type, with the primary constraint being that the elements all have the same type. If an array is given with mixed types, the expression evaluator will attempt to losslessly convert the elements to a common type. Thus, for example,

```
{1, 2.3}
```

has value

```
{1.0, 2.3}
```

Its type is *arrayType(double)*. The common type might be *arrayType(scalar)*, which is a union type (a type that can contain multiple distinct types). For example,

```
{1, 2.3, true}
```

has value

```
{1, 2.3, true}
```

The value is unchanged, although the type of the array is now *arrayType(scalar)*.

Array types also have the notion of length. For example, in Figure 2.5, the "Type" column may be used to specify the type of a port. Usually, this is not necessary to set the type, since the type inference mechanism will determine the type from the connections. However, it is sometimes necessary to set the type of an array. The Type column accepts values like "arrayType(int)", which specifies an array with an unknown length. "arrayType(int)" is not recommended as it expects a token of UnsizedArrayToken, whose only purpose is to have an unknown array size. Regular array tokens always have a known length. If you must specify an array type, Instead of using "arrayType(int)", specify a type like "arrayType(int,*n*)", where *n* is a non-zero integer that is the length of the array that is expected on the port.

The elements of the array can be given by expressions, as in the example "`{2*pi, 3*pi}`." Arrays can be nested; for example, "`{{1, 2}, {3, 4, 5}}`" is an array of arrays of integers. The elements of an array can be accessed as follows:

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

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

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

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

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

Addition, subtraction, multiplication, division, and modulo of arrays by scalars is also supported, as in the following examples:

```
>> {1.0, 2.0} / 2.0
{0.5, 1.0}
>> 1.0 / {2.0, 4.0}
{0.5, 0.25}
>> 3 * {2, 3}
{6, 9}
>> 12 / {3, 4}
{4, 3}
```

Arrays of length 1 are equivalent to scalars, as illustrated below:

```
>> {1.0, 2.0} / {2.0}
{0.5, 1.0}
>> {1.0} / {2.0, 4.0}
{0.5, 0.25}
>> {3} * {2, 3}
{6, 9}
>> {12} / {3, 4}
{4, 3}
```

A significant subtlety arises when using nested arrays. Note the following example:

```
>> {{1.0, 2.0}, {3.0, 1.0}} / {0.5, 2.0}
{{2.0, 4.0}, {1.5, 0.5}}
```

In this example, the left argument of the divide is an array with two elements, and the right argument is also an array with two elements. The divide is thus element wise. However, each division is the division of an array by a scalar. An array can be checked for equality with another array as follows:

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

For other comparisons of arrays, use the compare() function (see Table 2.5). As with scalars, testing for equality using the == or != operators tests the values, independent of type. For example,

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

You can extract a subarray by invoking the subarray() method as follows:

```
>> {1, 2, 3, 4}.subarray(2, 2)
{3, 4}
```

The first argument is the starting index of the subarray, and the second argument is the length.

You can also extract non-contiguous elements from an array using the extract() method. This method has two forms. The first form takes a *boolean* array of the same length as the original array which indicates which elements to extract, as in the following example:

```
>> {"red", "green", "blue"}.extract({true,false,true})
{"red", "blue"}
```

The second form takes an array of integers giving the indices to extract, as in the following example:

```
>> {"red","green","blue"}.extract({2,0,1,1})
{"blue", "red", "green", ""green''}
```

You can create an empty array with a specific element type using the emptyArray() function. For example, to create an empty array of integers, use:

```
>> emptyArray(int)
{}
```

You can combine arrays into a single array using the concatenate() function. For example,

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

2.4.2 Matrices

In Ptolemy II, arrays are ordered sets of tokens. Ptolemy II also supports matrices, which are more specialized than arrays. They contain only certain primitive types, currently *boolean*, *complex*, *double*, *fixedpoint*, *int*, and *long*. Currently *float*, *short* and *unsignedByte* matrices are not supported. Matrices cannot contain arbitrary tokens, so they cannot, for example, contain matrices. They are intended for data intensive computations. Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., "[1, 2, 3; 4, 5, 5+1]" gives a two by three integer matrix (2 rows and 3 columns). Note that an array or matrix element can be given by an expression. A row vector can be given as "[1, 2, 3]" and a column vector as "[1; 2; 3]". Some MATLAB-style array constructors are supported. For example, "[1:2:9]" gives an array

of odd numbers from 1 to 9, and is equivalent to "[1, 3, 5, 7, 9]." Similarly, "[1:2:9; 2:2:10]" is equivalent to "[1, 3, 5, 7, 9; 2, 4, 6, 8, 10]." In the syntax "[p:q:r]", *p* is the first element, *q* is the step between elements, and *r* is an upper bound on the last element. That is, the matrix will not contain an element larger than *r*. If a matrix with mixed types is specified, then the elements will be converted to a common type, if possible. Thus, for example, "[1.0, 1]" is equivalent to "[1.0, 1.0]," but "[1.0, 1L]" is illegal (because there is no common type to which both elements can be converted losslessly).

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

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

Matrix multiplication works as expected. For example, as seen in the expression evaluator (see figure 2.1),

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

Of course, if the dimensions of the matrix don't match, then you will get an error message. To do element wise multiplication, use the multipyElements() function (see Table 2.9). Matrix addition and subtraction are element wise, as expected, but the division operator is not supported. Element wise division can be accomplished with the divideElements() function, and multiplication by a matrix inverse can be accomplished using the inverse() function (see Table 2.9). A matrix can be raised to an *int*, *short* or *unsignedByte* power, which is equivalent to multiplying it by itself some number of times. For instance,

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

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

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

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

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

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

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

For other comparisons of matrices, use the compare() function (see Table 2.8). As with scalars, testing for equality using the == or != operators tests the values, independent of type. For example,

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

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

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

2.4.3 Records

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

Ordered records behave similarly to normal records except that they preserve the original ordering of the labels rather than alphabetizing them. Ordered records are delimited using square brackets rather than curly braces. For example, `[b="foo", a=1]` is an ordered record token in which 'b' will remain the first label.

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

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

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

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

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

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

yields the result

```
{foodCost=60}
```

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

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

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

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

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

Records can be compared, as in the following examples:

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

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

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

The order of the fields is irrelevant for normal (unordered) records. Hence

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

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

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

Equality comparisons for ordered records respect the original order of the fields. For example,

```
>> [a=1, b=2]==[b=2, a=1]
false
```

Additionally, ordered record fields are always reported in the order in which they are defined. For example,

```
>> [b=2, a=1]
[b=2, a=1]
```

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

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

Finally, You can create an empty record using the emptyRecord() function:

```
>> emptyRecord()
{}
```

2.5 Invoking Methods

Every element and subexpression in an expression represents an instance of the Token class in Ptolemy II (or more likely, a class derived from Token). The expression language supports invocation of any method of a given token, as long as the arguments of the method are of type Token and the return type is Token (or a class derived from Token, or something that the expression parser can easily convert to a token, such as a string, *double*, *int*, etc.). The syntax for this is

(*token*).*methodName*(*args*), where *methodName* is the name of the method and *args* is a comma-separated set of arguments. Each argument can itself be an expression. Note that the parentheses around the *token* are not required, but might be useful for clarity. As an example, the ArrayToken and RecordToken classes have a length() method, illustrated by the following examples:

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

each of which returns the integer 3.

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

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

which returns 3, and

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

which returns 2, and

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

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

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

2.6 Casting

The cast function can be used to explicitly cast a value into a type.

When the cast function is invoked with `cast(type, value)`, where `type` is the target type and `value` is the value to be cast, a new value is returned (if a predefined casting is applicable) that is in the specified type. For example, `cast(long, 1)` yields 1L, which is equal to 1 but is in the long data type, and `cast(string, 1)` yields "1", which is in the string data type.

2.6.1 Object Types

An object token encapsulates a Java object. Methods defined in the Java class of that object can be invoked in an expression. For example, in a model that contains an actor named `C`, `C` in an expression may refer to that actor in an object token.

An object token has a type, which is an object type that is specific for the class of the encapsulated Java object or any class that is a superclass of that class. For example, with `C` being a *Const* actor, when used in an expression `C` is of an object type that is specific to the Java class ptolemy.actor.lib.Const.

An object type specific to Java class `A` can be specified with `object("A")`, and its value is null. Comparison between object tokens is by reference with the Java objects that they encapsulate. Therefore, `object("A") == object("B")` is always true, because the values in both tokens are null.

2.6.2 Relationship between Object Types

An object type A is more specific than object type B if the Java class represented by A is a subclass of that represented by B.

For example, `object("ptolemy.actor.TypedIOPort")` is more specific than `object("ptolemy.actor.IOPort")` and `object("ptolemy.kernel.Port")`.

The most general object type is `object` (without any argument). Conceptually it encapsulates a null class (which, of course, does not exist in Java). The most specific object type is `object("ptolemy.data.type.ObjectType$BottomClass")` (which is not very useful in practice). The family of object types forms a lattice.

2.6.3 Object Tokens

Object tokens are tokens of object types. A predefined object token is `null`, which has null as the value and the null class as the Java class. Its type is `object`. It is special in that it can be cast into any object type. For example, you can cast it into a port with `cast(object("ptolemy.kernel.Port"), null)`. If you enter this in the expression evaluator, you shall see the stringification of the token as `"object(null: ptolemy.kernel.Port)"`.

(Notice that the string here is not a valid Ptolemy expression. In fact, most object tokens do not have string representations that are valid expressions, and therefore, they cannot be stored permanently in a Ptolemy model.)

Except for `null`, for a Ptolemy expression that evaluates to an object token, the Java class repre-

sented by that token's type is always the most specific class. For example, if `C` is a *Const* actor, then `C` in an expression refers to an object token that has actor `C` as its value, and `object("ptolemy.actor.lib.Const")` as its type. You can cast this type into a more general actor type by doing `cast(object("ptolemy.actor.Actor"), C)`.

2.6.4 Casting between Object Types

An object token can be cast into a different object type, as long as the target object type represents a Java class that is in the inheritance hierarchy of the encapsulated object. That class need not always be a superclass of the class that the object type represents.

For example, again let `C` be a *Const* actor. As discussed, the following expression casts it into a more general actor type:

```
cast(object("ptolemy.actor.Actor"), C)
```

The result of the cast is another object token with `C` as its value and `object("ptolemy.actor.Actor")` as its type. That token can be cast back into one of a more specific object type:

```
cast(object("ptolemy.actor.TypedAtomicActor"),
     cast(object("ptolemy.actor.Actor"), C))
```

This is valid because the value `C` is in any of the mentioned classes.

As mentioned, `null` is a special object token that can be cast into any object type. Any object token can also be cast into the most general object type, which is `object`. The only object that can be cast into the most specific object type, `object("ptolemy.data.type.ObjectType$BottomClass")`, is `null`.

2.6.5 Method Invocation

Native Java methods may be invoked on the objects encapsulated in object tokens.

For example, if `C` is a *Const* actor, `C.portList()` returns a list of its ports. The returned list itself is a Java object in the class java.util.List, so it is encapsulated in an object token. You may further invoke `C.portList().isEmpty()` to test whether the list is empty. In that case, the isEmpty() method is invoked on the returned list. The Java isEmpty() method returns a Java java.lang.Boolean value which corresponds to the Ptolemy *boolean* data type, so the value is converted into the latter type.

