2

Using Vergil

Authors: Steve Neuendorffer

2.1 Introduction

Vergil is the Graphical User Interface for Ptolemy II. This chapter will guide you though using Vergil to create and manipulate Ptolemy models. Figure 2.1 shows a simple Ptolemy II model in Vergil, showing the graph editor, one of several editors available in Vergil.

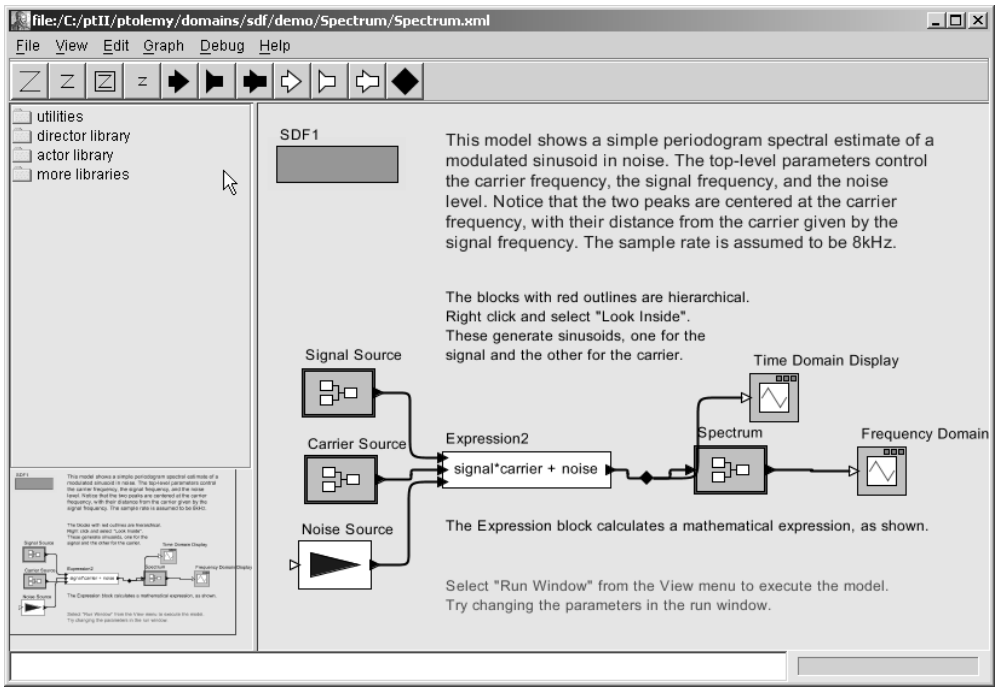

FIGURE 2.1. Example of a Vergil window.

2.2 Quick Start

The traditional first programming example is Hello World. Why break tradition?

First start Vergil. From the command line, enter “vergil”, or select Vergil from the Start menu. You should see a welcome screen 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.

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 before we run our example: 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, right click on the *Const* actor and select “Edit Parameters”. 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.

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 director to 1, which can be done

