2

Using Vergil

Authors: *Edward A. Lee*
Steve Neuendorffer

2.1 Introduction

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

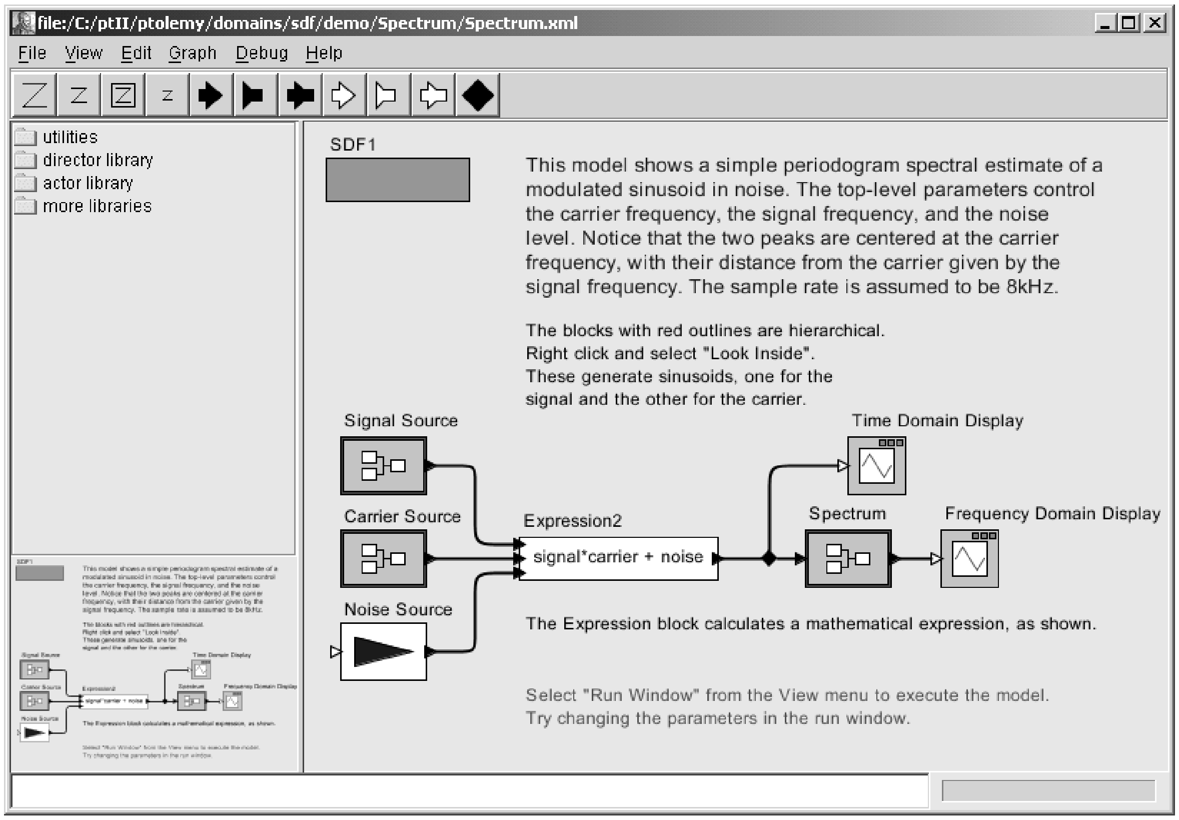
In this chapter, we describe how to graphically construct models using Vergil, a graphical user interface (GUI) for Ptolemy II. figure 2.1 shows a simple Ptolemy II model in Vergil, showing the graph editor, one of several editors available in Vergil. Keep in mind as you read this document that graphical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

2.2 Quick Start

The traditional first programming example is one that prints “Hello World.” Why break tradition?

2.2.1 Starting Vergil

First start Vergil. From the command line, enter “vergil”, or select Vergil or Ptolemy II in the Start menu, or click on a Web Start link on a web page supporting the web edition. You should see an initial welcome window that looks like the one in figure 2.2. Feel free to explore the links in this window. Most useful is probably the “Quick tour” link.



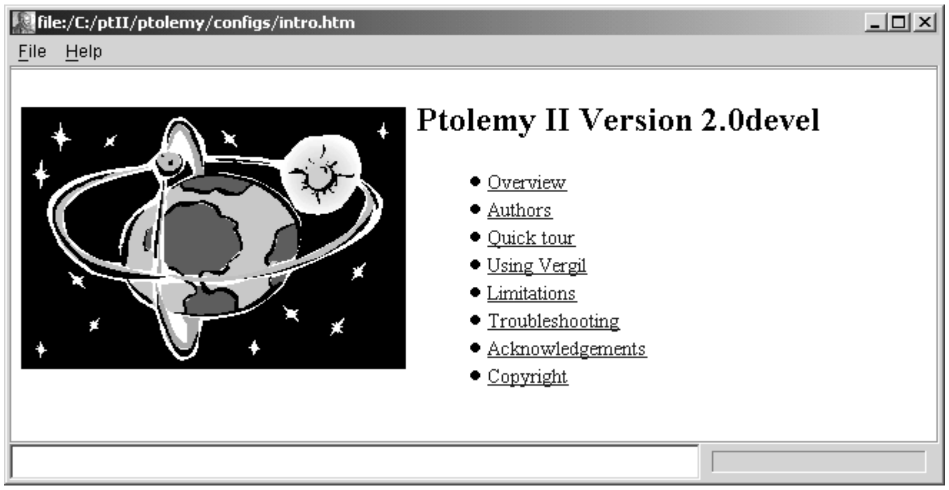
This model shows a simple periodogram spectral estimate of a modulated sinusoid in noise. The top-level parameters control the carrier frequency, the signal frequency, and the noise level. Notice that the two peaks are centered at the carrier frequency, with their distance from the carrier given by the signal frequency. The sample rate is assumed to be 8kHz.

The blocks with red outlines are hierarchical. Right click and select "Look Inside". These generate sinusoids, one for the signal and the other for the carrier.

The Expression block calculates a mathematical expression, as shown.

Select "Run Window" from the View menu to execute the model. Try changing the parameters in the run window.

FIGURE 2.1. Example of a Vergil window.



Ptolemy II Version 2.0devel

- Overview
- Authors
- Quick tour
- Using Vergil
- Limitations
- Troubleshooting
- Acknowledgements
- Copyright

FIGURE 2.2. Initial welcome window.

2.2.2 Creating a New Model

Create a new graph editor from the File->New menu in the welcome window. You should see something like the window shown in figure 2.3. 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 *actor library* in the palette, and go into the *sources* library. Find the *Const* actor and drag an instance over onto the blank page. Then go into the *sinks* library 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 *director library* and drag an *SDFDirector* onto the page. The director gives an execution meaning to the graph, but for now we don't have to be concerned about exactly what that is.

Now you should have something that looks like figure 2.4. The *Const* actor is going to create our string, and the *Display* actor is going to print it out for us. We need to take care of one small detail to make it look like figure 2.4: 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 on the *Const* actor icon and select “Configure”. You should see the dialog box in figure 2.5. 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. Note that because of a bug, you will need to explicitly add “.xml” to the end of the filename so that Vergil will properly process the file the next time you open that file.

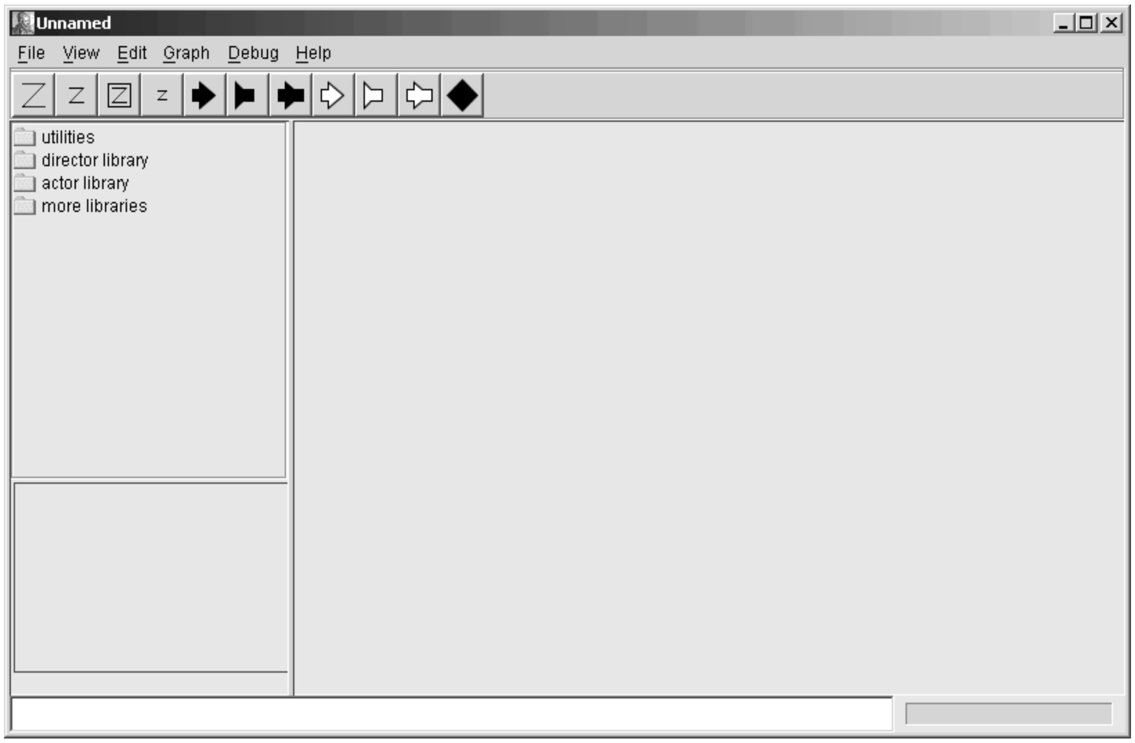


FIGURE 2.3. An empty Vergil Graph Editor.

2.2.3 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 2.6.

2.2.4 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 (although not particularly intelligently). We can now explore how to create and manipulate more complicated connections.

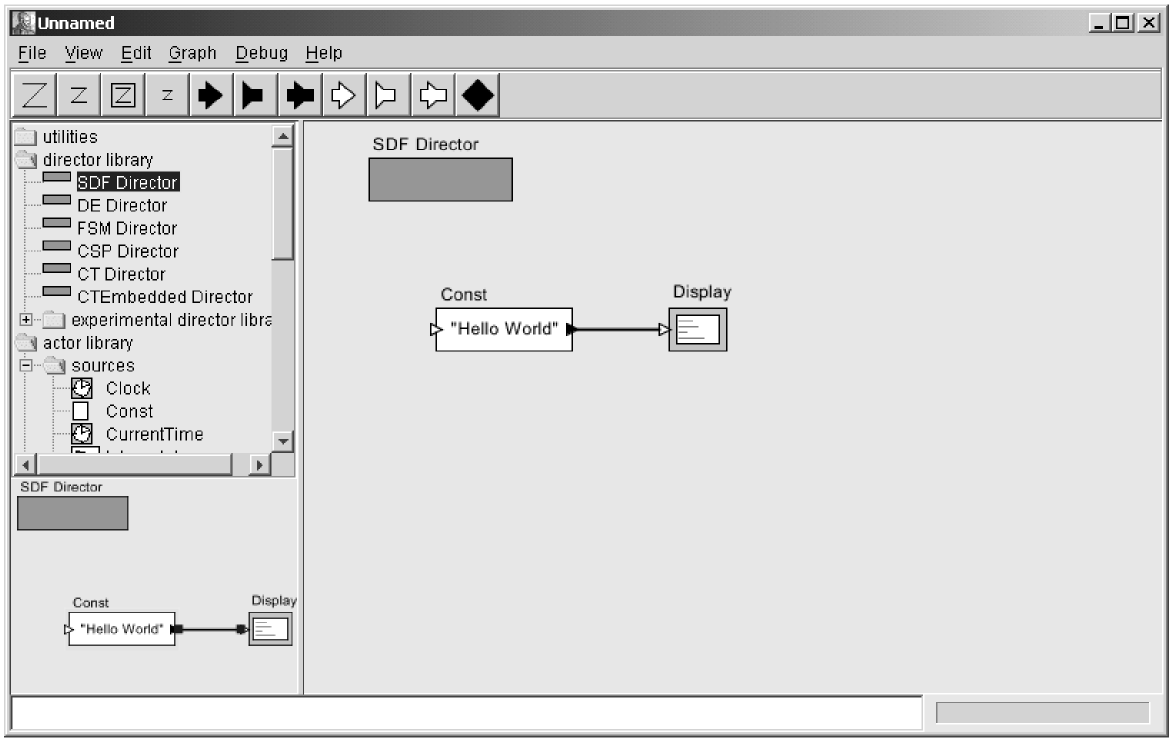
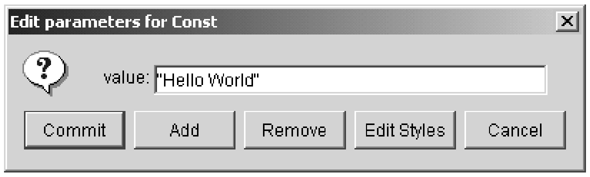


FIGURE 2.4. The Hello World example.



