An Informal Guide to Using Pablo

Ruth A. Aydt*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

May 12, 1992
Revised October 29, 1992
Revised Sept 6, 1993
Revised April 12, 1994

*Supported in part by the Advanced Research Projects Agency under ARPA Contract Number DABT63-91-K-0004, by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, by a grant from the Digital Equipment Corporation External Research Program, and by a collaborative research agreement with the Intel Supercomputer Systems Division.

Contents

Introduction to the Document and Tutorials

This informal document introduces the reader to the Pablo Performance Visualization system from a user's perspective via a series of detailed examples. The current release of the software does not include a comprehensive User's Manual, but the authors believe that after following and understanding the tutorials presented here, the reader will be ready to use the Pablo system for processing their own data files. We encourage the reader to execute the Pablo tutorials on their system as they read through the examples provided. The features of Pablo are introduced in the context of the tutorials – subsection headings of each tutorial chapter can be used to locate the text describing a given feature of the Pablo system. Additional on-line documentation for specific parts of the system is accessible through *Help* buttons located on the individual screens and dialogs that make up the Pablo user interface. The paper *The Pablo Performance Analysis Environment* provides an overview of the Pablo system and should be read before attempting the tutorials presented in this guide.

Pablo was designed to organize and visualize information collected from programs executing on parallel processors. However, in the first four tutorials we will be working with small files containing information about baseball games to provide an easy introduction to the basic operations which must be performed any time Pablo is used. Whereas performance data trace files usually contain many types of records and potentially thousands of records of each type, these fabricated baseball files have only two types of records and are quite short. The simplicity of the baseball files makes it easy for the reader to see how the input record types affect the choices presented to the user when configuring the Pablo visualization environment. In addition, the small number of records in these files makes it practical for the user to watch each data record "flow through" the Pablo execution graph for processing and display. The simple files allow the beginning user to concentrate more on understanding the environment and less on comprehending the details of the input data.

As stated, each baseball file has two types of records. Records of the first type contain information on the number of hot dogs sold during each inning of the game. Records of the second type contain information on each player in the game, including the team number, the player number, the number of hits, and the number of at-bats.

When given a file containing data, the obvious question is "How can I manipulate and view this data so that it will be most useful to me?" One option is simply to look at the file, but for that you don't need Pablo. In the tutorials, we ask questions about our data and use Pablo to answer the questions. In the process of answering our questions we introduce and demonstrate the various features of the Pablo system. Each tutorial will answer one or more questions about the data and demonstrate several features of the system. The later tutorials build on the earlier examples, covering in detail only those features being presented for the first time.

The final two examples take as input a performance trace data file that was captured using the Pablo instrumentation system. These tutorials do not cover the details of using Pablo; instead, they show how the basic building blocks provided by the Pablo visualization system can be used to manipulate and present information from actual performance trace files to gain insight into program behavior.

The trace data for tutorials five and six was gathered on an eight node iPSC/860 hypercube executing a parallel PDE solver code—a detailed explanation of the PDE program can be found in Appendix G. In addition to the default statistics reported by the instrumentation system, the source code was instrumented to produce trace records for various other events such as: *procedure* *entry*, *procedure exit*, *blocking send*, *non-blocking send*, *blocking receive*, and *non-blocking receive*. The Pablo instrumentation interface was used to choose which procedures should be instrumented – procedure entry and procedure exit events were collected for five different routines: *InitGrid* (initialize grid), *IterateEdge* (iterate edge), *Converged* (converge edge), *Iterate*, and *Convergence* (global converge). These procedures were assigned numeric IDs by the instrumentation system and the events are tagged with those identifiers in the trace data file.

Message send and receive events (both blocking and non-blocking) report the source and destination nodes, message length, and message type. For the file we will be using, there are eight fixed message types: *maximum number of iterations*, *convergence check frequency*, *grid dimension*, *partition width*, *partition data*, *send boundary points "up"*, *send boundary points "down"*, and *convergence value broadcast*. Each type in this list was assigned an integer ID which is included in the trace event records as the message type. For our files, the integers 0–7 were assigned, where 0 corresponds to *maximum number of iterations*, 1 to *convergence check frequency*, and so on.

As you work through the tutorials and become familiar with the system, try to think of other ways Pablo might be used to answer a particular question. Also, think of other questions you might want to ask – "using Pablo" is really building up execution graphs from generic modules and then configuring those modules to address questions specific to your data. The power of Pablo is realized when the full range of transformation and display options provided by the environment are utilized to find the "best" answers to the questions posed, and when having those answers raises even more questions.

The figures referred to in the tutorial chapters can be found in the appendices. Configuration parameters for the individual modules making up the execution graph in each example are also presented in tabular form in the appendices.

Comments, questions, and problem reports regarding the Pablo Performance Visualization system should be sent to *pablo-bugs@guitar.cs.uiuc.edu*.

1 Tutorial 1

In the first tutorial, we are interested in how many hot dogs are sold during the innings of game 1. We know the data for game 1 is in a file called *Game1.ascii*, and we want to display the information graphically on the screen. Having stated the question for our particular set of data, we can get started.

1.1 Starting Pablo

To begin, move into the System/Build directory and enter the Pablo environment by typing

runPablo

You should see a window labeled *Pablo.* This is the main Pablo window that is used to control the system and assemble execution graphs. Across the top of the window are several pull-down menus labeled *File*, *Run*, *Configure*, *Module*, and *Help.* Just below these menu items is the system status area showing the current state of the environment. It should report *Idle*, today's date and time, and a *NULL Configuration.*

FIGURE 1.1

1.2 Adding a Module; Changing the Default Module Name

At this point we need to select the generic Pablo modules that will be connected and tailored to answer our particular questions. To do this, position your mouse on the *Module* selection of the *Pablo* window, depress the left button, pull down the menu, and select the *Add...* item. You should see a new window entitled *Pablo Module Creation.*

FIGURE 1.2

The *Pablo Module Creation* window shows all of the possible modules that can be used to build up Pablo execution graphs. To answer our first question, we need the data from the *Game1.ascii* file, so we want a *FileInput* module. By default, a *FileInput* module with the name *FileInput-1* is always selected when the *Pablo Module Creation* window pops up. We want to be more descriptive, so let's change the module name to *GameStatistics.* [1] To make this change, position your mouse in the *Module name:* text editing box following the default name *FileInput-1.* Click the left mouse button to set the cursor focus and use your delete key to erase the default name. You may also erase the default name by positioning the mouse anywhere on the string *FileInput-1*, clicking the left mouse button twice to highlight the entire name, and pressing your delete key to erase the selected characters. After erasing the default, type in the new name – *GameStatistics.* To actually add the module to the execution graph, click the left mouse button on the *Add* box located near the bottom of the window. You should see a bubble appear in the main Pablo window with the name *GameStatistics* and a new display window labeled *GameStatistics.* The display window will be used to show the number of bytes processed by the file input module.

[1] The system will accept module names containing space and slash (/) characters. However, these characters will cause problems if you decide to save and reload your execution graph. It is best to avoid the space and slash characters when naming your modules.

For this first tutorial we simply want to map some data from our game file to a graphical display, in our case a bargraph. To select a generic bargraph module, click the left mouse button on the *BarGraph* item appearing under *Display* in the *Pablo Module Creation* window. The name *BarGraph-1* should be displayed in the *Module name:* box. This time we'll use the default name, so simply click on the *Add* button without first editing the module name. This *Add* click should cause a second bubble named *BarGraph-1* to appear in the main Pablo window, and a bargraph display window entitled *BarGraph-1* to appear on the screen.

We don't need any more modules to answer the question being asked in this tutorial, so select the *Dismiss* button on the *Pablo Module Creation* window.

1.3 Repositioning Display Windows and Module Bubbles

The display windows that appear when certain modules are added will initially be centered within the Pablo main window. You may use your standard window manager commands to move these windows anywhere on your screen. You will probably want to reposition the *BarGraph-1* display window in this tutorial because it partially obscures the *GameStatistics* display window. Display windows can be moved at any time during the Pablo session.

You may also want to reposition the module bubbles in the Pablo main window to form a more pleasing graph layout. To move a module bubble, position the mouse on the bubble to be moved, depress the middle mouse button, drag the module bubble to the new location, and release the button.

1.4 Connecting Modules to Form an Execution Graph

The next step is to connect the modules to form an execution graph. File input modules are always at the top of our execution graph. Data stored in files enters the execution graph at the file input modules and flows from them through connections to the processing and display modules which have been selected. In this example, we have a single file input module called *GameStatistics* and a single display module called *BarGraph-1*. To connect our two modules, pull down the *Module* menu from the main Pablo window and select the second item – *Connect...* . At this point, the system status line in the main window should show *Waiting for first module or exit*. Position the mouse on the file input module, *GameStatistics*, and click the left mouse button to select it as the source module. After this selection, the status line should display *Waiting for second module or exit*. Now move the mouse to the *BarGraph-1* module and click the left button to select it as the recipient of the data. [2] The system status line changes back to *Waiting for first module or exit*, and the screen display should be very similar to the one shown (bubble and window placement may vary).

FIGURE 1.3

Our simple execution graph for this tutorial is complete. It consists of two modules joined by a single connection. The execution graph connections are often referred to as *pipes*. A pipe carrying data out of a module is called an *output pipe*, while one supplying data to a module is called an *input pipe*. In this tutorial, the single pipe is an output pipe for module *GameStatistics* and an input pipe for module *BarGraph-1*.

[2] There is no direct way to remove connections once they have been established. Module deletion, introduced in tutorial two, provides an indirect method of removing unwanted connections.

To exit the module connection phase, position the mouse over any of the bubbles and click the right mouse button. The system status line should change to *Idle.*

1.5 The Module Configuration Phase

At this point we have built a generic execution graph consisting of a file input module and a bargraph display module. We haven't yet specified what data file we want to operate on, or what records and fields we want to display. It's now time to configure the generic modules to answer our specific question.

1.5.1 Entering the Module Configuration Phase

To begin the module configuration process, pull down the *Configure* menu and select the *Graph...* option.

This selection causes Pablo to sort your execution graph and present individual modules for configuration. The system status line will display the name of the module currently being configured. Pay close attention to this information because the order in which configuration occurs will not necessarily be the order in which modules were added to the graph. In this simple example things are pretty obvious, but that is not always the case. The configuration information requested from the user will vary somewhat depending on the module characteristics, but the user interface is consistent across all modules so you should quickly become familiar with the configuration process.

1.5.2 Configuring a File Input Module

We configure our file input module *GameStatistics* first. At the onset we said the information we were interested in was found in the file *Game1.ascii.* Pablo is set up to look in a particular directory for data files, and the standard Motif file selector dialog box is used to display files in that directory. [3] Use the Motif file selector to specify the file *Game1.ascii* and press *OK* to continue. After the file has been selected and opened successfully, the *GameStatistics* display window associated with the file input module should be updated with the name of the selected input file and its size in bytes. A portion of the horizontal bargraph in the *GameStatistics* display should be filled in corresponding to the number of bytes read from the beginning of the input file at this phase in the configuration process. These initial bytes contain information about the records in the file that will be used when configuring other modules in the graph with connections to the file input module.

Configuration continues with next module – in this case *BarGraph-1.*

1.5.3 Configuring a Display Module (BarGraph); Status Board

You will be presented with two windows entitled *Status Board* and *Bind Records to Input Ports and Output Record Fields.* The second of these will be referred to as *Bind Records* throughout the remainder of the document. The system status line in the main window shows *Configuring module BarGraph-1* and both the *Status Board* and *Bind Records* windows display the module name *BarGraph-1* below their respective title bars.

FIGURE 1.4

[3] If your input data file is not in the default directory, you can use the Motif file selector dialog to make the appropriate directory and file the active selection.

The *Status Board* window gives the current state of the module configuration, and the *Bind Records* window is used to interactively configure the module.

Look first at the *Status Board* window. It is made up of four panels labeled *Input Pipe*, *Input Port*, *Output Port*, and *Output Record*. The first panel, *Input Pipe*, shows sources of data for the module being configured. It displays a pipe icon for every pipe in the execution graph which terminates at this module. Each pipe icon is labeled with the name of the module where the pipe originates. For the module *BarGraph-1*, we see a single input pipe originating at the *GameStatistics* module. In addition to input values coming from other modules, it is possible for some modules to accept constant data values entered directly by the user. To represent this potential data source, a *User Input* icon appears in the *Input Pipe* panel of every *Status Board* window.

The *Input Port* panel appears second in the *Status Board* window and shows the input values required by the particular module being configured. In the case of *BarGraph-1*, we see a single icon labeled *Input* indicating the bargraph module takes one input value.

The third panel, *Output Port*, shows the output produced by the current module – none in the case of the display module *BarGraph-1*. Finally, the *Output Record* panel lists fields making up every output record produced when the module executes. This panel will only have entries if the module has output ports.

Notice that the borders around the Input Pipe icon *GameStatistics* and the Input Port icon *Input* are the same color. This indicates the input port *Input* will be filled with data from records on the input pipe *GameStatistics*. Since this module has a single input pipe and a single input port, the user interface automatically associates the pipe with the port. If there were multiple input pipes or multiple input ports, the user would be required to associate input sources with input ports prior to the appearance of the *Bind Records* window. We will see an example of this more complicated configuration in later tutorials.

1.5.4 Configuring a Display Module (BarGraph); Configuration Board Detail

Before moving on to the *Bind Records* window, position your mouse on the *GameStatistics* icon in the *Status Board* window and click the left button. You should see a new window entitled *Configuration Board Detail* displayed on the screen.

FIGURE 1.5

The *Configuration Board Detail* window gives in-depth information on the various entities represented by icons on the *Status Board*. Looking at the details for the *GameStatistics* input pipe we see the *Available Records* are *Concessions* and *Player Record*. The file *Game1.ascii* which we specified in the configuration of the *GameStatistics* module has been opened and read. The file contains two types of records, *Concessions* and *Player Record*. This information was passed down the pipe from the *GameStatistics* module to the *BarGraph-1* module and is being presented so that you can configure the display module to run with the type of records in the particular data file you selected. If you position your mouse on the *Concessions* entry and click the left mouse button, the *Available Fields* will be updated with the fields contained in the *Concessions* records. In a similar manner, you can click the left mouse button on the *Player Record* entry to display the fields contained in records of that type.

To see detailed information for the input port named *Input*, position the mouse on the *Input* icon in the *Status Board* window and click the left button. The *Configuration Board Detail* window is updated to list the traits that are valid for this input port – notice that scalar, vector, and two-dimensional arrays of type integer, float, and double are all accepted by the bargraph display module. The set of records which may provide data to the selected input port is also shown.

1.5.5 Configuring a Display Module (BarGraph); Bind Records

We are now ready to configure the module *BarGraph-1* using the *Bind Records* window interface. To begin the process, position your mouse on the item *Game Statistics* under the heading *Input Pipes* and click the left button. The window is updated to show the records available on the *GameStatistics* input pipe and the single input port that must be filled. Notice how this information corresponds to what we saw in the *Status Board* and *Configuration Board Detail* windows. To configure the module, we "bind" a particular field value from our incoming data to the input port.

We said we wanted to see the number of hot dogs sold. That information is contained in the Concessions record. Click on the *Concessions* item in the *Records* column of the *Bind Records* window. The *Fields* column will be updated with the field names *Inning* and *Hot Dogs Sold.* Each field item has a unique color associated with it. [4]

FIGURE 1.6

We are interested in the data values for the particular field *Hot Dogs Sold* and want to bind that field to the input port of our bargraph module. To perform the field to input port binding, first position the mouse on the *Hot Dogs Sold* item in the fields list and click the left button to select it as the field to be bound. Next, select the input port by positioning the mouse on the *Input* item and clicking with the left mouse button. This sequence causes the field *Hot Dogs Sold* to be bound to the input port *Input* – the color of the *Input* item should have changed to match that of the field *Hot Dogs Sold.* The user interface will actually allow you to select the field and the input port in either order when performing the binding process – you could have clicked first on *Input* and second on *Hot Dogs Sold* to achieve the same binding results.

Sometimes the selection is not registered properly and the binding is not made (no color change occurs). When that happens, use the left mouse button to reselect the field and input port items until the binding is successful as indicated by a change in the color of the input port item.

When the field to input port binding is complete, you will notice several changes to the *Bind Records* window in addition to the color change of the *Input* entry. The *Input Pipes* and *Records* areas of the window are desensitized to indicate that you must complete your work with the current record, *Concessions*, before you can make further selections in those areas. The letter "E" appears to the left of the *Concessions* record, indicating that extractions will be performed by the module on all records of that type. Near the bottom right of the screen, the buttons *Clear* and *Done* are active – we'll explain these two buttons here but don't press them until later!

The *Clear* button causes all bindings from fields to input ports and output fields to be removed. *Done* is sensitive when all the input ports and output fields have been bound to fields in the selected input record. Pressing *Done* indicates you have finished making binding selections for the current record and are ready to continue with the configuration process.

Before experimenting with the *Done* and *Clear* buttons, turn your attention back to the *Status Board* and *Configuration Board Detail* windows for a moment. Reselect the *Input* icon on the *Status*

[4] The black and white screen dumps found in the appendices do a poor job of showing the unique colors associated with the items in the Fields list – use your imagination!

Board and notice how the *Traits*, *Records*, and *Fields* now reflect the binding made in the *Bind Records* window. (You'll need to click on the *Concessions* item in the *Configuration Board Detail* window to display the fields information).

FIGURE 1.7

Returning to the *Bind Records* window, click first on the field *Innings* and then on the input port *Input* – notice the color change indicating that the port is now satisfied by the *Inning* field. Press the *Clear* button and notice the *Done* button is desensitized because the input ports are no longer satisfied. Finally, rebind the field *Hot Dogs Sold* to the input port *Input.* With the color of *Input* matching the color of *Hot Dogs Sold*, press the *Done* button to convey that you are finished with the bindings for the *Concessions* record. The *Continue* button is activated, signaling all necessary bindings have been made. Select *Continue* to close the *Bind Records* screen and proceed with the configuration process.

1.5.6 Configuring a Display Module (BarGraph); Parameter Configuration

It is now time to configure the bargraph display widget. You should see the window:

FIGURE 1.8

The default values appearing on this screen are controlled by the Pablo resource file. The bargraph display will remember a historical high water mark for up to *Historical marker interval* redisplays. The *Minimum value* and *Maximum value* are used to scale the data and present it appropriately in the display. The *Vertical and Horizontal axis labels* are for user convenience.

For our situation, we will reset the *Maximum value* to *2500* [5] and the *Horizontal axis label* to *Hot Dogs Sold.* This screen makes use of standard Motif text dialogs to accept parameter input. In general, use your mouse to select the text to be changed, clicking the left button to position the cursor. [6] Use your delete key to erase existing text, then enter the new text in the dialog. After all desired changes have been made, select *OK.* The *BarGraph-1* display should change to reflect the new maximum and horizontal label.

1.6 Saving a Configured Execution Graph

We have now finished the steps necessary to configure the execution graph in this example. All configuration screens should have disappeared, and the system status should show *Idle.* Before processing the data in the input file, we will save the execution graph and module configuration information. To do this, pull down the *File* menu and select *Save Configuration.* You will see a standard Motif file selection dialog labeled *Save Configuration to Directory.* The filter is specified in the Pablo defaults file, and you simply need to select (or create) a writable directory where the configuration information will be saved. For consistency, call it *Tutorial1.* Press *OK.* After a small delay during which the configuration is written to disk, the dialog window will be closed.

[5] For help in selecting minimum and maximum values that are reasonable for a given input file, see the program "FileStats" distributed with the Pablo software release. It scans an input file and reports the range of values for every field in the file.

[6] For some unknown reason, the cursor does not always appear in the text fields. It is "invisible" but there.

1.7 Executing a Configured Graph

We are now ready to run the configured graph and view our data. In this example, we expect to see displayed on the bargraph the data values telling us how many hot dogs were sold in an inning from every Concession record for game one. The *GameStatistics* display should also be updated as additional bytes are read from the input file.

To execute the graph, pull down the *Run* menu and select item three, *Run*. You should see a burst of activity on the displays, and the status line should change to *Executing graph continuously* and then quickly to *Execution complete*. At this point you are undoubtedly saying "Gee that seems like a lot of work setting things up for such a short run time – things went so fast I couldn't even tell what happened!"

1.8 Using a Display Module Callback (BarGraph)

What did happen? For each record in our game file that gave Concession information, we displayed how many hot dogs were sold on the bargraph display. As soon as the BarGraph-1 module saw a new Concession record, the information from the previous record was erased. The final display shows how many hot dogs were reported sold in the last Concession record in the file. If you look carefully at the *BarGraph-1* window, you should see that the display is registering a non-zero value. To find out the actual value being displayed, position your mouse in the *BarGraph-1* window and click the left button. Your screen should be as follows:

FIGURE 1.9

Most of the display modules are equipped with callbacks which can be used to obtain more detailed information about the values being displayed. By clicking the left button with your mouse positioned inside the boundaries of the display window, you activated the callback for the bargraph module. The callback information tells us there were 100 hot dogs sold during the inning that was reported last in the file. Notice we aren't sure what inning that was. If you think back to the phase where we configured the module BarGraph-1, you'll recall the Concession records contained two fields – *Inning* and *Hot Dogs Sold*. We did nothing with the *Inning* values, so we have no way of knowing when the 100 hot dogs were sold.

Take another look at the *BarGraph-1* window and notice the line near the top of the bargraph. This is the high-water mark that we referred to when configuring the bargraph display widget. It shows the highest level reached for the historical interval specified. In this example, the mark tells us that during some inning we sold almost 2500 hot dogs – even if things ran so fast that we never really saw that value displayed. We'll address the problem of not being able to tell how many hot dogs were sold in a particular inning and also slow down the display rate in the next tutorial.

Lower the callback information window by clicking the left mouse button on the *OK* selection in the window. Next, position your mouse in the *GameStatistics* display window and click the left button. A callback information window for that display will appear, showing the actual value being displayed and the percentage that value is of the maximum. Since we have processed the entire input file, the displayed value is equal to the total number of bytes in the file. Click on the *OK* button to lower the callback window.

1.9 Deleting the Execution Graph; Exiting Pablo

Before beginning the next tutorial, we must reinitialize Pablo to an *Idle* status with a *NULL Configuration.* To do this, pull down the *File* menu and select *Delete Graph...* . Verify that you really want to delete the graph by choosing *OK* in question dialog that appears after you select the *Delete Graph...* option. The bubbles will be cleared from the main Pablo window and the display windows will disappear. We are now ready to begin building a new Pablo execution graph.

If you wish to exit the Pablo environment entirely, select *Quit* from the *File* pull-down menu.

2 Tutorial 2

In the second tutorial we build on what we learned in the first, asking to see not only the hot dogs sold per inning, but also the correlation between the number of hot dogs sold and the particular inning the sales occurred. We introduce guidelines for avoiding and recovering from errors made while building an execution graph, and show how to execute the graph more slowly, allowing individual display updates to be viewed.

2.1 Loading a Saved Configuration

In this tutorial we will reuse our saved configuration from tutorial one. When loading a saved configuration, you must always begin with an empty main window – you cannot load a saved configuration if you have already started building a new execution graph in your current Pablo environment. If you selected the *Delete Graph...* item from the *File...* menu at the end of tutorial one, you should be ready to proceed. If you exited the Pablo environment, restart it now using the *runPablo* command.

To reload the saved configuration, pull down the *File* menu and select *Load Configuration....* You should see a standard Motif file selection dialog labeled *Load Configuration from Directory* – use it to specify the directory where you saved your tutorial one configuration.

The original execution graph should be displayed in the main Pablo window, and another file selection dialog entitled *Select Input File for GameStatistics* should appear. The selection box in this dialog will show the file *Game1.ascii* – the file we chose during the configuration of the *GameStatistics* module in the first tutorial. It is possible to use a different input file than the one you originally selected, with the restriction that the new file must have record types identical to the original file. [7] If you wanted to specify a different file, you would simply modify the selection box value to reflect your new choice. In our case, we will again use the file *Game1.ascii* so click the *OK* button without changing the selection box default value.

The system status area of the main Pablo window will be updated with the name of the loaded configuration, *Tutorial1*, and the date the configuration was saved.

If you'd like, you can pull down the *Run* menu and select the *Run* option to execute the configuration that was just loaded. The behavior should be exactly as it was in tutorial one. If you did choose to run the graph, you'll need pull down the *File* menu and select *Delete Graph...*, then reload the saved configuration as described above before continuing with tutorial two.

At this point, you should see your execution graph from tutorial one in the main Pablo window, two display windows, and a system status of idle. The *GameStatistics* display window should show the name of the input file (Game1.ascii), the total number of bytes in the file (539), and a bar representing the number of bytes processed from the beginning of the file. The bargraph display window, *BarGraph-1*, should report the minimum and maximum range but will not yet be updated with any values corresponding to hot dogs sold.

2.2 Adding Modules to a Configured Graph; Error Message Dialog Box

We will expand our original execution graph by adding new modules to track and display the hot dogs sold per inning. A *Synthesize Vector* module will be used to create a vector containing the

[7] In this release, selecting a new file with record types different than those in the original file will cause the system to exit after printing an error message that an expected record type was not found in the input file.

number of hot dogs sold by inning, with the inning represented by the index into the vector. A second *BarGraph* display module will be used to show the vector of data values.

To add these modules, pull down the *Module* menu and select the item *Add...* Using the procedure described in tutorial one and default module names, add a *Synthesize Vector* module and another *BarGraph* module. Your first attempt to add the *BarGraph* module with the default name will fail, as the name will be *BarGraph-1* and a module with that name already exists. You will be notified of this via an information dialog box with the message *Module with name BarGraph-1 already exists.* After reading the message, close the information dialog box by pressing the *OK* button. The *Module name:* entry in the *Pablo Module Creation* window will change to *BarGraph-2*. Selecting the *Add* button at this point will cause a new module bubble, *BarGraph-2*, to be added to the Pablo main window. A second bargraph display window entitled *BarGraph-2* will also be created on the screen.

Reposition the module bubbles and display windows to form a more pleasing execution graph and make better use of screen space. Use a layout similar to the one shown:

FIGURE 2.1

Finish building the expanded execution graph by connecting the new modules. To enter module connection mode, select the item *Connect...* from the *Module* pull-down menu. Using the left mouse button to select the source and destination modules for each pipe, make connections from *GameStatistics* to to *SynthesizeVector-1* and from *SynthesizeVector-1* to *BarGraph-2*. Use the right mouse button to leave the module connection mode after completing all connections.

2.3 Execution Graph Error Recovery

Before continuing with the module configuration phase of this tutorial, we'll offer some guidelines for avoiding and recovering from common errors made while building the execution graph. Each subsection outlines a potential problem and offers suggestions on how to recover without restarting from scratch.

2.3.1 Removing Unwanted Modules

Preplanning the execution graph can help you avoid adding modules that are not needed. If you do add a module that is unnecessary, perhaps by accidentally double-clicking the *Add* button in the *Pablo Module Creation* window, it is possible to remove the module using the *Module Delete* menu item.

Let's practice deleting a module at this time. Begin by adding an "unwanted" file input module using the *Pablo Module Creation* window which should still be popped up from the previous section. A new module bubble with the default name *FileInput-1* should appear in the Pablo main window together with a corresponding *bytes processed* display when you press the *Add* dialog button. To delete the module, pull down the *Module* menu on the main Pablo window and select *Delete....* The status line should display *Waiting for node selection or exit.* Position the mouse on the unwanted *FileInput-1* module bubble and press the left mouse button to delete the bubble and it's corresponding display. The system status should return to *Idle* – only one module can be deleted each time the *Module/Delete* option is selected.