2.7 Defining Functions

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

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

where `function` is the keyword for defining a function. The type of an argument can be left unspecified, in which case the expression language will attempt to infer it. The function body gives an expression that defines the return value of the function. The return type is always inferred based on the argument type and the expression. For example:

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

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

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

To apply the function to an argument, simply do

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

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

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

Functions can be passed as arguments to certain *higher-order functions* that have been defined (see Table 2.19) . For example, the iterate() function takes three arguments, a function, an integer, and an initial value to which to apply the function. It applies the function first to the initial value, then to the result of the application, then to that result, collecting the results into an array whose length is given by the second argument. For example, to get an array whose values are multiples of 3, try

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

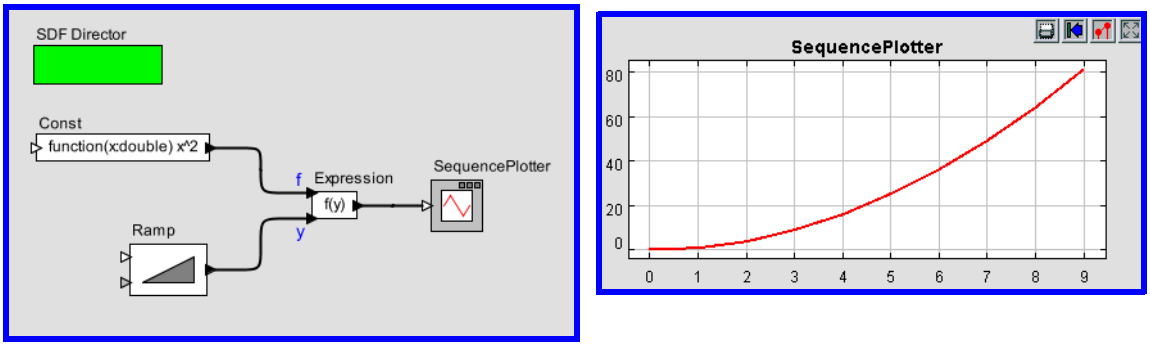

Figure 2.6: Example of a function being passed from one actor to another

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

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

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

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

Functions can also be passed along connections in a Ptolemy II model. Consider the model shown in figure 2.6. In that example, the *Const* actor defines a function that simply squares the argument. Its output, therefore, is a token with type function. That token is fed to the "f" input of the Expression actor. The expression uses this function by applying it to the token provided on the "y" input. That token, in turn, is supplied by the *Ramp* actor, so the result is the curve shown in the plot on the right.

A more elaborate use is shown in figure 2.7 In that example, the *Const* actor produces a function, which is then used by the Expression actor to create new function, which is then used by Expression2 to perform a calculation. The calculation performed here adds the output of the *Ramp* to the square of the output of the *Ramp*.

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

```
>> fact = function(x:int,f:(function(x,f) int)) (x<1?1:x*f(x-1,f))
(function(x:int, f:function(a0:general, a1:general) int) (x<1)?1:(x*f((x-1), f)))
>> factorial = function(x:int) fact(x,fact)
(function(x:int) (function(x:int, f:function(a0:general, a1:general) int)
 (x<1)?1:(x*f((x-1), f)))(x, (function(x:int,
```

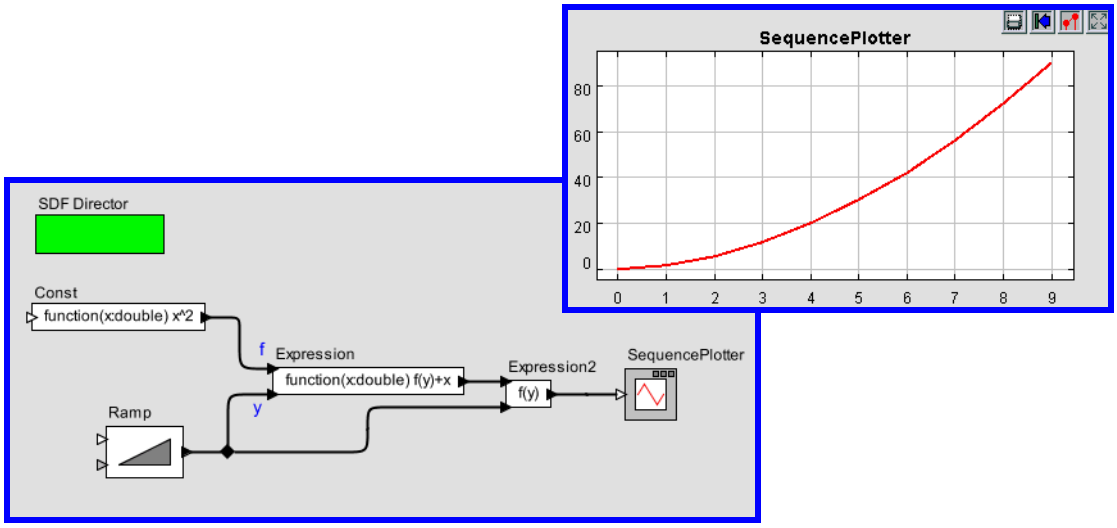

Figure 2.7: More elaborate example with functions passed between actors

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

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

2.8 Built-In Functions

The expression language includes a set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the classes shown in Table 2.2, which together provide a rich set [2]. The functions currently available are shown in the tables in the appendix, which also show the argument types and return types.

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices. Thus, for example, you can fill a row vector with a sine wave using an expression like

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

Or you can construct an array as follows,

[2] Moreover, the set of available can easily be extended if you are writing Java code by registering another class that includes static methods (see the PtParser class in the ptolemy.data.expr package).

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

Functions that operate on type *double* will also generally operate on *int*, *short*, or *unsignedByte*, because these can be losslessly converted to *double*, but not generally on *long* or *complex*. Tables of available functions are shown in the appendix. For example, Table 2.4 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int*, *short* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be *double*. For example,

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

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

Table 2.8 shows other arithmetic functions beyond the trigonometric functions. As with the trigonometric functions, those that indicate that they operate on *double* will also work on *int*, *short* and *unsignedByte*, and unless they indicate otherwise, they will return whatever they return when the argument is *double*. Those functions in the table that take scalar arguments will also operate on matrices and arrays. For example, since the table indicates that the max() function can take *int*, *int* as arguments, then by implication, it can also take *int*, *int*. For example,

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

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

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

Table 2.2: The classes whose static methods are available as functions in the expression language

java.lang.Math	ptolemy.math.IntegerMatrixMath
java.lang.Double	ptolemy.math.DoubleMatrixMath
java.lang.Integer	ptolemy.math.ComplexMatrixMath
java.lang.Long	ptolemy.math.LongMatrixMath
java.lang.String	ptolemy.math.IntegerArrayMath
ptolemy.data.MatrixToken.	ptolemy.math.DoubleArrayStat
ptolemy.data.RecordToken.	ptolemy.math.ComplexArrayMath
ptolemy.data.expr.UtilityFunctions	ptolemy.math.LongArrayMath
ptolemy.data.expr.FixPointFunctions	ptolemy.math.SignalProcessing
ptolemy.math.Complex	ptolemy.math.FixPoint
ptolemy.math.ExtendedMath	ptolemy.data.ObjectToken

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

Table 2.8 shows functions that only work with matrices, arrays, or records (that is, there is no corresponding scalar operation). Recall that most functions that operate on scalars will also operate on arrays and matrices. Table 2.12 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the eval() function is the most flexible.

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

eval() and traceEvaluation()

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

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

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

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

where the filename can be relative to the current working directory (where Ptolemy II was started, as reported by the Java Virtual Machine property `user.dir`), the user's home directory (as reported by the property `user.home`), or the classpath, which includes the directory tree in which Ptolemy II is installed. Note that if eval() is used in an Expression actor, then it will be impossible for the type system to infer any more specific output type than general. If you need the output type to be more specific, then you will need to cast the result of eval(). For example, to force it to type *double*:

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

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

random(), gaussian()

random() and gaussian() shown in Table 2.5 and Table 2.6 return one or more random numbers. With the minimum number of arguments (zero or two, respectively), they return a single number.

With one additional argument, they return an array of the specified length. With a second additional argument, they return a matrix with the specified number of rows and columns.

There is a key subtlety when using these functions in Ptolemy II. In particular, they are evaluated only when the expression within which they appear is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. Thus, for example, if the value parameter of the *Const* actor is set to `random()`, then its output will be a random constant, i.e., it will not change on each firing. The output will change, however, on successive runs of the model. In contrast, if this is used in an Expression actor, then each firing triggers an evaluation of the expression, and consequently will result in a new random number.

property()

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

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

For a complete list of Java Virtual Machine properties, see the Java documentation for java.lang.System.getProperties().

remainder()

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

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

Compare this to

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

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

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

DCT() and IDCT()

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

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos((2n+1)k\frac{\pi}{2D}) \tag{2.1}$$

for k from 0 to $D-1$, where N is the size of the specified array and D is the size of the DCT. If only one argument is given, then D is set to equal the next power of two larger than N. If a second argument is given, then its value is the order of the DCT, and the size of the DCT is 2^{order}. If a third argument is given, then it specifies the scaling factors s_k according to the following table:

Table 2.3: Normalization options for the DCT function

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

The default, if a third argument is not given, is "Normalized." The IDCT function is similar, and can also take one, two, or three arguments. The formula in this case is

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos((2n+1)k\frac{\pi}{2D}) \tag{2.2}$$

.

2.9 Folding

Ptolemy II supports a fold function, which can be used to program a loop in an expression.

- ```
  fold(
      function(x:int, e:int) x + 1,
      0, {1, 2, 3}
  )
  ```

  This computes the length of array {1, 2, 3}. The result is 3, which is equal to {1, 2, 3}.length(). Function `f` here is defined with anonymous function `function(x:int, e:int) x + 1`. Given `x` and arbitrary element `e`, it returns `x + 1`. It is invoked the number of times equal to the number of elements in array {1, 2, 3}. Therefore, `x` is increased 3 times from the starting value 0.

- ```
  fold(
      function(x:int, e:int) x + e,
      0, {1, 2, 3}
  )
  ```

  This computes the sum of all elements in array {1, 2, 3}.

- ```
  fold(
      function(x:arrayType(int), e:int)
          e % 2 == 0 ?  x :  x.append({e}),
      {}, {1, 2, 3, 4, 5}
  )
  ```

  This computes a subarray of array {1, 2, 3, 4, 5} that contains only odd numbers. The result is {1, 3, 5}.