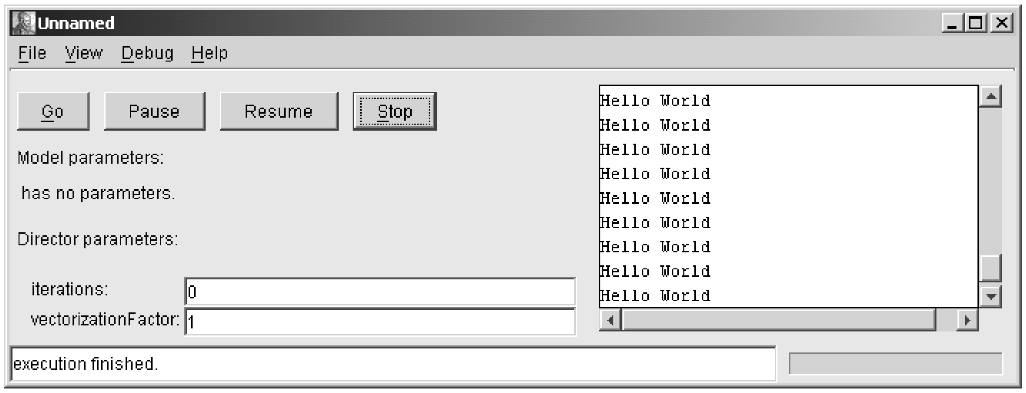
value: "Hello World"

FIGURE 2.5. The Const parameter editor.

First create a model in a new graph editor that includes an *SDFDirector*, a *Ramp* actor (found in the *sources*) library, a *Display* actor, and a *SequencePlotter* actor, found in the *sinks* library, as shown in figure 2.7. Suppose we wish to route the output of the *Ramp* to both the *Display* and the *Sequence-Plotter*. If we simply attempt to make the connections, we get the exception shown in figure 2.7. Don't panic! Exceptions are normal and common. The key line in this exception report is the last one, which says

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

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



Model parameters:

has no parameters.

Director parameters:

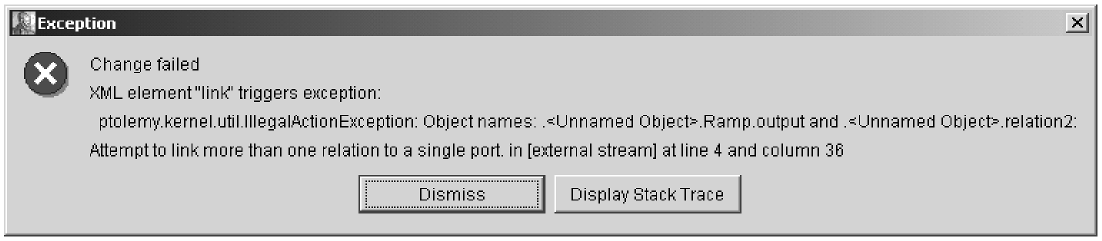
iterations: 0

vectorizationFactor: 1

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

execution finished.

FIGURE 2.6. Execution of the Hello World example.



Change failed

XML element "link" triggers exception:

ptolemy.kernel.util.IllegalActionException: Object names: .<Unnamed Object>.Ramp.output and .<Unnamed Object>.relation2:

Attempt to link more than one relation to a single port. in [external stream] at line 4 and column 36

FIGURE 2.7. Exception that occurs if you attempt to simply wire the output of the *Ramp* in figure 2.8 to the inputs of the other two actors.

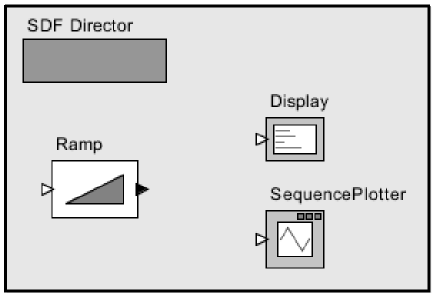


FIGURE 2.8. Three unconnected actors in a model.

```
    .<Unnamed Object>.Ramp.output and .<Unnamed Object>.relation3
```

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

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

So how do we get the output of the *Ramp* to the other two actors? We need an explicit *relation* in the diagram. A relation is represented in the diagram by a black diamond, as shown in figure 2.9. It can be created by either control-clicking on the background or by clicking on the button in the toolbar with the black diamond on it.

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

In the model shown in figure 2.9, the relation is used to broadcast the output from a single port to a number of places. The single port still has only one connection to it, a connection to a relation. Relations can also be used to control the routing of wires in the diagram. However, as of the 2.0 release of Ptolemy II, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

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

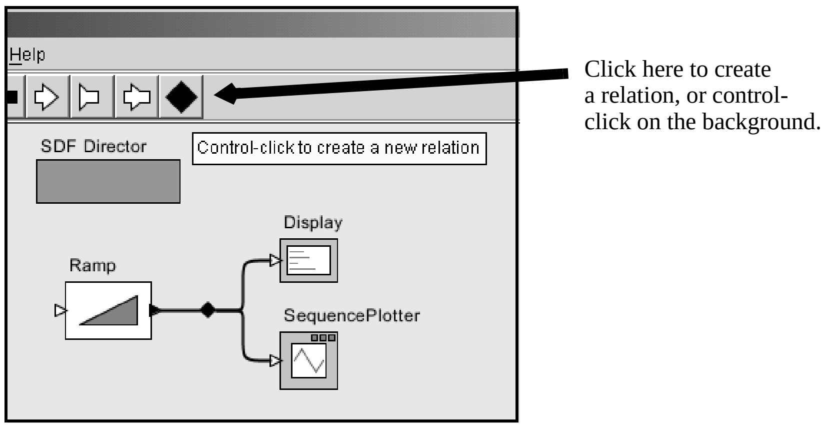


FIGURE 2.9. A relation can be used to broadcast an output from a single port.

2.3 Tokens and Data Types

In the example of figure 2.4, 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, but for now, try giving the value 1 (the integer with value one), or 1.0 (the floating-point number with value one), or {1.0} (An array containing a one), or {value=1, name="one"} (A record with two elements: an integer named “value” and a string named “name”), or even [1,0;0,1] (the two-by-two identity matrix). These are all expressions.

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

To explore data types a bit further, try creating the model in figure 2.10. The *Ramp* actor is listed under *sources* and the *AddSubtract* actor is listed under *math*. Set the *value* parameter of the constant to be 0 and the *iterations* parameter of the director to 5. Running the model should result in 5 numbers between 0 and 4, as shown in the figure. These are the values produced by the *Ramp*, which are having the value of the *Const* actor subtracted from them. Experiment with changing the value of the *Const* actor and see how it changes the 5 numbers at the output.

Now for the real test: change the value of the *Const* actor back to "Hello World". When you execute the model, you should see an exception window, as shown in figure 2.11. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a string value from an integer value, which doesn’t make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

Exceptions can be a very useful debugging tool, particularly if you are developing your own components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of figure 2.11. You should see the stack trace shown in figure 2.12. This window displays the execution sequence that resulted in the exception. For example, the line

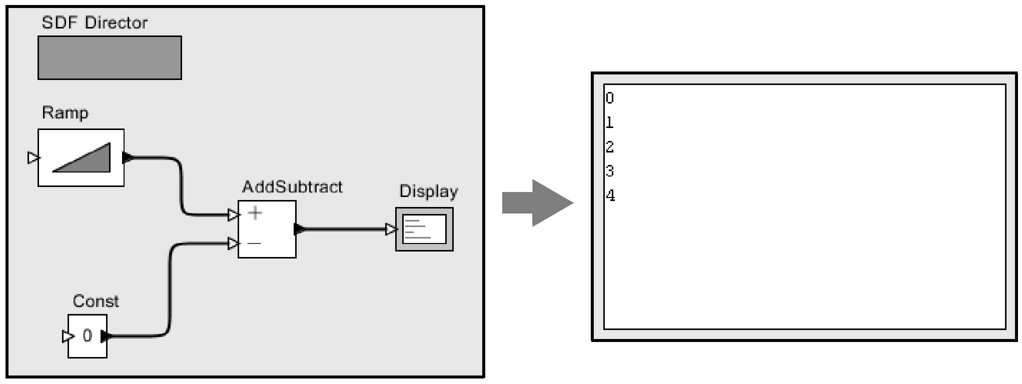
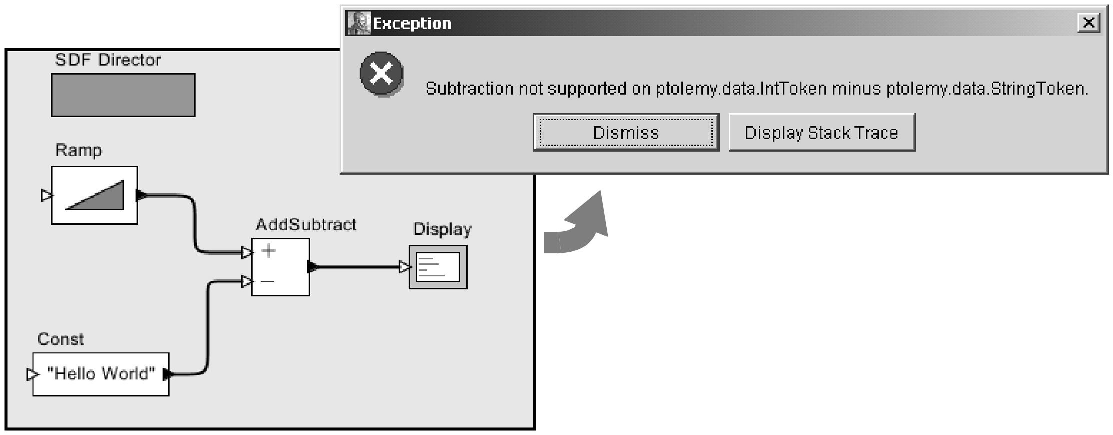


FIGURE 2.10. Another example, used to explore data types in Ptolemy II.