If you accidentally select the *Module/Delete* combination when you do not have any unwanted modules, it is possible to return to the idle status without removing a module. To try this, pull down the *Module* menu and select the *Delete* item. At this time you have the four "wanted" module bubbles in the main window (GameStatistics, BarGraph-1, SynthesizeVector-1, BarGraph-2) and the status line should display *Waiting for node selection or exit.* To leave delete mode without removing a module, position your mouse over any of the module bubbles and click the right mouse button.

2.3.2 Renaming Modules

Once a module bubble has been created, the only way to rename the module is to delete the original module, reselect the correct module type using the *Pablo Module Creation* dialog, edit the *Module name* entry, and press *Add* to create the new module with the correct name. If you can remember to double-check the module name before you actually add a module, you can save yourself some time.

2.3.3 Correcting Module Connections

As stated in the first tutorial, once a connection has been made between two modules, there is no direct way to remove it. This limitation can prove very frustrating if you've spent a considerable amount of time building up your graph and then add an incorrect connection that cannot be removed. Preplanning the execution graph and arranging the modules in the Pablo window so that source modules are always above destination modules can help minimize the likelyhood of error.

Using the *Pablo Module Creation* window, add three "dummy" modules of any type that we can use to demonstrate connection error recovery – call them *D1*, *D2*, and *D3*.

Sometimes connection errors occur because you've lost track of where you are in the connection process. Pull down the *Module* menu and select *Connect....* The status line should change to *Waiting for first module or exit.* Select *D1* as the first module – the status line should change to *Waiting for second module or exit.* Notice the system does not report the source module for the current connection – if you've forgotten what node you selected, don't guess! Instead, click the right mouse button over any module to return to the idle state, then reenter module connection mode and add any missing pipes.

Before continuing, make sure the three test modules *D1*, *D2*, and *D3* are aligned vertically in the main Pablo window. This alignment is the default layout and unless you moved them after they were added no further adjustments should be needed. Make a connection from *D1* to *D3* and look carefully at your graph. It is impossible to tell that *D2* has no input or output pipes. Reposition *D2* so that you can see it is not involved in the connection graph. It's wise not to have more than two modules aligned vertically or you may be mislead as to the actual pipe connections.

At this point you should still be in module connection mode with *D1* connected to *D3* and *D2* unconnected. Let's say that we really didn't want a connection from *D1* to *D3*, but rather wanted *D1* connected to *D2* and *D2* connected to *D3*. We can't delete connections, but we can delete modules. Leave the module connection mode by clicking with the right mouse button on any module bubble and select *Module/Delete....* Click the left mouse button on *D1* – one of the the two modules connected to the misplaced pipe. An information dialog will be displayed with the message *Sorry, can't delete a module that has output pipe(s).*. Click *OK* to close the dialog and reselect *Module/Delete....* This time, click with the left button on the module *D3* – the module and the incoming pipe from *D1* will be erased. Use the *Pablo Module Creation* window to add a new *D3* module to replace the one we deleted. You have now recovered from the bad connection and can link *D1* to *D2* and *D2* to *D3*.

Because of the restriction that modules with output pipes cannot be deleted, it is best to add the links from the topmost module bubbles in the execution graph before making any connections from modules lower in the graph. Consider the graph we entered for this tutorial – if you linked *SynthesizeVector-1* to *BarGraph-2* and then made a bad connection from *BarGraph-1* to *SynthesizeVector-1* you would have to delete and reinsert both *BarGraph-1* and *SynthesizeVector-1* to recover from the error, removing a valid link in the process. However, if you had not made the link to *Bargraph-2*, you could recover from a bad connection between *Bargraph-1* and *SynthesizeVector-1* by removing and reinserting the single module *Synthesize Vector-1*. Check each connection as you make it – the sooner you notice an error the less painful the recovery.

To conclude our detour on error recovery, delete the three test modules and dismiss the *Pablo Module Creation* window.

2.4 The Module Configuration Phase

It is now time to configure the generic modules in our execution graph to answer our specific questions. Selections for *GameStatistics* and *BarGraph-1* have been reloaded from the saved configuration, but the modules *SynthesizeVector-1* and *BarGraph-2* have not yet been configured. It is very important to configure all modules in the graph before beginning execution.

Pull down the *Configure* menu and select the *Graph...* option. Pablo will present you with modules for configuration – the exact order of presentation may vary from what is described here, but the steps for the individual modules will be identical.

2.4.1 Verifying the Configuration for a File Input Module

Once a file input module has been configured, it is not possible to change the selected input file. We specified the input file to be used during the *Load Configuration* phase earlier in the session. At this point, we see an information dialog box telling us the name of the file associated with our *GameStatistics* module. This notifies us of the selection, but does not allow us to modify it. Press the *OK* button in the dialog box to continue.

2.4.2 Configuring a Synthesis Module (Synthesize Vector); Operation Selection

The modules of the *Synthesis* type all build up complicated data structures (vectors, 2-d arrays, sets of pairs, sets of triples) from individual records containing indices (for vectors and arrays) and values. To configure a module of this type, the user is first asked to specify the action to be performed by the module as it builds up the selected data structure.

You should see the window:

FIGURE 2.2

The window contains lists of *Action* and *Operation* toggle buttons. The behavior of the synthesize vector module is controlled by the selections made here. If the *Replace* action is specified, the *Operation* selection is not used. If the *Update* action is specified, the *Operation* selection controls how the vector cell at the given index is updated. To simplify the explanation, let's adopt the following shorthand notations:

- New Cell Value (NCV) — value of vector cell after the module runs
- Old Cell Value (OCV) — value of vector cell before the module runs
- Field Value (FV) — value of the field bound to the input port "Cell Value"

The behavior of the synthesize vector module for various selections is as follows:

Selection	*Behavior*
Replace	NCV = FV
Update/Addition	NCV = OCV + FV
Update/Subtraction	NCV = OCV − FV
Update/Multiplication	NCV = OCV × FV
Update/Division	NCV = OCV ÷ FV
Update/Minimum[8]	NCV = Minimum (OCV, FV)
Update/Maximum	NCV = Maximum (OCV, FV)
Update/Count	NCV = OCV + 1

For this tutorial, we want each cell in the vector to represent the number of hot dogs sold for the corresponding inning. We could assume that there is a single Concessions record for each inning in the game, but that is not guaranteed. To be sure we account for all the hot dogs in each inning, select the action *Update* and the operation *Addition.* (Here's a good place to exercise your imagination and ask other questions... For example, what selection could you make to report the number of Concession records in the input file for each inning in the game? [9])

Your final selection should be as shown:

FIGURE 2.3

Select *OK* to continue the configuration process for module *SynthesizeVector-1.*

2.4.3 Configuring a Synthesis Module (Synthesize Vector); Bind Input Pipes

During this phase of the configuration process, the user is asked to associate fields from the incoming data records with the indices and values needed for the particular data structure being synthesized. In our case, we are building a vector of values and need to specify the index into the vector and the value to be used to update the indexed cell. You will be presented with two windows entitled *Status Board* and *Bind Input Pipes to Input Ports.*

FIGURE 2.4

[8] For the Update/Minimum selection you will probably want to set your initial cell value to something other than zero. Otherwise any positive field values will always be greater than the old cell value and the generated vector will always contain zeros. The initial cell value is specified later in the configuration of the module.

[9] The selection *Update/Count* would keep track of the number of Concession records seen for each inning of the game.

The *Status Board* window is similar to the one we saw in tutorial one. The synthesize vector module takes two input values, *Index* and *Cell Value*, and produces the output value *Vector*. As explained in tutorial one, you may use the left mouse button to select an icon in the *Status Board* window and have detailed information about it presented in a *Configuration Board Detail* display.

The *Bind Input Pipes to Input Ports* window did not appear in the first tutorial. Recall our *BarGraph-1* module had a single input pipe and a single input port – the system automatically associated the pipe with the port. This was conveyed to the user via the color borders around the *GameStatistics* and *Input* icons in the *Status Board*. The current module, *SynthesizeVector-1*, has two input ports so the user is asked to specify the input sources for those ports using the *Bind Input Pipes to Input Ports* window. [10]

In this tutorial, both the index into the vector and the data value used to update the vector cell will be taken from records on the input pipe *GameStatistics*. To make this configuration choice, position the mouse on the *Game Statistics* item in the *Bind Input Pipes to Input Ports* window and click the left mouse button. Next, move the mouse to the input port entry *Index* in the same window and again click the left button. This should "bind" the input pipe *Game Statistics* to the input port *Index*, causing the *Index* item to be colored the same as the *GameStatistics* item. This binding should also be registered in the *Status Board* window – the borders around the *GameStatistics* and *Index* icons should be colored alike. To perform the second binding, position the mouse on the *Cell Value* item in the *Binding Input Pipes* window and click the left button. Next, move the mouse to the *GameStatistics* item and click the left mouse button to select the input pipe. The *Cell Value* entry should assume the same color as the *Game Statistics* entry and the *Status Board* icon borders should be updated. Notice that you may select the input pipe and the input port in either order when performing the binding process.

At this point, we have bound the input pipe *Game Statistics* to both input ports – *Index* and *Cell Value*. Take a minute to experiment with the *Clear* button in the *Bind Input Pipes* window, and also try binding the *User Input* to one or both of the *Input Ports*. Notice that the *Continue* button is sensitive only when both input ports have been bound to valid input sources. After becoming comfortable with the screen, restore the bindings so that *GameStatistics* is bound to both *Index* and *Cell Value*, then press *Continue*.

2.4.4 Configuring a Synthesis Module (Synthesize Vector); Specify Output Record Fields

The *Bind Input Pipes to Input Ports* window will be closed and a new window entitled *Specify Output Record Fields* will appear on the screen. Since this module produces output, the user must specify the fields that will make up the output record being created.

FIGURE 2.5

Near the upper left of the window is the name of the output record we are creating; the default name <<*SynthesizeVector-1*>> has been generated based on the name of the module being configured. We will change the name of our output record. To do this, position the mouse on the default name and double-click with the left mouse button. Press your delete key to erase the default name, and enter the new name *Hot Dogs Per Inning*. The names of all records in the

[10] If a module has a single input port, that port cannot be bound to a user input value.

system must be unique, so be careful not to enter any record name from the original input file or from other modules for which recombinations have been performed. [11]

It is possible to have fields in the output record originating from three different sources:

- Fields taken from records on input pipes which are bound to input ports.
- User input values.
- Output port values which are the results produced by the module.

These three sources are represented by buttons along the left side of the *Specify Output Record Fields* window. By default, no fields are created in the output record from the input pipe or user input sources. A field is created in the output record for each output port associated with the module. To see this, use the left mouse button to select the items *GameStatistics*, *User Input*, and *Module Result*. The first two selections will not cause any fields to be displayed in the right section of the window. The third selection, *Module Results*, will cause the field *Vector* to be displayed.

FIGURE 2.6

For this module, we will use the default output field name *Vector*, and not create any additional fields in the output record – we'll see how to do that in a future tutorial. To close the *Specify Output Record Fields* window, select *Continue*. Notice the *Status Board* window is updated so the borders around the output port icon *Vector* and the output field icon *Vector* assume the same color.

2.4.5 Configuring a Synthesis Module (Synthesize Vector); Bind Records

This window is similar to the one we saw in tutorial one when we configured the module *BarGraph-1*. To begin the field binding process, click on the item *GameStatistics* with the left mouse button. As before, we are interested in the number of hot dogs sold – information contained in the *Concessions* record. Select that record using your left mouse button. The *Current Record:* near the upper right of the screen will be updated and the fields for the record will be displayed. This time we will not only perform field extraction on the *Concessions* data records, but also copy records of that type through to the output pipe connected to the module being configured. In our case, we have a single output pipe connected to the *SynthesizeVector-1* module – if there were more output pipes connected, the information sent to each would be identical. To perform the configuration described, make sure *Concessions* is the current record and select the *Toggle Copy Thru* button near the botton of the *Records* column. A small 'C' should appear next to the *Concessions* entry in the *Records* list.

FIGURE 2.7

Using the procedure detailed in tutorial one, bind the field *Inning* to the input port *Index*, and the field *Hot Dogs Sold* to the input port *Cell Value*. After the second binding is made, the *Done* button near the bottom right of the screen should become sensitized. Your *Configure* window should be similar to the one shown – the field and input port entries should be color-coded to represent the bindings made.

[11] In the current release, there is no verification at this point to make sure the record name entered is indeed unique. If you enter a duplicate name, you will see a warning message when the *Specify Output Record* window is closed and the output record name will revert to the default values.

FIGURE 2.8

Select the *DONE* button to signal that you have finished the field bindings for the *Concessions* record. Finally, select *Continue* to proceed with the next phase of the configuration.

2.4.6 Configuring a Synthesis Module (Synthesize Vector); Parameter Configuration

It is now time to configure the synthesize vector parameters. You should see the window:

FIGURE 2.9

Your choices from the *Operation Selection* window are shown to remind you that the module will be performing an *Addition* operation. The default values on the display are controlled by the Pablo software. For our situation, we will reset the *Initial Vector Size* to 1 and leave the *Maximum Vector Size* and *Initial Cell Value* unchanged. If an index value is seen that falls outside the range of the initial vector size, the vector length will automatically be increased to accommodate the larger index. The maximum vector size is primarily a safeguard to insure that an unexpectedly large index value does not cause the system to generate a huge output vector.

For this tutorial, the final selection should be as shown:

FIGURE 2.10

Select *OK* to finish the configuration of the module *SynthesizeVector-1*.

2.4.7 Configuring the Module BarGraph-2

You will be presented with a *Status Board* and *Bind Records* window to configure the module *BarGraph-2*. In the *Bind Records* window, click on the input pipe *SynthesizeVector-1* to see the records available on that pipe. Two records types should be listed – *Concessions* and *Hot Dogs Per Inning*. The first, Concessions, was copied through the synthesize vector module unchanged while the second, *Hot Dogs Per Inning*, was generated by the module.

To configure the module *BarGraph-2*, select the record *Hot Dogs Per Inning* and bind the field *Vector* to the input port *Input*, and select *DONE*. Press *Continue* to close the *Bind Records* window.

On the display widget configuration screen, set the maximum value to *2500* and the horizontal axis label to *Hot Dogs Sold by Inning*. Select *OK* to continue.

2.4.8 Reconfiguration of the Module BarGraph-1

Status Board and *Bind Records* windows will be displayed for the module *BarGraph-1* showing the previous configuration information for the module. We do not need to make any changes to the original bindings, so simply select *Continue* in the *Bind Records* window. Our original selections for the display configuration of *BarGraph-1* are also fine, so choose *OK* on that configuration screen.

2.5 Using the Run Panel to Execute a Configured Graph

At this point, we have finished configuring the expanded execution graph in this example. All configuration screens should have disappeared, and the system status should show *Idle.* Before executing the graph, use the procedure outlined in Section 1.6 to save your new execution graph and configuration selections in a directory called *Tutorial2.*

We are now ready to run the configured graph and view our data. In this example, we expect to see displayed on *BarGraph-1* the data values telling us how many hot dogs were sold in an inning from every Concession data record for game 1. We expect *BarGraph-2* to show a per-inning count of the number of hot dogs sold.

To execute the graph at a controlled rate, we will use the run panel. Pull down the *Run* menu and select the first option – *Run Panel....* A small window labeled *Pablo Execution Control* will be displayed.

FIGURE 2.11

Reset the *Number of graph steps* value in the Motif text widget to 1, and select the the *Execute graph* option *N times.* Use the left mouse button to click once on the *Run* button. You should see the *BarGraph-1* display updated and the *BarGraph-2* display expanded to show two bargraphs – the second having a non-zero value. Position the mouse in the second bar of the display *BarGraph-2* and click the left button to invoke the callback. You will see that the column is number one and the current value is 100. This tells us that there is a Concessions data record in the file for inning one, and 100 hot dogs were reported sold in that record. Notice that the first index into the vector is zero – since there is no inning zero we expect that this bar will never be updated. The screen should be similar to the one shown although the position of the various displays may vary:

FIGURE 2.12

Close the callback information display and continue selecting the *Run* button in the *Pablo Execution Control* window, observing how the screen is updated after each click. The information shown on the *BarGraph-1* display (which is exactly the same as what we saw in the first tutorial) should correspond to the information shown in the last bar updated on the *BarGraph-2* display. Notice that the bar representing hot dogs sold for inning 9 in the display *Bargraph-2* is updated before the bar for inning 8 – this tells us that the concession data for inning 9 appears in the file before the data for inning 8.

Experiment using callbacks to get exact information about the values shown on the bargraph displays. After you have run the graph several times with each run causing an update to the bargraph display, you will stop seeing updates but the system status line will continue to report *Idle.* During this time, Pablo is processing data records in the file that are not Concession records and therefore they do not affect the displays we have configured. Continue pressing the *Run* button until the system status line changes to *Execution complete.* This status indicates that all input data records have been processed.

The single-step mode of execution is acceptable for small input files, but unrealistic when processing files containing large numbers of records. When learning about the Pablo visualization system, it is instructive to “watch” each record be consumed and processed by the execution graph you have built. However, this is rarely done when using the environment to examine real data files.

2.6 Ending the Pablo Session

To end the Pablo session, pull down the *File* menu and select *Delete Graph* to prepare for a new session, or *Quit* to exit the Pablo environment.

3 Tutorial 3

In the third tutorial we use the Pablo visualization system to calculate batting averages, taking as input the data on hits and at-bats from a baseball game file. Several new types of modules are used and the configured execution graph is more complex than what we have seen in previous examples.

3.1 Planning the Execution Graph

When using Pablo to perform somewhat complicated data manipulation and display operations, it is wise to plan your execution graph in advance. Our baseball data includes records reporting the team, player number, number of hits, and times at-bat. By manipulating this information, we can calculate the batting average for each player on each team, and the average over all players in the game.

In this tutorial, we first outline an execution graph that will calculate and display the averages described and then go on to build, configure, and execute the graph we have planned. It is important to realize that other combinations of generic modules would also achieve the desired result – rarely is there a single "correct" configuration for a given problem. In fact, as you become more familiar with the functionality of the modules in the system, you will undoubtedly see an increasing number of possible solutions for any given situation.

Consider the annotated figure representing our planned execution graph:

FIGURE 3.1

When labeling the modules in the graph, we extended the default module names to make them more descriptive of the task each node performs in this tutorial. Most generic data-manipulation modules can perform more than one type of operation – we have indicated the planned operation of each module in our graph by including the operation as part of the module name. As you build up a library of saved configurations, you may find it helpful to add descriptive module labels corresponding to the behavior of the generic module type in the specific execution graph – weeks or months later, understanding the functionality of a graph with only default module names can be almost impossible.

The *FileInput-1:Game_File* module will read our baseball game data file. The module *BinaryMath-1:Divider* will operate on the *Player* records, dividing the hits by the times at-bat to calculate the batting average. The calculated average will be written to a new record together with the team and player number from the input player record. (Copying fields from an input record to an output record has not been discussed previously and will be explained in detail below.) The *SynthesizeArray-1:Replace* module will create an array of batting averages using the team and player number as row and column indices. This array will be displayed by *Matrix-1:Batting_Avgs*.

The records generated by *BinaryMath-1:Divider* will also be directed to the *HistoryMath-1:Count* module which will increment a counter each time a record is seen, in effect counting the number of player records processed. The array of batting averages by team and player created by *SynthesizeArray-1:Replace* will be used as input to the module *ReductionMath-1:Sum* where all elements in the array will be added together to produce a single output value representing the sum of the batting averages for all the players. *BinaryMath-2:Divider* will be used to divide this sum by the total number of player records seen so far to compute the average hitting performance. The module *Led-1:Overall_Batting_Avg* will show this value throughout the run.

3.2 Building the Execution Graph; Using the Pablo Module Creation Scroll Bar

Having planned our execution graph, we are now ready to enter the graph into the Pablo visualization system. If you already have a Pablo session running, reinitialize it to an *Idle* status with a *NULL Configuration* by selecting the *Delete Graph...* option from the *File* menu. If you are not currently running Pablo, start it using the *runPablo* command. Select *Add...* from the *Module* menu.

Add the modules indicated, changing the default names to the more descriptive ones used in FIGURE 3.1 and taking care not to use space or slash characters in the new names. The modules *Matrix* and *Led* will not be visible in the initial *Pablo Module Creation* window. Both are entries under the *Display* heading, but the entire set of display modules will not fit into the space allocated so a scrolling list is used. To move forward in the list and bring the *Led* and *Matrix* modules into the active window, position your mouse on the rectangular box just below the upward-arrow at the top of the scroll bar located on the right side of the *Display* list. Depressing the left mouse button, use the mouse to move the box down in the scroll bar causing the list to scroll forward. When *Led* and *Matrix* are visible in the active window, release the left mouse button. Select each one by clicking the left mouse button on the module name, editing the default name, and choosing *Add.* In a similar fashion, use the *Math* list scroll bar to locate the *ReductionMath* module so that it may be selected. Dismiss the *Pablo Module Creation* window when all eight modules have been added.

Use the middle mouse button in the main Pablo window to reposition the module bubbles to form a layout similar to the one shown in FIGURE 3.1. Select *Connect...* from the *Module* menu and add the connecting pipes – source modules appear above destination modules in the graph. Exit the connect module phase by clicking with the right mouse button on a module bubble.

3.3 Saving a Graph Layout

We will save our graph at this point to preserve our work in case we have a power failure before we are able to complete and save the entire configuration. There are also cases where the same graph layout with different configuration parameters can be used to solve different problems. In general, it is a good idea to save your layout after entering a complex graph if only as a hedge against Murphy's Law.

Recall from tutorial one that complete configurations are saved as multiple files under a specified directory. In contrast, Pablo saves a graph layout in a single file containing the names of the module bubbles, their locations, and the connections between them.

To save the layout in this tutorial, pull down the *File* menu and choose *Save Layout.* A file selection dialog box labeled *Save Layout to File* will appear. Specify the file where the layout should be saved, perhaps *Tutorial3-layout*, and press *OK.* The dialog box will be closed and you are ready to proceed.

3.4 The Module Configuration Phase

Enter the module configuration phase by selecting *Graph...* from the *Configure* pull-down menu. The order in which you are asked to configure your modules may vary from the one presented here, but the steps for the individual modules are identical. The system status line will always report the module currently being configured.

The configuration process will be guided by the notes we made when planning our execution graph – the operations of the data-manipulation modules are specified and the comments give an idea of the record types and fields we will bind to our input ports.

3.4.1 Configuring the Module FileInput-1:Game_File

As always, the first module presented for configuration is the file input module. Use the file selection dialog box to choose *Game2.ascii* as the input file for this tutorial.

3.4.2 Configuring a Binary Math Module (BinaryMath-1:Divider); Operation Selection

The *Binary Math* module is capable of performing one of several binary math operations on it's input data. It accepts two inputs, *Left Operand* (LO) and *Right Operand* (RO). Based on the operation chosen, it manipulates the operands to produce a *Result* value. The behavior of the binary math module for the available operations follows:

Binary Operation	*Behavior*
Addition	Result = LO + RO
Subtraction	Result = LO − RO
Multiplication	Result = LO × RO
Division	Result = LO ÷ RO
Minimum	Result = Minimum (LO, RO)
Maximum	Result = Maximum (LO, RO)
Average	Result = (LO + RO) ÷ 2
Double-precision Division	(double) Result = LO ÷ RO
Double-precision Average	(double) Result = (LO + RO) ÷ 2

The input operands may be of type integer, float, or double. For all operations except *Double-precision Division* and *Double-precision Average*, the result will have the same type as the most precise input operand. For example, if you compute the average of an integer and a float, the result will be a float. The *Double-precision* operations are necessary when you operate on less precise input operands, but want a double-precision result.

The *Binary Math* module will accept both scalars and arrays as inputs. If both input operands are arrays, they must have the same dimensions and the same number of elements in each dimension. When one input is a scalar and the other an array, the result will be an array having the same dimension and size as the input array. As an example, if the left operand is the scalar 10, the right operand the vector [5.5, .02], and the operation subtraction, the result will be the vector [4.5, 9.98].

For this tutorial, we want to divide the number of hits by the times at-bat to calculate the batting average for each player. Both of the inputs are integers – if we use the *Divide* operation, our result will be an integer as well. We need to use double-precision division to get an accurate batting average. Select *Double-precision_Division* from the operation menu for *BinaryMath-1:Divider* and press *OK* to continue.

3.4.3 Configuring a Binary Math Module (BinaryMath-1:Divider); Bind Input Pipes

Status Board and *Bind Input Pipes to Input Ports* windows will be displayed. Using the procedure outlined in tutorial two, use the *Bind Input Pipes to Input Ports* window to bind the input pipe *FileInput-1:Game_File* to both the *LeftOperand* and *RightOperand* input ports. Your bindings should be reflected in the *Status Board* display. Recall that you can select icons in the *Status Board* window for detailed display in a *Configuration Board Detail* panel. Press *Continue* in the *Bind Input Pipes to Input Ports* window to proceed with the configuration process.

3.4.4 Configuring a Binary Math Module (BinaryMath-1:Divider); Specify Output Record Fields – Non-Default Selections

The next phase of configuring *BinaryMath-1:Divider* involves specifying the fields to be included in the output records created by the module. A *Specify Output Record Fields* window similar to the one seen in tutorial two will be displayed. In this tutorial, we will include fields in the output record that originate from the input pipe, user input, and module result sources. While the user input fields are not required to answer the questions posed in this tutorial, including them provides a good opportunity for you to become familiar with all the options available when creating an output record.

Begin by changing the output record name from the default value <<*BinaryMath-1:Divider*>> to *Batting Avg for Player*. Next, select the input pipe entry *FileInput-1:Game_File* – the right side of the display should be activated and the heading changed to *Output Fields Satisfied By FileInput-1:Game_File*. Use your left mouse button to click on the *FileInput-1:Game_File* icon in the *Status Board* window. The *Configuration Board Detail* window will show two record types available on the input pipe – *Concessions* and *Player Record*. We will be using fields from the *Player Record* as input to our *BinaryMath-1:Divider* module. Select the *Player Record* entry in the *Configuration Board Detail* window to view the fields making up records of that type.

When populating output fields from input pipes, the system requires the fields must be taken from records which are bound to input ports of the module. For this particular module, we can therefore copy any of the fields from the *Player Record* type into the output record. We are interested in knowing which player has the calculated batting average, where any given player is identified by his team and player number. We see from the *Configuration Board Detail* display that this identifying information is available in the *Player Record*. We'll copy those fields from the input record to the output record for use by modules further down in the execution graph.

Position your cursor in the *Edit Field Name* box of the *Specify Output Record Fields* window and enter the field name *team*. Press the *Add* button and the new field will appear in the upper-right panel of the window. An icon representing the field will also appear in the *Output Record* section of the *Status Board* window. The border of the icon will be colored to indicate that it's source is the input pipe *FileInput-1:Game_File*. Return to the *Edit Field Name* box, clear the *team* entry using your erase key, type in the new field name *player number*, and select *Add*. Your screen should be similar to what is shown:

FIGURE 3.2

Click the *Done* button to signal that these two output fields are the only ones we will add which originate from the input pipe *FileInput-1:Game_File*.

Next, use the left mouse button to select the *User Input* item located on the left side of the *Specify Output Record Fields* window. The heading *Output Fields Satisfied by User Input* will appear on the right side of the window. Enter the field name *Bogus entry* in the *Edit Field Name* box and press *Add.* Notice the new output field appears in the *Status Board* display as well as in the *Specify Output Record Fields* window. Select *Reset* from the right panel of the *Specify Output* window – the entries for the active source, currently *User Input*, are reset to what they were when the source was selected. In this particular case, *Reset* clears the *Bogus entry* field. Using the *Edit Field Name* box, enter the field name *User Input One* and select *Add.* In a similar manner, add a second field *User Input Two.* Press *Done* to signal that all the fields originating from the user input source have been entered.