- Let `C` be an actor.

  ```
  fold(
      function(list:arrayType(string),
          port:object("ptolemy.kernel.Port"))
          port.connectedPortList().isEmpty() ?
              list.append({port}) :  list,
      {}, C.portList()
  )
  ```

  This returns a list of `C`'s ports that are not connected to any other port (with `connectedPortList()` being empty). Each port in the returned list is encapsulated in an ObjectToken.
  ```

2.10 Nil Tokens

Null or missing tokens are common in analytical systems like R and SAS where they are used to handle sparsely populated data sources. In database parlance, missing tokens are sometimes called null tokens. Since null is a Java keyword, we use the term "nil". Nil tokens are useful for analyzing real world data such as temperature where the value was not measured during every interval. In principle, an as yet unimplemented method such as an Javerage() method could properly handle nil tokens - when the average() method sees a nil token, it should be ignored. Note that this can lead to uncertainty. For example, if average() is expecting 30 values and 29 of them are nil, then the average will not be very accurate.

If an operation such as add(), divide(), modulo(), multiply(), one(), subtract(), zero() or their corresponding "reverse" operations includes a nil token, then the output is nil. If one of the arguments for isCloseTo() or isEqualTo() is nil, then the method returns false. Methods that return a nil token return a nil token with a specific type so that type safety is preserved. The following tokens have `NIL` values defined: ArrayToken, BooleanToken, ComplexToken, DoubleToken, FloatToken IntToken, LongToken, ShortToken, StringToken, Token, UnsignedByteToken. There is no nil token for the various matrix tokens because the underlying matrices are Java native type matrices that do not support nil.

The expression language defines a constant named `nil` that refers to the Token.NIL field. The cast() expression language function can be used to generate references to the NIL fields of the other classes. For example, "cast(int, nil)" will return a reference to the IntToken.NIL field.

2.11 Fixed Point Numbers

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

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

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

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

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

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

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

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

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

- To create a single FixPoint Token using the expression language:

  ```
  fix(5.34, 10, 4)
  ```

  This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with FixPoint values using the expression language:

  ```
  fix([ -.040609, -.001628, .17853 ], 10, 2)
  ```

  This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a FixPoint value with precision(10/2). The resulting FixMatrixToken will try to fit each element of the given *double* matrix into a 10 bit representation with 2 bits used for the integer part. By default the round quantizer is used.

- To create a single DoubleToken, which is the quantized version of the *double* value given, using the expression language:

  ```
  quantize(5.34, 10, 4)
  ```

  This will create a DoubleToken. The resulting DoubleToken contains the *double* value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with *doubles* quantized to a particular precision using the expression language:

  ```
  quantize([ -.040609, -.001628, .17853 ], 10, 2)
  ```

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

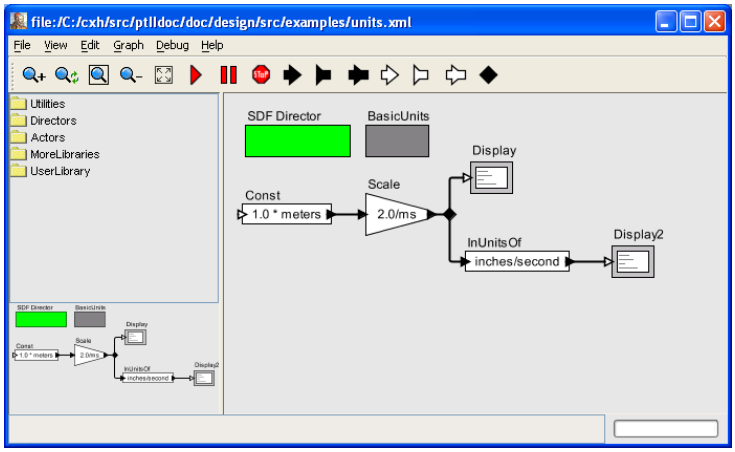

Figure 2.8: Example of a model that includes a unit system.

2.12 Units

Ptolemy II supports units systems, which are built on top of the expression language. Units systems allow parameter values to be expressed with units, such as “1.0 * cm”, which is equal to “0.01 * meters”. These are expressed this way (with the * for multiplication) because “cm” and “meters” are actually variables that become in scope when a units system icon is dragged in to a model. A few simple units systems are provided (mainly as examples) in the utilities library.

A model using one of the simple provided units systems is shown in figure 2.8 This unit system is called *BasicUnits*; the units it defines can be examined by double clicking on its icon, or by invoking “Customize” |“Configure”, as shown in figure 2.9. In that figure, we see that “meters”, “meter”, and “m” are defined, and are all synonymous. Moreover, “cm” is defined, and given value “0.01*meters”, and “in”, “inch” and “inches” are defined, all with value “2.54*cm”.

In the example in figure 2.8, a constant with value “1.0 * meter” is fed into a *Scale* actor with scale factor equal to “2.0/ms”. This produces a result with dimensions of length over time. If we feed this result directly into a *Display* actor, then it is displayed as “2000.0 meters/seconds”, as shown in figure 2.10, top display. The canonical units for length are meters, and for time are seconds.

In figure 2.8, we also take the result and feed it to the *InUnitsOf* actor, which divides its input by its argument, and checks to make sure that the result is unitless. This tells us that 2 meters/ms is equal to about 78,740 inches/second.

The *InUnitsOf* actor can be used to ensure that numbers are interpreted correctly in a model, which can be effective in catching certain kinds of critical errors. For example, if in figure 2.8, we had entered “seconds/inch” instead of “inches/second” in the *InUnitsOf* actor, we would have gotten the exception in figure 2.11 instead of the execution in figure 2.10.

Units systems are built entirely on the expression language infrastructure in Ptolemy II. The units

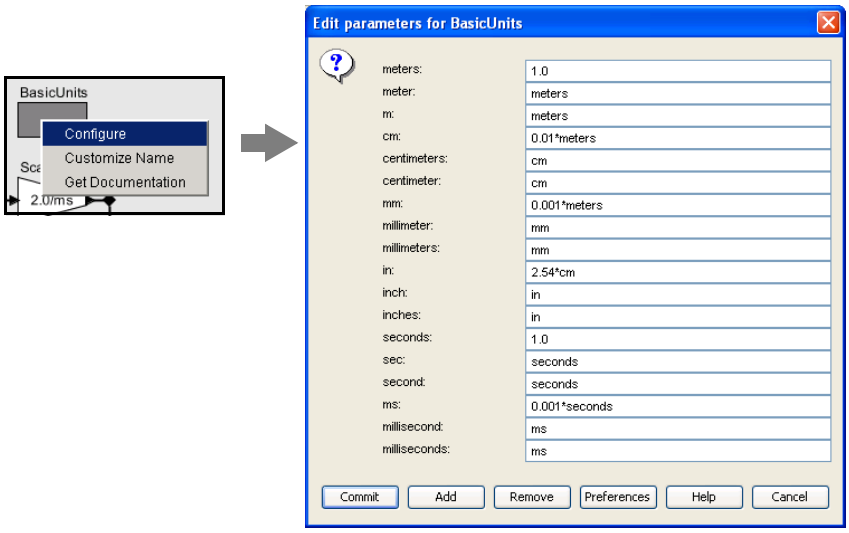

Figure 2.9: Units defined in a units system can be examined by double clicking or by right clicking and selecting “Customize” |“Configure”.

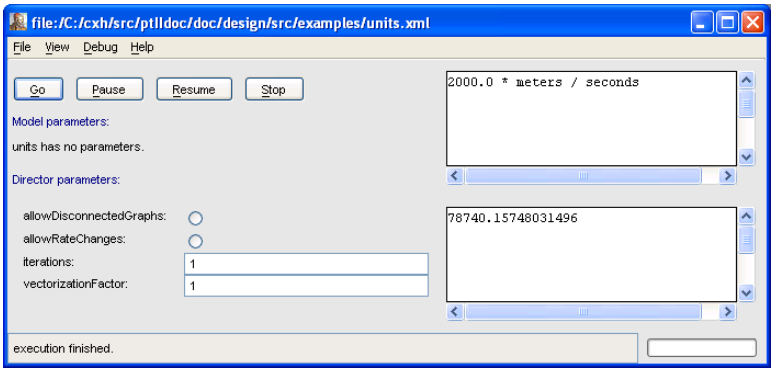

Figure 2.10: Result of running the model in figure 2.8

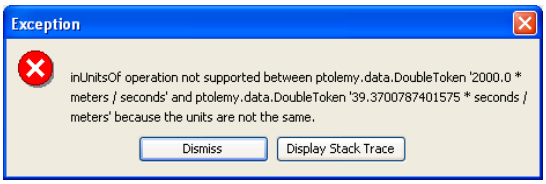

Figure 2.11: Example of an exception resulting from a units mismatch.

system icons actually represent instances of scope-extending attributes, which are attributes whose parameters are in scope as if those parameters were directly contained by the container of the scope extending attribute. That is, scope-extending attributes can define a collection of variables and constants that can be manipulated as a unit. Two fairly extensive units systems are provided, CGSUnitBase and ElectronicUnitBase. Nonetheless, these are intended as examples only, and can no doubt be significantly improved and extended.

2.A Functions

In this appendix, we tabulate the functions available in the expression language. Further explanation of many of these functions is given in section section 2.8 above.

The argument and return types are the widest type that can be used. For example, acos() will take any argument that can be losslessly cast to a *double*, such as unsigned byte, short, integer, float. *long*cannot be cast losslessly cast to *double*, so acos(1L) will fail.

2.A.1 Trigonometric Functions

Table 2.4: Trigonometric functions.

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

2.A.2 Basic Mathematical Functions

Table 2.5: Basic mathematical functions, part 1.

Function	Argument type(s)	Return type	Description
abs	*double* or *complex*	*double* or *int* or *long*(complex returns *double*	absolute value *complex* case: $abs(a+ib) = \|z\| = \sqrt{a^2+b^2}$
angle	*complex*	*double* in the range $[-\pi, \pi]$	angle or argument of the *complex* number: $\angle z$
ceil	*double* or float	*double*	ceiling function, which returns the smallest (closest to negative infinity) *double* value that is not less than the argument and is an integer.
compare	*double*, *double*	*int*	compare two numbers, returning -1, 0, or 1 if the first argument is less than, equal to, or greater than the second.
conjugate	*complex*	*complex*	*complex* conjugate
exp	*double* or *complex*	*double* in the range [0.0, ∞] or complex	exponential function ($e^{argument}$) complex case: $e^{a+ib} = e^a(\cos(b) + i\sin(b))$
floor	*double*	*double*	floor function, which is the largest (closest to positive infinity) value not greater than the argument that is an integer.
gaussian	*double*, *double* or *double*, *double*, *int*, or *double*, *double*, *int*, *int*	*double* or *arrayType(double)* or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see 2.8).
imag	*complex*	*double*	imaginary part
isInfinite	*double*	*boolean*	return true if the argument is infinite
isNaN	*double*	*boolean*	return true if the argument is "not a number"
log	*double* or *complex*	*double* or *complex*	natural logarithm *complex* case: $log(z) = log(abs(z) + i\angle z)$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2

Table 2.6: Basic mathematical functions, part 2.

Function	Argument type(s)	Return type	Description
max	*double*, *double* or double	a scalar of the same type as the arguments	maximum
min	*double*, *double* or double	a scalar of the same type as the arguments	minimum
pow	*double*, *double* or *complex*, *complex*	*double* or complex	first argument to the power of the second
random	no arguments or int or *int*, *int*	*double* or double or [*double*]	one or more random numbers between 0.0 and 1.0 (see 2.8)
real	*complex*	*double*	real part
remainder	*double*, *double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see 2.8).
round	*double*	*long*	round to the nearest long, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0L. If the argument is out of range, the result is either MaxLong or MinLong, depending on the sign.
roundToInt	*double*	*int*	round to the nearest *int*, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0. If the argument is out of range, the result is either MaxInt or MinInt, depending on the sign.
sgn	*double*	*int*	-1 if the argument is negative, 1 otherwise
sqrt	*double* or complex	*double* or complex	square root. If the argument is *double* with value less than zero, then the result is NaN. complex case: $sqrt(z) = \sqrt{\|z\|}(\cos(\frac{\angle z}{2}) + i\sin(\frac{\angle z}{2}))$

Table 2.7: Basic mathematical functions, part 3.

Function	Argument type(s)	Return type	Description
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians
within	*type*, *type*, *double*	*boolean*	return true if the first argument is in the neighborhood of the second, meaning that the distance is less than or equal to the third argument. The first two arguments can be any type for which such a distance is defined. For composite types, arrays, records, and matrices, then return true if the first two arguments have the same structure, and each corresponding element is in the neighborhood.

2.A.3 Matrix, Array, and Record Function.

Table 2.8: Functions that take or return matrices, arrays, or records, part 1.

Function	Argument type(s)	Return type	Description
arrayToMatrix	*arrayType(type)*, *int*, *int*	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
concatenate	*arrayType(type)*, *arrayType(type)*	*arrayType(type)*	Concatenate two arrays.
concatenate	*arrayType(arrayType(type))*	*arrayType(type)*	Concatenate arrays in an array of arrays.
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type*, *type*, *int*	*arrayType(type)*	Create an array with values starting with the first argument, incremented by the second argument, of length given by the third argument.
crop	[*int*], *int*, *int*, *int*, *int* or [*double*], *int*, *int*, *int*, *int* or [*complex*], *int*, *int*, *int*, *int* or [*long*], *int*, *int*, *int*, *int* or	[*int*] or [*double*] or [*complex*] or [*long*] or	Given a matrix of any type, return a submatrix starting at the specified row and column with the specified number of rows and columns.
determinant	[*double*] or [*complex*]	*double* or complex	Return the determinant of the specified matrix.
diag	*arrayType(type)*	[*type*]	Return a diagonal matrix with the values along the diagonal given by the specified array.
divideElements	[*type*], [*type*]	[*type*]	Return the element-by-element division of two matrices
emptyArray	*type*	*arrayType(type)*	Return an empty array whose element type matches the specified token.
emptyRecord		*record*	Return an empty record.

2.A.4 Functions for Evaluating Expressions

2.A.5 Signal Processing Functions

2.A.6 I/O Functions and Other Miscellaneous Functions

Table 2.9: Functions that take or return matrices, arrays, or records, part 2.

Function	Argument type(s)	Return type	Description
find	*arrayType(type)*, *type*	*arrayType(int)*	Return an array of the indices where elements of the specified array match the specified token.
find	*arrayType(boolean)*	*arrayType(int)*	Return an array of the indices where elements of the specified array have value true.
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = \frac{1}{i+j+1}$. A Hilbert matrix is nearly, but not quite singular.
identityMatrixComplex	*int*	[*complex*]	Return an identity matrix with the specified dimension.
identityMatrixDouble	*int*	[*double*]	Return an identity matrix with the specified dimension.
identityMatrixInt	*int*	[*int*]	Return an identity matrix with the specified dimension.
identityMatrixLong	*int*	[*long*]	Return an identity matrix with the specified dimension.
intersect	*record*, *record*	*record*	Return a record that contains only fields that are present in both arguments, where the value of the field is taken from the first record.
inverse	[*double*] or [*complex*]	[*double*] or [*complex*]	Return the inverse of the specified matrix, or throw an exception if it is singular.
matrixToArray	[*type*]	*arrayType(type)*	Create an array containing the values in the matrix
merge	*record*, *record*	*record*	Merge two records, giving priority to the first one when they have matching record labels.

Table 2.10: Functions that take or return matrices, arrays, or records, part 3.

Function	Argument type(s)	Return type	Description
multiplyElements	[*type*], [*type*]	[*type*]	Multiply element wise the two specified matrices.
orthonormalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal columns.
orthonormalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal rows.
repeat	*int*, *type*	*arrayType(type)*	Create an array by repeating the specified token the specified number of times.
sort	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.
sortAscending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.
sortDescending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in descending order. realScalar is any scalar token except *complex*.
subarray	*arrayType(type)*, *int*, *int*	*arrayType(type)*	Extract a subarray starting at the specified index with the specified length.

Table 2.11: Functions that take or return matrices, arrays, or records, part 4.

Function	Argument type(s)	Return type	Description
sum	*arrayType(type)* or [*type*]	*type*	Sum the elements of the specified array or matrix. This throws an exception if the elements do not support addition or if the array is empty (an empty matrix will return zero).
trace	[*type*]	*type*	Return the trace of the specified matrix.
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
zeroMatrixComplex	*int*, *int*	[*complex*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixDouble	*int*, *int*	[*double*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixInt	*int*, *int*	[*int*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixLong	*int*, *int*	[*long*]	Return a zero matrix with the specified number of rows and columns.

Table 2.12: Utility functions for evaluating expressions

Function	Argument type(s)	Return type	Description
eval	*string*	any type	evaluate the specified expression (see 2.8).
parseInt	*string* or *string*, *int*	*int*	return an *int* read from a string, using the given radix if a second argument is provided.
parseLong	*string* or *string*, *int*	*int*	return a long read from a string, using the given radix if a second argument is provided.
toBinaryString	*int* or *long*	*string*	return a binary representation of the argument
toOctalString	*int* or *long*	*string*	return an octal representation of the argument
toString	*double* or *int* or *int*, *int* or *long*	*string*	return a string representation of the argument, using the given radix if a second argument is provided.
traceEvaluation	*string*	*string*	evaluate the specified expression and report details on how it was evaluated (see 2.8).

Table 2.13: Functions performing signal processing operations, part 1.

Function	Argument type(s)	Return type	Description
close	*double*, *double*	*boolean*	Return true if the first argument is close to the second (within EPSILON, where EPSILON is a static public variable of this class).
convolve	*arrayType(double)*, *arrayType(double)* or *arrayType(complex)*, *arrayType(complex)*	*arrayType(double)* or *arrayType(complex)*	Convolve two arrays and return an array whose length is sum of the lengths of the two arguments minus one. Convolution of two arrays is the same as polynomial multiplication.
DCT	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the Discrete Cosine Transform of the specified array, using the specified (optional) length and normalization strategy (see 2.8).
downsample	*arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return a new array with every n-th element of the argument array, where n is the second argument. If a third argument is given, then it must be between 0 and $n-1$, and it specifies an offset into the array (by giving the index of the first output).

Table 2.14: Functions performing signal processing operations, part 2.

Function	Argument type(s)	Return type	Description
FFT	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, int *arrayType(complex)*, *int*	*arrayType(complex)*	Return the Fast Fourier Transform of the specified array. If the second argument is given with value n, then the length of the transform is 2^n. Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
generateBartlett Window	*int*	*arrayType(double)*	Bartlett (rectangular) window with the specified length. The end points have value 0.0, and if the length is odd, the center point has value 1.0. For length M + 1, the formula is: $w(n) = \begin{cases} 2\frac{n}{M}; & if 0 \leq n \leq \frac{M}{2} \\ 2-2\frac{n}{M}; & if \frac{M}{2} \leq n \leq M \end{cases}$
generateBlackman Window	*int*	*arrayType(double)*	Return a Blackman window with the specified length. For length M + 1, the formula is: $w(n) = 0.42 + 0.5\cos(\frac{2\pi n}{M}) + 0.08\cos(\frac{4\pi n}{M})$
generateBlackman HarrisWindow	*int*	*arrayType(double)*	Return a Blackman-Harris window with the specified length. For length M + 1, the formula is: $w(n) = 0.35875 + 0.48829\cos(\frac{2\pi n}{M}) + 0.14128\cos(\frac{4\pi n}{M}) + 0.01168\cos(\frac{6\pi n}{M})$

Table 2.15: Functions performing signal processing operations, part 3.

Function	Argument type(s)	Return type	Description
generateGaussian Curve	arrayType(double), arrayType(double), *int*	*arrayType(double)*	Return a Gaussian curve with the specified standard deviation, extent, and length. The extent is a multiple of the standard deviation. For instance, to get 100 samples of a Gaussian curve with standard deviation 1.0 out to four standard deviations, use generateGaussianCurve(1.0, 4.0, 100).
generateHamming Window	*int*	*arrayType(double)*	Return a Hamming window with the specified length. For length M + 1, the formula is: $w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{M})$
generateHanning Window	*int*	*arrayType(double)*	Return a Hanning window with the specified length. For length M + 1, the formula is: $w(n) = 0.5 - 0.5\cos(\frac{2\pi n}{M})$
generatePolynomial Curve	*arrayType(double)*, *double*, *double*, *int*	*arrayType(double)*	Return samples of a curve specified by a polynomial. The first argument is an array with the polynomial coefficients, beginning with the constant term, the linear term, the squared term, etc. The second argument is the value of the polynomial variable at which to begin, and the third argument is the increment on this variable for each successive sample. The final argument is the length of the returned array.

Table 2.16: Functions performing signal processing operations, part 4.

Function	Argument type(s)	Return type	Description
generateRaised CosinePulse	*double*, *double*, *int*	*arrayType(double)*	Return an array containing a symmetric raised-cosine pulse. This pulse is widely used in communication systems, and is called a "raised cosine pulse" because the magnitude its Fourier transform has a shape that ranges from rectangular (if the excess bandwidth is zero) to a cosine curved that has been raised to be non-negative (for excess bandwidth of 1.0). The elements of the returned array are samples of the function: $h(t) = \frac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}} \times \frac{\cos(\frac{x\pi t}{T})}{1-(\frac{2xt}{T})^2}$, where x is the excess bandwidth (the first argument) and T is the number of samples from the center of the pulse to the first zero crossing (the second argument). The samples are taken with a sampling interval of 1.0, and the returned array is symmetric and has a length equal to the third argument. With an excess Bandwidth of 0.0, this pulse is a sinc pulse.
generateRectangular Window	*int*	*arrayType(double)*	Return an array filled with 1.0 of the specified length. This is a rectangular window.

Table 2.17: Functions performing signal processing operations, part 5.

Function	Argument type(s)	Return type	Description
IDCT	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see 2.8).
IFFT	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, *int* *arrayType(complex)*, *int*	*arrayType(complex)*	inverse fast Fourier transform of the specified array. If the second argument is given with value n, then the length of the transform is 2^n. Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroTo Frequency	*arrayType(complex)*, *arrayType(complex)*, *complex*, *int*	*arrayType(complex)*	Given an array of pole locations, an array of zero locations, a gain term, and a size, return an array of the specified size representing the frequency response specified by these poles, zeros, and gain. This is calculated by walking around the unit circle and forming the product of the distances to the zeros, dividing by the product of the distances to the poles, and multiplying by the gain.

Table 2.18: Functions performing signal processing operations, part 6.

Function	Argument type(s)	Return type	Description
sinc	*double*	*double*	Return the sinc function, $\sin(x)/x$, where special care is taken to ensure that 1.0 is returned if the argument is 0.0.
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	*arrayType(double)*	*arrayType(double)*	Modify the specified array to unwrap the angles. That is, if the difference between successive values is greater than π in magnitude, then the second value is modified by multiples of 2π until the difference is less than or equal to π. In addition, the first element is modified so that its difference from zero is less than or equal to π in magnitude.
upsample	*arrayType(double)*, *int*	*arrayType(double)*	Return a new array that is the result of inserting $n-1$ zeroes between each successive sample in the input array, where n is the second argument. The returned array has length nL, where L is the length of the argument array. It is required that $n > 0$.

Table 2.19: Miscellaneous functions, part 1.

Function	Argument type(s)	Return type	Description
asURL	*string*	*string*	Return a URL representation of the argument.
cast	type1, type2	type1	Return the second argument converted to the type of the first, or throw an exception if the conversion is invalid.
constants	none	*record*	Return a record identifying all the globally defined constants in the expression language.
findFile	*string*	*string*	Given a file name relative to the user directory, current directory, or classpath, return the absolute file name of the first match, or return the name unchanged if no match is found.
filter	function, *arrayType(type)*	*arrayType(type)*	Extract a sub-array consisting of all of the elements of an array for which the given predicate function returns true.
filter	function, *arrayType(type)*, *int*	*arrayType(type)*	Extract a sub-array with a limited size consisting of all of the elements of an array for which the given predicate function returns true.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	function, *int*, *type*	*arrayType(type)*	Return an array that results from first applying the specified function to the third argument, then applying it to the result of that application, and repeating to get an array whose length is given by the second argument.

Table 2.20: Miscellaneous functions, part 2.

Function	Argument type(s)	Return type	Description
map	function, *arrayType(type)*	*arrayType(type)*	Return an array that results from applying the specified function to the elements of the specified array.
property	*string*	*string*	Return a system property with the specified name from the environment, or an empty string if there is none. Some useful properties are java.version, ptolemy.ptII.dir, ptolemy.ptII.dirAsURL, and user.dir.
readFile	*string*	*string*	Get the string text in the specified file, or throw an exception if the file cannot be found. The file can be absolute, or relative to the current working directory (user.dir), the user's home directory (user.home), or the classpath. The readfile function is often used with the eval function.
readResource	*string*	*string*	Get the string text in the specified resource (which is a file found relative to the classpath), or throw an exception if the file cannot be found.
totalMemory	none	*long*	Return the approximate number of bytes used by current objects plus those available for future object allocation.

Chapter 3

Ptolemy Project Coding Style

Authors: Christopher X. Brooks, Edward A. Lee

3.1 Motivation

Collaborative software projects benefit when participants read code created by other participants. The objective of a coding style is to reduce the fatigue induced by unimportant formatting differences and differences in naming conventions. Although individual programmers will undoubtedly have preferences and habits that differ from the recommendations here, the benefits that flow from following these recommendations far outweigh the inconveniences. Published papers in journals are subject to similar stylistic and layout constraints, so such constraints are not new to the academic community.

Software written by the Ptolemy Project participants follows this style guide. Although many of these conventions are arbitrary, the resulting consistency makes reading the code much easier, once you get used to the conventions. We recommend that if you extend Ptolemy II in any way, that you follow these conventions. To be included in future versions of Ptolemy II, the code must follow the conventions.

In general, we follow the Sun Java Style guide (http://java.sun.com/docs/codeconv/). We encourage new developers to use Eclipse (http://www.eclipse.org) as their development platform. Eclipse includes a Java Formatter, and we have found that the Java Conventions style is very close to our requirements. For information about setting up Eclipse to follow the Ptolemy II coding style, see http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm, which is a copy of $PTII/doc/coding/eclipse.htm, where $PTII is the location of your Ptolemy II installation. A file template that follows these rules can be found in $PTII/doc/coding/templates/JavaTemplate.java In addition useful tools are provided in the directories under $PTII/util/ to help enforce the standards.

- lisp/ptjavastyle.el is a lisp module for GNU Emacs that has appropriate indenting rules. This file works well with Emacs under both Unix and Windows.
- testsuite/ptspell is a shell script that checks Java code and prints out an alphabetical list of unrecognized spellings. It properly handles namesWithEmbeddedCapitalization and has a list of author names. This script works best under Unix. Under Windows, it would require the installation of the `ispell` command as /usr/local/bin/ispell. To run this script, type