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

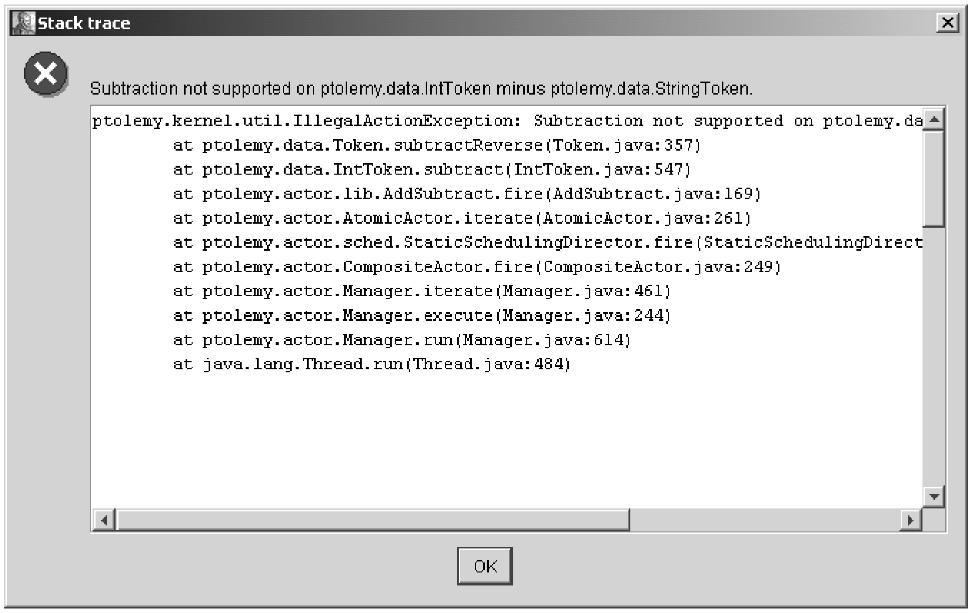
indicates that the exception occurred within the subtract() method of the class ptolemy.data.IntToken, at line 547 of the source file IntToken.java. Since Ptolemy II is distributed with source code (except in the Web Edition), this can be very useful information. For type errors, you probably do not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

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



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

FIGURE 2.11. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

```
ptolemy.kernel.util.IllegalActionException: Subtraction not supported on ptolemy.da
        at ptolemy.data.Token.subtractReverse(Token.java:357)
        at ptolemy.data.IntToken.subtract(IntToken.java:547)
        at ptolemy.actor.lib.AddSubtract.fire(AddSubtract.java:169)
        at ptolemy.actor.AtomicActor.iterate(AtomicActor.java:261)
        at ptolemy.actor.sched.StaticSchedulingDirector.fire(StaticSchedulingDirect
        at ptolemy.actor.CompositeActor.fire(CompositeActor.java:249)
        at ptolemy.actor.Manager.iterate(Manager.java:461)
        at ptolemy.actor.Manager.execute(Manager.java:244)
        at ptolemy.actor.Manager.run(Manager.java:614)
        at java.lang.Thread.run(Thread.java:484)
```

FIGURE 2.12. Stack trace for the exception shown in figure 2.11.

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

Let's try a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in figure 2.13. You can do this by selecting the connection and deleting it (using the delete key), then adding a new connection, or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like "0HelloWorld", as shown in the figure.

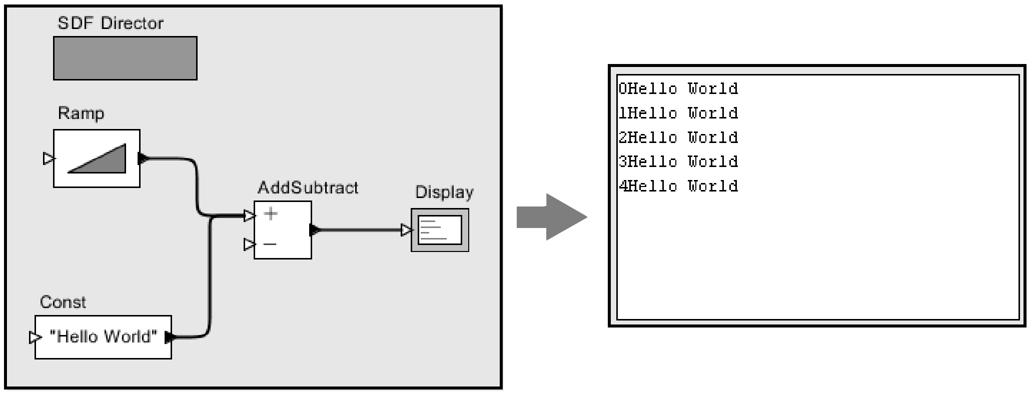
There are two interesting things going on here. The first is that, as in Java, strings are added by concatenating them. The second is that the integers from the *Ramp* are converted to strings and concatenated with the string "Hello World". All the connections to a multiport must have the same type.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

Ptolemy II automatically converts the integers to strings when integers are provided to an actor that requires strings. But in this case, why does the *AddSubtract* actor require strings? Because it would not work to require integers; the string "Hello World" would have to be converted to an integer. As a rough guideline, Ptolemy II will perform automatic type conversions when there is no loss of information. An integer can be converted to a string, but not vice versa. An integer can be converted to a double, but not vice versa. An integer can be converted to a long, but not vice versa. The details are explained in the Data chapter, but many users will not need to understand the full sophistication of the system. You should find that most of the time it will just do what you expect.

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

2.4 Hierarchy

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



```
0Hello World
1Hello World
2Hello World
3Hello World
4Hello World
```

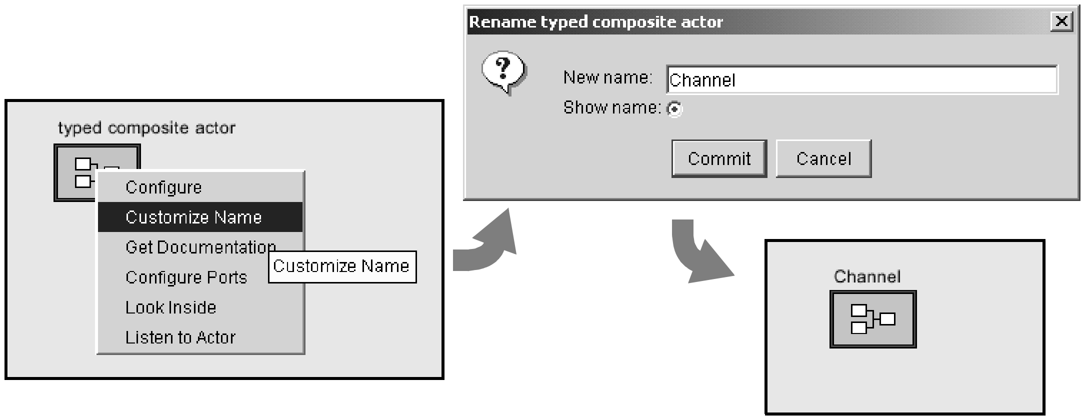
FIGURE 2.13. Addition of a string to an integer.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select “Customize Name”, and give the composite a better name, like “Channel”, as shown in figure 2.14. Then, using the context menu again, select “Look Inside” on the actor. You should get a blank graph editor, as shown in figure 2.15. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

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



New name: Channel
Show name: ●

FIGURE 2.14. Changing the name of an actor.

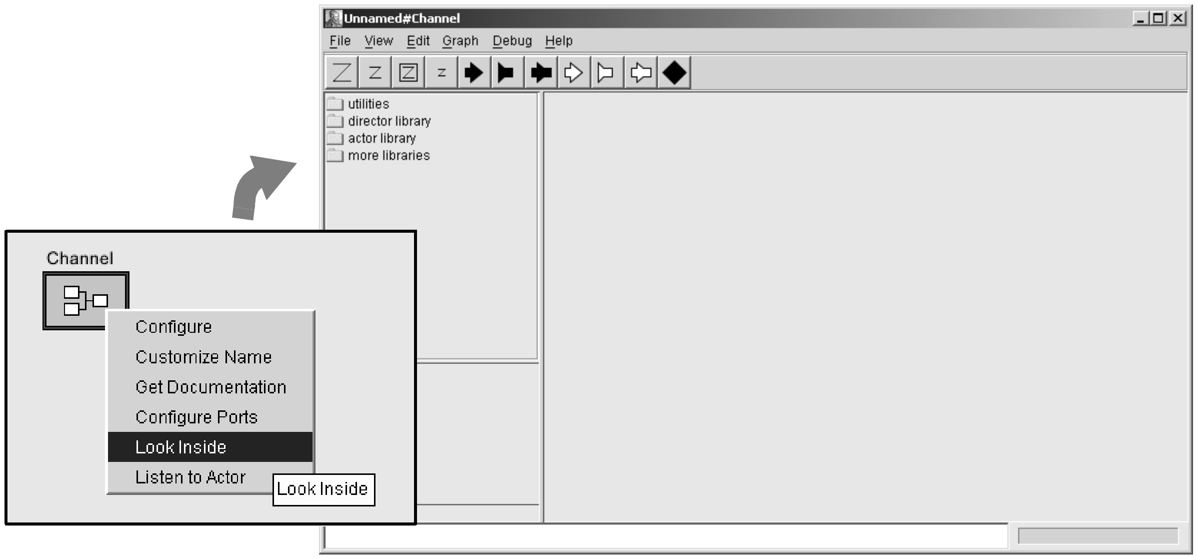


FIGURE 2.15. Looking inside a composite actor.

The buttons are summarized in figure 2.16. Create an input port and an output port and rename them *input* and *output* right clicking on the ports and selecting “Customize Name”. Note that, as shown in figure 2.17, you can also right click on the background of the composite actor and select *Configure Ports* to change whether a port is an input, an output, or a multiport. The resulting dialog also allows you to set the type of the port, although much of the time you will not need to do this, since the type inference mechanism in Ptolemy II will figure it out from the connections.

Then using these ports, create the diagram shown in figure 2.18[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close

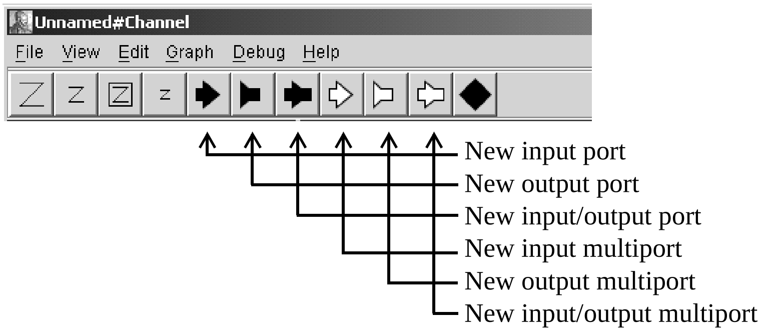
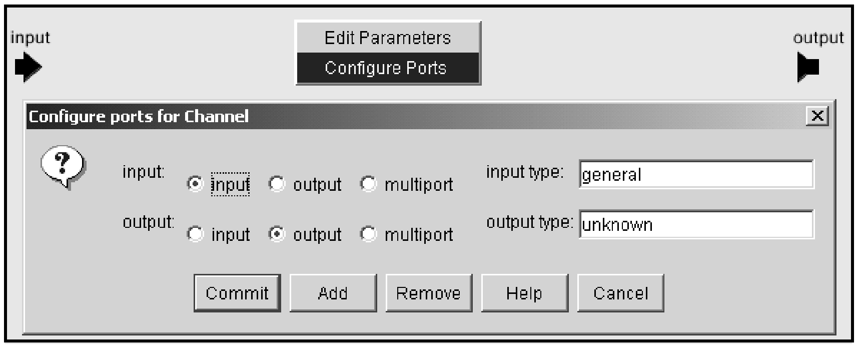


FIGURE 2.16. Summary of toolbar buttons for creating new ports.



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

FIGURE 2.17. Right clicking on the background brings up a dialog that can be used to configure ports.