Select the *Module Result* item on the *Specify Output Record Fields* window. The default entry, *Result*, corresponding to the module's output port is displayed. For this module, we will rename the default output field to be more descriptive. To do this, use the left mouse button to select the *Result* entry from the right panel – it will be highlighted and also appear in the *Edit Field Name* box. Press the *Delete* button to erase the entry. Type in the new field name *batting average* and press the *Add* button. Select *Done* to complete the addition of output fields originating from the module result source.

At this point, the output record name should be *Batting Avg for Player* and there should be five fields in the output record – *team, player number, User Input One, User Input Two*, and batting average. These five fields appear in the *Status Board* display which also indicates the first two are satisfied by the input pipe *FileInput-1:Game_File* and the second two by the user input source.

FIGURE 3.3

You can select the *FileInput-1:Game_File*, *User Input*, and *Module Result* items from the left side of the *Specify Output Record Fields* window to review the output fields for each source and make changes as desired. The *Reset* button at the bottom of the window will reset the *Output Record Name* and all output fields to their original values when the window first appeared. Once you are comfortable with the window, verify the output record has the name *Batting Avg for Player* and the five fields listed above, then select *Continue* to proceed with the configuration process for the *BinaryMath-1:Divider* module.

3.4.5 Configuring a Binary Math Module (BinaryMath-1:Divider); Specify User Input Values

A window entitled *Specify User Input Values* will be displayed on your screen. We specified earlier in the configuration of this module that no input ports and two output fields, *User Input One* and *User Input Two*, would contain user input values – those choices are reflected in the *Specify User Input Values* window.

FIGURE 3.4

Standard Motif text widgets accept your input data. Integers, floating point numbers, and single characters are valid entries. [12] We will set *User Input One* to the integer "10" and *User*

[12]For input port values, additional checks are made to verify the value entered matches the traits accepted by the input port being filled. Non-scalar input ports cannot be filled by user input.

Input Two to the floating point number "22.22". To enter each value, position your cursor in the appropriate output record field box and type the numeric value followed by a carriage return. The carriage return is not absolutely necessary, but it allows the system to check the value for validity immediately after it is entered. If you do not terminate the entry with a carriage return, the value will not be checked until you select *Continue*. When you have made the entries, press the *Continue* button.

3.4.6 Configuring a Binary Math Module (BinaryMath-1:Divider); Bind Records

The *Bind Records* configuration window will be displayed for the *BinaryMath-1:Divider* module. Using the procedure outlined in tutorial one, select the input pipe *FileInput-1:Game_File* and the *Player Record* on that pipe. Bind the field *Hits* to the input port *LeftOperand* and the field *At-Bats* to the input port *RightOperand*. The two output fields we configured with this input pipe as their source must also be bound to fields in the selected record before we can continue. Using the left mouse button, bind the input field *Team* to the output field *team* and the input field *Player Number* to the output field *player number*. As with the input ports, the output fields will be colored to reflect the bindings made. When all input ports and output fields have been bound, select *Done*.

To demonstrate the *Discard Record* button which appears at the bottom of the Records panel on this screen, select the *Concessions* record with the left mouse button and then select *Toggle Copy Thru*. Bind the field *Inning* to all input ports and output fields and press *Done*. The actions just taken configure the module *BinaryMath-1:Divider* to copy through all records of type *Concessions* and to execute and produce an output record every time a record of type *Concessions* or *Player Record* is seen. Notice that we have configured the module to run using input from two distinct record types. In general, there is no limit to the number of records you may extract from on a given input pipe – the restriction is that every input port and output field bound to that pipe must be filled by fields from the records for which extraction is performed. From our earlier discussion and design of the execution graph for this tutorial, we know this configuration is incorrect for our situation – we should not be concerned with *Concessions* records. To correct the problem, press the *Discard Record* button while *Concessions* is the current record. The small letters 'C' and 'E' which indicated copy through and extract will be removed from the *Concessions* item and the configuration is now correct.

Click on the *Module Results* item in the *Bind Records* window. The output port *Result* is listed in the *Fields* panel and is bound to the renamed output field *batting average*. If the module being configured has more than one output port and you rename the output fields, you must perform the output port to output field binding explicitly at this point in the configuration process. When the system can deduce the correct bindings, they are made automatically. The *Status Board* display is helpful in determining what fields have been bound and the *Continue* button of the *Bind Records* window will remain desensitized until all necessary bindings have been made.

Press *Continue* to move to the next module.

3.4.7 Configuring a History Math Module (HistoryMath-1:Count)

The generic *History Math* module can perform four different operations: *Accumulate*, *Minimum*, *Maximum* and *Count*.

The result produced by the module depends not only on the current operand value, but also on operand values seen since the beginning of the graph's execution. *Accumulate* sets the module result to the sum of all the operands seen by the module. *Minimum* and *Maximum* respectively set the module result to the minimum and maximum of all the operands seen by the module. *Count* sets the module result to the number of operands seen by the module. For this module, *HistoryMath-1:Count*, choose the *Count* operation from the *Historical Values* list and click *OK* to continue.

In the *Specify Output Record Fields* window for the module *HistoryMath-1:Count*, use the default output record name <<*HistoryMath-1:Count*>>. Select the *Module Result* item and change the default output field from *Result* to the more descriptive *batting records seen.* (To change the output field name, first delete the old field and then add the new one). Select *Done* and then *Continue.*

Select the input pipe *BinaryMath-1:Divider* in the *Bind Records* window for *HistoryMath-1:Count* and the single record type *Batting Avg for Player* will be listed. Select that record and bind the field *batting average* to the input port *Operand.* In this tutorial we want to count the number of times a record of the specified type is seen, so the actual field we use to trigger the count event doesn't really matter. For operations other than count, the binding to the operand in the *History Math* module is more critical. The other fields of the input record *Batting Avg for Player* are not used, so press *Done.*

The *Continue* button is activated, indicating that we have completed the required bindings. Also, the *Status Board* window indicates that the output field *batting records seen* has been automatically bound to the output port *Result* as we would expect. Press *Continue.*

3.4.8 Configuring a Synthesize Array Module (SynthesizeArray-1:Replace)

We wish to build up a 2-dimensional array where the row corresponds to the team, the column corresponds to the player number, and the array cell value represents the batting average of the player. We will configure the *SynthesizeArray-1:Replace* module to do this.

As with the *SynthesizeVector* module used in tutorial two, the configuration of the *SynthesizeArray* module begins with an operation selection window. As indicated by the name we chose for this module in planning our execution graph, we will be replacing each cell value with the batting average field from the records produced by our *BinaryMath-1:Divider* module. We are assuming that a "correct" input file will contain a single data record for each player. The default action shown in the operation selection window is *Replace*, so simply select *OK* to continue.

Using the *Bind Input Pipes to Input Ports* window, bind the input pipe *BinaryMath-1:Divider* to the three input ports *Index 0*, *Index 1*, and *Cell Value.* Press *Continue.*

Change the default output record name to *Batting Avg by Team/Player* in the *Specify Output Record Fields* window and select *Continue.* (Our output record will contain a single output field with the default name *Array*).

Use the *Bind Records* screen to select the *Batting Avg for Player* record from the input pipe *BinaryMath-1:Divider.* Bind the field *team* to the input port *Index 0*, the field *player number* to the input port *Index 1*, and the field *batting average* to the input port *Cell Value.* Notice the two user input values appearing in the *Batting Avg for Player* record are not being used. Select *Done* and then *Continue.*

It is now time to configure the synthesize array parameters – you should see a window labeled *SynthesizeArray-1:Replace* displaying the default parameters. For this tutorial, reset the *Initial Array Size (Dimension 0)* to 2 as we expect two teams in the baseball game. Also reset the *Initial Array Size (Dimension 1)* to 9 as we should have at least nine players per team. If an index value is seen that falls outside the range of the initial array size, the array will automatically be increased to accommodate the larger index. The maximum array sizes are primarily safeguards to insure that an unexpectedly large index value does not cause the system to generate a huge output array. Select *Ok* to finish the configuration of the module.

3.4.9 Configuring a Reduction Math Module (ReductionMath-1:Sum)

The generic *Reduction Math* module accepts an array of any dimension [13] and performs operations on the individual cell values in the array, calculating a scalar result value. The operations available are *Summation*, *Product*, *Mean*, *Minimum*, and *Maximum*. As an example, if the input is the two-dimensional array [[10, 30], [1, 2], [3, 1]], and the operation *Product*, the output would be the scalar value 1800.

In this tutorial, we use the module *ReductionMath-1:Sum* to sum the individual batting averages reported in the cells of the array we created in the module *SynthesizeArray-1:Replace*.

Using the *Specify Output Record Fields* window, change the output record name to *Sum of all Batting Avgs*. Select the *Module Result* item and delete the default output field *Result*. Add a new field to replace it named *sum*. Select *Done* and *Continue*.

From input pipe *SynthesizeArray-1:Replace*, bind the field *Array* in the record *Batting Avg by Team/Player* to the input port *Operand*. Select *Done* and *Continue*.

3.4.10 Configuring a Module with 2 Input Pipes (BinaryMath-2:Divider); Specify Module Execution

Our second *Binary Math* module will also be used to perform double-precision division – select that option from the operations menu and click *OK* to continue.

The *BinaryMath-2:Divider* module has two input pipes – prior to this each module has had a single input pipe. The two pipes are shown in both the *Status Board* and *Bind Input Pipes To Input Ports* windows. Use the latter of these windows to bind the input pipe *ReductionMath-1:Sum* to the input port *LeftOperand*, and the input pipe *HistoryMath-1:Count* to the input port *Right Operand*. Press *Continue*.

In the *Specify Output Record Fields* window, change the output record name to *Overall Batting Average* and continue.

Using the *Bind Records* window, bind input pipe *ReductionMath-1:Sum*, record *Sum of all Batting Avgs*, field *sum* to input port *LeftOperand*. Next, bind input pipe *HistoryMath-1:Count*, record <<*HistoryMath-1:Count*>>, field *batting records seen* to input port *Right Operand*. Continue.

A new type of window entitled *Specify Module Execution* will appear on the screen.

FIGURE 3.5

When a module has multiple input pipes bound to multiple input ports it is possible that data will not be available to all input ports at the same time. A module can execute only when it has data on all of it's input ports. Usually, a module takes data from a particular data record only once and then discards it – the module executes in *Synchronous* mode. However, there are cases where this is not the desired behavior. In those cases, you would like the module to execute whenever it has data on all input ports and at least one of the input port data values is new. The data values

[13] A one-dimensional array is a vector.

on the other input ports are reused by the module. This is called *Asynchronous* execution mode. In *Asynchronous* mode, an input port data value is no longer reused once another record arrives on the input pipe bound to that port.

In this tutorial, we know from the nature of our graph that the records will arrive on both pipes at the same rate, so it doesn't matter if we select *Synchronous* or *Asynchronous* execution mode. There are cases where the behavior will be quite different depending on the selection made. [14]

Press *Continue* to close the *Module Execution* window and move to the next module.

3.4.11 Configuring the Module Led-1:Overall_Batting_Avg

From the *Bind Records* window for module *Led-1:Overall_Batting_Avg*, bind input pipe *BinaryMath-2:Divider*, record *Overall Batting Average*, field *Result* to the input port *Input.* Select *Done* and *Continue.*

To configure the led display widget, change the *LED levels* value to 20, and the *Maximum value* to 1.0 as we know all our batting averages will fall between 0 and 1. We will not label the axes. Click on *OK* to close the dialog box.

3.4.12 Configuring the Module Matrix-1:Batting_Avgs

A small dialog box asking the type of data input will be popped up for the module *Matrix-1:Batting_Avgs.* The synthesize array module produces the input for this display and it generates arrays as output. Therefore, use the default selection *Array or Vector* and press *OK* to continue.

Bind input pipe *SynthesizeArray-1:Replace*, record *Batting Avg by Team/Player*, field *Array* to the input port *Input.* Press *Done* and *Continue.*

In the *Matrix FU Config* dialog box, change the *Initial Number of Rows* and the *Initial Number of Columns* both to 1 – the matrix display will expand as needed to match the size of the incoming array. Set the *Maximum value* to 1.0, add the *Vertical axis label* "Team" and the*Horizontal axis label* "Player". Click *OK* to close the dialog box.

3.5 Saving the Configuration and Running the Execution Graph

Reposition the matrix and led displays if necessary, and save the execution graph and module configuration information by selecting *Save Configuration* from the *File* pull-down menu.

We are now ready to run the configured graph and view our data. You can use the *Run Panel* to execute the graph at a controlled rate, or select the *Run* option to process the input file as fast as possible. If you do select *Run*, you may stop the execution at any time by pressing *Stop.* Our

[14] Consider a baseball statistics file having a different format than those we have been working with. In that file, a *Hit* record occurs for each hit and an *AtBat* record occurs for each at-bat. A running batting average could be computed by dividing the number of *Hit* records seen by the number of *AtBat* records. We could use two history math modules configured with the count operation to keep the running count of each type of record. The output of those modules, say *HitCnt* and *AtBatCnt*, could be directed into a binary math divider module. A *HitCnt* record would be produced for each *Hit* record in the file, and likewise an *AtBatCnt* would be produced for each *AtBat* record. The *HitCnt* and *AtBatCnt* records would not arrive at the binary math divider module at the same rate. However, we would want the divider module to report the most recent average – it should produce an output whenever a new *HitCnt* **or** *AtBatCnt* record is seen. It should run in *Asynchronous* mode. If we chose *Synchronous* mode in this example, our divider module would only produce an output when a new *HitCnt* and a new *AtBatCnt* record were available. This would mean the batting average would always be reported as "1.0"; any *AtBat* records for which no *Hit* occurred would never be accepted by the divider module – they would remain unprocessed in the input pipe.

input file for this tutorial is quite short, so you may not have time to select *Stop* before the end of the data is reached. However, for large data files, the *Run/Stop* combination is quite useful.

Recall from our planned execution graph in Section 3.1 that the display *Matrix-1:Batting_Avgs* will be showing the batting averages for all players, with the row corresponding to the team and the column corresponding to the player number. The color of the matrix cells will vary depending on the cell value. High values are represented by reds, middle values by greens, and low values by blues. [15] Twenty-eight unique colors are divided evenly over the range of values to be displayed – the same color will be used to represent several different cell values when the number of actual values is greater than twenty-eight. If you need to see an actual data value, click the left mouse button on the cell of interest and the exact value will be displayed. The display *Led-1:Overall_Batting_Avg* will show the average hitting performance for all players whose data records have been processed by the system.

Run the graph until the system status changes to *Execution complete.*

FIGURE 3.6

Notice that the matrix display contains several cells which were never updated – we cannot tell from the display if those cells represent players who had no hits, or if they correspond to team/player combinations that did not play in the game (no record in the input file). We'll try to answer that question in tutorial four.

3.6 Ending the Pablo Session

To end the Pablo session, pull down the *File* menu and select *Delete Graph* to prepare for a new session, or *Quit* to exit the Pablo environment.

[15] If you think of the RGB color specification, the range of colors in the Pablo widgets correspond to the Red, Green, and Blue components respectively carrying more weight as the data values the represent go from maximum to minimum.

4 Tutorial 4

In the fourth tutorial we build on our saved configuration from tutorial three, adding two new modules to calculate and display which players participated in the game.

4.1 Building on a Saved Configuration

To begin, select *Load Configuration* from the *File* pull-down menu. You should see a standard Motif file selection dialog labeled *Load Configuration from Directory* – use it to specify the directory where you saved your tutorial three configuration. We will again be using the input file *Game2.ascii*, the default value in the dialog *Select Input File for FileInput-1:Game_File*. Your saved execution graph and the matrix, led, and input file bytes processed displays should be shown.

Pull down the *Module* window and select *Add...*. Add a *SynthesizeArrayElement* module, naming it *SynthesizeArrayElement-1:Count*. Also add a display module of type *Bubble* named *Bubble-1:Active_Players*. Dismiss the *Pablo Module Creation* window. Reposition the *Bubble-1:Active_Players* display and move the new module bubbles in the main Pablo window to a place on the screen where they do not overlap any of the original module bubbles. A sample layout is shown:

FIGURE 4.1

Select *Connect...* from the *Module* pull-down menu. Make connections from *BinaryMath-1:Divider* to *SynthesizeArrayElement-1:Count* and from *SynthesizeArrayElement-1:Count* to *Bubble-1:Active_Players*. Exit the module connection phase by clicking the right mouse button on a module bubble. The system status should be *Idle*.

4.2 Configuring Individual Modules

We have added two new modules to our saved configuration from tutorial three. In tutorial two, we added to a saved configuration and then selected *Configure/Graph* to configure all modules in the execution graph – both old and new. The graph in that case was quite small, so verifying the configuration for the original modules wasn't overly time-consuming. In our current graph we have two new modules which have not been configured and eight old modules for which configuration is complete – we will configure only the new modules to save time.

Pull down the *Configure* menu and select the *Module...* option. The system status line should change to *Waiting for node selection or exit*. When configuring individual modules, you must begin with the new module closest to the root of the execution graph and work your way down to the leaf nodes. In our case, that means we must first configure *SynthesizeArrayElement-1:Count* and then *Bubble-1:Active_Players*. It is important to follow this configuration order and also to remember to configure all new modules you have added.

4.2.1 Configuring the Module SynthesizeArrayElement-1:Count

The *SynthesizeArray* and *SynthesizeArrayElement* modules are very similar. Both take three inputs (index0, index1, cell value) and build up two dimensional arrays of data based on the action and operation selected at configuration time. The difference lies in the output generated by the modules – *SynthesizeArray* outputs the entire array it has built each time it executes, while *SynthesizeArrayElement* only outputs the indices and value for the one cell in the array that changed as a result of the execution. [16] Depending on the array size, replacing a *SynthesizeArray* module with a *SynthesizeArrayElement* module can result in considerable speed-up of the graph execution.

Position the mouse on the *SynthesizeArrayElement-1:Count* module bubble and click the left mouse button. You should see an operation selection window for the module – choose the *Update* action and the *Count* operation; press *OK* to close the window. You will see *Status Board* and *Bind InputPipes to InputPorts* windows. Notice in the *Status Board* display the three output ports for this module – recall the *SynthesizeArray* module had a single output port.

Use the *Bind Input Pipes to InputPorts* window to bind the input pipe *BinaryMath-1:Divider* to all three input ports. Continue. Change the output record name to *Player Records by Team/Player* and close the *Specify Output Record Fields* window. From the input pipe *BinaryMath-1:Divider* record *Batting Avg for Player*, bind *team* to *Index0*, *player number* to *Index1*, and *batting average* to *Cell Value*. Press *Done* and *Continue*. Finally, change the module parameters for *SynthesizeArrayElement-1:Count*, making '1' the *Initial Array Size* for both dimensions.

The module as configured will generate array elements with row indices corresponding to teams, column indices corresponding to players, and cell values representing the number of batting average records seen for particular team/player combinations. This data will tell us if no team/player record was seen, or if more than one record appears for a given team/player combination (flagging an error in the data file).

Close the parameter configuration window by clicking on the *OK* button – the system status should change to *Idle*.

4.2.2 Configuring the Module Bubble-1:Active_Players

Pull down the *Configure* menu and again select the *Module...* option. The system status line should change to *Waiting for node selection or exit* – click on the *Bubble-1:Active_Players* module with the left mouse button.

Choose the *Array Indices and Element* option from the data input type dialog box and press *OK*. Bind the input pipe *SynthesizeArrayElement-1:Count* to the three input ports and continue. From the input pipe record *Player Records by Team/Player*, bind *Subscript 0* to *Index 0*, *Subscript 1* to *Index 1*, and *Matrix Element* to *Array Element*. Press *Done* and *Continue*. Change the initial number of rows and columns to '1', and the maximum value to '4'. We use '4' as the maximum value instead of '1' because the bubble modules display the cell values using both color and bubble size. If we set the maximum to '1', the bubbles for the cells with that value will be quite large – choosing the maximum of '4' makes the display somewhat clearer. Change the vertical label to "team" and the horizontal label to "player number". Click *OK* on the parameter screen to close the window.

4.3 Executing the Configured Graph

At this point, we have finished configuring the expanded execution graph in this example. All configuration screens should have disappeared and the system status should show *Idle*. Before

[16] SynthesizeArrayElement maintains the complete array internally.

running the graph, save your new execution graph and configuration selections in a directory called *Tutorial4*.

Run the configured graph – the final display should be similar to the one shown:

FIGURE 4.2

This display tells us the first team fielded eleven players, while the second fielded thirteen. Players 6, 7, 8, and 10 on the first team, and players 2, 8, 10, and 12 on the second team all played but did not get any hits. (Player numbers begin at 0.) The overall batting average was .243353 and the person with the best batting average was player number 1 on the second team – his average was .666667. Try using callbacks to verify this information on your own display.

4.4 Rerunning a Graph; Reconfiguring Module Parameters

Now that we know the highest batting average was .666667, we can use that information to reset our maximum values for the matrix and led displays and rerun the execution graph. In the current version of Pablo, it isn't possible to rerun a graph directly, but we can choose *Delete Graph...* from the *File* pull-down menu to complete the current session, and then select *Load Configuration* and reload our configuration for Tutorial 4.

We demonstrated earlier in this tutorial how to configure individual modules – it is also possible to reconfigure the parameters of a module after the initial configuration phase. Pull down the *Configure* menu and select the *Module Parameters...* item. The status line will change to *Waiting for node selection or exit.* At this point, you can position your mouse over a module bubble and click the right button to cancel the choice or click the left button to select the module you wish to reconfigure. Position your mouse over the *Matrix-1:Batting_Avgs* module and click the left button to select it. You will be shown the configured data input type; press *OK*. In the screen *Matrix FU Config:Matrix-1:Batting_Avgs*, change the maximum value to '.7' and click *OK*. The matrix display should be updated to show the new maximum. Follow a similar procedure to change the maximum on the led display.

Execute the graph with the new parameters and notice the differences in the led and matrix displays. The actual data values have not changed, but each display represents a smaller range of values therefore a greater number of the available colors will be seen.

4.5 Ending the Pablo Session

To end the Pablo session, pull down the *File* menu and select *Delete Graph* to prepare for a new session, or *Quit* to exit the Pablo environment.

5 Tutorial 5

In this tutorial we use the Pablo visualization system to examine procedure entry and exit events collected by the Pablo instrumentation system when a parallel PDE solver was executed on an eight-node iPSC/860 machine. Unlike the first four tutorials, we will not present a detailed explanation of the configuration process in this example. Instead, we will outline the execution graph used the the example and interpret the displayed results. The reader is directed to Appendix E for the exact configuration specifications, and to Appendix G for a discussion of the PDE algorithm.

5.1 The Execution Graph

Consider the annotated figure representing our execution graph:

FIGURE 5.1

The *FileInput-1:Pablo_Trace* module at the top of the graph will read our performance trace data file.

Looking first at the left-branch of the execution graph, the module *SynthesizeVector-1:Sum* will operate on procedure exit trace events. It will use the procedure number as an index into the vector it is building, and accumulate in the vector elements the total time spent in each procedure. The display module *BarGraph-1:Cumulative_time* will show the cumulative time spent in each procedure throughout the course of the program execution.

Following the middle branch of the execution graph, the *SynthesizeArray-1:Count* module will perform field extraction on procedure entry trace event records. The rows of the array correspond to processors and the columns correspond to procedures. Each time a procedure entry trace event is seen by the module, the corresponding array element will be incremented. The display module *Matrix-1:Procedure_count* will show the number of times each procedure is called on each processor.

The right-branch of the execution graph contains two modules – *Scale-1:toNanoSecs* and *Chart-1:Procedure_duration*. The module *Scale-1:toNanoSecs* will perform field extractions from procedure exit trace events and scale the exclusive duration fields given in seconds to nanoseconds for input into the *Chart-1:Procedure_duration* module. The Chart display module will show the duration of time spent in each procedure as reported in the procedure exit trace event records—there is no attempt to correlate the durations to a given processor or procedure. The display simply shows the variance between times spent in different procedures throughout the trace.

From this execution graph we should gain insight into which procedures account for the majority of the execution time (optimize those first), and also see if the various processors are executing approximately the same number of procedure calls. [17]

5.2 Entering and Running the Graph

Begin with an empty Pablo window and follow the instructions in the previous tutorials to enter the execution graph shown in FIGURE 5.1. Using the configuration tables in Appendix E as a guide, configure the system. [18]

[17] You might want to modify or add to this execution graph to show the breakdown of time spent in each procedure on each processor instead of just reporting the number of times each procedure was called on each processor.

[18] The minimum and maximum values for the various modules were arrived upon using the "FileStats" program provided with the Pablo distribution and trial runs of the execution graph. When selecting minimum and maximum

Save your configured graph, select the *Run Panel* from the Run pull-down menu, and press the *Run* button to begin execution.

The input file in this example is quite large (refer to the *FileInput-1:Pablo_Trace* bargraph display)—you can press the *Stop* button on the run panel to stop execution at any point and use the display callbacks to view the data being displayed in more detail.

5.3 Interpreting the Results

Recall from the introduction that procedure entry and procedure exit events were collected for five different procedures. The procedure names and the numeric indices assigned to each by the instrumentation system are: *InitGrid*(12), *IterateEdge*(7), *Converged*(0), *Iterate*(8), and *Convergence*(6). Also remember that the trace file was collected on an eight node iPSC/860 machine. Consider the displays as the graph executes:

FIGURE 5.2

BarGraph-1:Cumulative_time shows the cumulative time spent in each of the five procedures. It is clear that procedures 6 *Convergence*, and 8 *Iterate*, account for the majority of the execution time. Optimization efforts should be directed first toward those procedures. The display *Matrix-1:Procedure_count* shows that the procedure calls are distributed evenly across all processors. Procedures 0, 6, 7, and 8 all execute the same number of times on all processors, while procedure 12 executed only one time on each. Invoke the display callback on the last element of *BarGraph-1:Cumulative_time* to convince yourself that procedure 12, *InitGrid*, did indeed accumulate some time.

The strip-chart display, *Chart-1:Procedure_duration*, shows the variance in procedure durations.

configuration parameters for your own execution graphs, the *Configure Module Parameters* menu item outlined in tutorial four can be used to adjust inappropriate values as the graph is executing.

6 Tutorial 6

In this tutorial we use the Pablo visualization system to examine message passing events collected by the Pablo instrumentation system when a parallel PDE solver was executed on an eight-node iPSC/860 machine. As in tutorial five, we will not present a detailed explanation of the configuration process. Instead, we will outline the execution graph used the the example and interpret the displayed results. The reader is directed to Appendix F for the exact configuration specifications, and to Appendix G for a discussion of the PDE algorithm.

6.1 The Execution Graph

Consider the annotated figure representing our execution graph:

FIGURE 6.1

The *FileInput-1:Pablo_Trace* module at the top of the graph will read our performance trace data file. The module *FilterRecords-1:Filter_Msgs* will not actually perform any data manipulation or display functions. Rather, it will be used to discard all records that are not related to message passing begin events. The message passing begin events—*Blocking Send Begin*, *Blocking Receive Begin*, *Nonblocking Send Begin*, and *Nonblocking Receive Begin*—will be copied through the module untouched. This *FilterRecords* module could be left out of the graph; however, it will simplify the configuration process for the modules just below it, and will speed up processing somewhat by limiting the number of records passed to the downstream nodes.

The execution graph contains three types of display modules: kiviats, contour plots, and bubble displays. There are two displays of each type—one will report information for blocking (synchronous) messages, and the other for non-blocking (asynchronous) messages.

The kiviat modules, *Kiviat:Cnt_Brecv* and *Kiviat:Cnt_NBrecv*, will show the number of receive events indexed by message type. *Contour:TotalSize_Bsend* and *Contour:TotalSize_NBsend* will be contour plot displays showing the total number of bytes sent from any processor (message source) to any other processor (message destination). The bubble displays, *Bubble:Cnt_Bsend* and *Bubble:Cnt_NBsend*, will show how many messages were sent between any pair of processors. The various synthesis modules will build up the data structures used by the display modules, extracting fields from the appropriate event records and performing *Addition* or *Count* operations as indicated in FIGURE 6.1.