  ```
  $PTII/util/testsuite/ptspell *.java
  ```

- testsuite/chkjava is a shell script for checking various other potentially bad things in Java code, such as debugging code, and FIXME's. This script works under both Unix and Windows. To run this script, type:

  ```
  $PTII/util/testsuite/chkjava *.java
  ```

- adm/bin/fix-files is a shell script that fixes common problems in files. To run this script, type:

  ```
  $PTII/adm/bin/fix-files *.java
  ```

3.2 Anatomy of a File

A Java file has the structure shown in figures 3.1 and 3.2.

The key points to note about this organization are:

- The file is divided into sections with highly visible delimiters. The sections contain constructors, public variables (including ports and parameters for actor definitions), public methods, protected variables, protected members, private methods, and private variables, in that order. Note in particular that although it is customary in the Java community to list private variables at the beginning of a class definition, we put them at the end. They are not part of the public interface, and thus should not be the first thing you see.
- Within each section, method order to easily search for a particular method (in printouts, for example, finding a method can be very difficult if the order is arbitrary, and use of printouts during design and code reviews is very convenient). If you wish to group methods together, try to name them so that they have a common prefix. Static methods are generally mixed with non-static methods.

The key sections are explained below.

3.2.1 Copyright

The copyright used in Ptolemy II is shown in figure 3.3.

Figure 3.1: Anatomy of a Java file, part1.

```
/* One line description of the class.

   copyright notice
*/
package MyPackageName;

// Imports go here, in alphabetical order, with no wildcards.

//////////////////////////////////////////////////////////////////////////
//// ClassName

/**
   Describe your class here, in complete sentences.
   What does it do?  What is its intended use?

   @author yourname
   @version $Id: codingStyle.tex,v 1.22 2010/10/12 07:02:10 cxh Exp $
   @see classname (refer to relevant classes, but not the base class)
   @since Ptolemy II x.x
   @Pt.ProposedRating Red (yourname)
   @Pt.AcceptedRating Red (reviewmoderator)
*/
public class ClassName {
    /** Create an instance with ... (describe the properties of the
     *  instance). Use the imperative case here.
     *  @param parameterName Description of the parameter.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    public ClassName(ParameterClass parameterName) throws ExceptionClass {
    }

    ////////////////////////////////////////////////////////////////////
    ////                         public variables                   ////

    /** Description of the variable. */
    public int variableName;

    ////////////////////////////////////////////////////////////////////
    ////                         public methods                     ////

    /** Do something... (Use the imperative case here, such as:
     *  "Return the most recently recorded event.", not
     *  "Returns the most recently recorded event.")
     *  @param parameterName Description of the parameter.
     *  @return Description of the returned value.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    public int publicMethodName(ParameterClass parameterName)
        throws ExceptionClass {
        return 1;
    }

    ////////////////////////////////////////////////////////////////////
    ////                         protected methods                  ////

    /** Describe your method, again using imperative case.
     *  @see RelevantClass#methodName()
     *  @param parameterName Description of the parameter.
     *  @return Description of the returned value.
     *  @exception ExceptionClass If ... (describe what
     *   causes the exception to be thrown).
     */
    protected int _protectedMethodName(ParameterClass parameterName)
        throws ExceptionClass {
        return 1;
    }

    ////////////////////////////////////////////////////////////////////
    ////                         protected variables                ////

    /** Description of the variable. */
    protected int _aProtectedVariable;
```

Figure 3.2: Anatomy of a Java file, part2.

```
    ////////////////////////////////////////////////////////////////
    ////                       private methods                  ////
    // Private methods need not have Javadoc comments, although it can
    // be more convenient if they do, since they may at some point
    // become protected methods.
    private int _privateMethodName() {
        return 1;
    }

    ////////////////////////////////////////////////////////////////
    ////                       private variables                ////
    // Private variables need not have Javadoc comments, although it can
    // be more convenient if they do, since they may at some point
    // become protected variables.
    private int _aPrivateVariable;
}
```

Figure 3.3: Copyright notice used in Ptolemy II.

```
Copyright (c) 1999-2010 The Regents of the University of California.
All rights reserved.

Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

PT_COPYRIGHT_VERSION_2
COPYRIGHTENDKEY
```

This style of copyright is often referred to the community as a "BSD" copyright because it was used for the "Berkeley Standard Distribution" of Unix. It is much more liberal that the commonly used "GPL" or "GNU Public License," which encumbers the software and derivative works with the requirement that they carry the source code and the same copyright agreement. The BSD copyright requires that the software and derivative work carry the identity of the copyright owner, as embodied in the lines:

```
Copyright (c) 1999-2010 The Regents of the University of California.
All rights reserved.
```

The copyright also requires that copies and derivative works include the disclaimer of liability in BOLD. It specifically does not require that copies of the software or derivative works carry the middle paragraph, so such copies and derivative works need not grant similarly liberal rights to users of the software.

The intent of the BSD copyright is to maximize the potential impact of the software by enabling uses of the software that are inconsistent with disclosing the source code or granting free redistribution rights. For example, a commercial enterprise can extend the software, adding value, and sell the original software embodied with the extensions. Economic principles indicate that granting free redistribution rights may render the enterprise business model untenable, so many business enterprises avoid software with GPL licenses. Economic principles also indicate that, in theory, fair pricing of derivative works must be based on the value of the extensions, the packaging, or the associated services provided by the enterprise. The pricing cannot reflect the value of the free software, since an informed consumer will, in theory, obtain that free software from another source.

Software with a BSD license can also be more easily included in defense or national-security related applications, where free redistribution of source code and licenses may be inconsistent with the mission of the software. Ptolemy II can include other software with copyrights that are different from the BSD copyright. In general, we do not include software with the GNU General Public License (GPL) license, because provisions of the GPL license require that software with which GLP'd code is integrated also be encumbered by the GPL license. In the past, we have made an exception for GPL'd code that is aggregated with Ptolemy II but not directly combined with Ptolemy II. For example cvs2cl.pl was shipped with Ptolemy II. This file is a GPL'd Perl script that access the CVS database and generates a ChangeLog file. This script is not directly called by Ptolemy II, and we include it as a "mere aggregation" and thus Ptolemy II does not fall under the GPL. Note that we do not include GPL'd Java files that are compiled and then called from Ptolemy II because this would combine Ptolemy II with the GPL'd code and thus encumber Ptolemy II with the GPL.

Another GNU license is the GNU Library General Public License now known as the GNU Lesser General Public License (LGPL). We try to avoid packages that have this license, but we on occasion we have included them with Ptolemy II. The LGPL license is less strict than the GPL - the LGPL permits linking with other packages without encumbering the other package. In general, it is best if you

avoid GNU code. If you are considering using code with the GPL or LGPL, we encourage you to carefully read the license and to also consult the GNU GPL FAQ at http://www.gnu.org/licenses/gpl-faq.html. We also avoid including software with proprietary copyrights that do not permit redistribution of the software.

The date of the copyright for newly created files should be the current year:

```
Copyright (c) 2010 The Regents of the University of California.
All rights reserved.
```

If a file is a copy of a previously copyrighted file, then the start date of the new file should be the same as that of the original file:

```
Copyright (c) 1999-2010 The Regents of the University of California.
All rights reserved.
```

Ideally, files should have at most one copyright from one institution. Files with multiple copyrights are often in legal limbo if the copyrights conflict. If necessary, two institutions can share the same copyright:

```
Copyright (c) 2010 The Ptolemy Institute and The Regents of the
University of California.
All rights reserved.
```

Ptolemy II includes a copyright management system that will display the copyrights of packages that are included in Ptolemy II at runtime. To see what packages are used in a particular Ptolemy configuration, do "Help" |"About" |"Copyright". Currently, URLs such as `about:` and `about:copyright` are handled specially. If, within Ptolemy, the user clicks on a link with a target URL of `about:copyright`, then we eventually invoke code within $PTII/ptolemy/actor/gui/GenerateCopyrights.java. This class searches the runtime environment for particular packages and generates a web page with the links to the appropriate copyrights if certain packages are found.

3.2.2 Imports

The imports section identifies the classes outside the current package on which this class depends. The package structure of Ptolemy II is carefully constructed so that core packages do not depend on

more elaborate packages. This limited dependencies makes it possible to create derivative works that leverage the core but drastically modify or replace the more advanced capabilities. By convention, we list imports by full class name, as follows:

```
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.Entity;
import ptolemy.kernel.Port;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.Locatable;
import ptolemy.kernel.util.NameDuplicationException;
```

in particular, we do not use the wildcards supported by Java, as in:

```
import ptolemy.kernel.*;
import ptolemy.kernel.util.*;
```

The reason that we discourage wildcards is that the full class names in import statements makes it easier find classes that are referenced in the code. If you use an IDE such as Eclipse, it is trivially easy to generate the import list in this form, so there is no reason to not do it. Imports are ordered alphabetically by package first, then by class name, as shown above.

3.3 Comment Structure

Good comments are essential to readable code. In Ptolemy II, comments fall into two categories, Javadoc comments, which become part of the generated documentation, and code comments, which do not. Javadoc comments are used to explain the interface to a class, and code comments are used to explain how it works. Both Javadoc and code comments should be complete sentences and complete thoughts, capitalized at the beginning and with a period at the end. Spelling and grammar should be correct.

3.3.1 Javadoc and HTML

Javadoc is a program distributed with Java that generates HTML documentation files from Java source code files [1]. Javadoc comments begin with "`/**`" and end with "`*/`". The comment immediately preceding a method, member, or class documents that method, member, or class. Ptolemy II classes include Javadoc documentation for all classes and all public and protected members and

[1] See http://java.sun.com/j2se/javadoc/writingdoccomments/ for guidelines from Sun Microsystems on writing Javadoc comments.

methods. Members and methods should appear in alphabetical order within their protection category (public, protected etc.) so that it is easy to find them in the Javadoc output. When writing Javadoc comments, pay special attention to the first sentence of each Javadoc comment. This first sentence is used as a summary in the Javadocs. It is extremely helpful if the first sentence is a cogent and complete summary. Javadoc comments can include embedded HTML formatting. For example, by convention, in actor documentation, we set in italics the names of the ports and parameters using the syntax:

```
/** Read inputs from the <i>input</i> port ... */
```

The Javadoc program gives extensive diagnostics when run on a source file. Our policy is to format the comments until there are no Javadoc warnings. Private members and methods need not be documented by Javadoc comments. The doccheck tool from
http://java.sun.com/j2se/javadoc/doccheck/index.html gives even more extensive diagnostics in HTML format. We encourage developers to run doccheck and fix all warnings. The nightly build at http://chess.eecs.berkeley.edu/ptexternal/nightly/ includes a run of doccheck.

3.3.2 Class documentation

The class documentation is the Javadoc comment that immediately precedes the class definition line. It is a particularly important part of the documentation. It should describe what the class does and how it is intended to be used. When writing it, put yourself in the mind of the user of your class. What does that person need to know? In particular, that person probably does not need to know how you accomplish what the class does. She only needs to know what you accomplish. A class may be intended to be a base class that is extended by other programmers. In this case, there may be two distinct sections to the documentation. The first section should describe how a user of the class should use the class. The second section should describe how a programmer can meaningfully extend the class. Only the second section should reference protected members or methods. The first section has no use for them. Of course, if the class is abstract, it cannot be used directly and the first section can be omitted.

Comments should include honest information about the limitations of a class.

Each class comment should also include the following Javadoc tags:

- @author - The @author tag should list the authors and contributors of a class, for example:

  ```
  @author Claudius Ptolemaus, Contributor: Tycho Brahe
  ```

  If you are creating a new file that is based on an older file, move the authors of the older file towards the end:

  ```
  @author Copernicus, Based on Galileo.java by Claudius Ptolemaus, Contributor: Tycho Brahe
  ```

The general rule is that only people who actually contributed to the code should be listed as authors. So, in the case of a new file, the authors should only be people who edited the file. Note that all the authors should be listed on one line. Javadoc will not include authors listed on a separate line.

- @version - The @version tag includes text that Subversion automatically substitutes in the version. The @version tag starts out with: `@version $Id$` When the file is committed using Subversion, the `@version $Id$` gets substituted, so the tag might look like: `@version $Id: makefile 43472 2006-08-21 23:16:56Z cxh $`

  Note that for Subversion keyword substitution to work properly, the file must have the svn:keyword attribute set. In addition, it is best if the svn:native property is set. Below is how to check the values for a file named README.txt:

```
bash-3.2$ svn proplist README.txt
Properties on 'README.txt':
  svn:keywords
  svn:eol-style
bash-3.2$ svn propget svn:keywords README.txt
Author Date Id Revision
bash-3.2$ svn propget svn:eol-style README.txt
native
```

  To set the properties on a file:

```
svn propset svn:keywords "Author Date Id Revision" filename
svn propset svn:eol-style native filename
```

  For details about properly configuring your Subversion environment, see http://chess.eecs.berkeley.edu/ptexternal/wiki/Main/Subversion#KeywordSubstitution

- @since - The @since tag refers the release that the class first appeared in. Usually, this is one decimal place after the current release. For example if the current release is 8.0.1, then the @since tag on a new file would read:

```
@since Ptolemy II 8.1
```

  Adding an @since tag to a new class is optional, we usually update these tags by running a script when we do a release. However, authors should be aware of their meaning. Note that the @since tag can also be used when a method is added to an existing class, which will help users notice new features in older code.

- @Pt.ProposedRating
- @Pt.AcceptedRating Code rating tags, discussed below.

3.3.3 Code rating

The Javadoc tags `@Pt.ProposedRating` and `@Pt.AcceptedRating` contain code rating information. Each tag includes the color (one of `red`, `yellow`, `green` or `blue`) and the Subversion login of the person responsible for the proposed or accepted rating level, for example:

```
@Pt.ProposedRating blue ptolemy
@Pt.AcceptedRating green ptolemy
```

The intent of the code rating is to clearly identify to readers of the file the level of maturity of the contents. The Ptolemy Project encourages experimentation, and experimentation often involves creating immature code, or even "throw-away" code. Such code is red. We use a lightweight software engineering process documented in "Software Practice in the Ptolemy Project,"[17] to raise the code to higher ratings. That paper documents the ratings as:

- Red code is untrusted code. This means that we have no confidence in the design or implementation (if there is one) of this code or design, and that anyone that uses it can expect it to change substantially and without notice. All code starts at red.
- Yellow code is code with a trusted design. We have a reasonable degree of confidence in the design, and do not expect it to change in any substantial way. However, we do expect the API to shift around a little during development.
- Green code is code with a trusted implementation. We have confidence that the implementation is sound, based on test suites and practical application of the code. If possible, we try not to release important code unless it is green.
- Blue marks polished and complete code, and also represents a firm commitment to backwards-compatibility. Blue code is completely reviewed, tested, documented, and stressed in actual usage.

The Javadoc doclet at $PTII/doc/doclets/RatingTaglet.java adds the ratings to the Javadoc output.

3.3.4 Constructor documentation

Constructor documentation usually begins with the phrase "Construct an instance that ..." and goes on to give the properties of that instance. Note the use of the imperative case. A constructor is a command to construct an instance of a class. What it does is construct an instance.

3.3.5 Method documentation

Method documentation needs to state what the method does and how it should be used. For example:

```
/** Mark the object invalid, indicating that when a method
 *  is next called to get information from the object, that
 *  information needs to be reconstructed from the database.
 */
public void invalidate() {
   _valid = false;
}
```

By contrast, here is a poor method comment:

```
/** Set the variable _valid to false.
 */
public void invalidate() {
   _valid = false;
}
```

While this certainly describes what the method does from the perspective of the coder, it says nothing useful from the perspective of the user of the class, who cannot see the (presumably private) variable _valid nor how that variable is used. On closer examination, this comment describes how the method is accomplishing what it does, but it does not describe what it accomplishes. Here is an even worse method comment:

```
/** Invalidate this object.
 */
public void invalidate() {
   _valid = false;
}
```

This says absolutely nothing. Note the use of the imperative case in all of the above comments. It is common in the Java community to use the following style for documenting methods:

```
/** Sets the expression of this variable.
 *  @param expression The expression for this variable.
 */
public void setExpression(String expression) {
...
}
```

We use instead the imperative case, as in

```
/** Set the expression of this variable.
 *  @param expression The expression for this variable.
 */
public void setExpression(String expression) {
...
}
```

The reason we do this is that our sentence is a well-formed, grammatical English sentence, while the usual convention is not (it is missing the subject). Moreover, calling a method is a command "do this," so it seems reasonable that the documentation say "Do this." The use of imperative case has a large impact on how interfaces are documented, especially when using the listener design pattern. For instance, the java.awt.event.ItemListener interface has the method:

```
/** Invoked when an item has been selected or deselected.
 *  The code written for this method performs the operations
 *  that need to occur when an item is selected (or deselected).
 */
void itemStateChanged(ItemEvent e);
```

A naive attempt to rewrite this in imperative tense might result in:

```
/** Notify this object that an item has been selected or deselected.
 */
void itemStateChanged(ItemEvent e);
```

However, this sentence does not capture what the method does. The method may be called in order to notify the listener, but the method does not "notify this object". The correct way to concisely document this method in imperative case (and with meaningful names) is:

```
/** React to the selection or deselection of an item.
 */
void itemStateChanged(ItemEvent event);
```

The above is defining an interface (no implementation is given). To define the implementation, it is also necessary to describe what the method does:

```
/** React to the selection or deselection of an item by doing...
 */
void itemStateChanged(ItemEvent event) { ... implementation ... }
```

Comments for base class methods that are intended to be overridden should include information about what the method generally does, plus information that a programmer may need to override it. If the derived class uses the base class method (by calling `super.methodName()`), but then appends to its behavior, then the documentation in the derived class should describe both what the base class does and what the derived class does.

3.3.6 Referring to methods in comments

By convention, method names are set in the default font, but followed by empty parentheses, as in

```
/** The fire() method is called when ... */
```

The parentheses are empty even if the method takes arguments. The arguments are not shown. If the method is overloaded (has several versions with different argument sets), then the text of the documentation needs to distinguish which version is being used. Other methods in the same class may be linked to with the @link ... Javadoc tag. For example, to link to a foo() method that takes a String:

```
* Unlike the {@link #foo(String)} method, this method ...
```

Methods and members in the same package should have an octothorpe (# sign) prepended. Methods and members in other classes should use the fully qualified class name:

```
{@link ptolemy.util.StringUtilities.substitute(String, String, String)}
```

Links to methods should include the types of the arguments. To run Javadoc on the classes in the current directory, run `make docs`, which will create the HTML javadoc output in the doc/codeDoc subdirectory. To run Javadoc for all the common packages, run `cd $PTII/doc; make docs`. The output will appear in $PTII/doc/codeDoc. Actor documentation can be viewed from within Vergil, right clicking on an actor and selecting "Documentation" |"Get Documentation".

3.3.7 Tags in method documentation

Methods should include Javadoc tags @param (one for each parameter), @return (unless the return type is void), and @exception (unless no exceptions are thrown). Note that we do not use the @throws tag, and that @returns is not a legitimate Javadoc tag, use @return instead. The annotation for the arguments (the @param statement) need not be a complete sentence, since it is usually presented in tabular format. However, we do capitalize it and end it with a period. Exceptions that are thrown by a method need to be identified in the Javadoc comment. An @exception tag should read like this:

```
* @exception MyException If such and such occurs.
```

Notice that the body always starts with "If", not "Thrown if", or anything else. Just look at the Javadoc output to see why. In the case of an interface or base class that does not throw the exception, use the following:

```
* @exception MyException Not thrown in this base class. Derived
*  classes may throw it if such and such happens.
```

The exception still has to be declared so that derived classes can throw it, so it needs to be documented as well.

3.3.8 FIXME annotations

We use the keyword "FIXME" in comments to mark places in the code with known problems. For example:

```
// FIXME: The following cast may not always be safe.
Foo foo = (Foo)bar;
```

By default, Eclipse will highlight FIXMEs.

3.4 Code Structure

3.4.1 Names of classes and variables

In general, the names of classes, methods and members should consist of complete words separated using internal capitalization [2]. Class names, and only class names, have their first letter capitalized, as in AtomicActor. Method and member names are not capitalized, except at internal word boundaries, as in getContainer(). Protected or private members and methods are preceded by a leading underscore "_" as in _protectedMethod(). Static final constants should be in uppercase, with words separated by underscores, as in INFINITE_CAPACITY. A leading underscore should be used if the constant is protected or private. Package names should be short and not capitalized, as in "de" for the discrete-event domain. In Java, there is no limit to name sizes (as it should be). Do not hesitate to use long names.

[2] Yes, there are exceptions (NamedObj, CrossRefList, IOPort). Many discussions dealt with these names, and we still regret not making them complete words.

3.4.2 Indentation and brackets

Nested statements should be indented by 4 characters, as in:

```
if (container != null) {
    Manager manager = container.getManager();
    if (manager != null) {
        manager.requestChange(change);
    }
}
```

Closing brackets should be on a line by themselves, aligned with the beginning of the line that contains the open bracket. Please avoid using the Tab character in source files. The reason for this is that code becomes unreadable when the Tab character is interpreted differently by different programs. Your text editor should be configured to react to the Tab key by inserting spaces rather than the tab character. To set up Emacs to follow the Ptolemy II indentation style, see $PTII/util/lisp/ptemacs.el. To set up Eclipse to follow the Ptolemy II indentation style, see the instructions in $PTII/doc/coding/eclipse.htm. Long lines should be broken up into many small lines. The easiest places to break long lines are usually just before operators, with the operator appearing on the next line. Long strings can be broken up using the + operator in Java, with the + starting the next line. Continuation lines are indented by 8 characters, as in the throws clause of the constructor in figure 3.1.

3.4.3 Spaces

Use a space after each comma:

```
Right: foo(a, b);
Wrong: foo(a,b);
```

Use spaces around operators such as plus, minus, multiply, divide or equals signs, after semicolons and after keywords like if, else, for, do, while, try, catch and throws:

```
Right: a = b + 1;
Wrong: a=b+1;

Right: for(i = 0; i < 10; i += 2)
Wrong: for (i=0 ;i<10;i+=2)

Right: if ( a == b) {
Wrong: if(a==b)
```

Note that the Eclipse clean up facility will fix these problems, see
http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm.

3.4.4 Exceptions

A number of exceptions are provided in the kernel.util package. Use these exceptions when possible because they provide convenient constructor arguments of type Nameable that identify the source of the exception by name in a consistent way.

A key decision you need to make is whether to use a compile-time exception or a run-time exception. A run-time exception is one that implements the RuntimeException interface. Run-time exceptions are more convenient in that they do not need to be explicitly declared by methods that throw them. However, this can have the effect of masking problems in the code.

The convention we follow is that a run-time exception is acceptable only if the cause of the exception can be tested for prior to calling the method. This is called a *testable precondition*. For example, if a particular method will fail if the argument is negative, and this fact is documented, then the method can throw a run-time exception if the argument is negative. On the other hand, consider a method that takes a string argument and evaluates it as an expression. The expression may be malformed, in which case an exception will be thrown. Can this be a run-time exception? No, because to determine whether the expression is malformed, you really need to invoke the evaluator. Making this a compile-time exception forces the caller to explicitly deal with the exception, or to declare that it too throws the same exception. In general, we prefer to use compile-time exceptions wherever possible.

When throwing an exception, the detail message should be a complete sentence that includes a string that fully describes what caused the exception. For example

```
throw IllegalActionException(this,
        "Cannot append an object of type: "
            + obj.getClass().getName() + " because "
            + "it does not implement Cloneable.");
```

Note that the exception not only gives a way to identify the objects that caused the exception, but also why the exception occurred. There is no need to include in the message an identification of the "this" object passed as the first argument to the exception constructor. That object will be identified when the exception is reported to the user.

If an exception is caught, be sure to use exception chaining to include the original exception. For example:

```
String fileName = foo();
try {
```

```
    // Try to open the file
} catch (IOException ex) {
    throw new IllegalActionException(this, ex,
            "Failed to open '" + fileName + "'");
}
```

Note that it is almost always an error to call printStackTrace() because if Ptolemy was invoked from a menu choice, then there is no stderr or stdout and the stack trace will not be visible. It is frequently an error to catch and ignore an exception because the error condition sometimes actually does matter. Often errors are caught and ignored because the call tree does not properly handle exceptions. Burying the problem by ignoring an exception only makes the problem worse. The proper solution is to throw an exception, often IllegalActionException and declare that methods throw the exception. It is often an error to call MessageHandler methods directly because these methods interrupt the flow of execution in non-gui or non-command shell environment.

3.4.5 Code Cleaning

Code cleaning is the act of homogenizing the coding style, looking for and repairing common problems. Fortunately, Eclipse includes a file formatter and a cleaner that fixes many common problems. Software that is to be formally released should be cleaned according to the guidelines set forth in $PTII/doc/coding/releasemgt.htm

3.5 Directory naming conventions

Individual demonstrations should be in directories under a `demo/` directory. The name of the directory, and the name of the model should match and both begin with capital letters. The demos should be capitalized so that it is possible to generate code for demonstrations. For example, the Butterfly demonstration is in `sdf/demo/Butterfly/Butterfly.xml`. All other directories begin with lower case letters and should consist solely of lower case letters. Java package names with embedded upper case letters are not encouraged.

3.6 Makefiles

The Ptolemy tree uses makefiles to provide a manifest of what files should be shipped with the release and to break the system up into modules. The advantage of this system is that we ship only the files that are necessary.

There are a few different types of makefiles

- makefiles that have no subdirectories that contain source code. For example, $PTII/ptolemy/kernel/util contains Java files but has no subdirectories that contain Java source code.
- makefiles that have one or more subdirectories that contain source code. For example, $PTII/ptolemy/kernel contains Java files and ptolemy/kernel has subdirectories such as $PTII/ptolemy/kernel/util that contain Java source code.
- makefiles in directories that do not have source code in the current directory, but have subdirectories that contain source code. For example, the ptolemy directory does not contain Java files, but does contain subdirectories that contain Java files.
- makefiles that contain tests. For example, ptolemy/kernel/util/test contains Tcl tests.
- makefiles in $PTII/mk that are included by other makefiles.

3.6.1 An example makefile

Below are the sections of $PTII/ptolemy/kernel/util/makefile.

```
# Makefile for the Java classes used to implement the Ptolemy kernel
```

Each makefile has a one line description of the purpose of the makefile.

```
#
# @Authors: Christopher Brooks, based on a file by Thomas M. Parks
#
```

The authors for the makefile.

```
# @version $Id: makefile 56450 2009-12-06 06:54:56Z eal $
```

The version control version. We use `$Id$`, which gets expanded by Subversion to include the revision number, the date of the last revision and the login of the person who made the last revision.

```
# @Copyright (c) 1997-2010 The Regents of the University of California.
```

The copyright year should be the start with the year the makefile file was first created, so when creating a new file, use the current year, for example "Copyright (c) 2010 The Regents . . ."

```
# All rights reserved.
#
# Permission is hereby granted, without written agreement and without
# license or royalty fees, to use, copy, modify, and distribute this
# software and its documentation for any purpose, provided that the
# above copyright notice and the following two paragraphs appear in all
```

```
# copies of this software.
#
# IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
# FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
# THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
# PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
# CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
# ENHANCEMENTS, OR MODIFICATIONS.
#
#  PT_COPYRIGHT_VERSION_2
#  COPYRIGHTENDKEY
```

The new BSD copyright appears in each makefile.

```
ME =  ptolemy/kernel/util
```

The makefile variable ME is set to the directory where that includes the makefile. Since this makefile is in ptII/ptolemy/kernel/util, ME is set to that directory. The ME variable is primarily used by make to print informational messages.

```
DIRS =  test
```

The DIRS makefile variable lists each subdirectory in which make is to be run. Any subdirectory that contains a makefile should be listed in DIRS.

```
# Root of the Ptolemy II directory
ROOT = ../../..
```

The ROOT makefile variable is a relative path to the $PTII directory. This variable is relative so as to avoid problems if the $PTII environment variable is not set or is set to a different version of Ptolemy II.

```
CLASSPATH =  $(ROOT)
```

Set the Java classpath to the value of the ROOT makefile variable, which should be the same as the $PTII environment variable. Note that if this makefile contains Java files that require third party software contained in jar files not usually found in the Java classpath, then CLASSPATH would be set to include those jar files, for example CLASSPATH = $(ROOT)$(CLASSPATHSEPARATOR)$(DIVA_JAR) would include ptII/lib/diva.jar, where the DIVA_JAR makefile variable is defined in ptII.mk

```
# Get configuration info
CONFIG = $(ROOT)/mk/ptII.mk
include $(CONFIG)
```

The above includes $PTII/mk/ptII.mk. The way the makefiles work is that the $PTII/configure script examines the environment, and then reads in the $PTII/mk/ptII.mk.in file, substitutes in user specific values and creates $PTII/mk/ptII.mk. Each makefile refers to $PTII/mk/ptII.mk, which defines variable settings such as the location of the compilers.

```
# Flags to pass to javadoc. (Override value in ptII.mk)
JDOCFLAGS =  -author -version -public $(JDOCBREAKITERATOR) $(JDOCMEMORY) $(JDOCTAG)
```

Directory specific makefile variables appear here. This variable sets JDOCFLAGS, which is used if "make docs" is run in this directory. JDOCFLAGS is not often used, we include it here for completeness.

```
# Used to build jar files
PTPACKAGE =  util
PTCLASSJAR =  $(PTPACKAGE).jar
```

PTPACKAGE is the directory name of this directory. In this example, the makefile is in ptolemy/kernel/util, so PTPACKAGE is set to util. PTPACKAGE is used by PTCLASSJAR to name the jar file when make install is run. For this file, running `make install` will create util.jar. If a directory contains subdirectories that have source files, then PTCLASSJAR is not set and PTCLASSALLJAR and PTCLASSALLJARS is set, see below.

```
JSRCS = \
        AbstractSettableAttribute.java \
        Attribute.java \
        BasicModelErrorHandler.java \
```

And so on . . .

```
        ValueListener.java \
        Workspace.java
```

A list of all the .java files to be included. The reason that each Java file is listed separately is to avoid shipping test files and random kruft. Each file that is listed should follow this style guide.

```
EXTRA_SRCS =        $(JSRCS)
```

EXTRA_SRCS contains all the source files that should be present. If there are files such as icons or .xml files that should be included, then OTHER_FILES_TO_BE_JARED is set to include those files and the makefile would include:
EXTRA_SRCS = $(JSRCS) $(OTHER_FILES_TO_BE_JARED)

```
# Sources that may or may not be present, but if they are present, we don't
# want make checkjunk to barf on them.
# Don't include demo or DIRS here, or else 'make sources' will run 'make demo'
MISC_FILES =        $(DIRS)

# make checkjunk will not report OPTIONAL_FILES as trash
# make distclean removes OPTIONAL_FILES
OPTIONAL_FILES = \
        doc \
        'CrossRefList$$1.class' \
        'CrossRefList$$CrossRef.class' \
```

And so on . . .

```
       'Workspace$$ReadDepth.class' \
        $(PTCLASSJAR)
```

MISC_FILES and OPTIONAL_FILES are used by the "make checkjunk" command. The checkjunk target prints out the names of files that should not be present. We use checkjunk as part of the release process. MISC_FILES should *not* include the demo directory or else running make sources will invoke the demos. To determine the value of OPTIONAL_FILES, run `make checkjunk` and add the missing .class files. Since the inner classes have $ in their name, we need to use single quotes around the inner class name and repeat the $ to stop make from performing substitution.

```
JCLASS = $(JSRCS:%.java=%.class)
```

JCLASS uses a make macro to read the value of JSRCS and substitute in .class for .java. JCLASS is used to determine what .class files should be created when make is run.