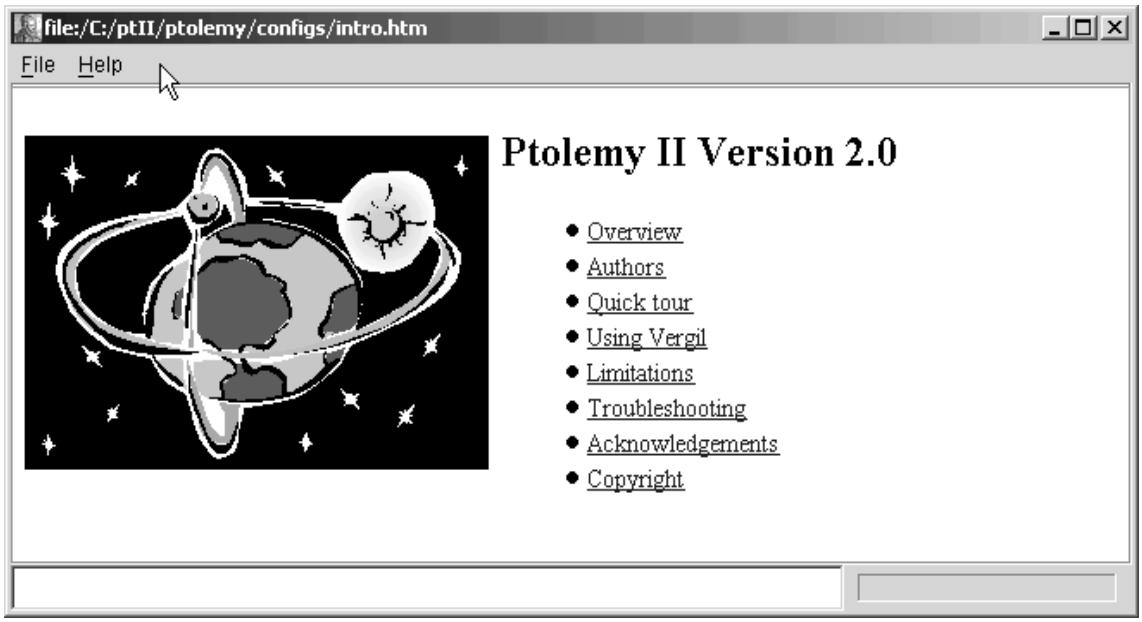

FIGURE 2.2. Welcome window.

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.

2.3 Data Types and the Type System

So what is really going on here? The *Const* actor is creating values on its output port. The *Display* actor is consuming data values from its input port and displaying them in the run window. The value that is created by the *Const* actor can be any type of object. For example, 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 of two elements: an integer named value and a string named name), or even [1,1;1,1] (a two by two matrix). They all seem pretty much the same in the Display, but Ptolemy knows the difference between them! To see the difference, try creating the model in Figure 2.7. The *Ramp* actor is listed under *sources* and the *AddSubtract* actor is listed under *math*. Set

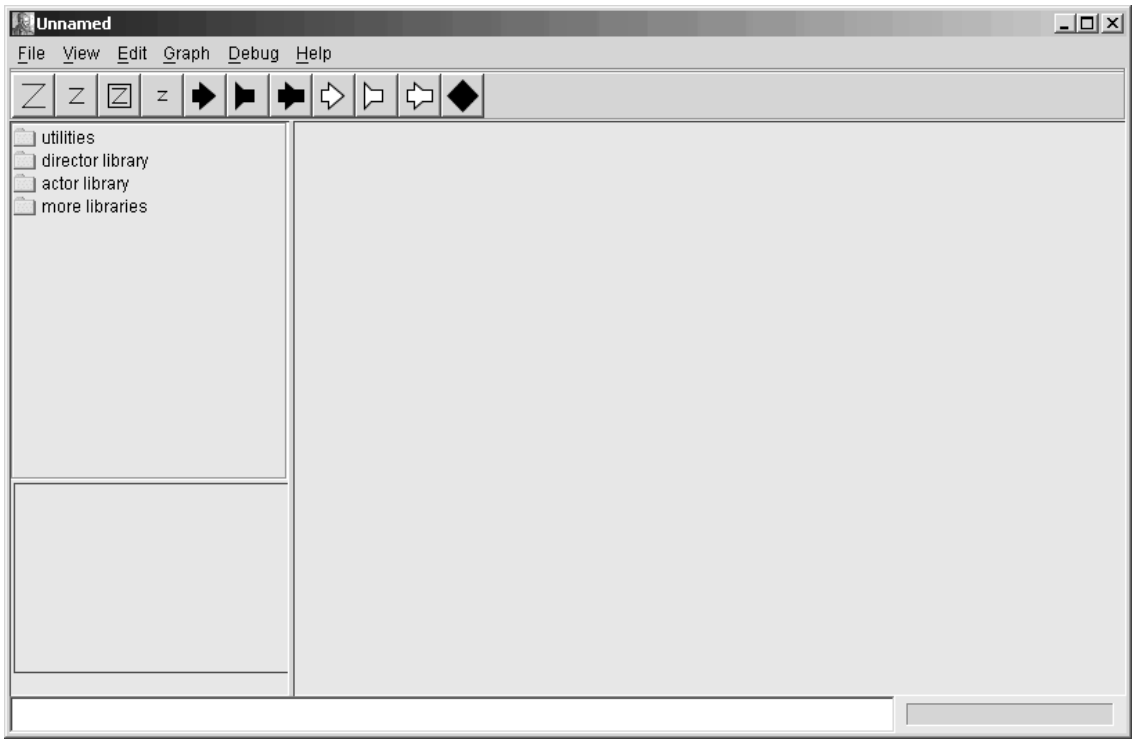

FIGURE 2.3. An empty Vergil Graph Editor.