From this execution graph we hope to better understand the message passing patterns of the parallel PDE code. Perhaps bottlenecks will be uncovered or errant messages detected.

6.2 Running the Graph

Follow the instructions in tutorials one–four to enter the execution graph shown in FIGURE 6.1. Using the configuration tables in Appendix F to guide you, configure the system. Save your configured graph, pull down the run panel, and select the *Run* option to begin execution. The input file in this example is quite large—you can press the *Stop* button on the run panel to stop execution at any point and view the data being displayed in more detail using the display callbacks.

6.3 Interpreting the Results

After your graph has been executing for awhile, the display should be similar to the one shown:

FIGURE 6.2

From the short discussion in the Introduction, we know that the PDE code has been instrumented with eight distinct message types: *maximum number of iterations*, *convergence check frequency*, *grid dimension*, *partition width*, *partition data*, *send boundary points "up"*, *send boundary points "down"*, and *convergence value broadcast.* In addition to these fixed message types, there are additional types generated for the condensation messages used by the global convergence algorithm (refer to Appendix G, Figure 3). The types assigned to these messages equal eight plus the number of the node sending the message. For our eight node trace, these will correspond to message types 8 through 15.

If we look first at the kiviat displays, we see that there have been no non-blocking receive events. The program must do only synchronous (blocking) receives. Using the callbacks on the display *Kiviat:Cnt_Brecv*, you will see that message types 5–*partition data*, 6–*send boundary points "up"*, and 7–*send boundary points "down"* occur much more frequently than message types 0–4. There are no occurrences of message type 8 – node zero does not send any condensation messages with the algorithm used. Message types 9-15 occur an equal number of times – nodes one through seven each send the same number of messages in the global convergence condensation algorithm.

Both the bubble and contour displays show the message distribution pattern is different for blocking sends than for non-blocking sends. While the number of blocking and non-blocking messages sent are the same for this program (use callbacks on the bubble displays), the number of bytes sent is much larger for non-blocking sends. [19] Refer to Appendix G for a discussion of the communication algorithms used by the PDE solver—they are clearly evident in the bubble and contour displays of this tutorial. Our displays tell us the message passing load is fairly well distributed, and we do not see any errant messages (such as a processor sending a message to itself).

The graph in this example could be used to analyze trace data from a variety of programs executed on distributed-memory parallel processors. Depending on the underlying algorithms, the communication patterns displayed would vary, but as long as the message events recorded were identical, the configured Pablo execution graph could be reused.

[19] You might want to add modules to calculate and display the average message size for each message type. Also, our graph does not show if a given message type is always sent in the same manner—synchronous versus asynchronous; you could add modules to display that information.

A Figures and Configuration Parameters for Tutorial 1

A.1 Figures 1.1 through 1.9

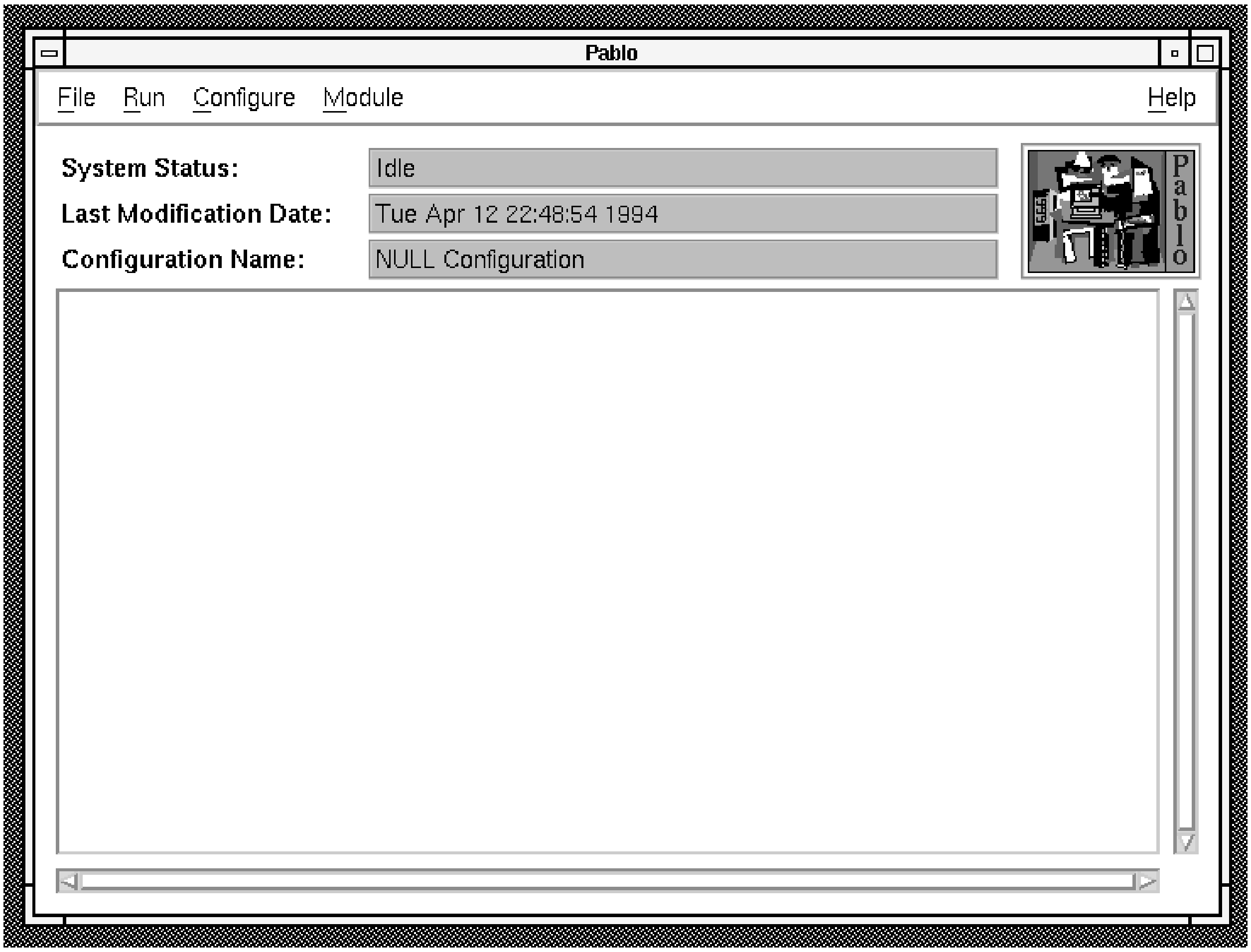


FIGURE 1.1

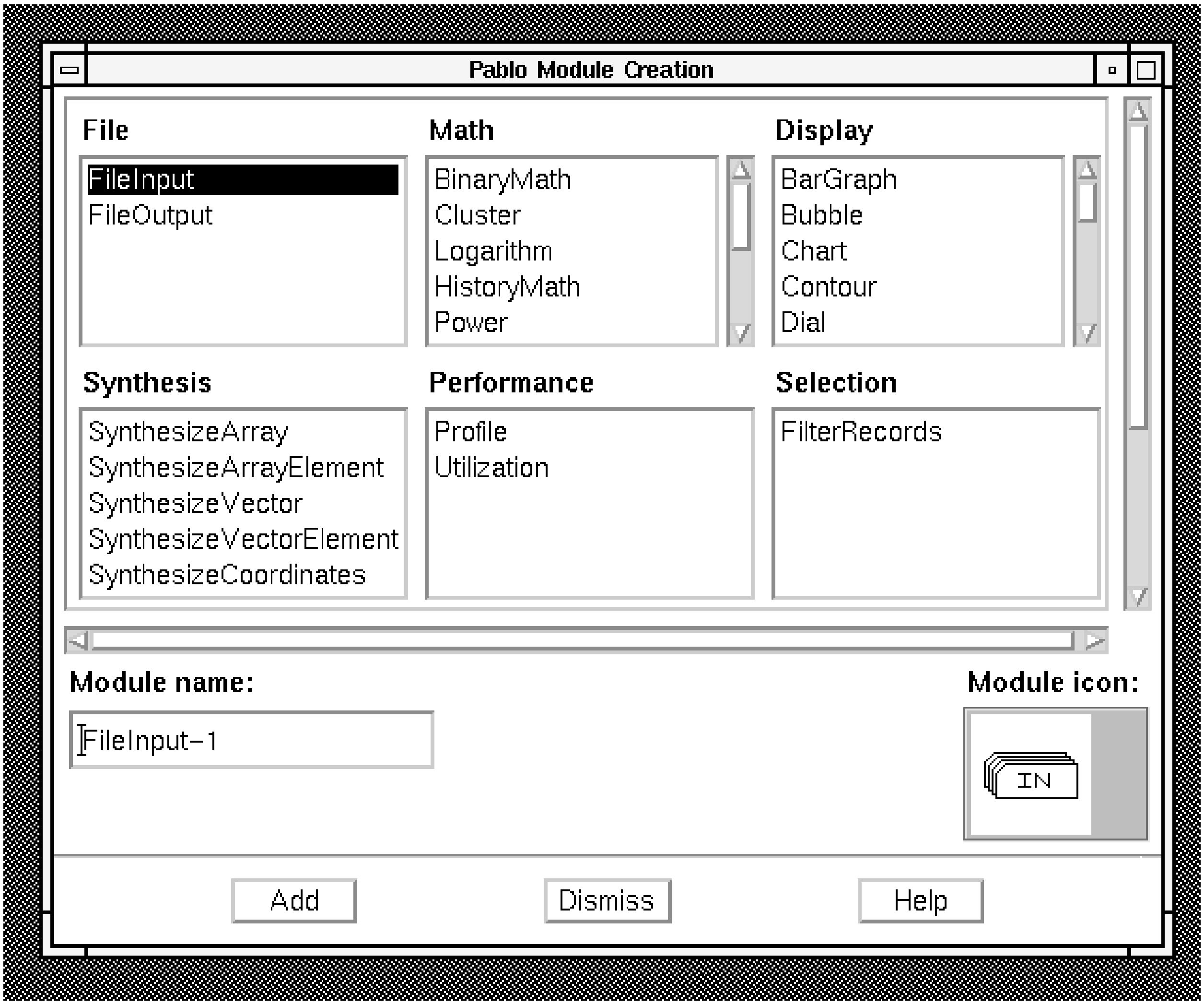

Pablo Module Creation
File
FileInput
FileOutput
Math
BinaryMath
Cluster
Logarithm
HistoryMath
Power
Display
BarGraph
Bubble
Chart
Contour
Dial
Synthesis
SynthesizeArray
SynthesizeArrayElement
SynthesizeVector
SynthesizeVectorElement
SynthesizeCoordinates
Performance
Profile
Utilization
Selection
FilterRecords
Module name:
FileInput-1
Module icon:
IN
Add
Dismiss
Help

FIGURE 1.2

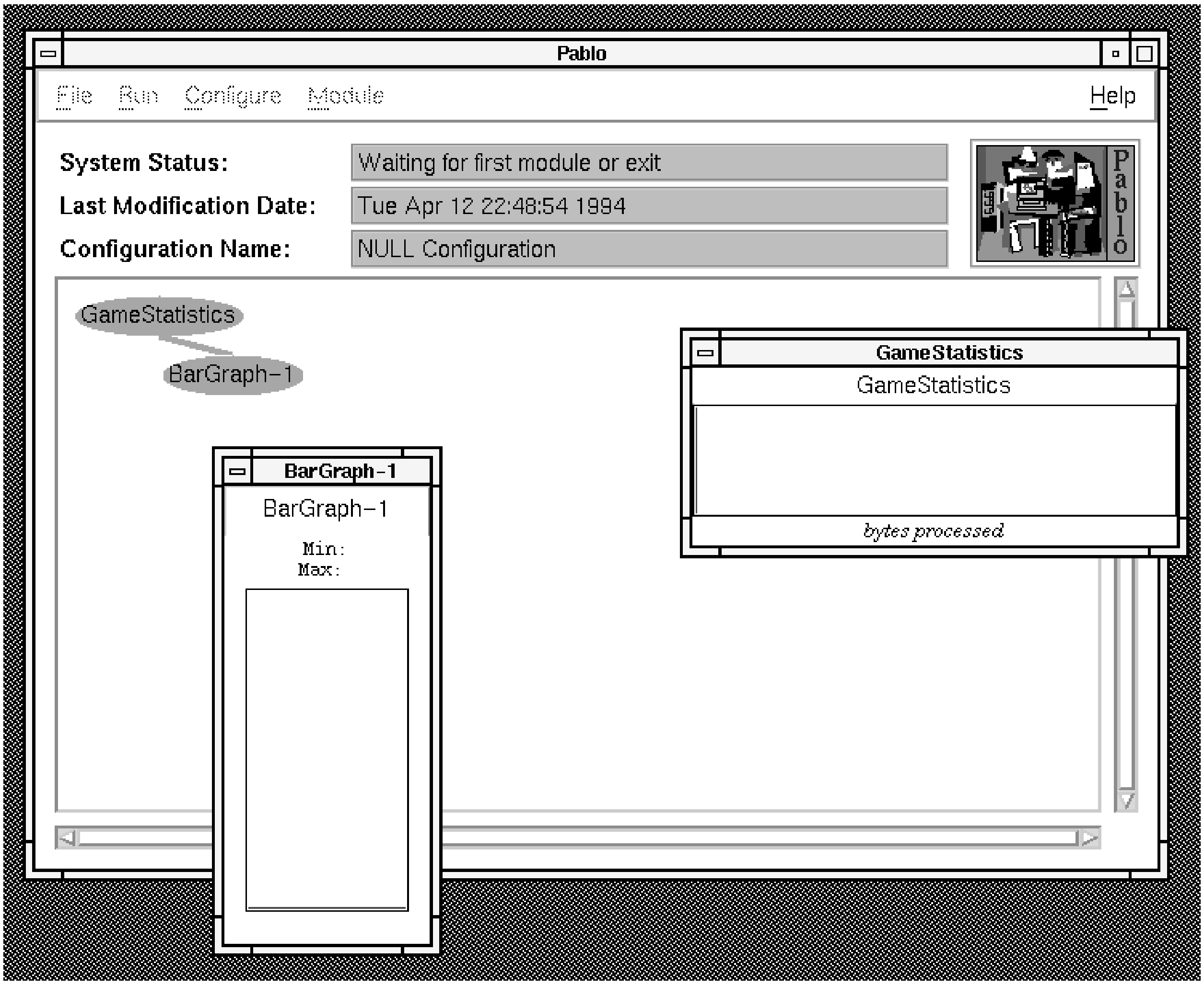
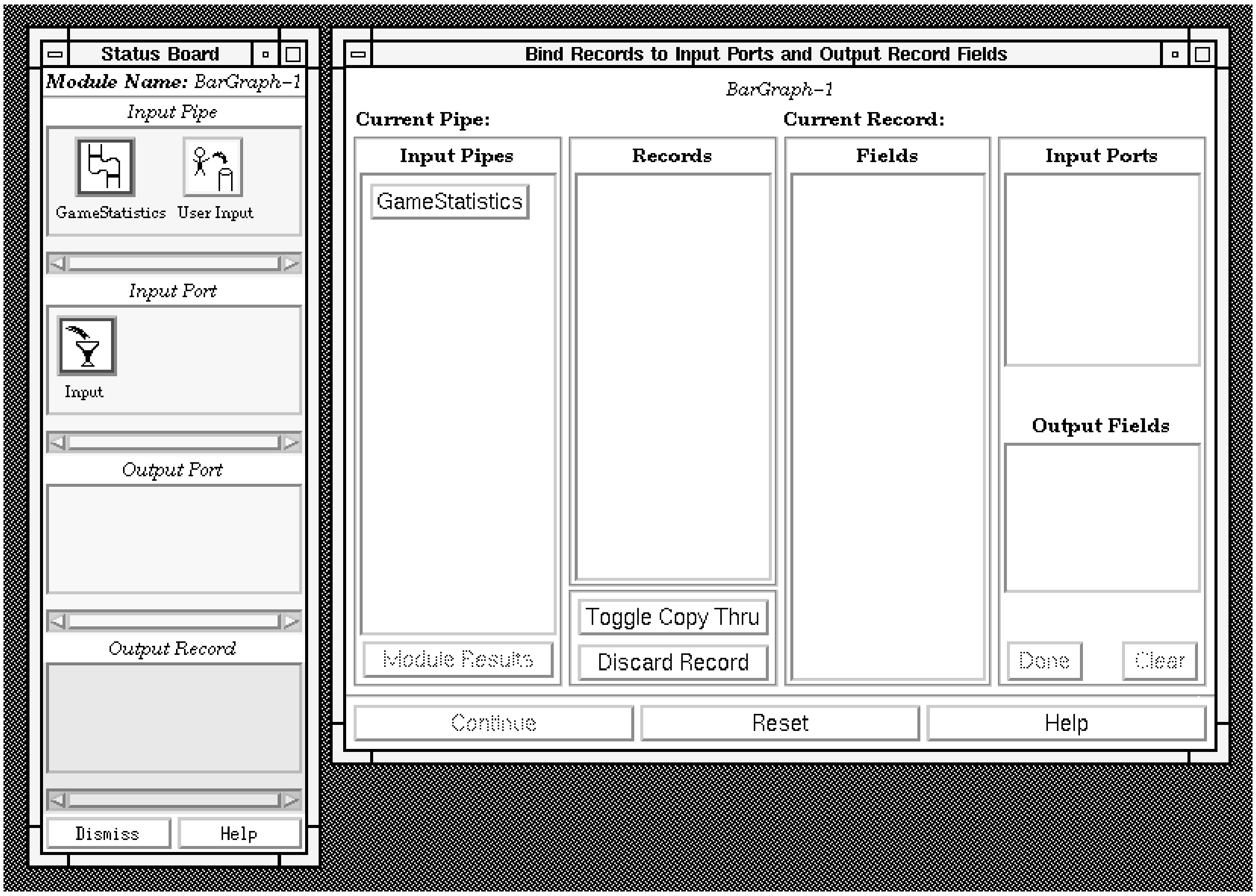

Pablo
File
Run
Configure
Module
Help
System Status:
Waiting for first module or exit
Last Modification Date:
Tue Apr 12 22:48:54 1994
Configuration Name:
NULL Configuration
Pablo
GameStatistics
BarGraph−1
GameStatistics
GameStatistics
bytes processed
BarGraph−1
BarGraph−1
Min:
Max:

FIGURE 1.3



Status Board
Module Name: BarGraph-1
Input Pipe
GameStatistics
User Input
Input Port
Input
Output Port
Output Record
Dismiss
Help
Bind Records to Input Ports and Output Record Fields
BarGraph-1
Current Pipe:
Current Record:
Input Pipes
GameStatistics
Module Results
Records
Toggle Copy Thru
Discard Record
Fields
Input Ports
Output Fields
Done
Clear
Continue
Reset
Help

FIGURE 1.4

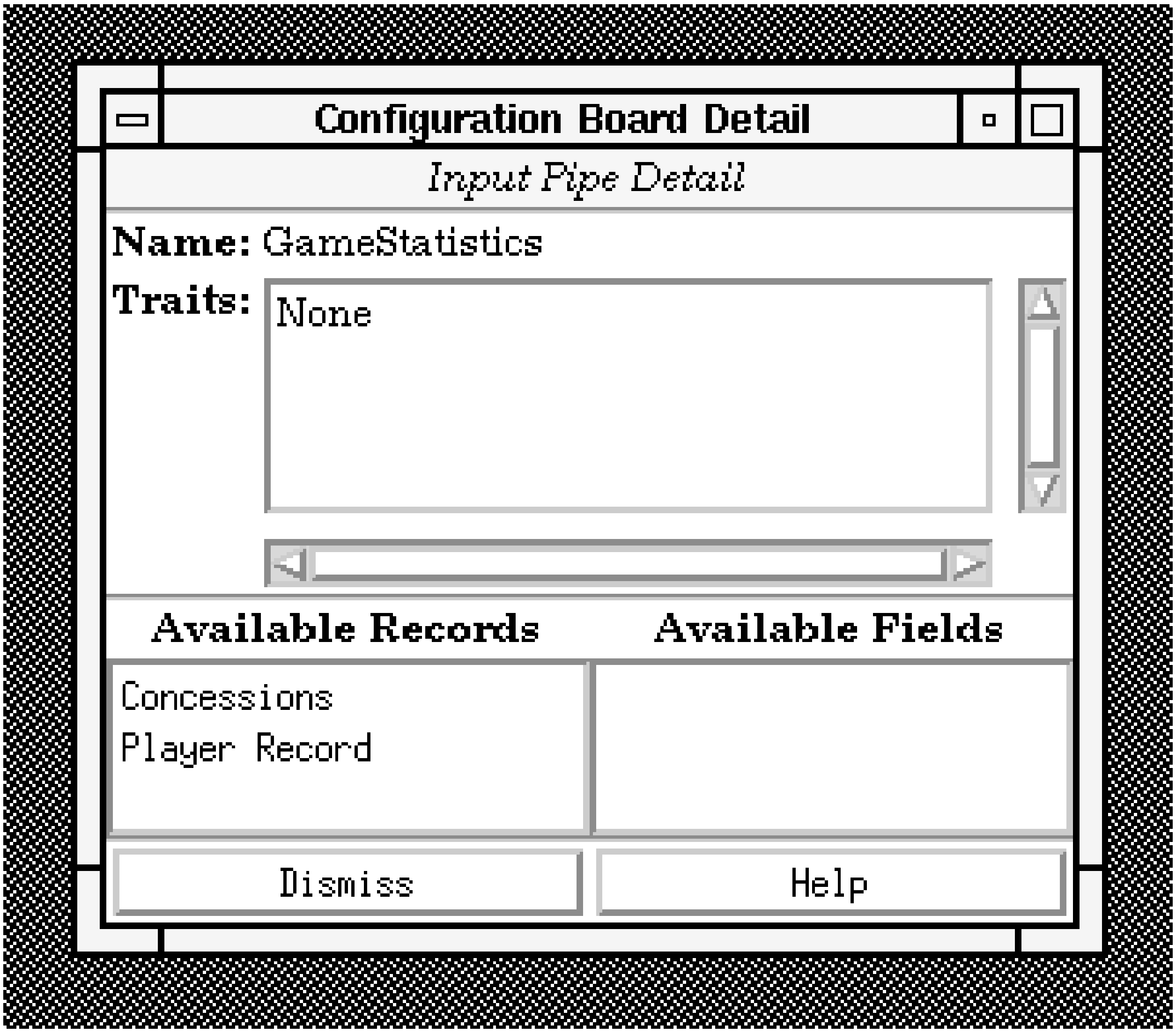
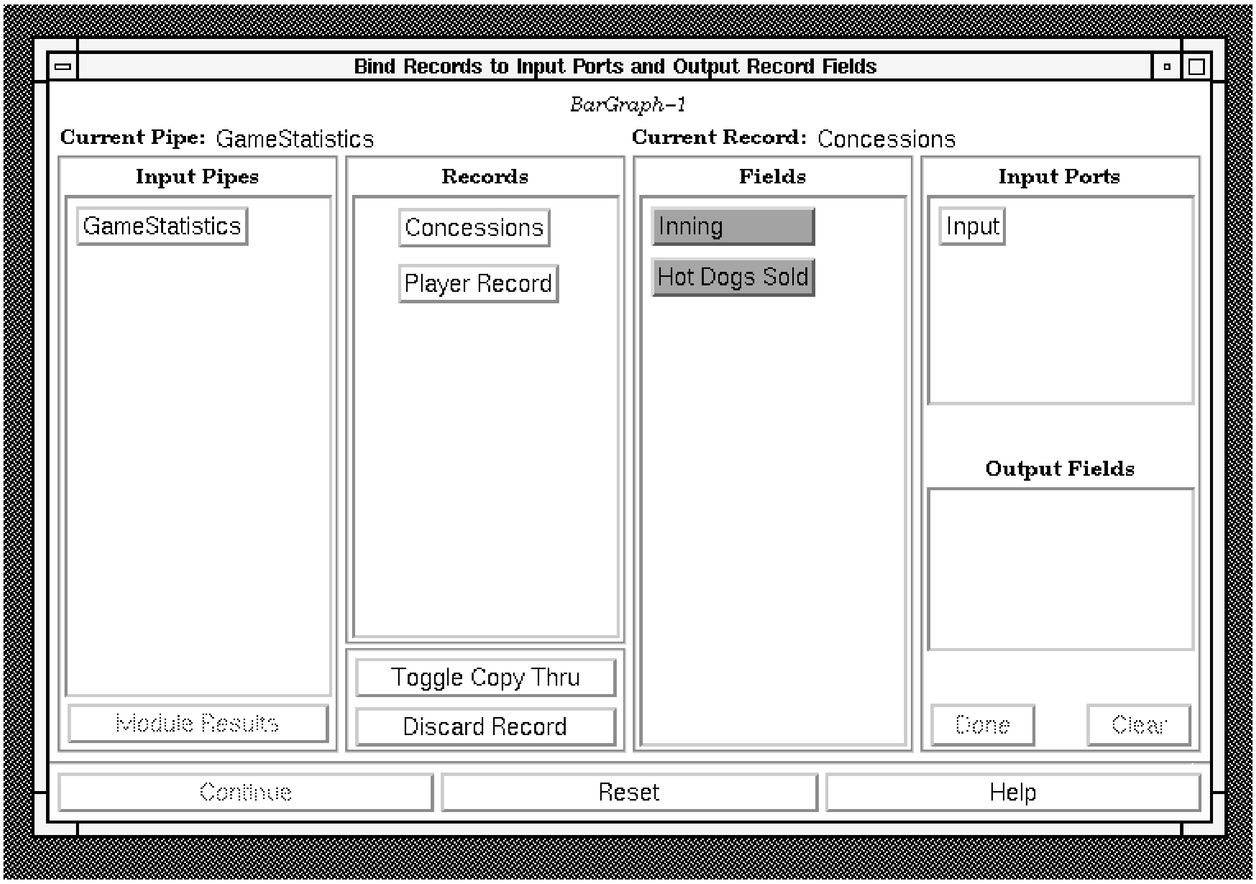

Configuration Board Detail
Input Pipe Detail
Name: GameStatistics
Traits:
None
Available Records
Available Fields
Concessions
Player Record
Dismiss
Help

FIGURE 1.5



Bind Records to Input Ports and Output Record Fields
BarGraph-1
Current Pipe: GameStatistics
Current Record: Concessions
Input Pipes
GameStatistics
Module Results
Records
Concessions
Player Record
Toggle Copy Thru
Discard Record
Fields
Inning
Hot Dogs Sold
Input Ports
Input
Output Fields
Done
Clear
Continue
Reset
Help