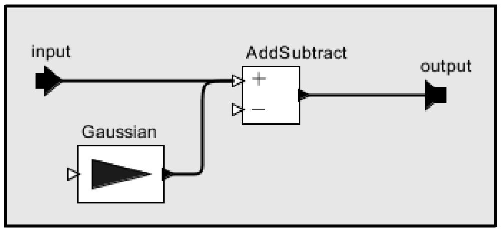


FIGURE 2.18. A simple channel model defined as a composite actor.

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

this editor and return to the previous one, you should be able to easily create the model shown in figure 2.19. The *Sinewave* actor is listed under *sources*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline (try looking inside). If you execute this model (you will probably want to set the iterations to something reasonable, like 100), you should see something like figure 2.20.

2.4.3 Setting the Types of Ports

In the above example, we never needed to define the types of any ports. The types were inferred from the connections. Indeed, this is usually the case in Ptolemy II, but occasionally, you will need to set the types of the ports. Notice in figure 2.17 that there is a position in the dialog box that configures ports for specifying the type. The natural question you might have is "how do I specify the type?"

The answer to this question seems too obvious: you specify a *prototype*, an expression that has the type that you want to specify. Thus, to specify that a port has type *boolean*, you could enter into the dialog of figure 2.17 the value "true" or "false". This, however, might not be too clear. For this reason, Ptolemy II has a built-in constant named "boolean" that happens to have value "true". Thus, if you specify that the type is "*boolean*", then, well, you have specified that the type is boolean.

There are several other constants that similarly can be used to specify types. They are *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. So how would you specify that the type of a port is a double matrix? Easy:

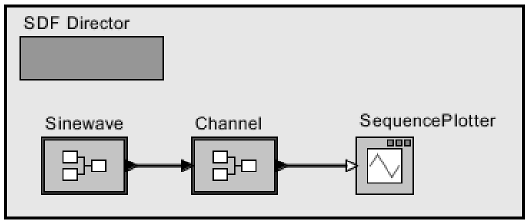
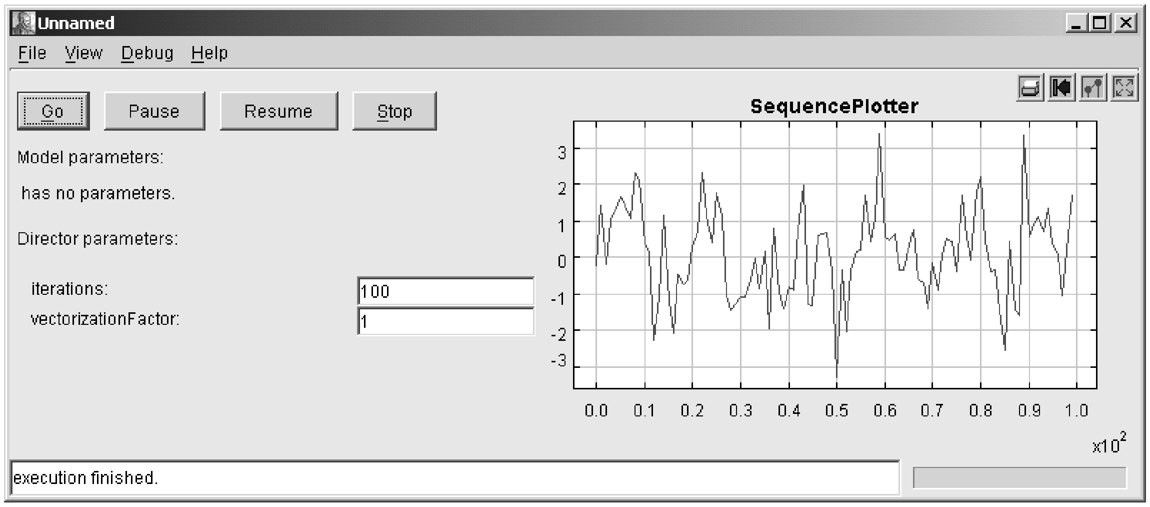


FIGURE 2.19. A simple signal processing example that adds noise to a sinusoidal signal.



Model parameters:
has no parameters.
Director parameters:
iterations: 100
vectorizationFactor: 1

execution finished.

FIGURE 2.20. The output of the simple signal processing model in figure 2.19.

```
[double]
```

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

```
{complex}
```

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

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

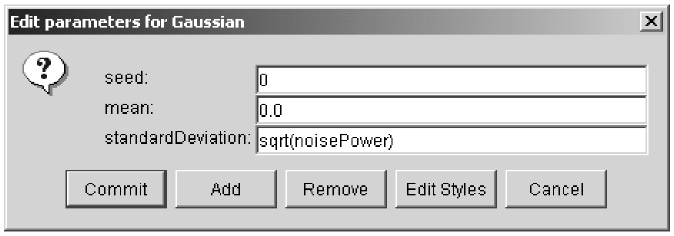
2.5 Annotations and Parameterization

In this section, we will enhance the model in figure 2.19 in a number of ways.

First, notice from figure 2.20 that the noise overwhelms the sinusoid, making it barely visible. A useful channel model would have a parameter that sets the level of the noise. Look inside the channel model, and add a parameter by dragging one in from the *utilities* library, as shown in figure 2.21. Right click on the parameter to change its name to "noisePower". (In order to be able to use this parameter in expressions, the name cannot have any spaces in it.) Also, right click or double click on the parameter to change its default value to 0.1.

Now we can use this parameter. First, let's use it to set the amount of noise. The *Gaussian* actor has a parameter called *standardDeviation*. In this case, the power of the noise is equal to the variance of the Gaussian, not the standard deviation. If you recall from basic statistics, the standard deviation is equal to the square root of the variance. Change the *standardDeviation* parameter of the *Gaussian* actor so its value is "sqrt(noisePower)", as shown in figure 2.22. This is an expression that references the *noisePower* parameter. We will explain the expression language in the next section. But first, let check our improved model. Return to the top-level model, and edit the parameters of the *Channel* actor (by either double clicking or right clicking and selecting "Configure"). Change the noise power from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in figure 2.23.

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



seed: 0
mean: 0.0
standardDeviation: sqrt(noisePower)

FIGURE 2.22. The standard deviation of the *Gaussian* actor is set to the square root of the noise power.

dialog can be obtained by either double clicking on the *Channel* icon, or by right clicking and selecting "Configure", or by right clicking on the background inside the composite and selecting "Edit Parameters".

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

2.6 Expressions

The values of parameters can be expressions. We have already seen a simple one,

```
sqrt(noisePower)
```

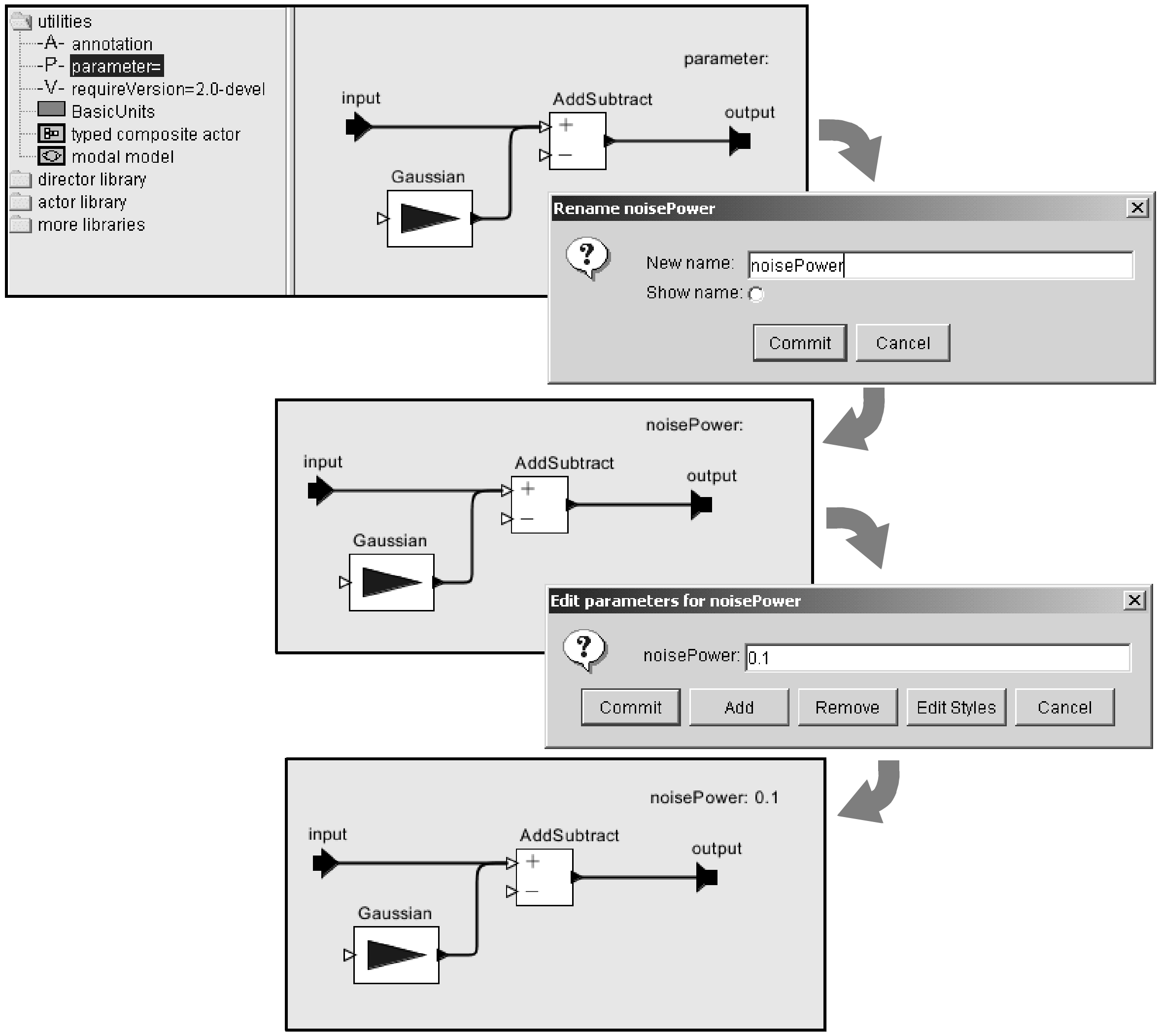


FIGURE 2.21. Adding a parameter to the channel model.

Expressions can be used to specify the value of a parameter, or to specify the calculation performed by the *Expression* actor when it fires.

2.6.1 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output an no inputs, as shown in figure 2.24(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).