FIGURE 2.5. The Const parameter editor.

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. 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:

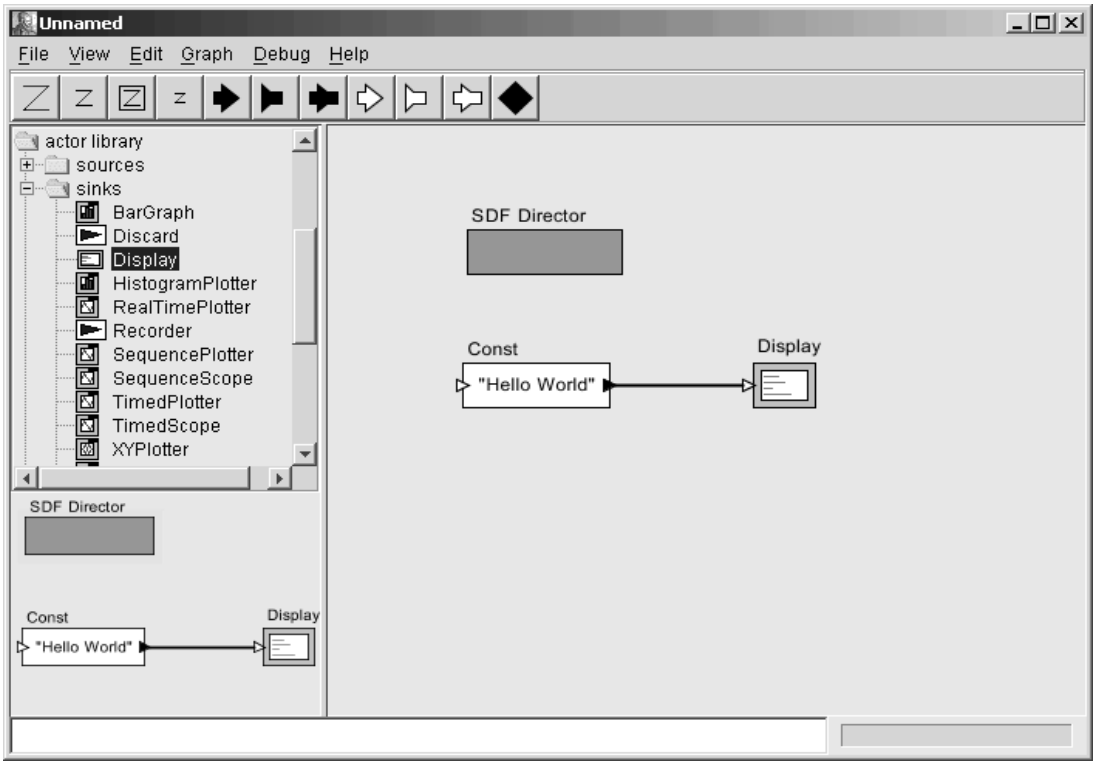

FIGURE 2.4. The Hello World example.