FIGURE 1.6

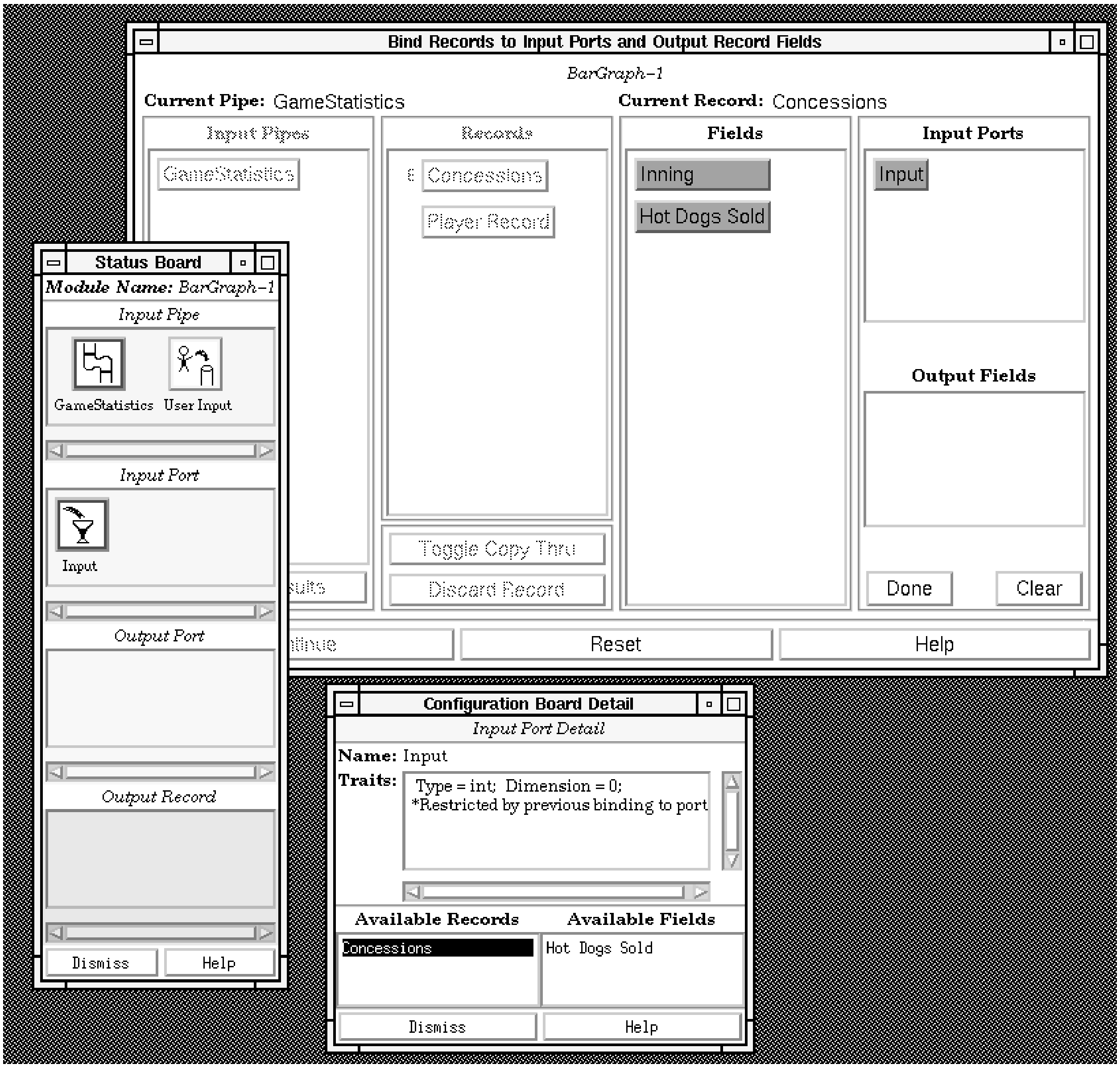

Bind Records to Input Ports and Output Record Fields
BarGraph-1
Current Pipe: GameStatistics
Current Record: Concessions
Input Pipes
GameStatistics
Records
Concessions
Player Record
Fields
Inning
Hot Dogs Sold
Input Ports
Input
Output Fields
Toggle Copy Thru
Discard Record
Done
Clear
Reset
Help
Status Board
Module Name: BarGraph-1
Input Pipe
GameStatistics
User Input
Input Port
Input
Output Port
Output Record
Dismiss
Help
Configuration Board Detail
Input Port Detail
Name: Input
Traits:
Type = int; Dimension = 0;
*Restricted by previous binding to port
Available Records
Available Fields
Concessions
Hot Dogs Sold
Dismiss
Help

FIGURE 1.7

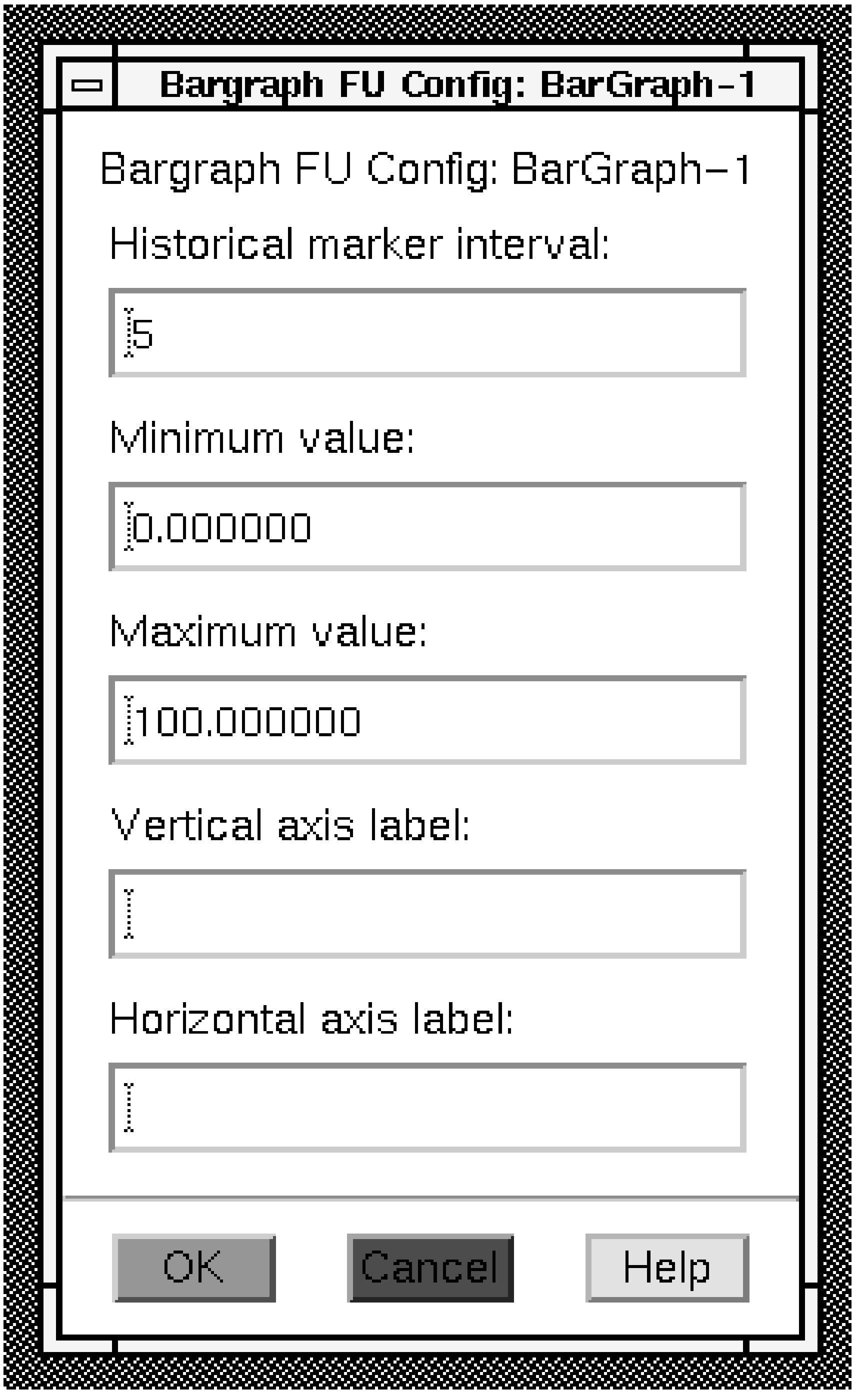

Bargraph FU Config: BarGraph-1
Bargraph FU Config: BarGraph−1
Historical marker interval:
5
Minimum value:
0.000000
Maximum value:
100.000000
Vertical axis label:
Horizontal axis label:
OK
Cancel
Help

FIGURE 1.8

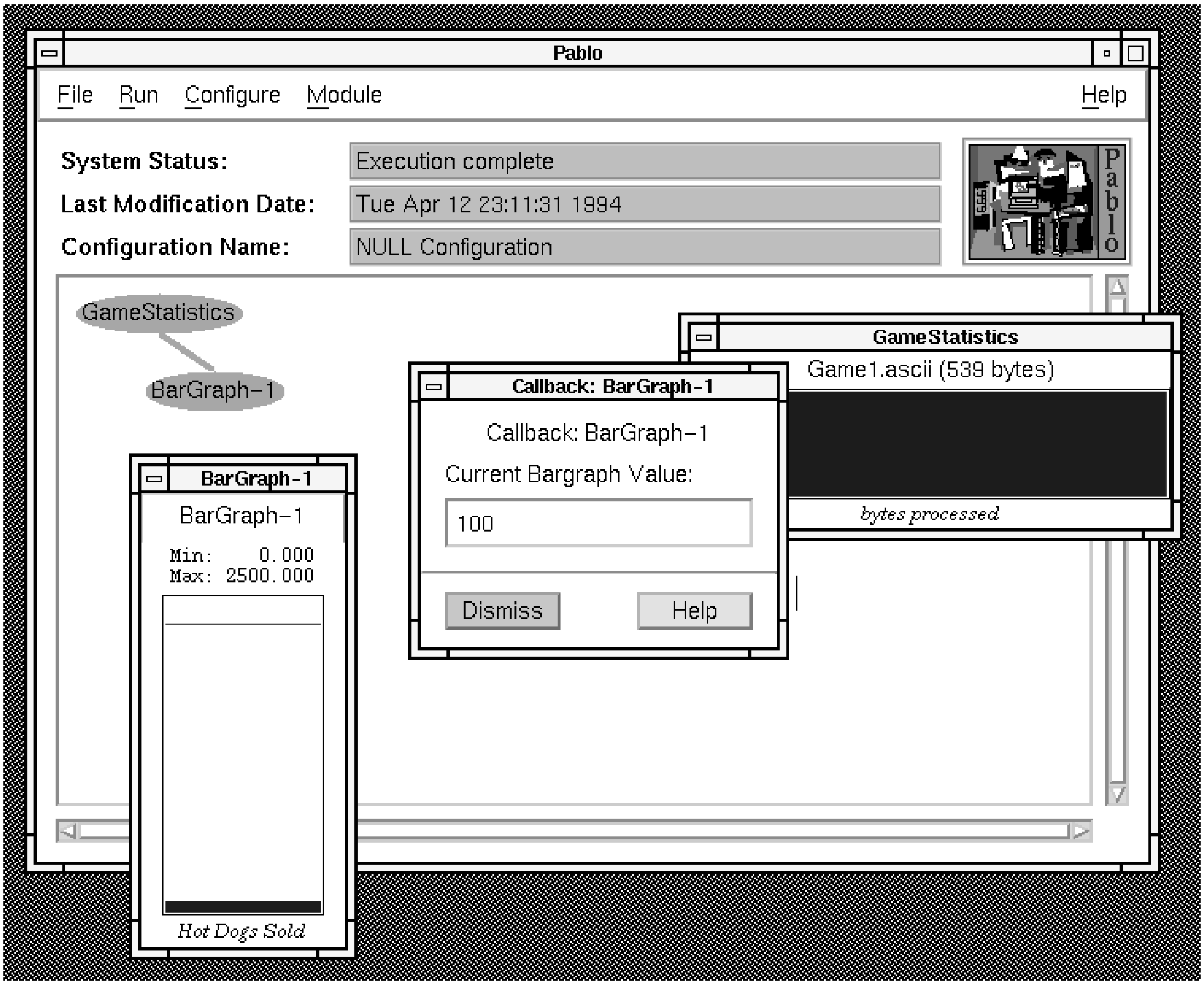

Pablo
File
Run
Configure
Module
Help
System Status:
Execution complete
Last Modification Date:
Tue Apr 12 23:11:31 1994
Configuration Name:
NULL Configuration
Pablo
GameStatistics
BarGraph-1
GameStatistics
Game1.ascii (539 bytes)
bytes processed
Callback: BarGraph-1
Callback: BarGraph-1
Current Bargraph Value:
100
Dismiss
Help
BarGraph-1
BarGraph-1
Min: 0.000
Max: 2500.000
Hot Dogs Sold

FIGURE 1.9

A.2 Configuration Parameters for Individual Modules

A.2.1 GameStatistics

Module Name:	GameStatistics
Description:	FileInput module.
Selected Input File:	Game1.ascii

A.2.2 BarGraph-1

Module Name:	BarGraph-1
Description:	BarGraph display module. Shows number of hot dogs sold for an inning.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: GameStatistics

Action	Record	Field	Bound To
Extract	Concessions	Hot Dogs Sold	Input
Discard	Player Record		

Parameter Configuration

Parameter	Value
Historical marker interval	5
Minimum value	0.00
Maximum value	2500.00
Vertical axis label	
Horizontal axis label	Hot Dogs Sold

B Figures and Configuration Parameters for Tutorial 2

B.1 Figures 2.1 through 2.12

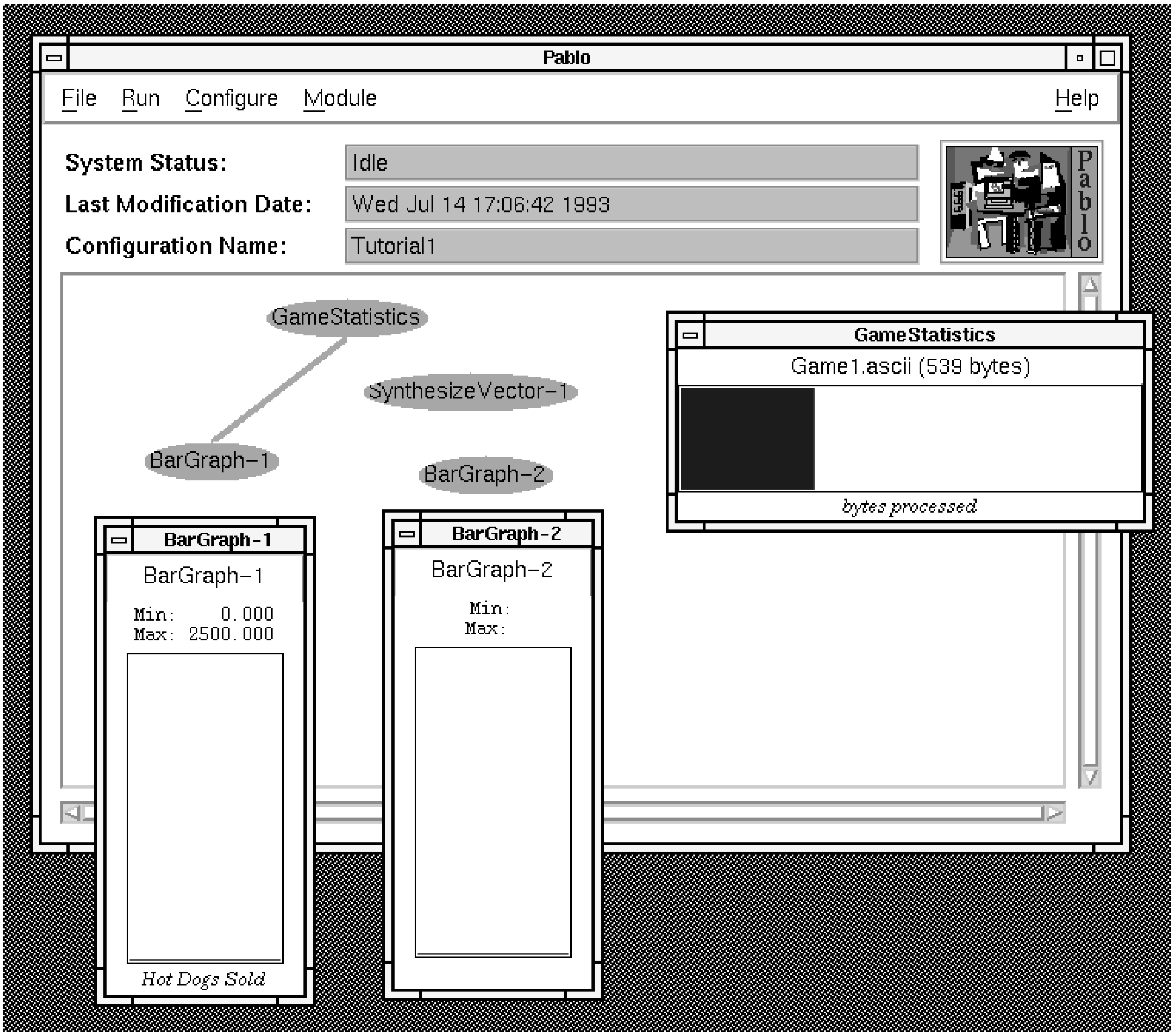


FIGURE 2.1

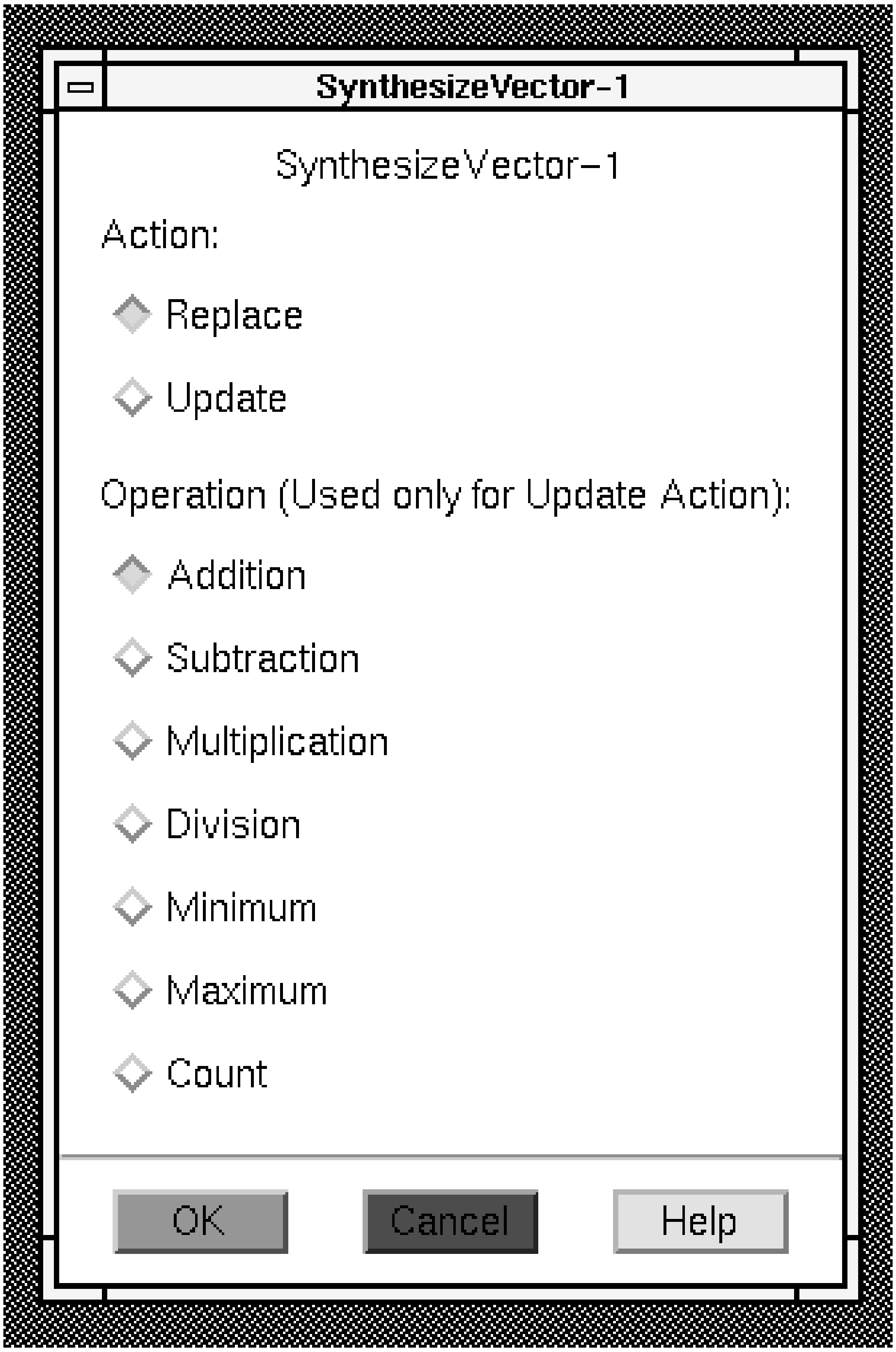

SynthesizeVector-1
SynthesizeVector-1
Action:
Replace
Update
Operation (Used only for Update Action):
Addition
Subtraction
Multiplication
Division
Minimum
Maximum
Count
OK
Cancel
Help

FIGURE 2.2

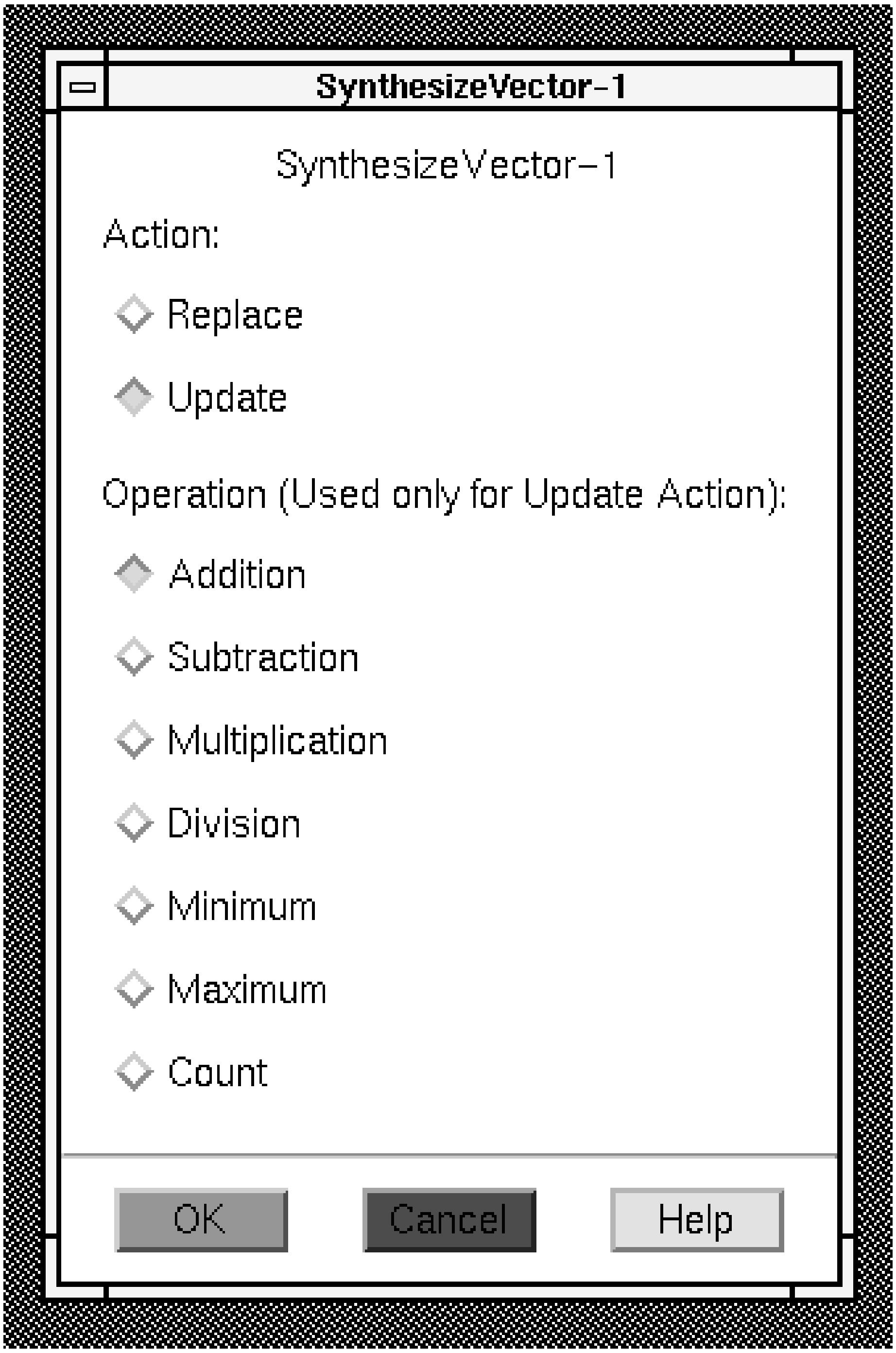
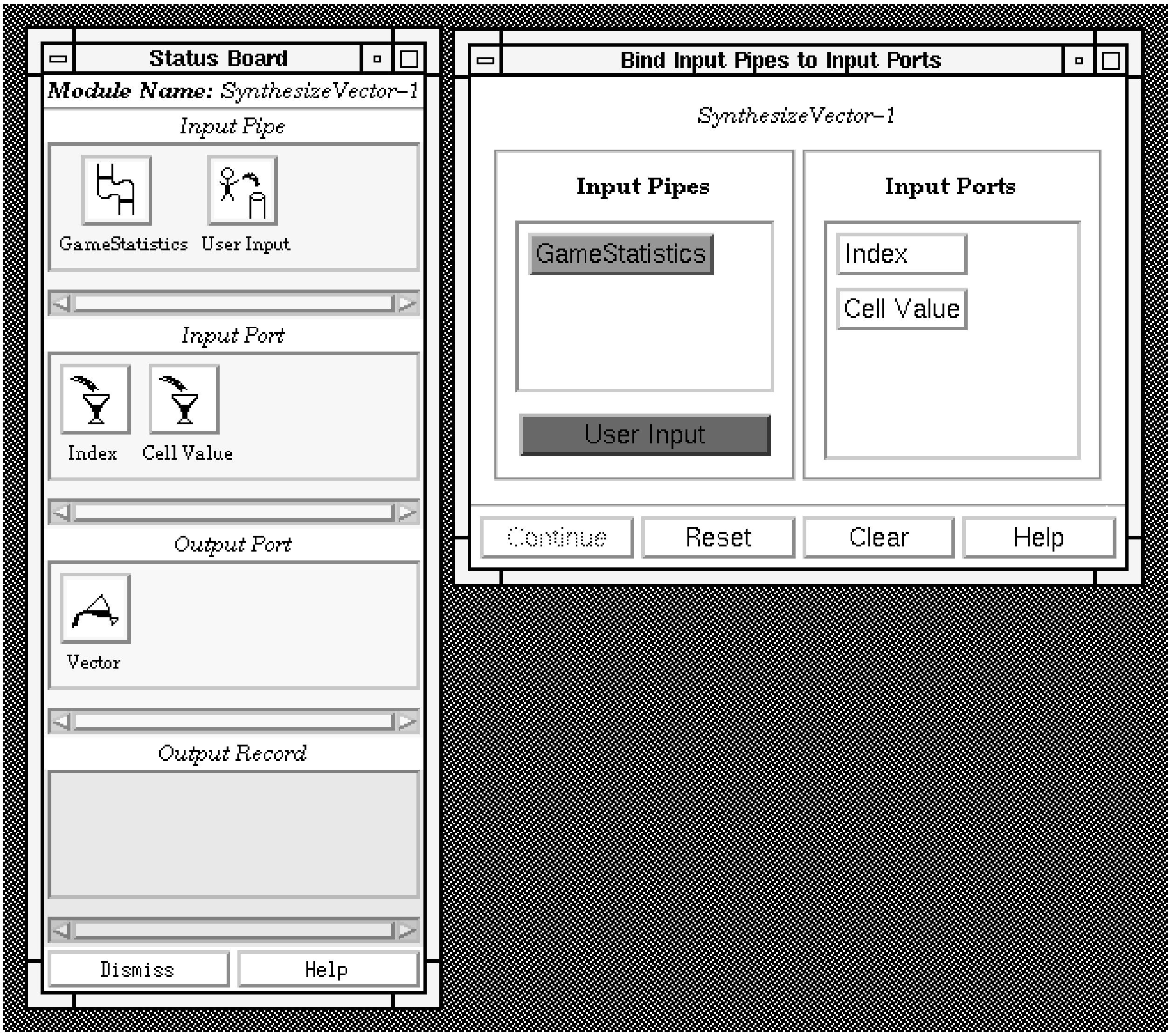
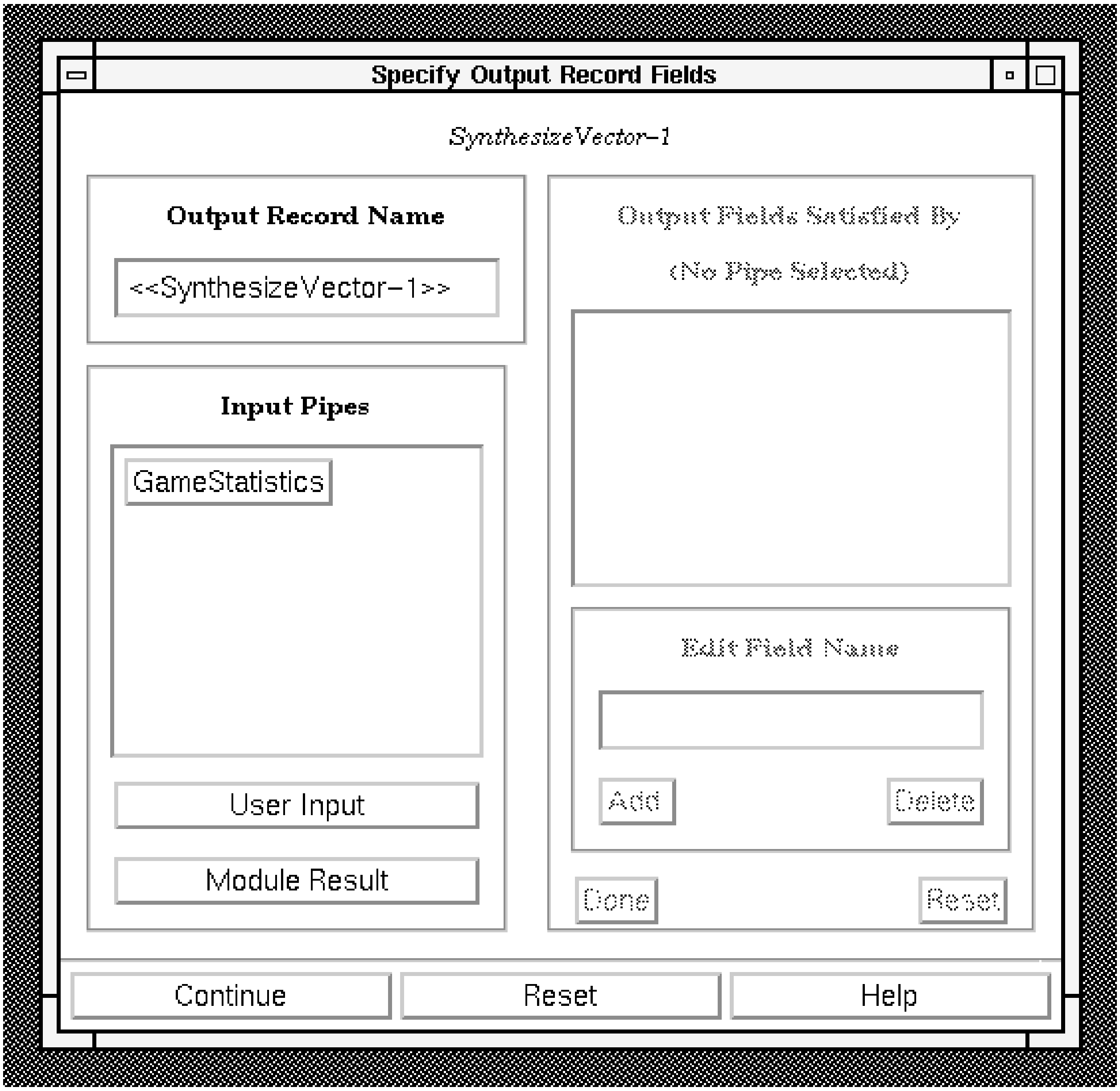