2.6.2 Constants and Literals

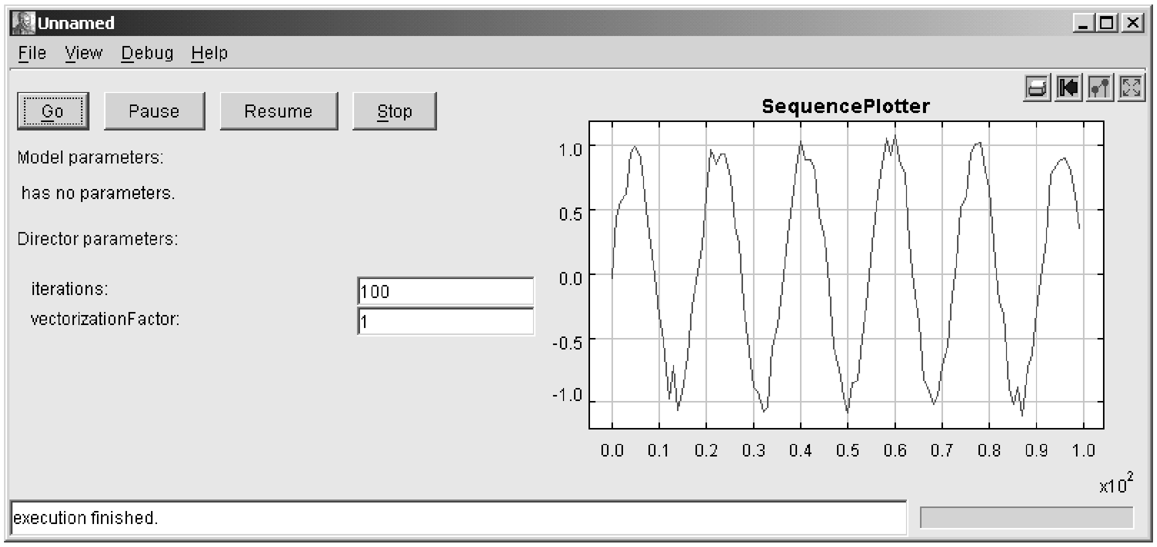
Expressions can include references to variables and some constants. By default, the constants supported are PI, pi, E, e, true, false, i, and j. for example,

```
PI/2.0
```

is a valid expression, and can be given as the value of a parameter that can accept doubles. The constants i and j are complex numbers with value equal to 0.0 + 1.0i. In addition, literal string constants are supported. Anything between quotes, “...”, is interpreted as a string constant. Numerical values without decimal points, such as “10” or “-3” are integers. Numerical values with decimal points, such as “10.0” or “3.14159” are doubles. Integers followed by the character “l” (el) or “L” are long integers.

2.6.3 Operators

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



Model parameters:

has no parameters.

Director parameters:

iterations: 100

vectorizationFactor: 1

execution finished.

FIGURE 2.23. The output of the simple signal processing model in figure 2.19.

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

```
    boolean ? value1 : value2
```

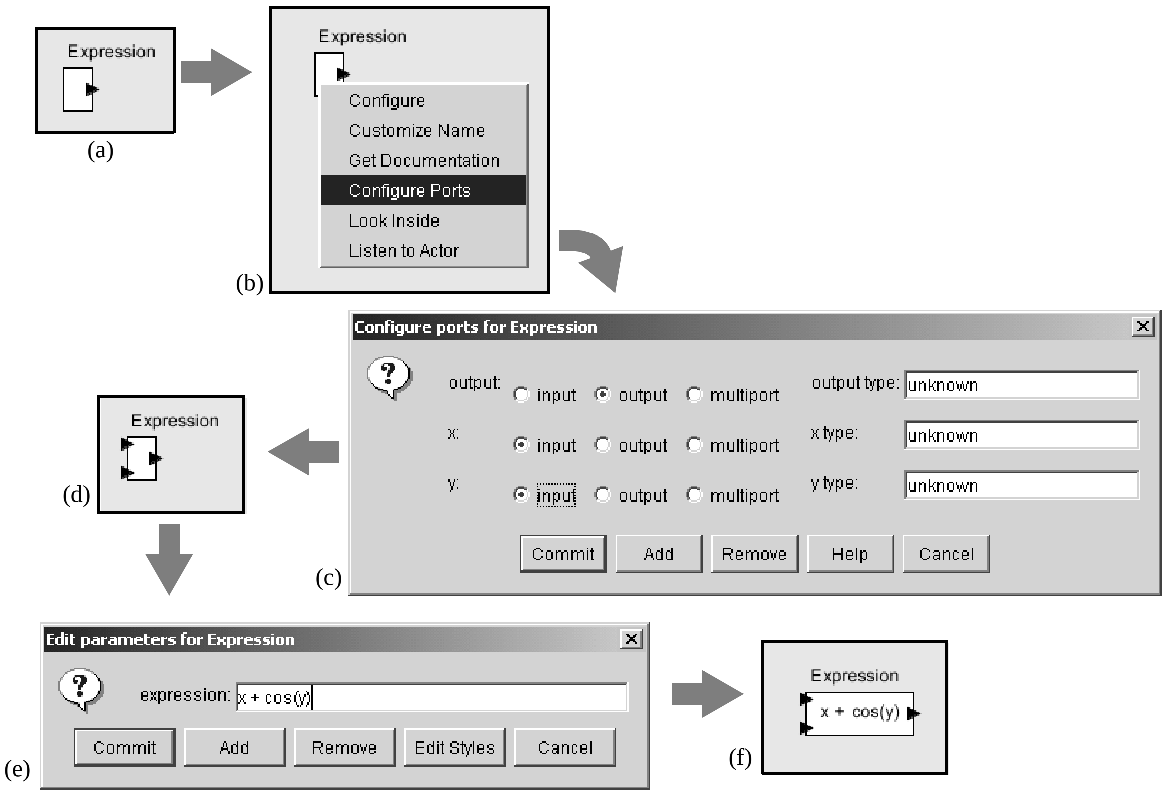
If the boolean is true, `value1` is returned, else `value2` is returned. The logical boolean operators are &&, ||, !, & and |. They operate on booleans and return booleans. Note that the difference between logical && and logical & is that & evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical || and |. This approach is borrowed from Java.

2.6.4 Variables

Expressions can contain references by name to parameters within the ***scope*** of the expression. Consider a parameter *P* in actor *X* which is in turn contained by composite actor *Y*. The scope of an expression for *P* includes all the parameters contained by *X* and *Y*, plus those of the container of *Y*, its container, etc. That is, the scope includes any parameters defined above in the hierarchy. You can add parameters to actors (composite or not) by right clicking on the actor, selecting "Configure" and then clicking on "Add", or by dragging in a parameter from the *utilities* library.

2.6.5 Arrays

Arrays are specified with curly brackets. E.g., "{1, 2, 3}" is an array of integers, while "`{"x", "y", "z"}`" is an array of strings. An array is an ordered list of tokens of any type, with the only con-



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

expression: x + cos(y)

FIGURE 2.24. Illustration of the *Expression* actor.

straint being that the elements all have the same type. Thus, for example, “{1, 2.3}” is illegal because the first element is an integer and the second is a double. The elements of the array can be given by expressions, as in the example “{2*pi, 3*pi}.” Arrays can be nested; for example, “{{1, 2}, {3, 4, 5}}” is an array of arrays of integers.

2.6.6 Matrices

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

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

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

2.6.7 Records

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

2.6.8 Functions

The expression language includes an extensible set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the java.lang.Math class and the ptolemy.data.expr.UtilityFunctions class. This can easily be extended by registering another class that includes static methods. The functions currently available are shown in figures 2.25 and 2.26, with the argument types and return types[1].

One slightly subtle function is the random() function shown in figure 2.25. It takes no arguments, and hence is written “`random()`”. It returns a random number. However, this function is evaluated only when the expression within which it appears is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. The random() function is not called again. Thus,

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

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

FIGURE 2.25. Functions available to the expression language from the java.lang.Math class.

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.

2.6.9 Methods

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

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

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

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

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

FIGURE 2.26. Functions available to the expression language from the ptolemy.data.expr.UtilityFunctions class. This class is still at a preliminary stage, and the function it provides will grow over time.

which returns the integer 3.

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

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

which returns 3, and

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

which returns 2, and

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

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

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

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

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

which returns 1.