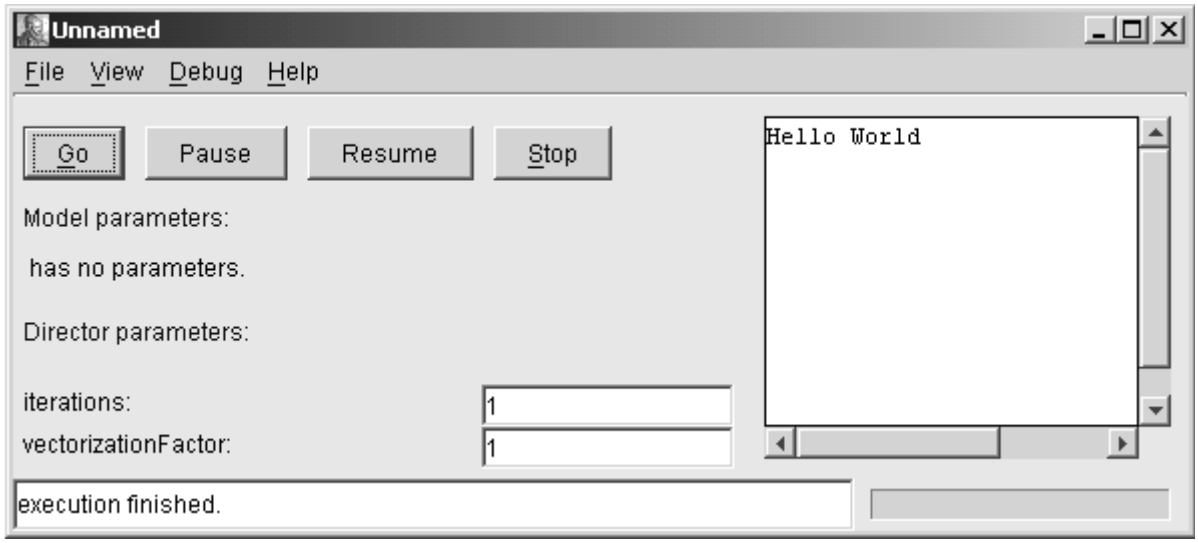

FIGURE 2.6. Execution of the Hello World example.

Change the value of the Const actor back to “Hello World”. When you execute the model, you should see an error window popup. Not to worry, this window is just telling you that you have tried to subtract a string value from an integer value, which doesn’t make much sense at all.

Let’s try a small change to the model to get something that is executable. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port. 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[1]. Notice that the upper port is hollow; 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”.

There are actually two things going on here. The first is that all the connections to the same port must have the same type. Ptolemy automatically converts the integers from the *Ramp* to strings. The second is that the strings are added together as strings usually are in Java, which means concatenating them.

2.4 Hierarchy

Let’s look at a slightly more interesting problem. In this case, a small signal processing problem, where we are interested in recovering a signal based only on noisy measurements of it. First open a new document and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add the noise to our measurements. First, using the context menu (right click over the composite actor), select “Rename” and give the composite a good name, like “Channel”. Then, using the context menu again, select “look inside” on the actor. You should get a blank graph editor. The original graph editor is still open. To see it, move the new one using its title bar.

First we have to add some external ports. There are several ways to do this, but clicking on the port toolbar button is probably the easiest. The port toolbar button is the small black triangle at the upper left. Create two ports and rename them *input* and *output*. Using the context menu on the background,

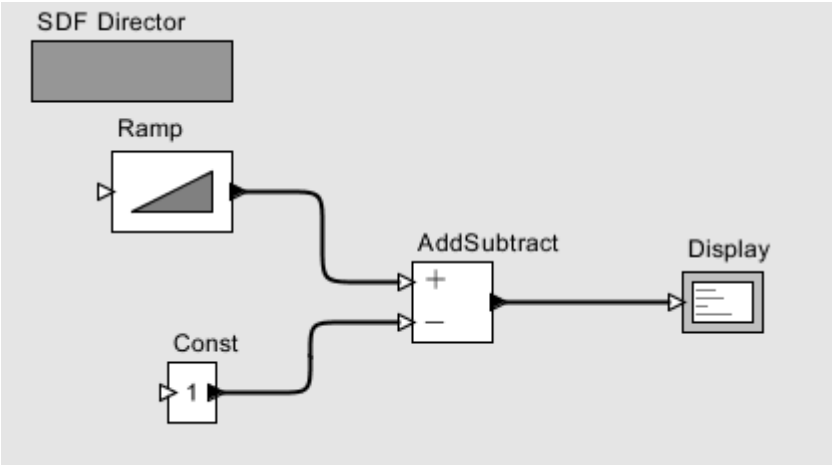

FIGURE 2.7. Another example

1. **Hint**: The connection can sometimes be difficult to select by clicking on it, since you have to precisely on it. To select it more easily, drag out a small box that overlaps it.

select *Configure ports* and set *input* to be an input port and *output* to be an output port. Then using these ports, create the diagram shown in Figure 2.8[1].