SynthesizeVector-1
SynthesizeVector-1
Action:
Replace
Update
Operation (Used only for Update Action):
Addition
Subtraction
Multiplication
Division
Minimum
Maximum
Count
OK
Cancel
Help

FIGURE 2.3



Status Board
Module Name: SynthesizeVector-1
Input Pipe
GameStatistics
User Input
Input Port
Index
Cell Value
Output Port
Vector
Output Record
Dismiss
Help
Bind Input Pipes to Input Ports
SynthesizeVector-1
Input Pipes
GameStatistics
User Input
Input Ports
Index
Cell Value
Continue
Reset
Clear
Help

FIGURE 2.4



Specify Output Record Fields
SynthesizeVector–1
Output Record Name
<<SynthesizeVector–1>>
Input Pipes
GameStatistics
User Input
Module Result
Output Fields Satisfied By
(No Pipe Selected)
Edit Field Name
Add
Delete
Done
Reset
Continue
Reset
Help

FIGURE 2.5

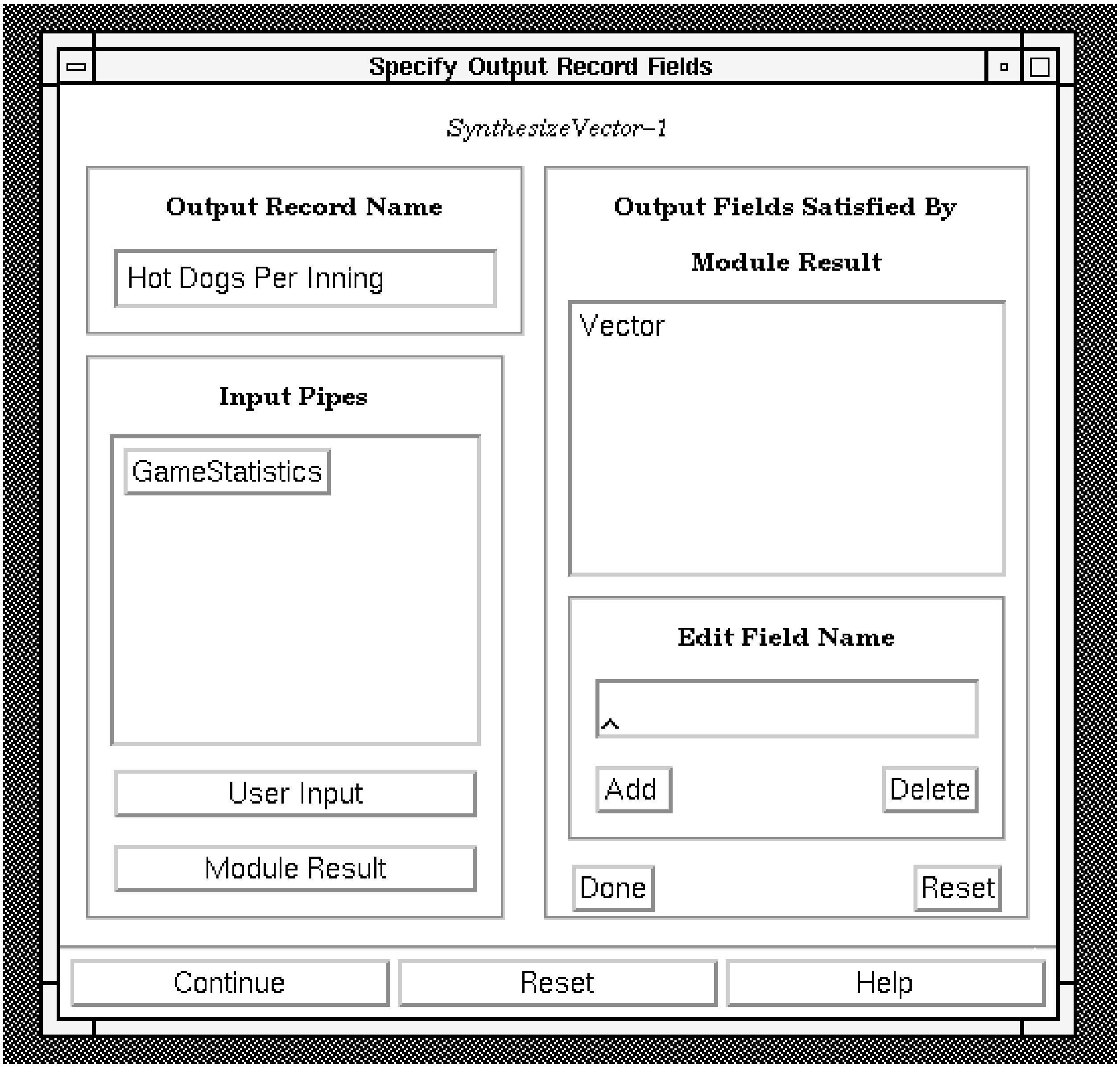

Specify Output Record Fields
SynthesizeVector–1
Output Record Name
Hot Dogs Per Inning
Input Pipes
GameStatistics
User Input
Module Result
Output Fields Satisfied By
Module Result
Vector
Edit Field Name
Add
Delete
Done
Reset
Continue
Reset
Help

FIGURE 2.6

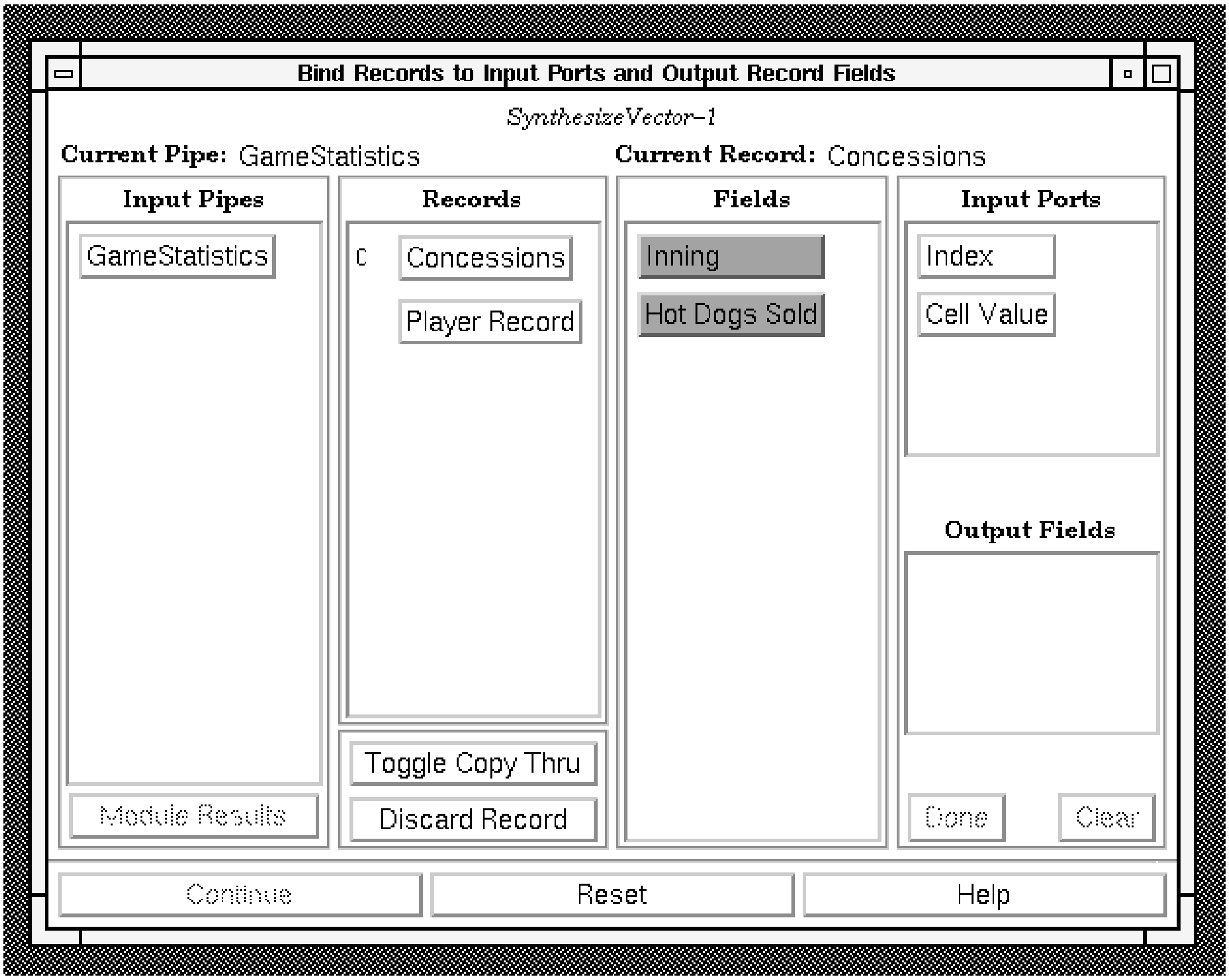

Bind Records to Input Ports and Output Record Fields
SynthesizeVector–1
Current Pipe: GameStatistics
Current Record: Concessions
Input Pipes
GameStatistics
Module Results
Records
C
Concessions
Player Record
Toggle Copy Thru
Discard Record
Fields
Inning
Hot Dogs Sold
Input Ports
Index
Cell Value
Output Fields
Done
Clear
Continue
Reset
Help

FIGURE 2.7

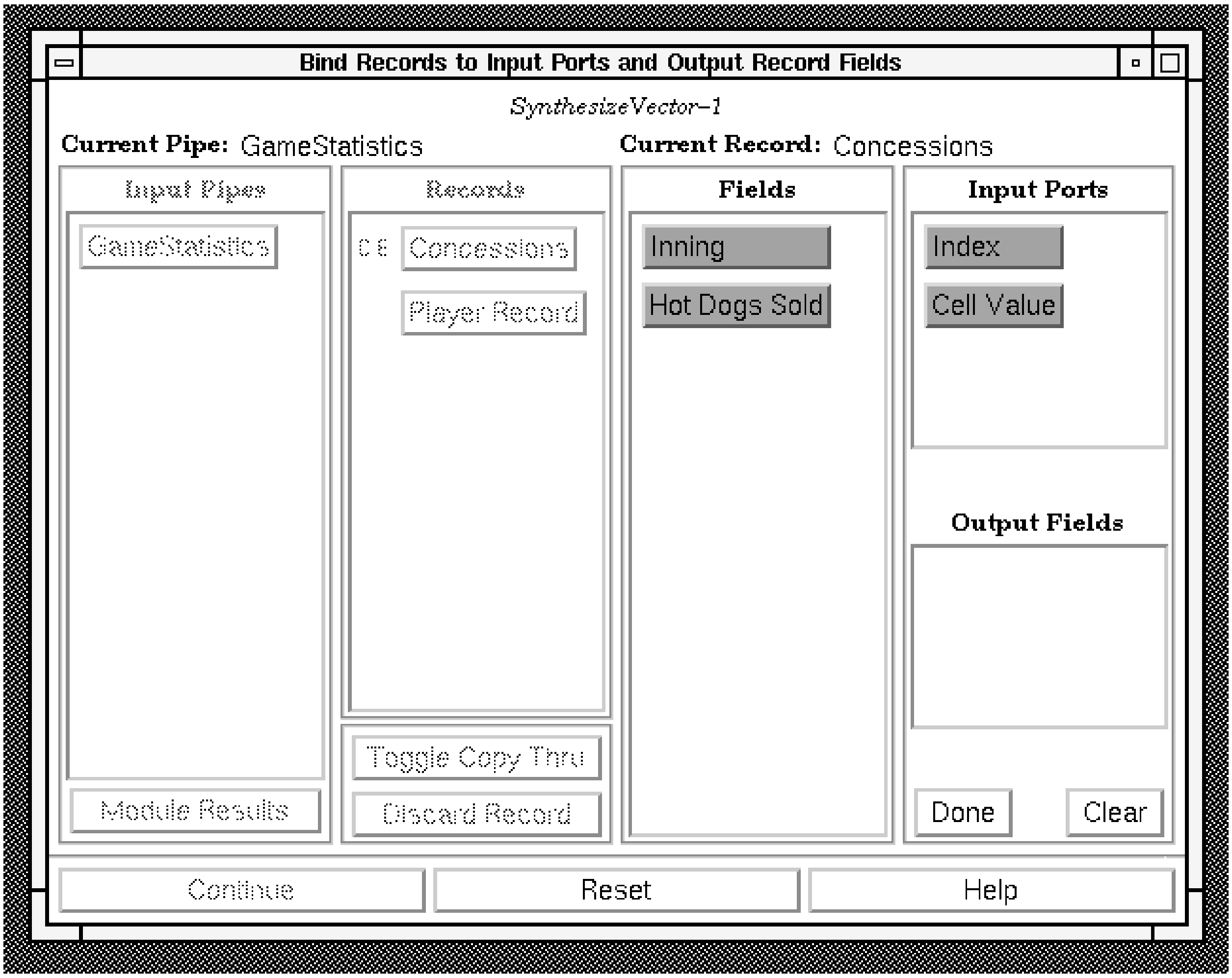

Bind Records to Input Ports and Output Record Fields
SynthesizeVector-1
Current Pipe: GameStatistics
Current Record: Concessions
Input Pipes
GameStatistics
Module Results
Records
Concessions
Player Record
Toggle Copy Thru
Discard Record
Fields
Inning
Hot Dogs Sold
Input Ports
Index
Cell Value
Output Fields
Done
Clear
Continue
Reset
Help

FIGURE 2.8

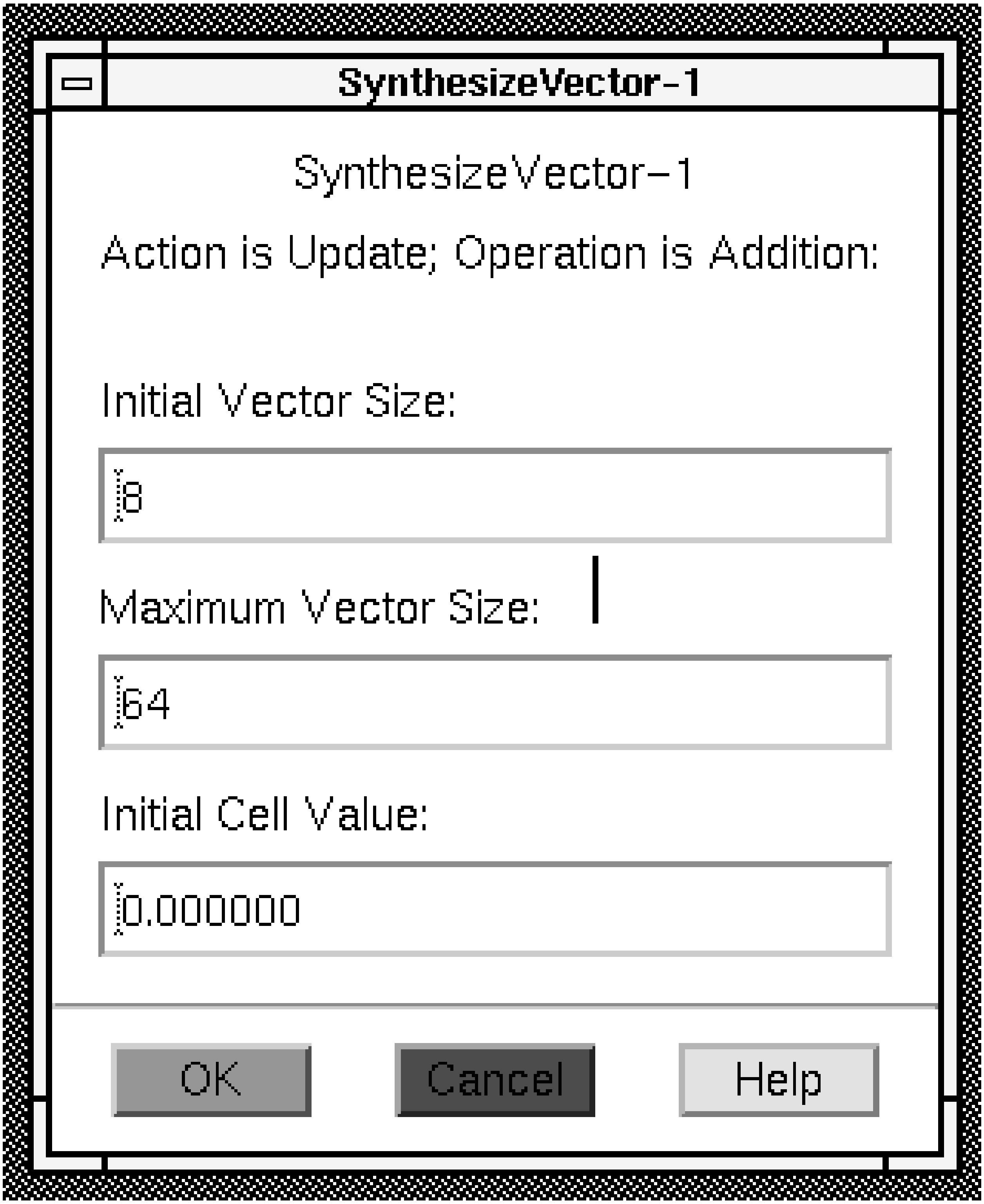

SynthesizeVector-1
SynthesizeVector-1
Action is Update; Operation is Addition:
Initial Vector Size:
8
Maximum Vector Size:
64
Initial Cell Value:
0.000000
OK
Cancel
Help

FIGURE 2.9

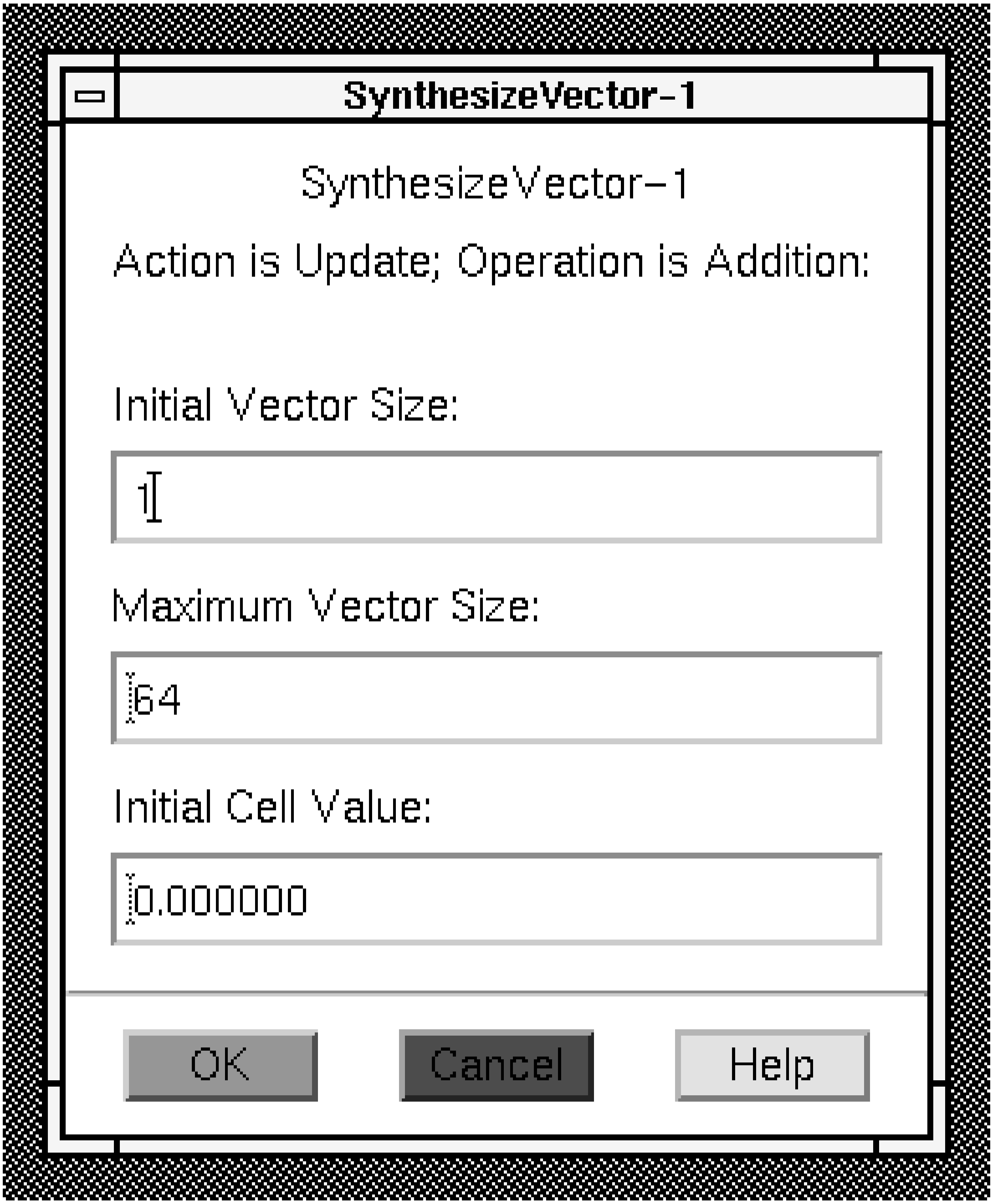

SynthesizeVector-1
SynthesizeVector−1
Action is Update; Operation is Addition:
Initial Vector Size:
1
Maximum Vector Size:
64
Initial Cell Value:
0.000000
OK
Cancel
Help

FIGURE 2.10

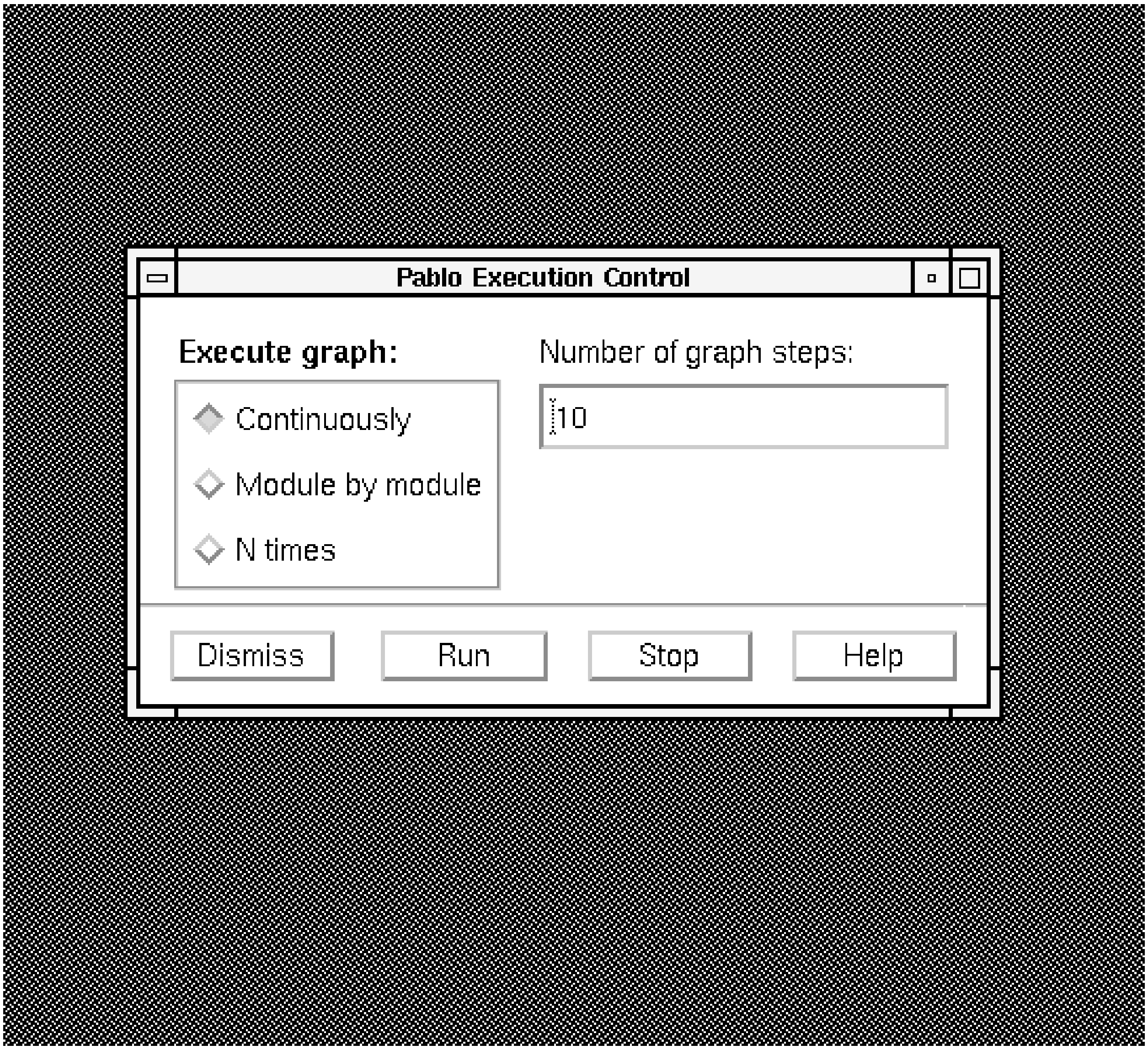

Pablo Execution Control
Execute graph:
Continuously
Module by module
N times
Number of graph steps:
10
Dismiss
Run
Stop
Help