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

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

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

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

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

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

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

2.6.10 Supported Types

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

2.6.11 Comments

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

2.6.12 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, integerBits, fractionBits)
```

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

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

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

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

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

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

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

- To create a single FixPoint Token using the expression language:
  ```
  fix(5.34, 10, 4)
  ```
  This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.
- To create a Matrix with FixPoint values using the expression language:
  ```
  fix([ -.040609, -.001628, .17853 ], 10,  2)
  ```

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

- 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.

2.7 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 section 2.27, that help. These buttons permit zooming in and out. The “Zoom reset” button restores the zoom factor to the “normal” one, and the “Zoom fit” calculates the zoom factor so that the entire model is visible in the editor window.

In addition, it is possible to pan over a model. Consider the window shown in figure 2.28. Here, we have zoomed in so that icons are larger than the default. The *pan window* at the lower left shows the entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the “Zoom fit” button in the toolbar results in the editor area showing the entire model, just as the pan window does.

2.8 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

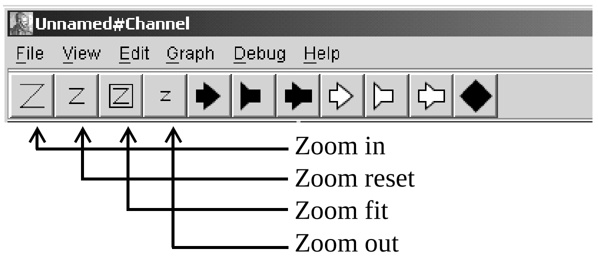


FIGURE 2.27. Summary of toolbar buttons for zooming and fitting.

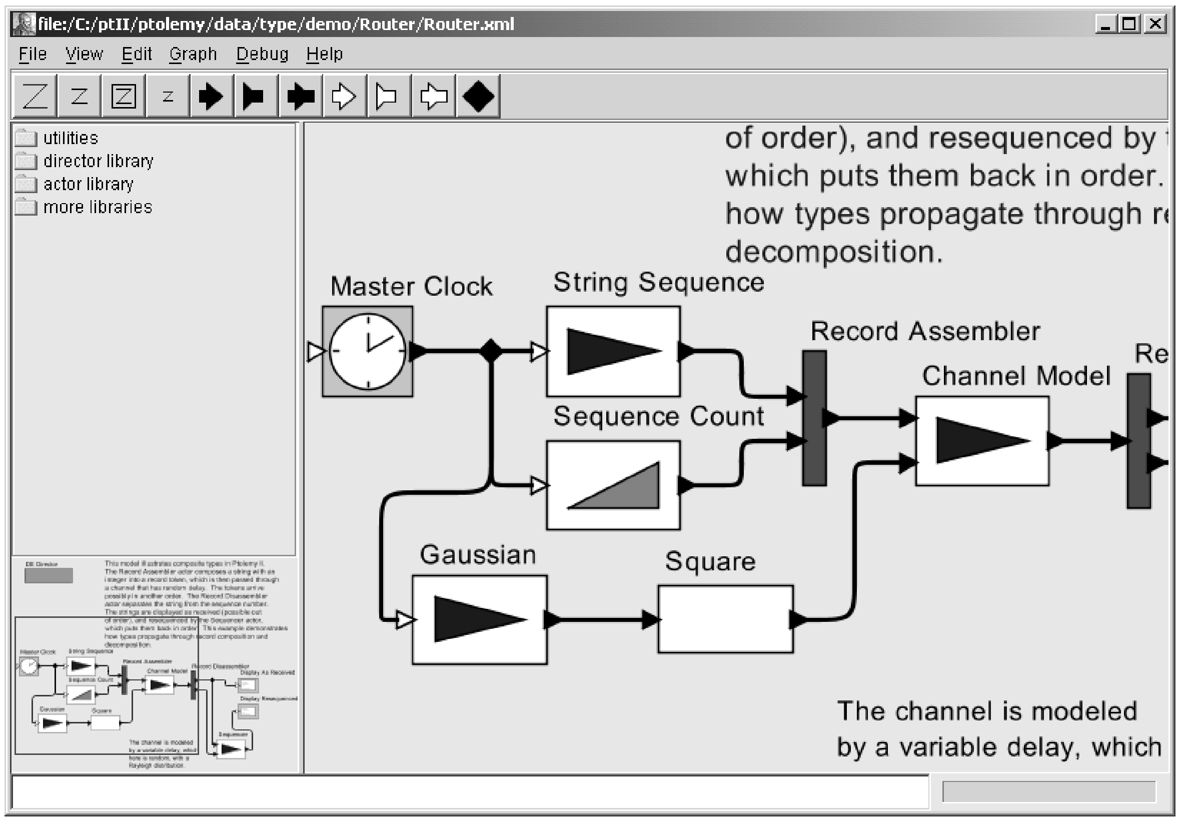
meaning (semantics) to a diagram. It specifies what a connection means, and how the diagram should be executed. In Ptolemy II terminology, the director realizes a *domain*. Thus, when you construct a model with an SDF director, you have constructed a model "in the SDF domain."

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

But there are several other models of computation available in Ptolemy II. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are using. Here, we give a very brief introduction to some of the domains. But we begin first by explaining some of the subtleties in SDF.

2.8.1 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 number of output values that are created by each actor is determined by the number of iterations.



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

The channel is modeled
by a variable delay, which

FIGURE 2.28. 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.

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

One such actor is a spectral estimation actor. Figure 2.29 shows a system that computes the spectrum of the same noisy sine wave that we constructed in figure 2.19. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 2.30 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 for details.

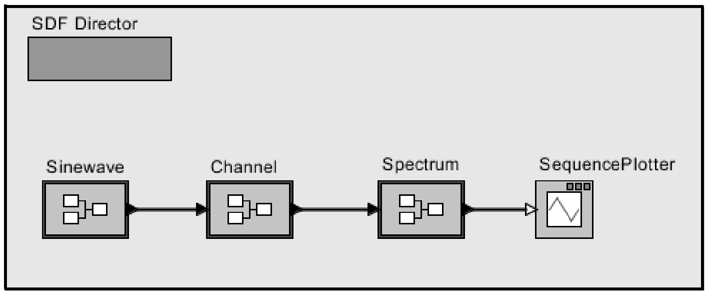
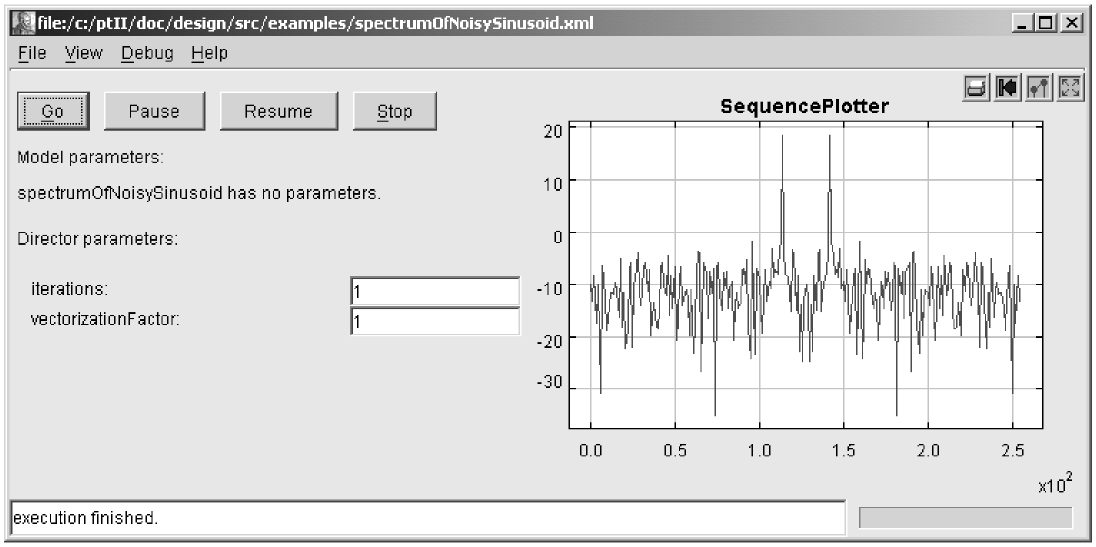


FIGURE 2.29. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave*, *Channel*, and *SequencePlotter*, and one firing of *Spectrum*.



Model parameters:

spectrumOfNoisySinusoid has no parameters.

Director parameters:

iterations: 1

vectorizationFactor: 1

execution finished.

FIGURE 2.30. A single iteration of the SDF model in figure 2.29 produces 256 output tokens.

A second subtlety with SDF models is that if there is a feedback loop, as in figure 2.31, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *flow control* library). Without this actor, the loop will deadlock. The *SampleDelay* actor produces initial tokens on its output, before the model begins firing. The initial tokens produced are given by a the *initialOutputs* parameter, which specifies an array of tokens. These initial tokens enable downstream actors and break the circular dependencies that would result otherwise from a feedback loop.

A final issue to consider with the SDF domain is time. Notice that in all the examples above we have suggested using the *SequencePlotter* actor, not the *TimedPlotter* actor, which is in the same *sinks* library. This is because the SDF domain does not include in its semantics a notion of time. Time does not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. An experimental domain, discrete time (DT), adds time to SDF. This turns out to be somewhat subtle for multirate systems, and this domain has not yet (as of version 2.0) reached the maturity of SDF.

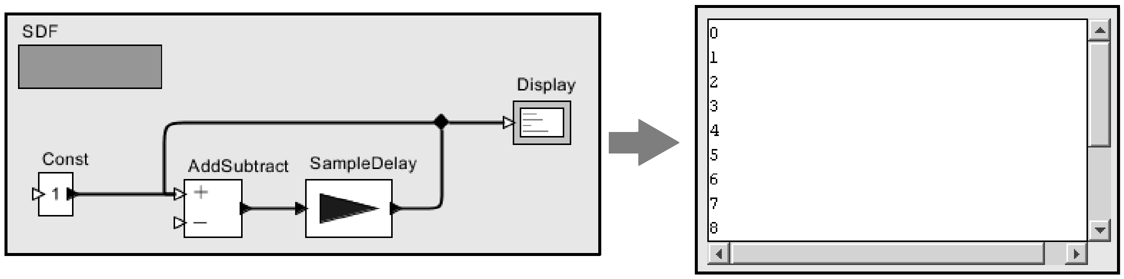
The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.8.2 Discrete-Event Systems

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

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

Consider the DE model shown in figure 2.32. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead,



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