The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *sources* library. Now if you close this editor and return to the previous one, you should be able to easily create the model shown in figure 2.9. The *Sinewave* actor is listed under *signal processing*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline. If you execute this model (you will probably want to set the iterations to something reasonable, like 200), you should see something like Figure 2.10.

2.5 Parameters and Expressions

The values of parameters can be expressions. The details of the expression language are described in chapter 4, but we give a brief summary here. First, 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 exam-

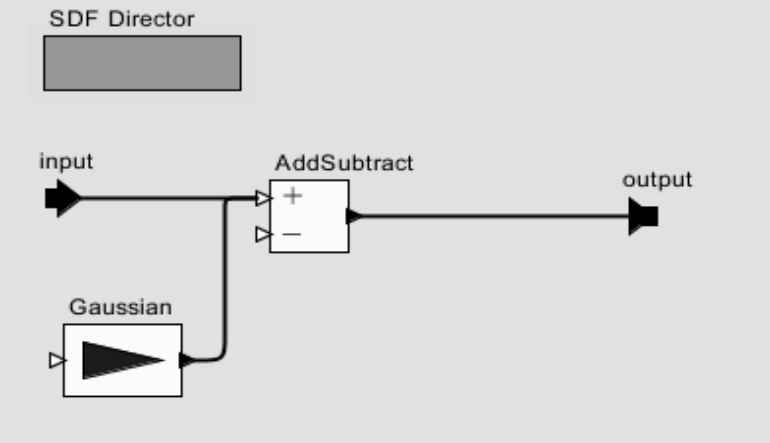

FIGURE 2.8. An example of a hierarchical model.

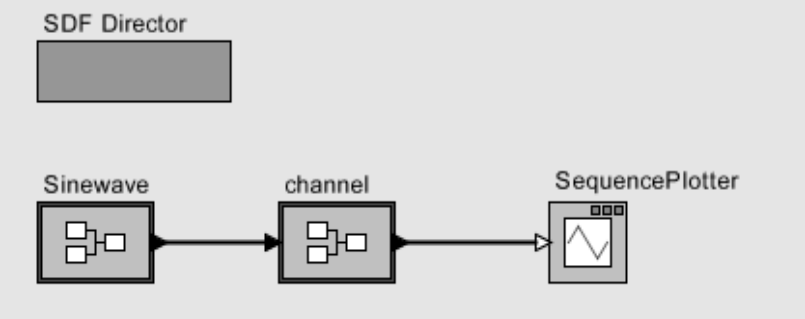

FIGURE 2.9. A simple signal processing example.