FIGURE 2.11

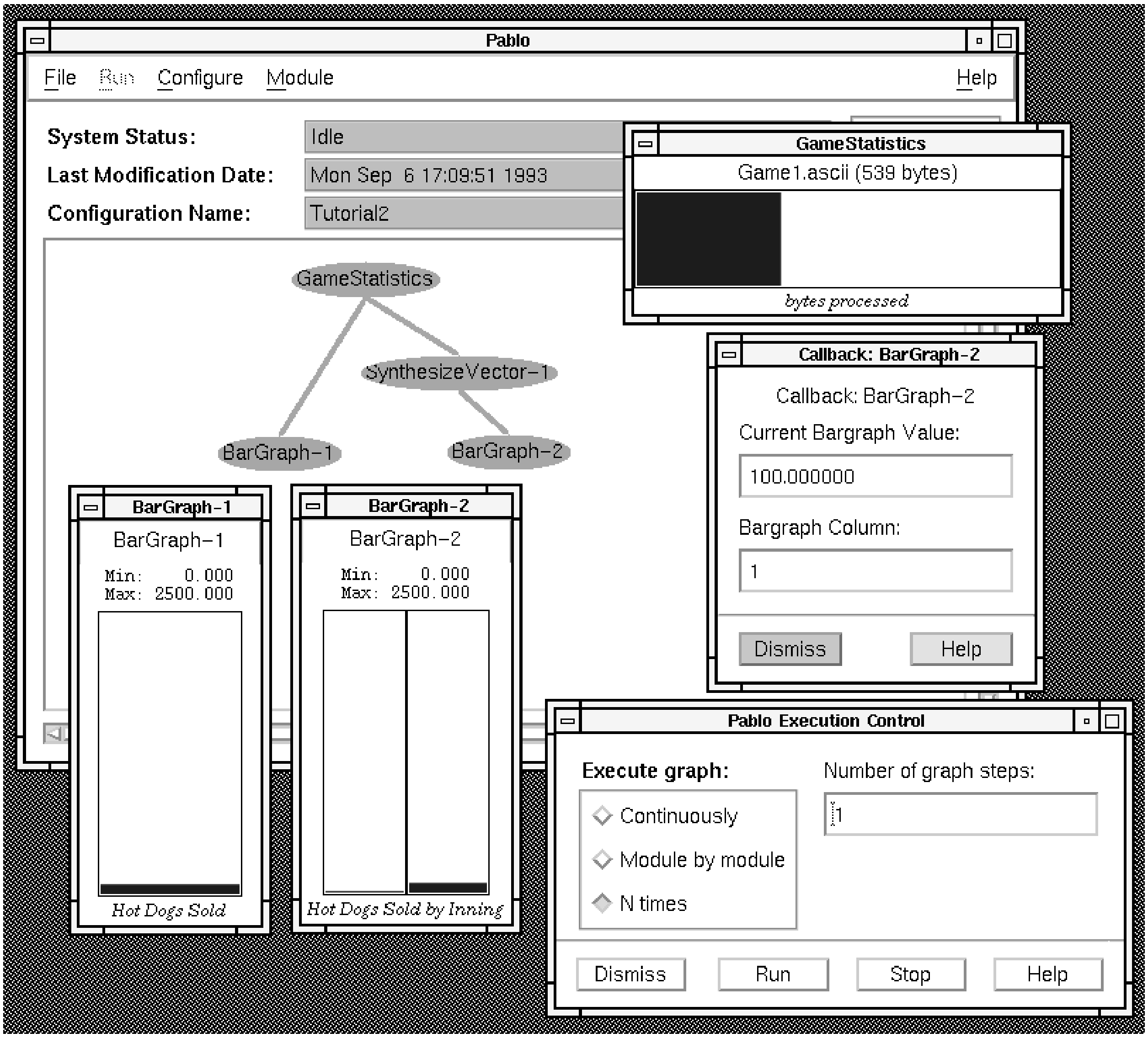

Pablo
File
Run
Configure
Module
Help
System Status:
Idle
Last Modification Date:
Mon Sep 6 17:09:51 1993
Configuration Name:
Tutorial2
GameStatistics
SynthesizeVector-1
BarGraph-1
BarGraph-2
GameStatistics
Game1.ascii (539 bytes)
bytes processed
Callback: BarGraph-2
Callback: BarGraph-2
Current Bargraph Value:
100.000000
Bargraph Column:
1
Dismiss
Help
BarGraph-1
BarGraph-1
Min: 0.000
Max: 2500.000
Hot Dogs Sold
BarGraph-2
BarGraph-2
Min: 0.000
Max: 2500.000
Hot Dogs Sold by Inning
Pablo Execution Control
Execute graph:
Continuously
Module by module
N times
Number of graph steps:
1
Dismiss
Run
Stop
Help

FIGURE 2.12

B.2 Configuration Parameters for Individual Modules

B.2.1 GameStatistics

Module Name:	GameStatistics
Description:	FileInput module.
Selected Input File:	Game1.ascii

B.2.2 BarGraph-1

Module Name:	BarGraph-1
Description:	BarGraph display module. Shows number of hot dogs sold for an inning.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: GameStatistics

Action	Record	Field	Bound To
Extract	Concessions	Hot Dogs Sold	Input
Discard	Player Record		

Parameter Configuration

Parameter	Value
Historical marker interval	5
Minimum value	0.00
Maximum value	2500.00
Vertical axis label	
Horizontal axis label	Hot Dogs Sold

B.2.3 SynthesizeVector-1

Module Name: SynthesizeVector-1

Description: SynthesizeVector synthesis module.
Builds vector of hot dogs sold indexed by inning.
Totals amounts if more than one concessions record per inning.

Operation Selection: Update/Addition

Bind Input Pipes to Input Ports

Input Pipe	Input Port
GameStatistics	Index Cell Value

Specify Output Record Fields

Output Record Name: Hot Dogs Per Inning	
Input Source	**Output Field**
Module Result	Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: GameStatistics			
Action	**Record**	**Field**	**Bound To**
Copy/Extr	Concessions	Inning	Index
		Hot Dogs Sold	Cell Value
Discard	Player Record		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Vector	Vector*

Parameter Configuration

Parameter	Value
Initial Vector Size	1
Maximum Vector Size	64
Initial Cell Value	0.00

B.2.4 BarGraph-2

Module Name: BarGraph-2
Description: BarGraph display module.
Shows number of hot dogs sold for each inning.
This display shows a vector of bargraphs.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeVector-1

Action	Record	Field	Bound To
Discard	Concessions		
Extract	Hot Dogs Per Inning	Vector	Input

Parameter Configuration

Parameter	Value
Historical marker interval	5
Minimum value	0.00
Maximum value	2500.00
Vertical axis label	
Horizontal axis label	Hot Dogs Sold by Inning

C Figures and Configuration Parameters for Tutorial 3

C.1 Figures 3.1 through 3.6

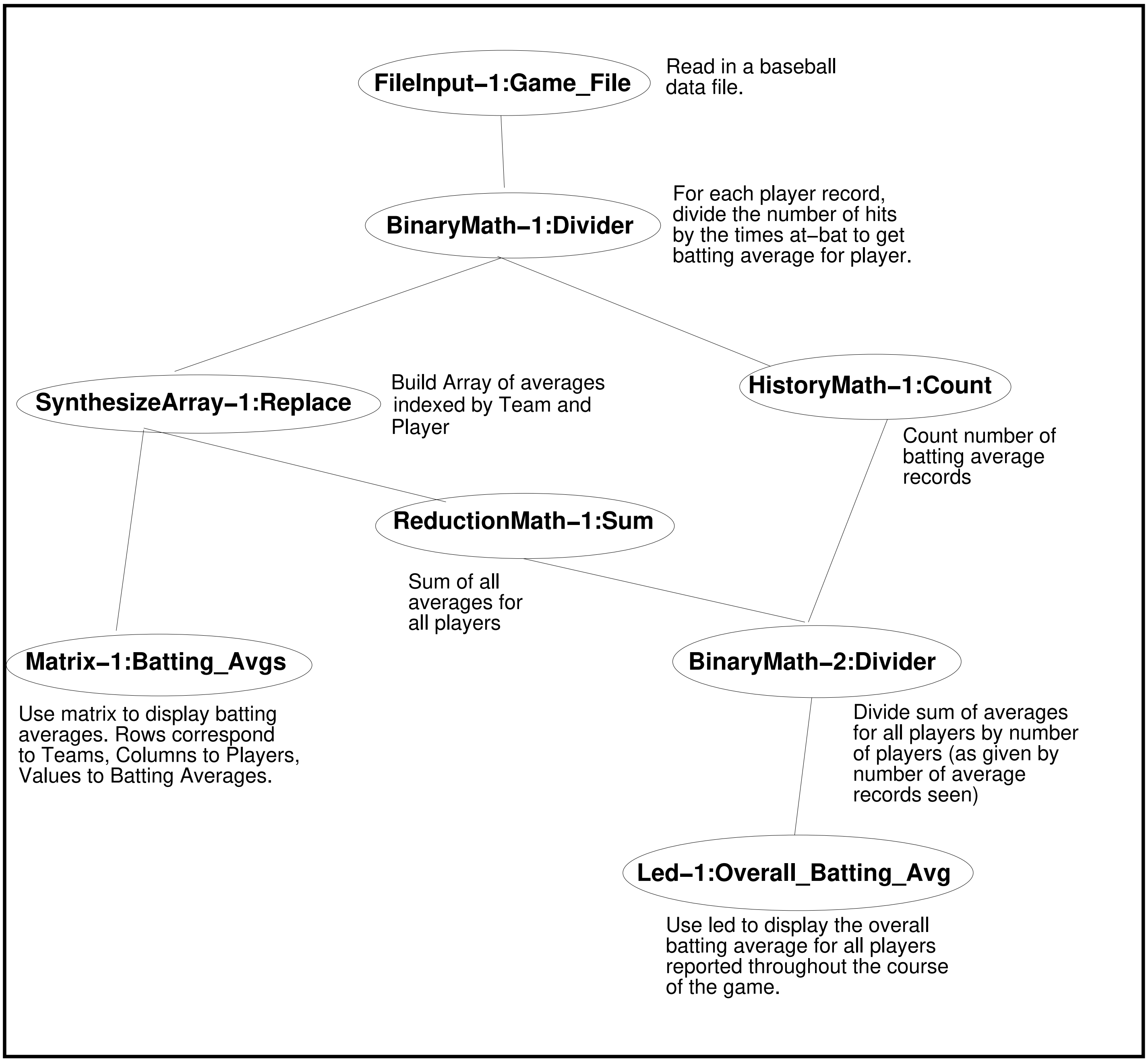


FIGURE 3.1

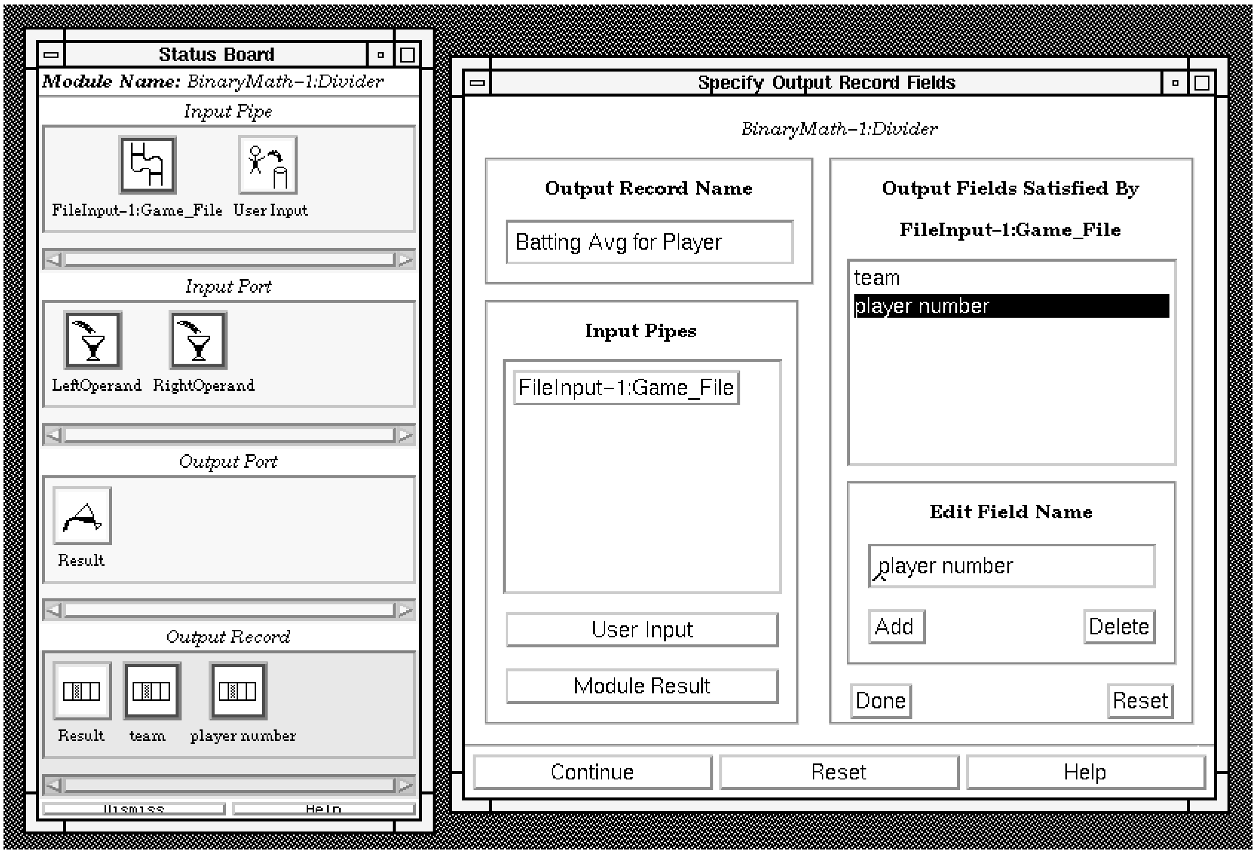
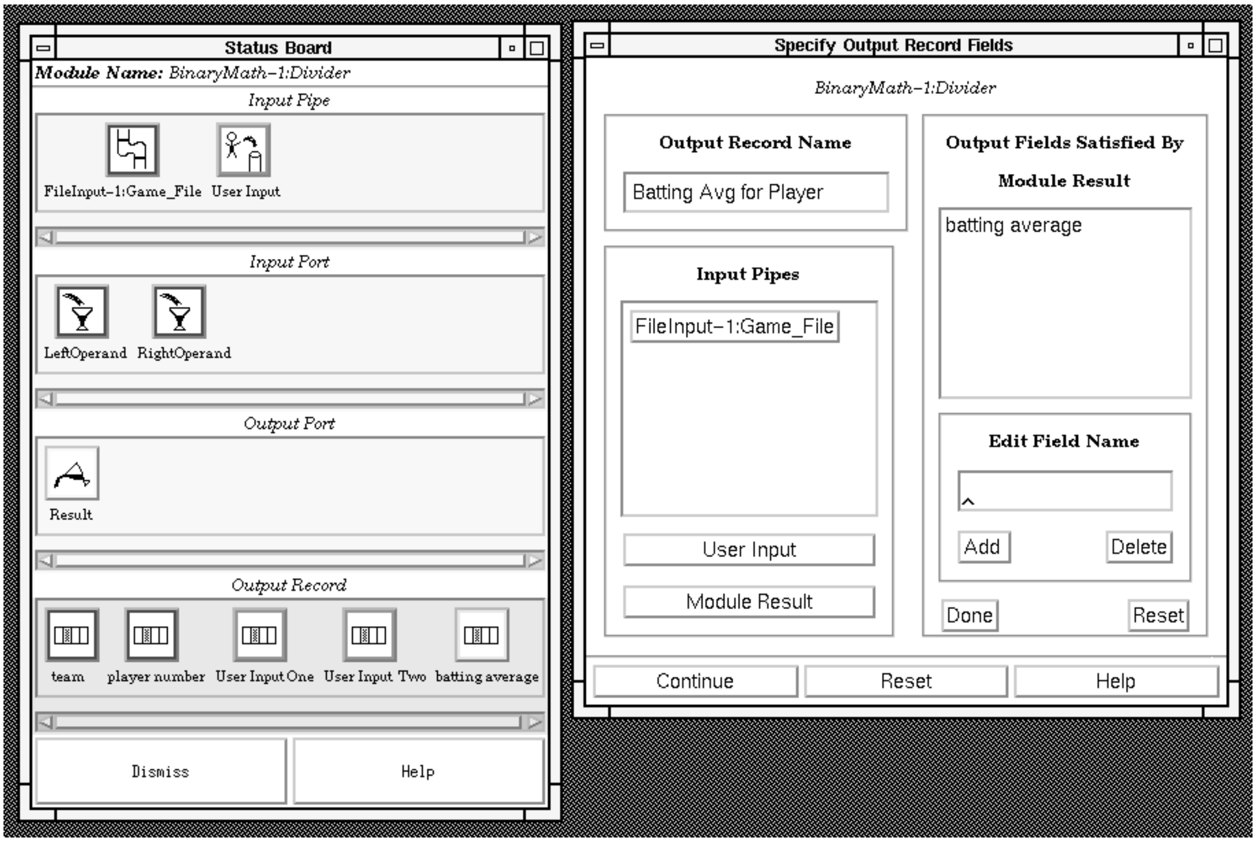

Status Board
Module Name: BinaryMath–1:Divider
Input Pipe
FileInput–1:Game_File
User Input
Input Port
LeftOperand
RightOperand
Output Port
Result
Output Record
Result
team
player number
Dismiss
Help
Specify Output Record Fields
BinaryMath–1:Divider
Output Record Name
Batting Avg for Player
Output Fields Satisfied By
FileInput–1:Game_File
team
player number
Input Pipes
FileInput–1:Game_File
User Input
Module Result
Edit Field Name
player number
Add
Delete
Done
Reset
Continue
Reset
Help

FIGURE 3.2



Status Board
Module Name: BinaryMath-1:Divider
Input Pipe
FileInput-1:Game_File
User Input
Input Port
LeftOperand
RightOperand
Output Port
Result
Output Record
team
player number
User Input One
User Input Two
batting average
Dismiss
Help
Specify Output Record Fields
BinaryMath-1:Divider
Output Record Name
Batting Avg for Player
Output Fields Satisfied By Module Result
batting average
Input Pipes
FileInput-1:Game_File
User Input
Module Result
Edit Field Name
Add
Delete
Done
Reset
Continue
Reset
Help

FIGURE 3.3

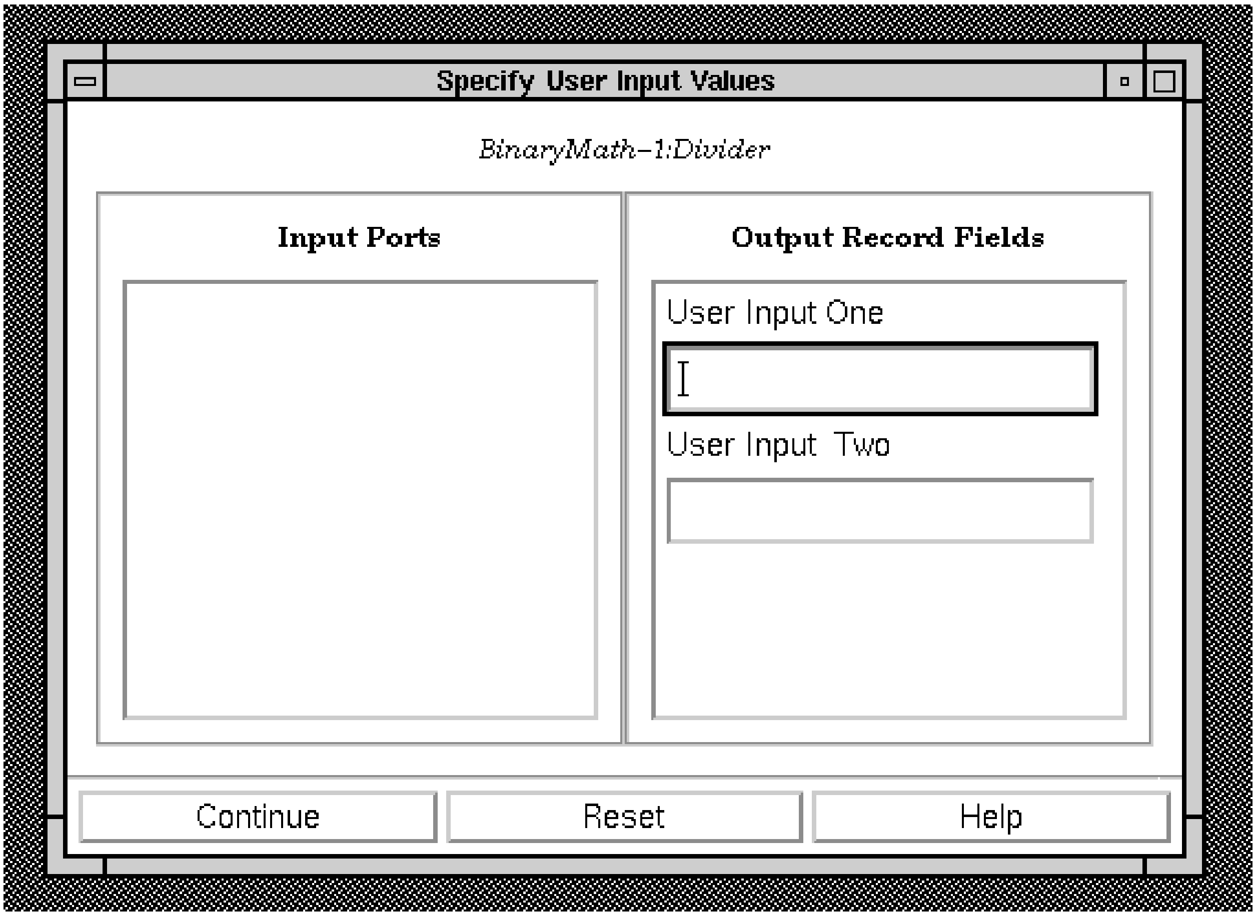

Specify User Input Values
BinaryMath–1:Divider
Input Ports
Output Record Fields
User Input One
User Input Two
Continue
Reset
Help

FIGURE 3.4

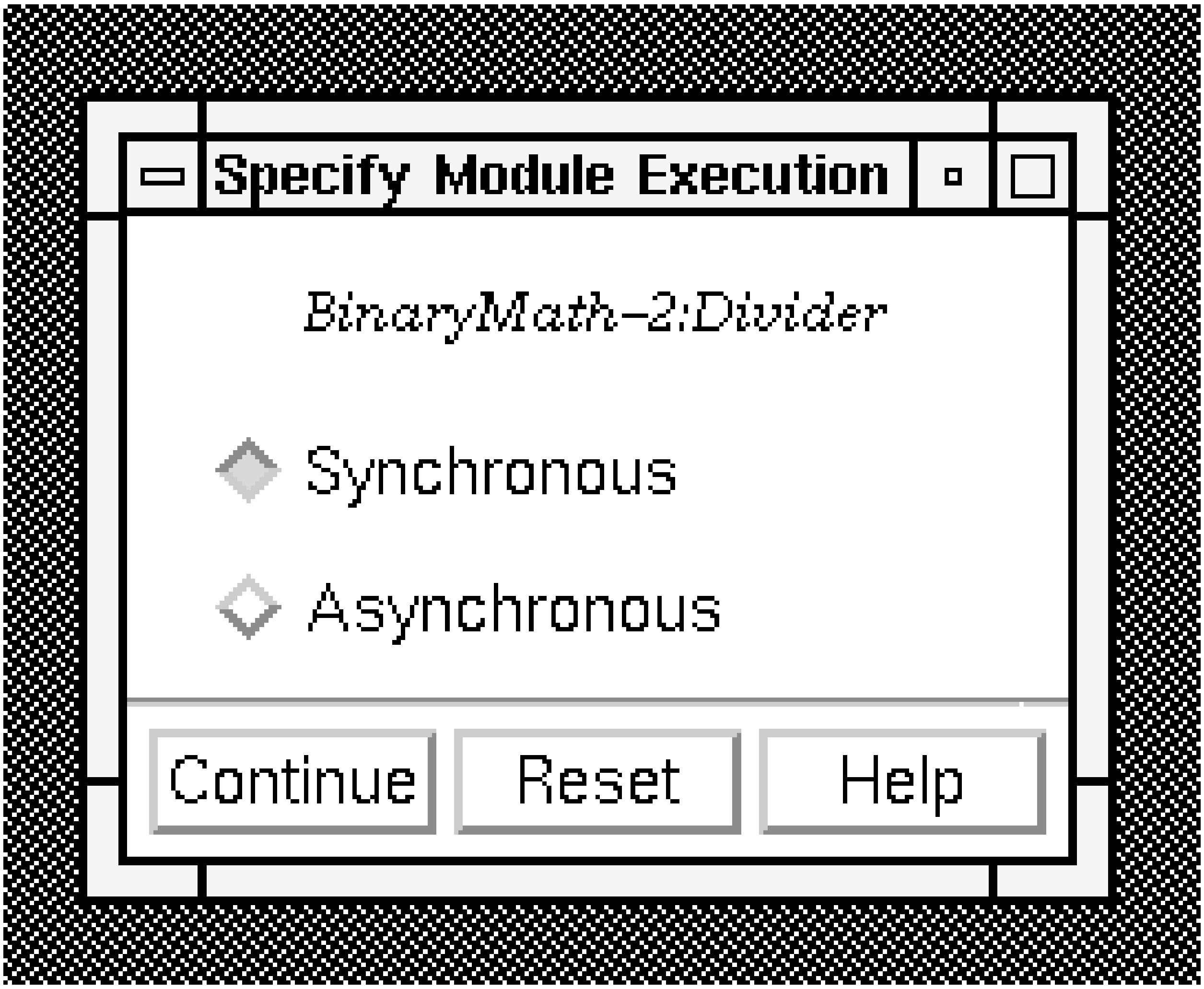
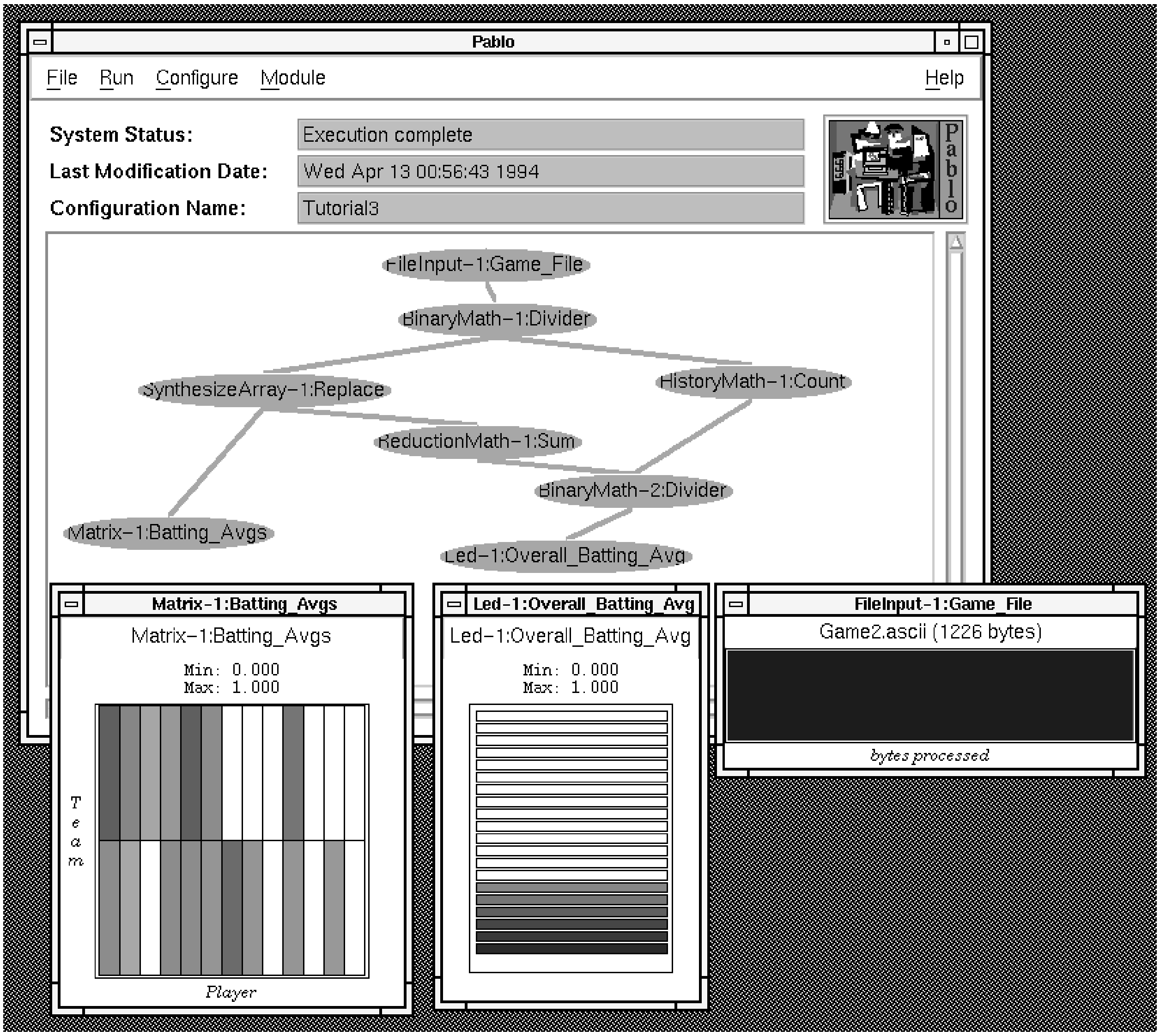