FIGURE 2.31. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the time stamp of the input). The *WallClockTime* actor an event with the same time stamp as the input, but whose value is the current real time, in seconds since initialization of the model.

The plot in figure 2.32 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

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

Consider the model shown in figure 2.33, 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.

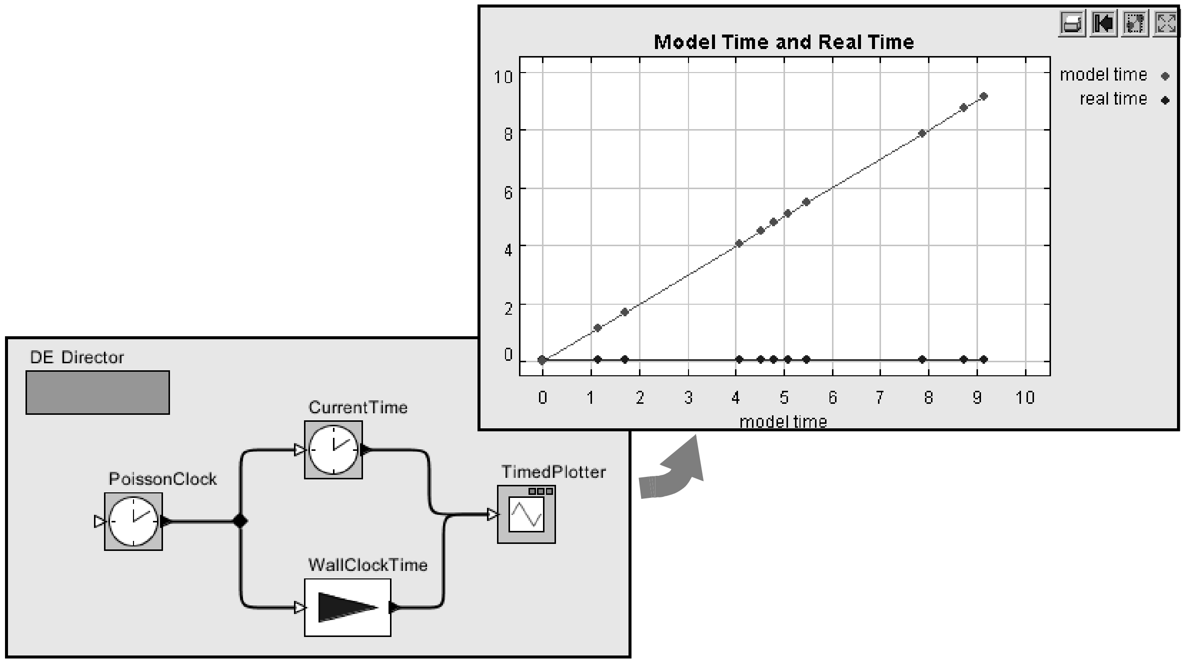


FIGURE 2.32. Model time vs. real time (wall clock time).

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

How does the director deliver on the intuition that the *Previous* actor should be fired first? Before executing the model, the DE director constructs a *topological sort* of the model. A topological sort is simply a list of the actors in data-precedence order. For the model in figure 2.33, there is only one allowable topological sort:

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

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

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

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *domain specific* library under *discrete-event* (this library is shown on the left in figure 2.34). Consider for example the model shown in figure 2.34. 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

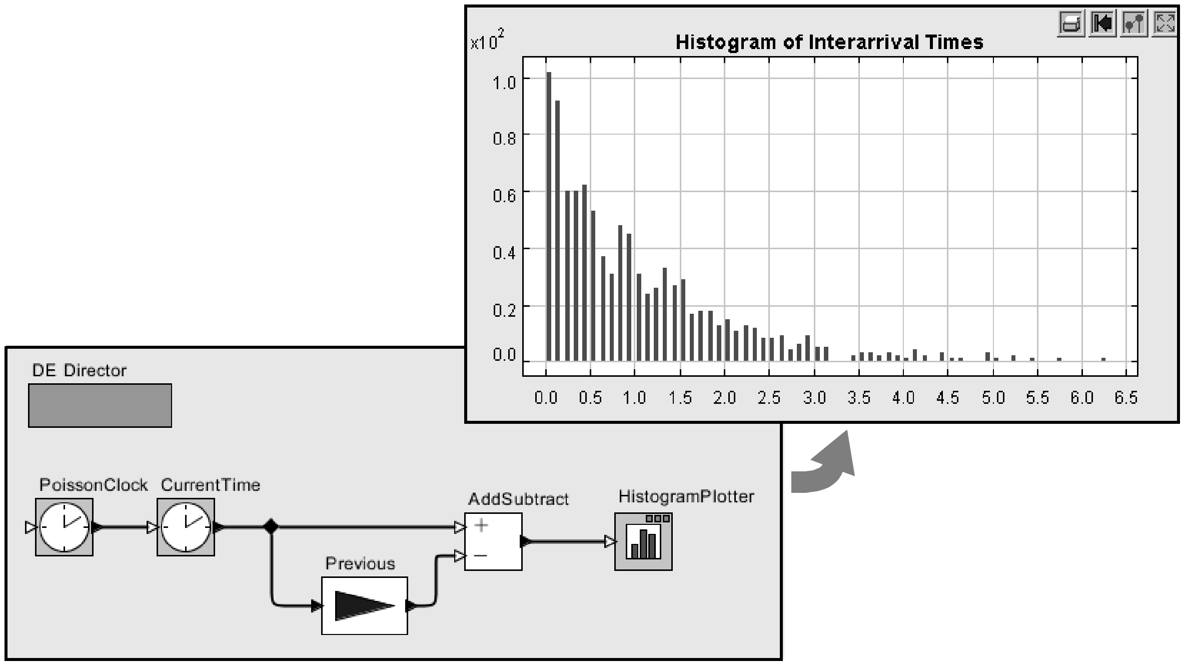


FIGURE 2.33. Histogram of interarrival times, illustrating handling of simultaneous events.

composite actor that the DE director cannot recognize as a delay. The *TimedDelay* actor with a delay of 0.0 can be thought of as a way to let the director know that there is a time delay in the preceding actor, without specifying the amount of the time delay.

2.8.3 Continuous-Time Systems

The continuous-time domain (CT) is another relatively mature domain with semantics considerably different from either DE or SDF. In CT, the signals sent along connections between actors are continuous-time signals, or in some cases, discrete-events that behave similarly to those in DE, with some restrictions. The typical application of the CT domain is to model differential equations. Consider the following set of three differential equations:

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= (\lambda - x_3)x_1 - x_2 \\ \dot{x}_3 &= x_1 \cdot x_2 - b \cdot x_3 \end{aligned} \tag{1}$$

There are three variables, x_1, x_2, and x_3, and three constants, σ, λ, and b. The variables vary contin-

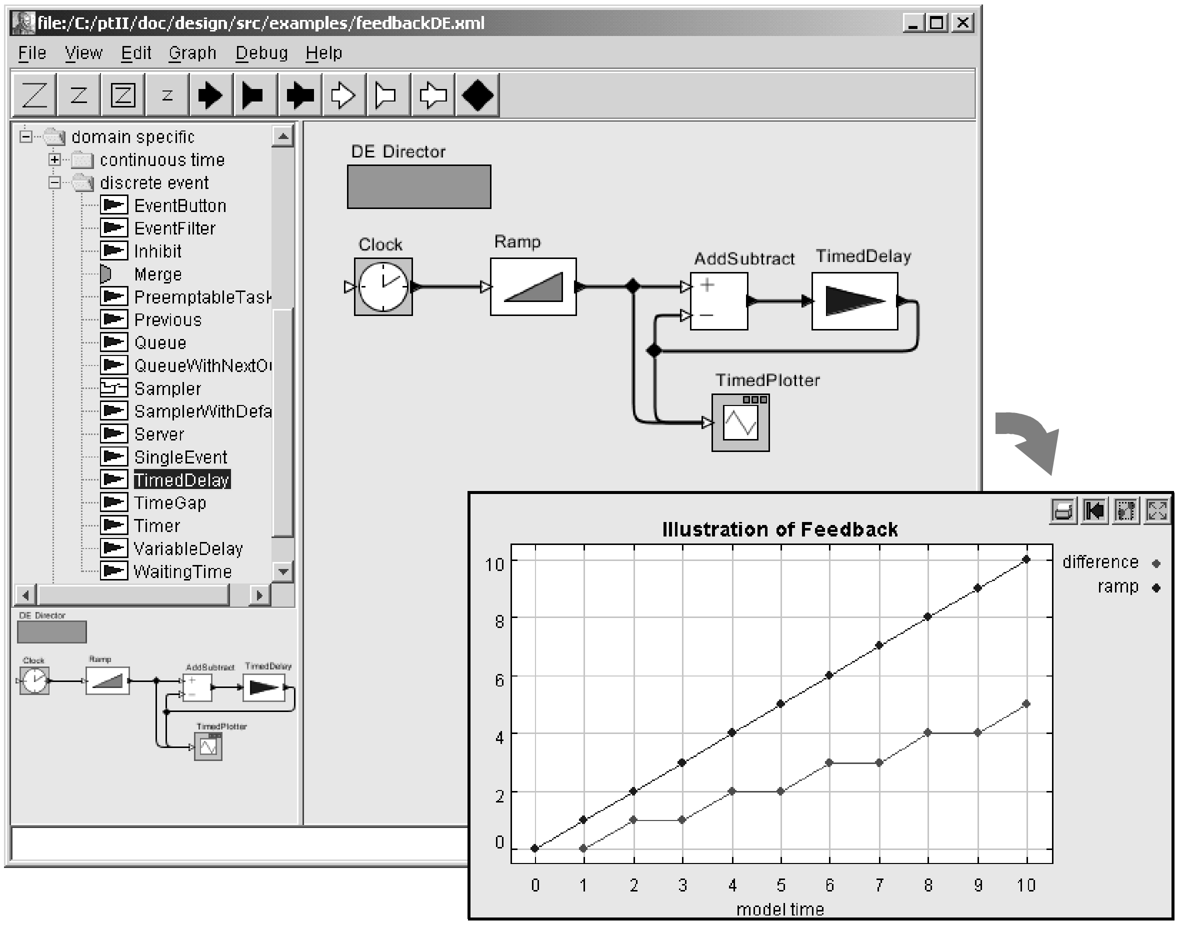


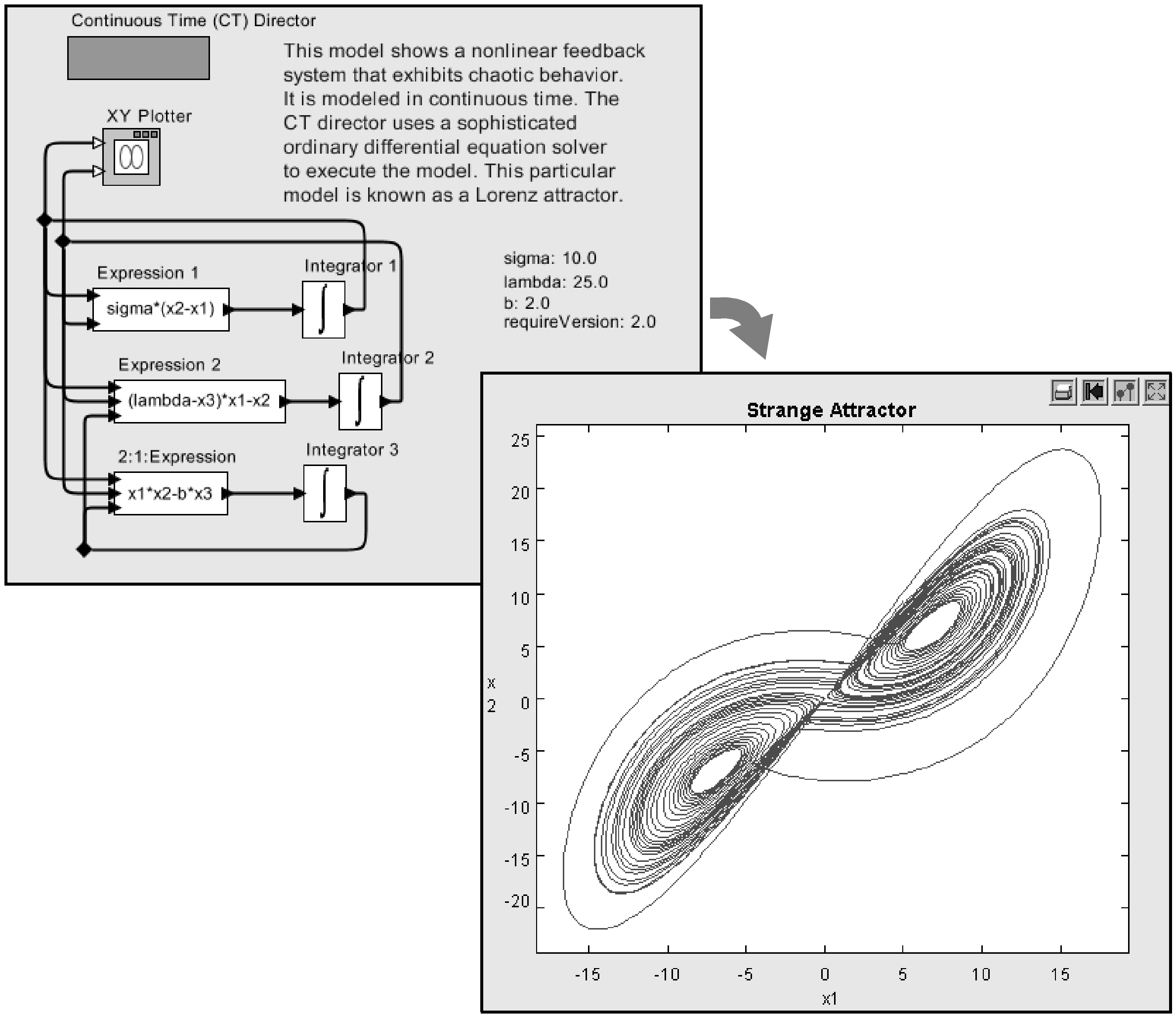
FIGURE 2.34. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

uously with time, and hence represent continuous-time signals. The notation $\dot{x}_1$ refers to the time derivative of x_1.

A model of these differential equations in the CT domain is shown in figure 2.35. As is customary in modeling differential equations, we use *integrators* instead of differentiators. Integrators are much more numerically robust. They are arranged in a feedback loop, so that the input to an integrator is simply the derivative of the output. Thus, the output of *Integrator 1* is x_1, and its input is $\dot{x}_1$. A feedback loop is used to specify the value of $\dot{x}_1$ in terms of x_1, x_2, and x_3.

This set of differential equations describe a famous chaotic system called a Lorenz attractor. It is a special case of a family of nonlinear feedback systems that exhibit *strange attractor* behavior. The "attractors" are the two nodes in the plot in figure 2.35 that the trace seems to be alternately orbiting.

The model in figure 2.35 illustrates several points. First, in CT, every feedback loop must contain an integrator. Second, the *XYPlotter* actor is used to plot x_2 vs. x_1. Third, three instances of the *Expres-*



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

sigma: 10.0
lambda: 25.0
b: 2.0
requireVersion: 2.0

FIGURE 2.35. Realization of the Lorenz attractor model in the CT domain.

sion actor are used instead of complex block diagrams to specify arithmetic expressions. Use of the *Expression* actor is explained in section 2.6.1.

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

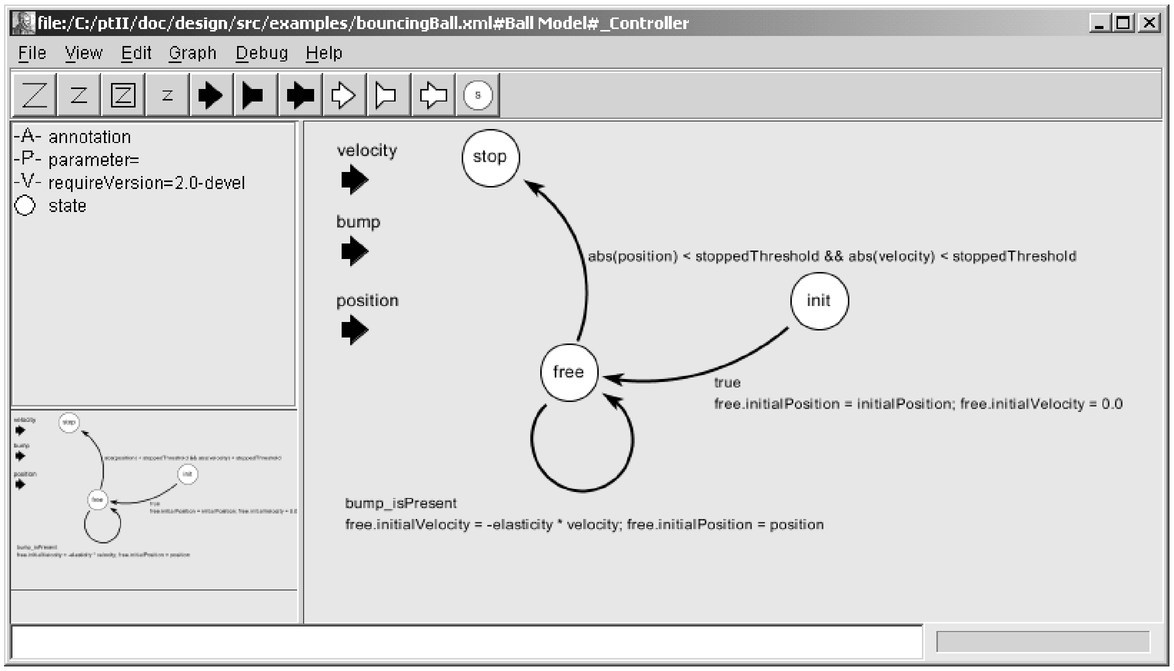
2.8.4 FSM and Modal Models

The finite-state machine domain (FSM) in Ptolemy II is a relatively less mature domain (but mature enough to be useful) with semantics very different from the domains covered so far. An FSM model looks different in Vergil. An example is shown in figure 2.36. Notice that the component library on the left and the toolbar at the top are different for this model. We will explain how to construct this model.

First, the FSM domain is almost always used in combination with other domains in Ptolemy II to create *modal models*. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a *state* in a finite-state machine. The circles in figure 2.36 are states, and the arcs between circles are *transitions* between states.

A modal model is typically a component in a larger model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will probably need to add ports. Figure 2.37 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. It represents a bouncing ball. Three outputs have been added, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, and then look inside, you will get an FSM editor like that in figure 2.36, except that it will be almost blank. The only items in it will be the ports you have added. You may want to move these ports to reasonable locations.


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

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