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

ple,

```
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.

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.

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.

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. You can add parameters to actors (composite or not) by right clicking on the actor, selecting "Configure" and then clicking on "Add."

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 constraint being that the elements all have the same type. Thus, for example, "{1, 2.3}" is illegal because

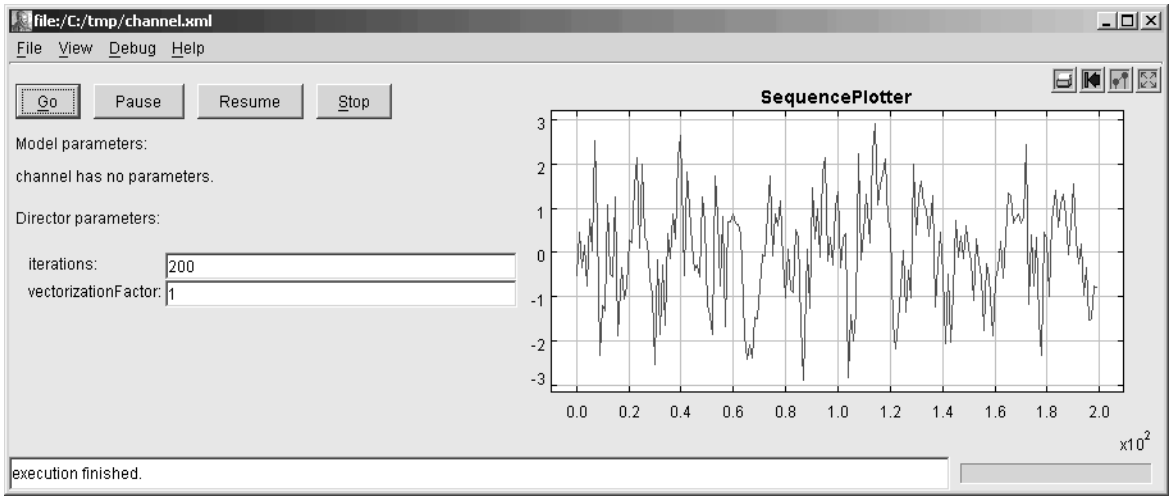

FIGURE 2.10. The output of the simple signal processing example above.

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.

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]."

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

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).

The 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 4.5 and 4.4.

One slightly subtle function is the random() function. 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, 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.

Every element and subexpression in an expression represents an instance of Token (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*).*name*(*args*), where *name* 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()
```

which returns the integer 3.

The MatrixToken classes have three particularly useful methods, illustrated in the following exam-