```
all: jclass
install: jclass $(PTCLASSJAR)
```

The all rule is the first rule in the makefile, so if the command make is run with no arguments, then the all rule is run. The all rule runs the jclass rule, which compiles the java files. The install rule is run if make install is run. The install rule is like the all rule in that the java files are compiled. The install rule also depends on the value of PTCLASSJAR makefile variable, which means that make install also creates util.jar

```
# Get the rest of the rules
include $(ROOT)/mk/ptcommon.mk
```

The rest of the rules are defined in ptcommon.mk

3.6.2 jar files

If a directory contains subdirectories that contain sources or resources necessary at runtime, then the jar file in that directory should contain the contents of the jar files in the subdirectories. For example, $PTII/ptolemy/kernel.jar contains the .class files from $PTII/ptolemy/kernel/util and other subdirectories.

Using $PTII/ptolemy/kernel/makefile as an example, we discuss the lines that are different from the example above.

```
DIRS =                  util attributes undo test
```

DIRS contains each subdirectory in which make will be run

```
# Used to build jar files
PTPACKAGE =          kernel
PTCLASSJAR =
```

Note that in ptolemy/kernel/util, we set PTCLASSJAR, but here it is empty.

```
# Include the .class files from these jars in PTCLASSALLJAR
PTCLASSALLJARS = \
                 attributes/attributes.jar \
                 undo/undo.jar \
                 util/util.jar
```

PTCLASSALLJARS is set to include each jar file that is to be included in this jar file. Note that we don't include test/test.jar because the test directory contains the test harness and test suites and is not necessary at run time

```
PTCLASSALLJAR = $(PTPACKAGE).jar
```

PTCLASSALLJAR is set to the name of the jar file to be created, which in this case is kernel.jar.

```
install: jclass jars
```

The install rule depends on the jars target. The jars target is defined in ptcommon.mk. The jars target depends on PTCLASSALLJAR, so if PTCLASSALLJAR is set, then make unjars each jar file listed in PTCLASSALLJARS and creates the jar file named by PTCLASSALLJAR

3.7 Subversion Keywords

If you are checking files in to the Ptolemy II Subversion repository, then you must set two svn properties:

- svn:keywords must be set to "Author Date Id Revision"
- svn:eol-style must be set to "native"

To enable keyword substitution, such as Id being changed to

`@version $Id: Foo.java 43472 2006-08-21 23:16:56Z cxh $,`

you need to set up ~/.subversion/config so that each file extension has the appropriate settings. See http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/eclipse.htm#Subversive for details which involve adding $PTII/ptII/doc/coding/svn-config-auto-props.txt to ~/.subversion/config

Why is it necessary to add have a pattern for every file? The answer is that Subversion decides that everything is a binary file and that it is safer to check things in and not modify them. However, there should be a repository wide way to set up config instead of requiring each user to do so.

To test out keyword substitution on new files, follow the steps below. If you have read/write permission to the source.eecs.berkeley.edu SVN repositories, then use the svntest repository. If you don't have write permission on the source.eecs.berkeley.edu repositories, then use your local repository.

```
bash-3.2$ svn co svn+ssh://source.eecs.berkeley.edu/chess/svntest
A    svntest/README.txt
Checked out revision 7.
bash-3.2$ cd svntest


bash-3.2$ echo '$Id$' > testfile.txt

bash-3.2$ svn add testfile.txt
A         testfile.txt
bash-3.2$ svn commit -m "A test for svn keywords: testfile.txt " testfile.txt
Adding         testfile.txt
Transmitting file data .
Committed revision 8.
bash-3.2$ cat testfile.txt
@version $Id: testfile.txt 1.1 2010-03-31 18:18:22Z cxh $
bash-3.2$ svn proplist testfile.txt
Properties on 'testfile.txt':
  svn:keywords
  svn:eol-style
```

Note that testfile.txt had `$Id$` properly substituted. If testfile.txt had only `$Id$` and not something like
`$Id: codingStyle.tex,v 1.22 2010/10/12 07:02:10 cxh Exp $` then keywords were not being substituted and that ~/.subversion/config had a problem.

3.7.1 Checking Keyword Substitution

To check keyword substitution on a file:

```
bash-3.2$ svn proplist README.txt
Properties on 'README.txt':
  svn:keywords
  svn:eol-style
bash-3.2$ svn propget svn:keywords README.txt
Author Date Id Revision
bash-3.2$ svn propget svn:eol-style README.txt
native
```

See $PTIII/doc/coding/releasemgt.htm for information about how to use $PTII/adm/bin/svnpropcheck to check many files.

3.7.2 Fixing Keyword Substitution

To set the keywords in a file called MyClass.java:

```
svn propset svn:keywords "Author Date Id Revision" MyClass.java
svn propset svn:eol-style native MyClass.java
svn commit -m "Fixed svn keywords" MyClass.java
```

3.7.3 Setting svn:ignore

Directories that contain .class files should have svn:ignore set. It is also helpful if svn:ignore is set to ignore the jar file that is created by make install. For example, in the package ptolemy.foo.bar, a makefile called bar.jar will be created by make install. One way to set multiple values in svn:ignore is to create a file /tmp/i and add what is to be ignored:

```
svn propget svn:ignore . > /tmp/i
```

If the directory has svn:ignore set, then /tmp/i will contain the files to be ignored. If the directory does not have svn:ignore set, then /tmp/i will be ignored. Edit /tmp/i and add files to be ignored:

```
*.class
bar.jar
```

Then run

```
svn propset svn:ignore -F /tmp/i .
svn commit -N -m "Added *.class and bar.jar to svn:ignore" .
```

We use the -N option to commit just the directory.

3.8 Checklist for new files

Below is a checklist for common issues with new Ptolemy II files.

3.8.1 Infrastructure

1. Is the java file listed in the makefile? (section 3.6.1)

2. Are the subversion properties svn:keywords and svn:eol-style set? (section 3.7)

3.8.2 File Structure

1. Copyright - Does the file have the copyright? (section 3.2.1)

2. Is the copyright year correct? New files should have the just the current year (section 3.6.1)

3.8.3 Class comment

1. Is the first sentence of the class comment a cogent and complete summary? (section 3.3.1)

2. Are these tags present? (section 3.3.2):

```
@author
@version
@since
@Pt.ProposedRating
@Pt.AcceptedRating
```

3. Are the constructors, methods and variables separated by the appropriate comment lines? (section 3.2)

3.8.4 Constructor, method, field and inner class Javadoc documentation.

1. Within each section, is each Javadoc comment alphabetized? (section 3.2)
2. Is the first sentence a cogent and complete summary in the imperative case? (section 3.3.5)
3. Are all the parameters of each method clearly documented? (section 3.3.7)
4. Are the descriptions of each exception useful?
5. Did you run doccheck and review the results? See http://chess.eecs.berkeley.edu/ptexternal/nightly/ (section 3.3.1)

3.8.5 Overall

1. Did you run spell check on the program and fix errors? (section 3.1)
2. Did you format the file using Eclipse? (section 3.1)
3. Did you fix the imports using Eclipse? (section 3.1)
4. Did you add the new file to the makefile?

3.9 Checklist for creating a new demonstration

Ptolemy II ships with many demonstrations that have a consistent look and feel and a high level of quality. Below is a checklist for creating a new demonstration.

1. Does the name of the demonstration match the directory? Demonstrations should be in directories like demo/Foo/Foo.xml so that the code generators can easily find them. Model names should definitely be well-formed Java identifiers, so Foo-Bar.xml is not correct, use FooBar.xml instead. Model names should not use underscores, the names should follow the coding convention for class names so that if we generate code for a model, then the code follows the coding style.
2. Does the model name begin with a capital letter? Most demonstrations start with a capital letter because if we generate code for them, then the corresponding class should start with a capital letter.
3. Is there a makefile in the directory that contains the demonstration and does the upper level makefile in the demos directory include the jar file produced in the directory? If your model is demo/Foo/Foo.xml, then demo/makefile should include Foo/Foo.jar in PTCLASSALLJARS.
4. Does the demonstration have a title?

5. Does the demonstration have an annotation that describes what the models does and why it is of interest?

6. Does the demonstration have any limitation or requirements for third-party software or hardware clearly marked in red. Usually these limitations use a smaller font.

7. Does the demonstration make sense? Some models require third party software or hardware and might not be the best demonstration.

8. Is there a gray author annotation in the bottom corner? The color values should be 0.4,0.4,0.4,1.0, which appears as grey. One trick is to copy the author annotation from another demonstration.

9. Do all the models use paths that are relative and not specific to your machine? In general, it is best if fileOrURL parameters start with $CLASSPATH so that they can be found no matter how Ptolemy is invoked.

10. Has the demonstration been added to $PTII/doc/coding/completeDemos.htm?

11. Is there a separate test that will exercise a model similar to the demo? Usually these tests go in directories like domains/sdf/test/auto because the domains/*/test/auto models are run after all of Ptolemy has been built.

3.10 Checklist for creating a new directory

To create a directory that contains Java file follow the steps below:

1. Copy a makefile from a similar directory. (section 3.6)

2. Verify that the makefile works by running the command below

```
make sources
make
make install
```

3. Check the style of the makefile: first line, author, copyright date etc. (section 3.6)

4. Add the package to doc/makefile so that Javadoc is created.

5. Create package.html and README.txt by running adm/bin/mkpackagehtml. For example, to create these files in the package ptolemy.foo.bar:

```
$PTII/adm/bin/mkpackagehtml ptolemy.foo.bar
cd $PTII/ptolemy/foo/bar
svn add README.txt package.html
```

6. Don't forget to edit package.html and add descriptive text that describes the package.
7. Set the svn:ignore properties (section 3.7.3)

Bibliography

[1] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA, 2004. http://doi.acm.org/10.1145/984622.984675.

[2] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. VisualSense: Visual modeling for wireless and sensor network systems. Technical Report UCB/ERL M05/25, EECS Department, University of California, July 15 2005. http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/9575.html.

[3] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. HyVisual: A hybrid system visual modeler. Technical Report UCB/ERL M05/ 24, University of California, July 15 2005. http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/9574.html.

[4] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html.

[5] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (volume 2: Ptolemy II software architecture). Technical Report UCB/EECS-2008-29, EECS Department, University of California, Berkeley, Apr 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-29.html.

[6] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (volume 3: Ptolemy II domains). Technical Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, Apr 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-37.html.

[7] E. Cheong, E. A. Lee, and Y. Zhao. Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, page 302, San Diego, CA, 2005. http://doi.acm.org/10.1145/1098918.1098967.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003. http://chess.eecs.berkeley.edu/pubs/488.html.

[9] E. A. Lee. Finite State Machines and Modal Models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

[10] E. A. Lee, X. Liu, and S. Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):1–26, 2009. http://ptolemy.eecs.berkeley.edu/publications/papers/07/classesandInheritance/index.htm.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. http://ptolemy.eecs.berkeley.edu/publications/papers/87/synchdataflow/.

[12] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. http://ptolemy.eecs.berkeley.edu/papers/95/processNets/.

[13] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages pp. 25–53, Zurich, Switzerland, 2005. Springer-Verlag. http://chess.eecs.berkeley.edu/pubs/669.html.

[14] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, 2007. ACM. http://doi.acm.org/10.1145/1289927.1289949.

[15] J. M.-K. Leung, T. Mandl, E. A. Lee, E. Latronico, C. Shelton, S. Tripakis, and B. Lickly. Scalable semantic annotation using lattice-based ontologies. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 393–407. ACM/IEEE, October 2009. (recipient of the MODELS 2009 Distinguished Paper Award) http://chess.eecs.berkeley.edu/pubs/611.html.

[16] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency & Computation: Practice & Experience*, 18(10):1039 – 1065, 2006. http://dx.doi.org/10.1002/cpe.v18:10.

[17] H. J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. Technical Report Series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley, April 1999. http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac/.

[18] Y. Xiong. An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1 2002. http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/3981.html.

Index