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

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

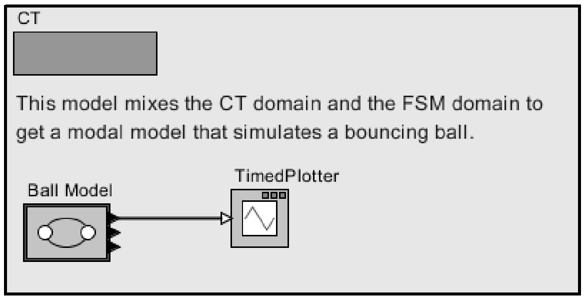
FIGURE 2.36. Finite-state machine model used in the bouncing ball example.

To create a finite-state machine like that in figure 2.36, drag in states (white circles). You can rename these states by right clicking on them and selecting “Customize Name”. Choose names that are pertinent to your application. In figure 2.36, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting “Edit Parameters”, and specifying an initial state name. In this example, the initial state is *init*.

To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or right clicking and selecting “Configure”) allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in figure 2.38. In that dialog, we see the following:

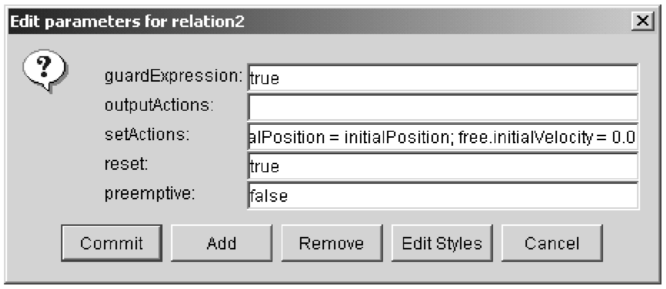
- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below).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

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



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

FIGURE 2.37. Top-level of the bouncing ball example. The *Ball Model* actor is an instance of *modal model* from the *utilities* library. It has been renamed.



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

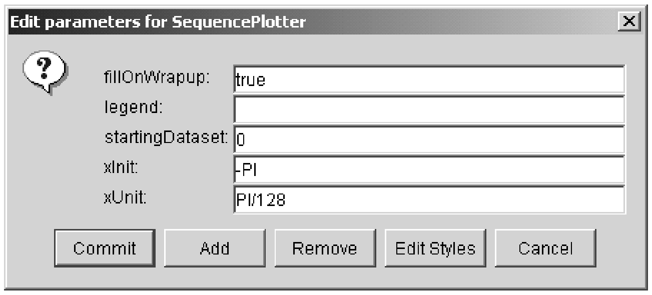
FIGURE 2.38. Transition dialog for the transition from *init* to *free* in figure 2.36.

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

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

To create a refinement for a state, right click on the state, and select “Add Refinement”. You will see a dialog like that in figure 2.42. You can specify the class name for the refinement, but for now, it is best to accept the default. Once you have created a refinement, you can look inside a state. For the bouncing ball example, the refinement of the *free* state is shown in figure 2.40. This model exhibits certain key properties of state refinements:

- Refinements must contain directors. In this case, the CTEmbeddedDirector is used. When a continuous-time model is used inside a mode, this director must be used instead of the default CTDirector (see the CT chapter for details).
- The refinement has the same ports as the modal model, and can read input value and specify output values. When the state machine is in the state of which this is the refinement, this model will be executed to read the inputs and produce the outputs.
- In this case, the refinement simply defines the laws of gravity. An acceleration of -10 m/sec^2 (roughly) is integrated to get the velocity. This, in turn, is integrated to get the vertical position.
- A *ZeroCrossingDetector* actor is used to detect when the vertical position of the actor is zero. This results in production of an event on the (discrete) output *bump*. Examining figure 2.36, you can see


Edit parameters for SequencePlotter

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

FIGURE 2.39. Parameters of the SequencePlotter actor.

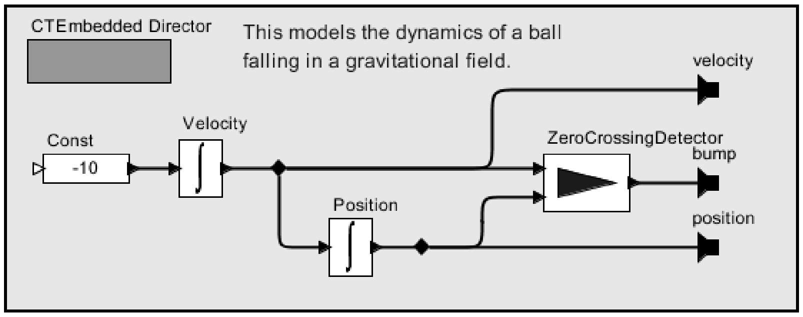


FIGURE 2.40. Refinement of the *free* state of the modal model in figure 2.36.

that this event triggers a state transition back to the same *free* state, but where the *initialVelocity* parameter is changed to reverse the sign and attenuate it by the *elasticity.* This results in the ball bouncing, and losing energy.

As you can see from figure 2.36, when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop,* which has no refinement. This results in the model producing no further output. The result of an execution is shown in figure 2.41. Notice that the ball bounces until it stops, after which there are no further outputs.

This model illustrates an interesting property of the CT domain. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. Try removing the *stop* state from the FSM, and re-run the model. What happens? Why?

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

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

2.9 Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in figure 2.30 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces

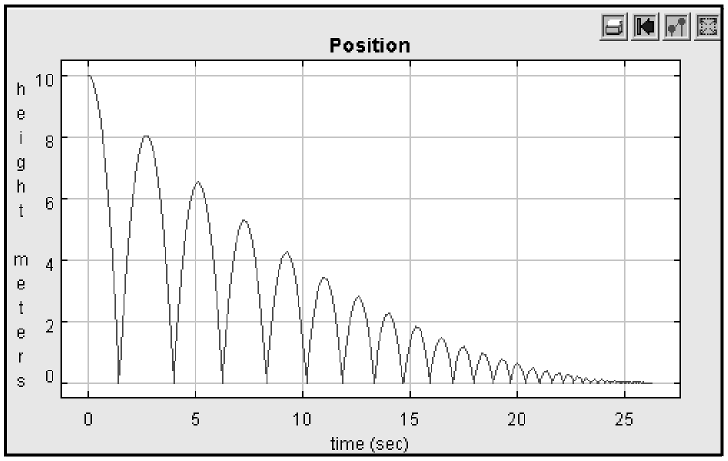


FIGURE 2.41. Result of execution of the bouncing ball model.

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

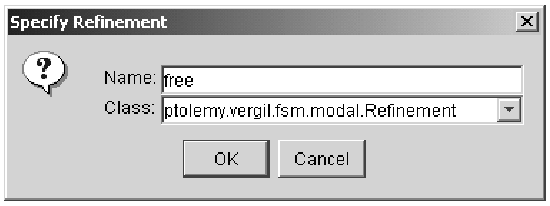
256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

The *SequencePlotter* actor has some pertinent parameters, shown in figure 2.39. The *xInit* parameter specifies the value to use on the horizontal axis for the first token. The *xUnit* parameter specifies the value to increment this by for each subsequent token. Setting these to "-PI" and "PI/128" respectively results in the plot shown in figure 2.43.

This plot is better, but still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in figure 2.44, filled in with values that result in the plot shown in figure 2.45. 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


Specify Refinement

Name: free
Class: ptolemy.vergil.fsm.modal.Refinement

FIGURE 2.42. Dialog for creating a refinement of a state.

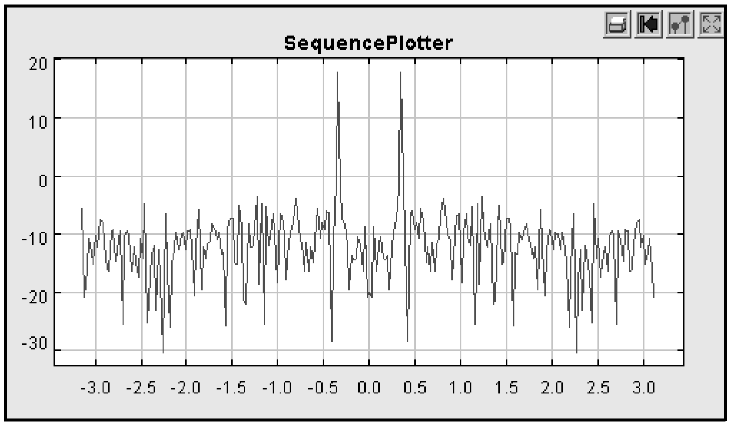
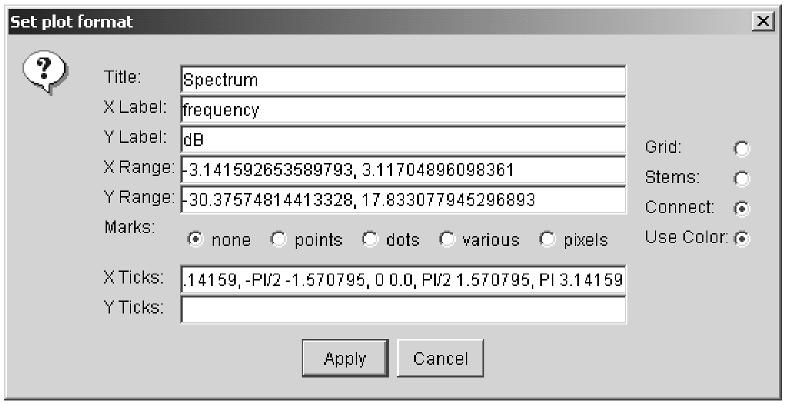


FIGURE 2.43. Better labeled plot, where the horizontal axis now properly represents the frequency values.

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.



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

FIGURE 2.44. Format control window for a plot.

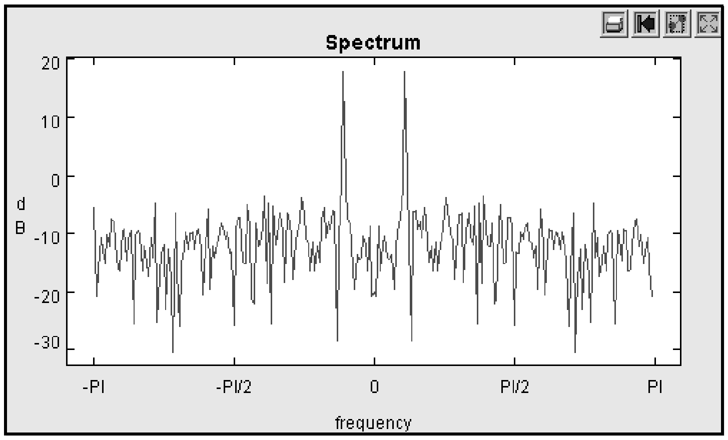


FIGURE 2.45. Still better labeled plot.