ples:

```
[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 types currently supported in the language are boolean, complex, fixed point, double, int, long, arrays, matrices, records, and string. Note that there is no float or byte. 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 in section 9.6.4.

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

2.6 Broadcast Relations

In the previous section we showed only one noisy measurement of the original signal. Now let's try to remove some of the noise. First, make three copies of the channel by selecting the one we created before, copying and pasting. We want to feed the original signal through all four channels and average the outputs of the channel. To broadcast the output of the Sinewave to more than one place, first create a relation (represented by a diamond), and then connect each of the ports to the relation. The relation can be created using the toolbar, or by control-clicking on the background. This should allow you to create the model shown in Figure 2.11. The parameter of the *Const* actor is 4, and the *MultiplyDivide* actor is found in the *math* library.

2.7 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 value it depends on. The number of output values that is created is determined by the number of iterations.

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 invoked (*fired*). Some require several input samples (*tokens*) before they can be fired, and produce several tokens when they are fired.

One such actor is a spectral estimation actor. Figure 2.12 shows a system that computes the spectrum of a sine wave. The spectrum actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 2.13 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**. 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.

2.8 Using the Plotter

The plot shown in figure 2.13 is not particularly satisfying. It has no title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the Spectrum actor produces

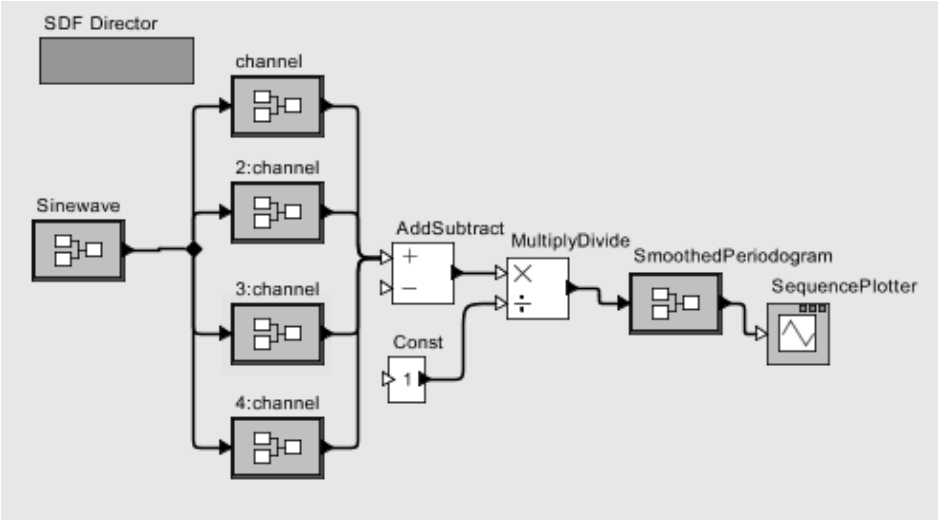

FIGURE 2.11. An example of a broadcast relation.

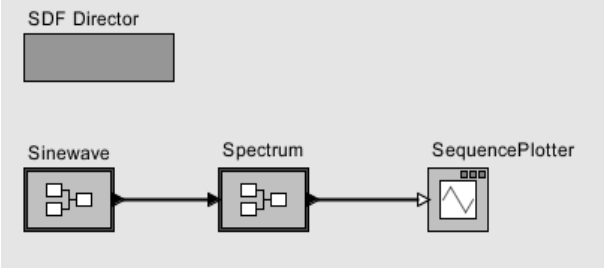

FIGURE 2.12. A multirate SDF model.

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 2.14. 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 2.15.

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 2.16, filled in with values that result in the plot shown in figure 2.17. 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"

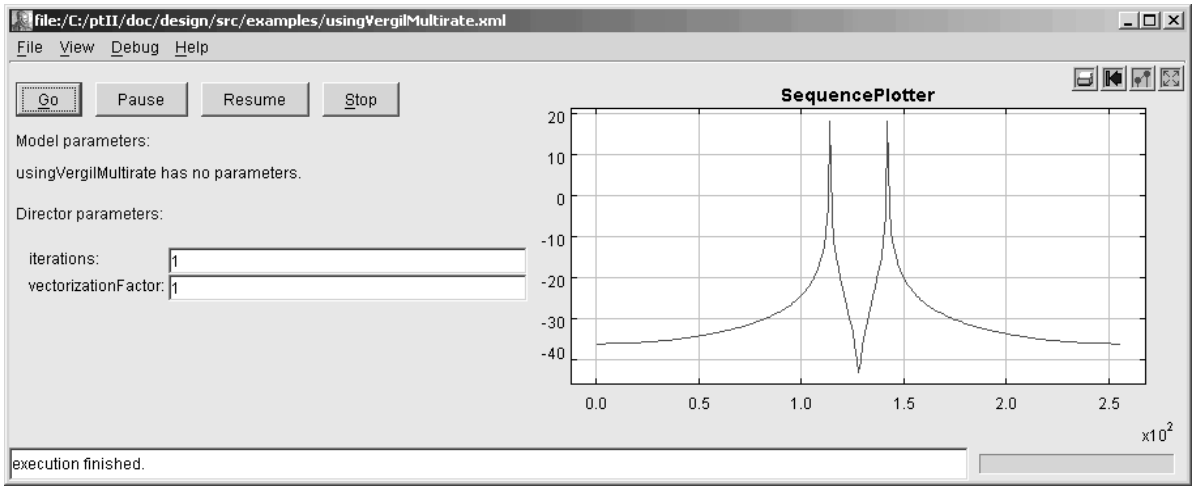

FIGURE 2.13. Execution of the multirate SDF model.

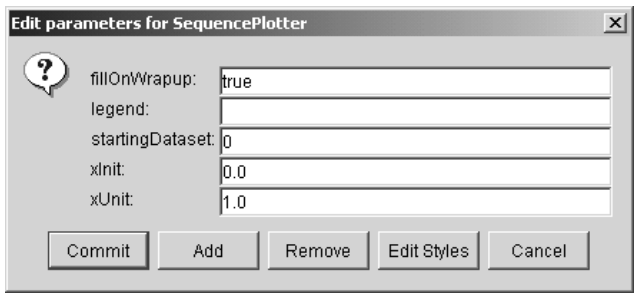

FIGURE 2.14. Parameters of the SequencePlotter actor.

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

- Connecting lines can be eliminated by deselecting “connect”
- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

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

  The syntax in general is:

  label value, label value, ...

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

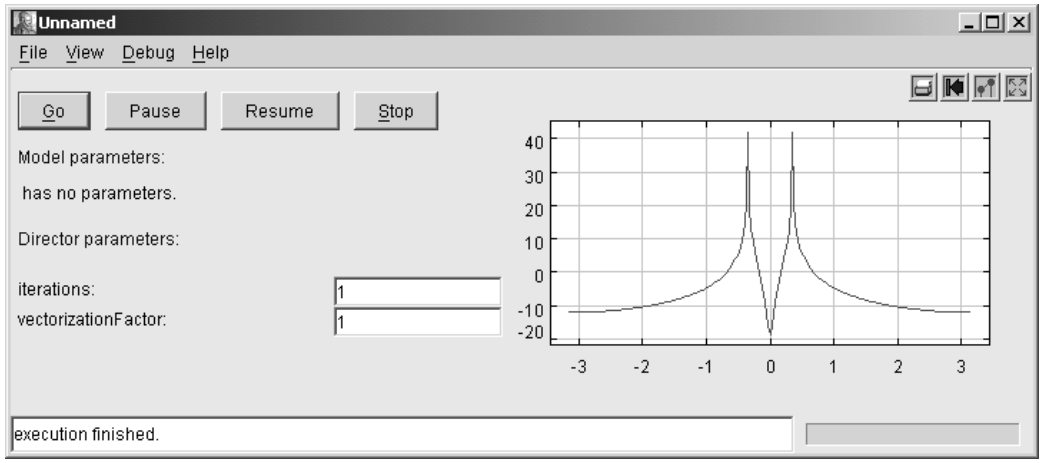

FIGURE 2.15. Better labeled plot, where the horizontal axis now properly represents the frequency values.

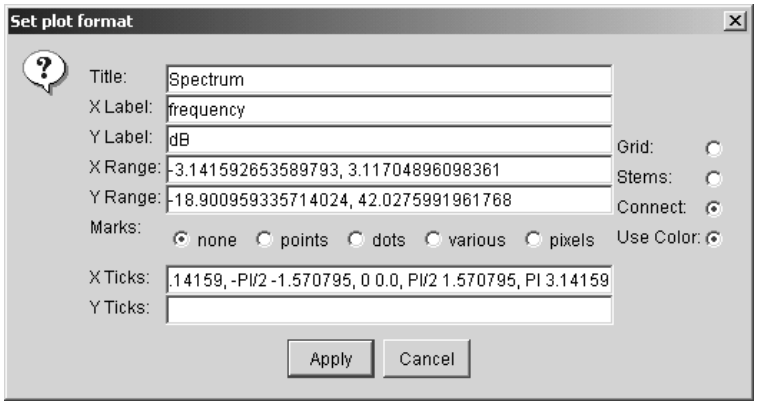

FIGURE 2.16. Format control window for a plot.

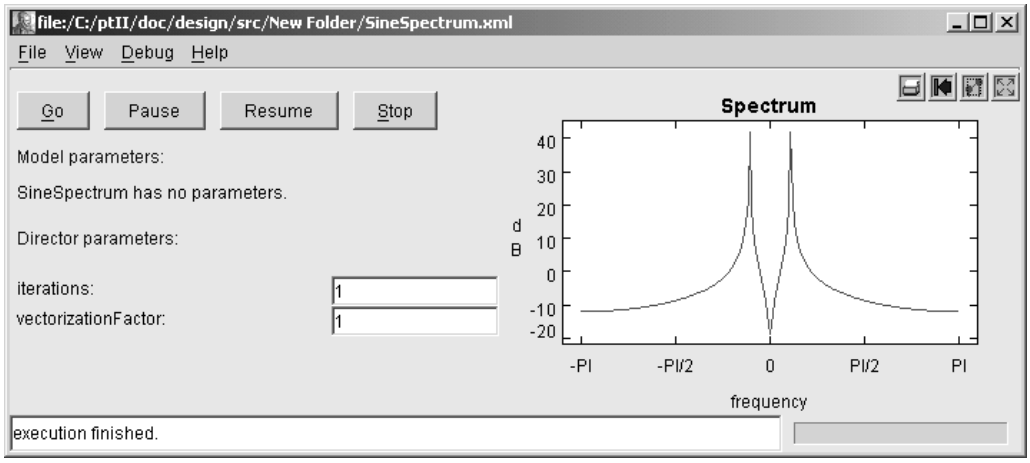

FIGURE 2.17. Still better labeled plot.