Specify Module Execution
BinaryMath-2:Divider
Synchronous
Asynchronous
Continue
Reset
Help

FIGURE 3.5



Pablo
File
Run
Configure
Module
Help
System Status:
Execution complete
Last Modification Date:
Wed Apr 13 00:56:43 1994
Configuration Name:
Tutorial3
FileInput−1:Game_File
BinaryMath−1:Divider
SynthesizeArray−1:Replace
HistoryMath−1:Count
ReductionMath−1:Sum
BinaryMath−2:Divider
Matrix−1:Batting_Avgs
Led−1:Overall_Batting_Avg
Matrix−1:Batting_Avgs
Matrix−1:Batting_Avgs
Min: 0.000
Max: 1.000
Team
Player
Led−1:Overall_Batting_Avg
Led−1:Overall_Batting_Avg
Min: 0.000
Max: 1.000
FileInput−1:Game_File
Game2.ascii (1226 bytes)
bytes processed

FIGURE 3.6

C.2 Configuration Parameters for Individual Modules

C.2.1 FileInput-1:Game_File

Module Name:	FileInput-1:Game_File
Description:	File input module.
Selected Input File:	Game2.ascii

C.2.2 BinaryMath-1:Divider

Module Name:	BinaryMath-1:Divider
Description:	BinaryMath math unit. Divides number of hits by times at-bat to calculate batting average. Output record also has player and team information.
Binary Operation:	Double-precision_Division

Bind Input Pipes to Input Ports

Input Pipe	**Input Port**
FileInput-1:Game_File	LeftOperand
	RightOperand

Specify Output Record Fields

Output Record Name: Batting Avg for Player	
Input Source	**Output Field**
FileInput-1:Game_File	team
	player number
User Input	User Input One
	User Input Two
Module Result	batting average

Specify User Input Values

Output Record Field	**Value**
User Input One	10
User Input Two	22.22

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Game_File			
Action	**Record**	**Field**	**Bound To**
Discard	Concessions		
Extract	Player Record	Hits	LeftOperand
		At-Bats	RightOperand
		Team	team*
		Player Number	player number*

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	batting average*

C.2.3 SynthesizeArray-1:Replace

Module Name: SynthesizeArray-1:Replace

Description: SynthesizeArray synthesis unit.
Build array of batting averages indexed by team and player.

Operation Selection: Replace/don't care

Bind Input Pipes to Input Ports

Input Pipe	Input Port
BinaryMath-1:Divider	Index0
	Index1
	Cell Value

Specify Output Record Fields

Output Record Name: Batting Avg by Team/Player	
Input Source	**Output Field**
Module Result	Array

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-1:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg for Player	team	Index0
		player number	Index1
		batting average	Cell Value

Module Results			
Action	**Record**	**Field**	**Bound To**
		Array	Array*

Parameter Configuration

Parameter	Value
Initial Array Size (Dimension 0)	2
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	9
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

C.2.4 Matrix-1:Batting_Avgs

Module Name:	Matrix-1:Batting_Avgs
Description:	Matrix display unit. Displays batting averages by team and player.
Data Input Type:	Array or Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArray-1:Replace			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg by Team/Player	Array	Input

Parameter Configuration

Parameter	Value
Initial Number of Rows	1
Initial Number of Columns	1
Minimum value	0.00
Maximum value	1.00
Vertical axis label	Team
Horizontal axis label	Player

C.2.5 ReductionMath-1:Sum

Module Name: ReductionMath-1:Sum
Description: ReductionMath math unit.
Sums batting averages of all players.

Reduction Operation: Summation

Specify Output Record Fields

Output Record Name: Sum of all Batting Avgs	
Input Source	**Output Field**
Module Result	sum

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArray-1:Replace			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg by Team/Player	Array	Operand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	sum*

C.2.6 HistoryMath-1:Count

Module Name:	HistoryMath-1:Count
Description:	History math unit. Counts number of batting average records.

Historical Operation: Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
BinaryMath-1:Divider	Operand

Specify Output Record Fields

Output Record Name: <<HistoryMath-1:Count>>	
Input Source	**Output Field**
Module Result	batting records seen

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-1:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg for Player	batting average	Operand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Operand	batting records seen *

C.2.7 BinaryMath-2:Divider

Module Name:	BinaryMath-2:Divider
Description:	BinaryMath math unit. Divides sum of averages for all players by number of players.
Binary Operation:	Double-precision_Division

Bind Input Pipes to Input Ports

Input Pipe	Input Port
ReductionMath-1:Sum	Left Operand
HistoryMath-1:Count	Right Operand

Specify Output Record Fields

Output Record Name: Overall Batting Average	
Input Source	**Output Field**
Module Result	Result

Bind Records To Input Ports and Output Record Fields*

Input Pipe: ReductionMath-1:Sum			
Action	**Record**	**Field**	**Bound To**
Extract	Sum of all Batting Avgs	sum	LeftOperand

Input Pipe: HistoryMath-1:Count			
Action	**Record**	**Field**	**Bound To**
Extract	<<HistoryMath-1:Count>>	batting records seen	RightOperand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	Result*

Execution Mode: Synchronous

C.2.8 Led-1:Overall_Batting_Avg

Module Name: Led-1:Overall_Batting_Avg
Description: Led display unit.
Displays overall batting average for all players.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-2:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Overall Batting Avearage	Result	Input

Parameter Configuration

Parameter	**Value**
Historical marker interval	5
LED levels	20
Minimum value	0.00
Maximum value	1.00
Vertical axis label	
Horizontal axis label	

D Figures and Configuration Parameters for Tutorial 4

D.1 Figures 4.1 and 4.2

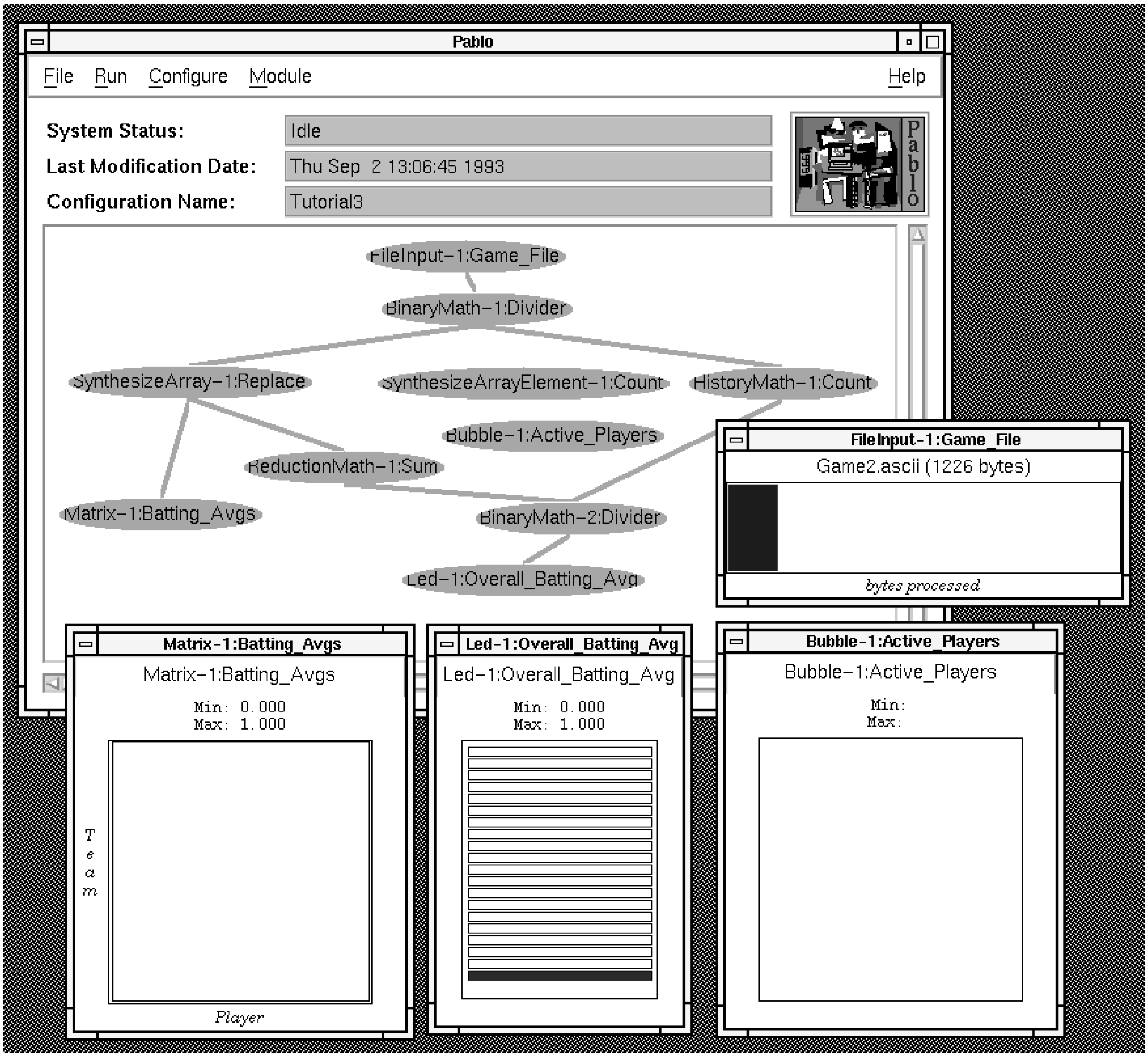


FIGURE 4.1

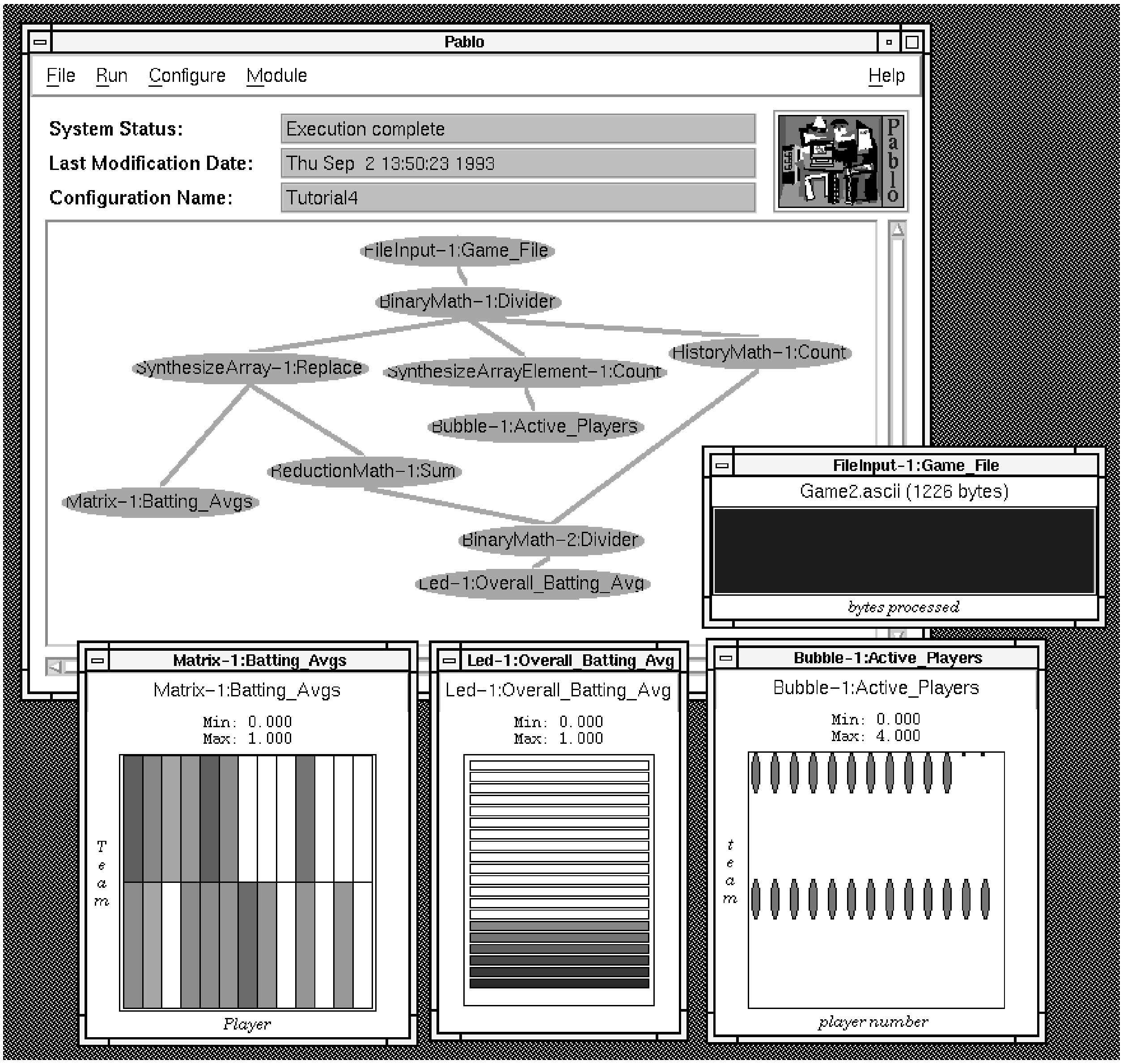

Pablo
File Run Configure Module Help
System Status: Execution complete
Last Modification Date: Thu Sep 2 13:50:23 1993
Configuration Name: Tutorial4
FileInput-1:Game_File
BinaryMath-1:Divider
SynthesizeArray-1:Replace
SynthesizeArrayElement-1:Count
HistoryMath-1:Count
Bubble-1:Active_Players
ReductionMath-1:Sum
Matrix-1:Batting_Avgs
BinaryMath-2:Divider
Led-1:Overall_Batting_Avg
FileInput-1:Game_File
Game2.ascii (1226 bytes)
bytes processed
Matrix-1:Batting_Avgs
Matrix-1:Batting_Avgs
Min: 0.000
Max: 1.000
Team
Player
Led-1:Overall_Batting_Avg
Led-1:Overall_Batting_Avg
Min: 0.000
Max: 1.000
Bubble-1:Active_Players
Bubble-1:Active_Players
Min: 0.000
Max: 4.000
team
player number

FIGURE 4.2

D.2 Configuration Parameters for Individual Modules

D.2.1 FileInput-1:Game_File

Module Name:	FileInput-1:Game_File
Description:	File input module.
Selected Input File:	Game2.ascii

D.2.2 BinaryMath-1:Divider

Module Name: BinaryMath-1:Divider
Description: BinaryMath math unit.
Divides number of hits by times at-bat to calculate batting average.
Output record also has player and team information.

Binary Operation: Double-precision_Division

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FileInput-1:Game_File	LeftOperand
	RightOperand

Specify Output Record Fields

Output Record Name: Batting Avg for Player	
Input Source	**Output Field**
FileInput-1:Game_File	team
	player number
User Input	User Input One
	User Input Two
Module Result	batting average

Specify User Input Values

Output Record Field	Value
User Input One	10
User Input Two	22.22

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Game_File			
Action	**Record**	**Field**	**Bound To**
Discard	Concessions		
Extract	Player Record	Hits	LeftOperand
		At-Bats	RightOperand
		Team	team*
		Player Number	player number*

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	batting average*

D.2.3 SynthesizeArray-1:Replace

Module Name:	SynthesizeArray-1:Replace
Description:	SynthesizeArray synthesis unit. Build array of batting averages indexed by team and player.
Operation Selection:	Replace/don't care

Bind Input Pipes to Input Ports

Input Pipe	Input Port
BinaryMath-1:Divider	Index0
	Index1
	Cell Value

Specify Output Record Fields

Output Record Name: Batting Avg by Team/Player	
Input Source	**Output Field**
Module Result	Array

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-1:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg for Player	team	Index0
		player number	Index1
		batting average	Cell Value

Module Results			
Action	**Record**	**Field**	**Bound To**
		Array	Array*

Parameter Configuration

Parameter	Value
Initial Array Size (Dimension 0)	2
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	9
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

D.2.4 Matrix-1:Batting_Avgs

Module Name:	Matrix-1:Batting_Avgs
Description:	Matrix display unit. Displays batting averages by team and player.
Data Input Type:	Array or Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArray-1:Replace

Action	Record	Field	Bound To
Extract	Batting Avg by Team/Player	Array	Input

Parameter Configuration

Parameter	Value
Initial Number of Rows	1
Initial Number of Columns	1
Minimum value	0.00
Maximum value	1.00
Vertical axis label	Team
Horizontal axis label	Player

D.2.5 ReductionMath-1:Sum

Module Name: ReductionMath-1:Sum
Description: ReductionMath math unit.
Sums batting averages of all players.

Reduction Operation: Summation

Specify Output Record Fields

Output Record Name: Sum of all Batting Avgs	
Input Source	**Output Field**
Module Result	sum

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArray-1:Replace			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg by Team/Player	Array	Operand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	sum*

D.2.6 SynthesizeArrayElement-1:Count

Module Name: SynthesizeArrayElement-1:Count

Description: SynthesizeArrayElement synthesis module.
Build array containing count of batting average records indexed by team and player.
Output array indices and element.

Operation Selection: Update/Count

Bind Input Pipes to Input Ports:

Input Pipe	Input Port
BinaryMath-1:Divider	Index0
	Index1
	Cell Value

Specify Output Record Fields

Output Record Name: Player Records by Team/Player	
Input Source	**Output Field**
Module Result	Subscript0
	Subscript1
	Matrix Element

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-1:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg for Player	team	Index0
		player number	Index1
		batting average	Cell Value

Module Results			
Action	**Record**	**Field**	**Bound To**
		Subscript0	Subscript0*
		Subscript1	Subscript1*
		Matrix Element	Matrix Element*

Parameter Configuration

Parameter	Value
Initial Array Size (Dimension 0)	1
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	1
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

D.2.7 Bubble-1:Active_Players

Module Name: Bubble-1:Active_Players

Description: Bubble display unit.
Displays number of player records seen indexed by team and player

Data Input Type: Array Indices and Element

Bind Input Pipes to Input Ports

Input Pipe	Input Port
SynthesizeArrayElement-1:Count	Index0 Index1 Array Element

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArrayElement-1:Count

Action	Record	Field	Bound To
Extract	Player Records by Team/Player	Subscript0 Subscript1 Matrix Element	Index 0 Index 1 Array Element

Parameter Configuration

Parameter	Value
Initial Number of Rows	1
Initial Number of Columns	1
Minimum value	0.00
Maximum value	4.00
Vertical axis label	team
Horizontal axis label	player number

D.2.8 HistoryMath-1:Count

Module Name: HistoryMath-1:Count
Description: History math unit.
Counts number of batting average records.

Historical Operation: Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
BinaryMath-1:Divider	Operand

Specify Output Record Fields

Output Record Name: <<HistoryMath-1:Count>>	
Input Source	**Output Field**
Module Result	batting records seen

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-1:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Batting Avg for Player	batting average	Operand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Operand	batting records seen *

D.2.9 BinaryMath-2:Divider

Module Name: BinaryMath-2:Divider
Description: BinaryMath math unit.
Divides sum of averages for all players by number of players.

Binary Operation: Double-precision_Division

Bind Input Pipes to Input Ports

Input Pipe	Input Port
ReductionMath-1:Sum	Left Operand
HistoryMath-1:Count	Right Operand

Specify Output Record Fields

Output Record Name: Overall Batting Average	
Input Source	**Output Field**
Module Result	Result

Bind Records To Input Ports and Output Record Fields*

Input Pipe: ReductionMath-1:Sum			
Action	**Record**	**Field**	**Bound To**
Extract	Sum of all Batting Avgs	sum	LeftOperand

Input Pipe: HistoryMath-1:Count			
Action	**Record**	**Field**	**Bound To**
Extract	<<HistoryMath-1:Count>>	batting records seen	RightOperand

Module Results			
Action	**Record**	**Field**	**Bound To**
		Result	Result*

Execution Mode: Synchronous

D.2.10 Led-1:Overall_Batting_Avg

Module Name: Led-1:Overall_Batting_Avg

Description: Led display unit.
Displays overall batting average for all players.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: BinaryMath-2:Divider			
Action	**Record**	**Field**	**Bound To**
Extract	Overall Batting Avearage	Result	Input

Parameter Configuration

Parameter	Value
Historical marker interval	5
LED levels	20
Minimum value	0.00
Maximum value	1.00
Vertical axis label	
Horizontal axis label	

E Figures and Configuration Parameters for Tutorial 5

E.1 Figures 5.1 through 5.2

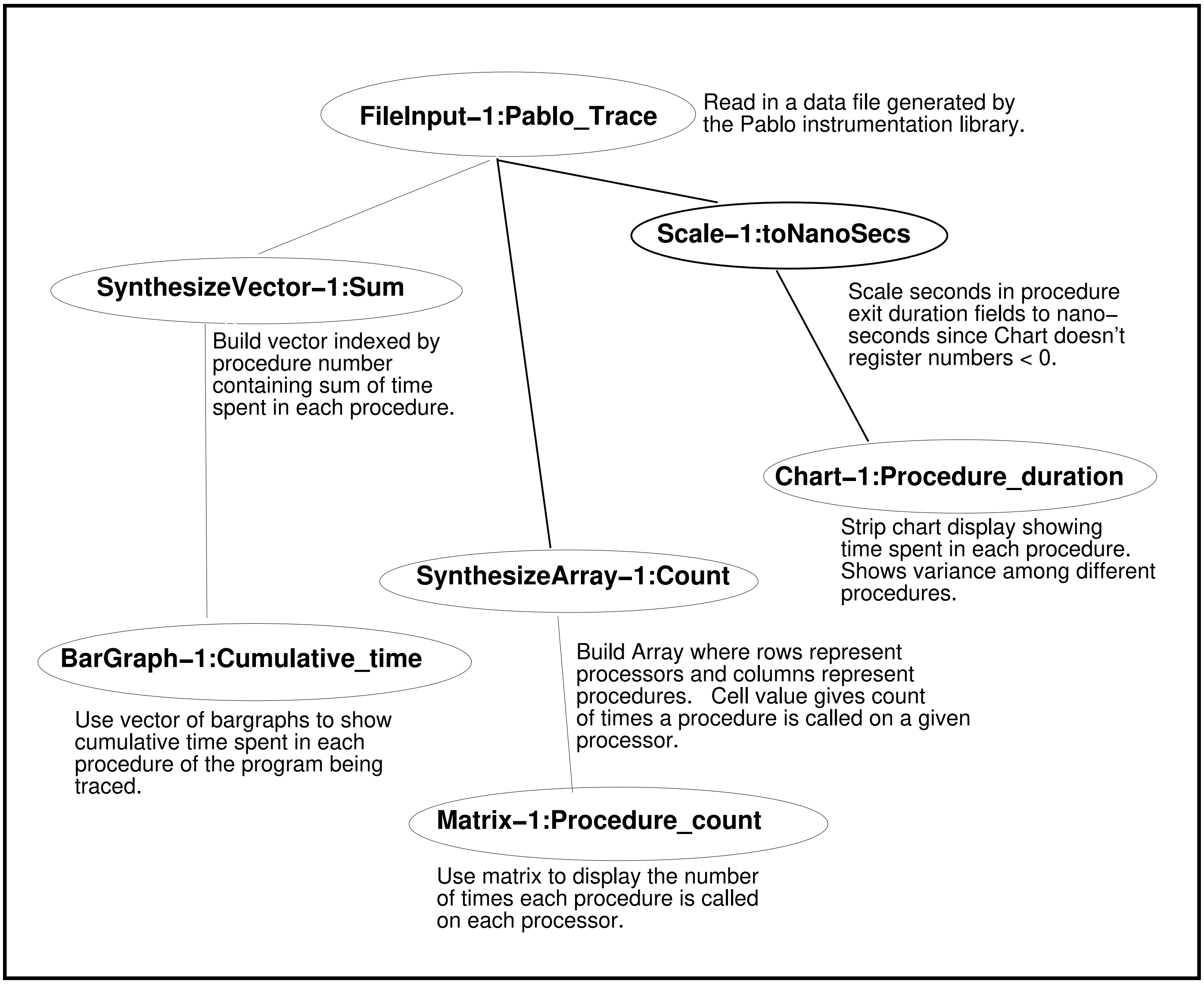


FIGURE 5.1

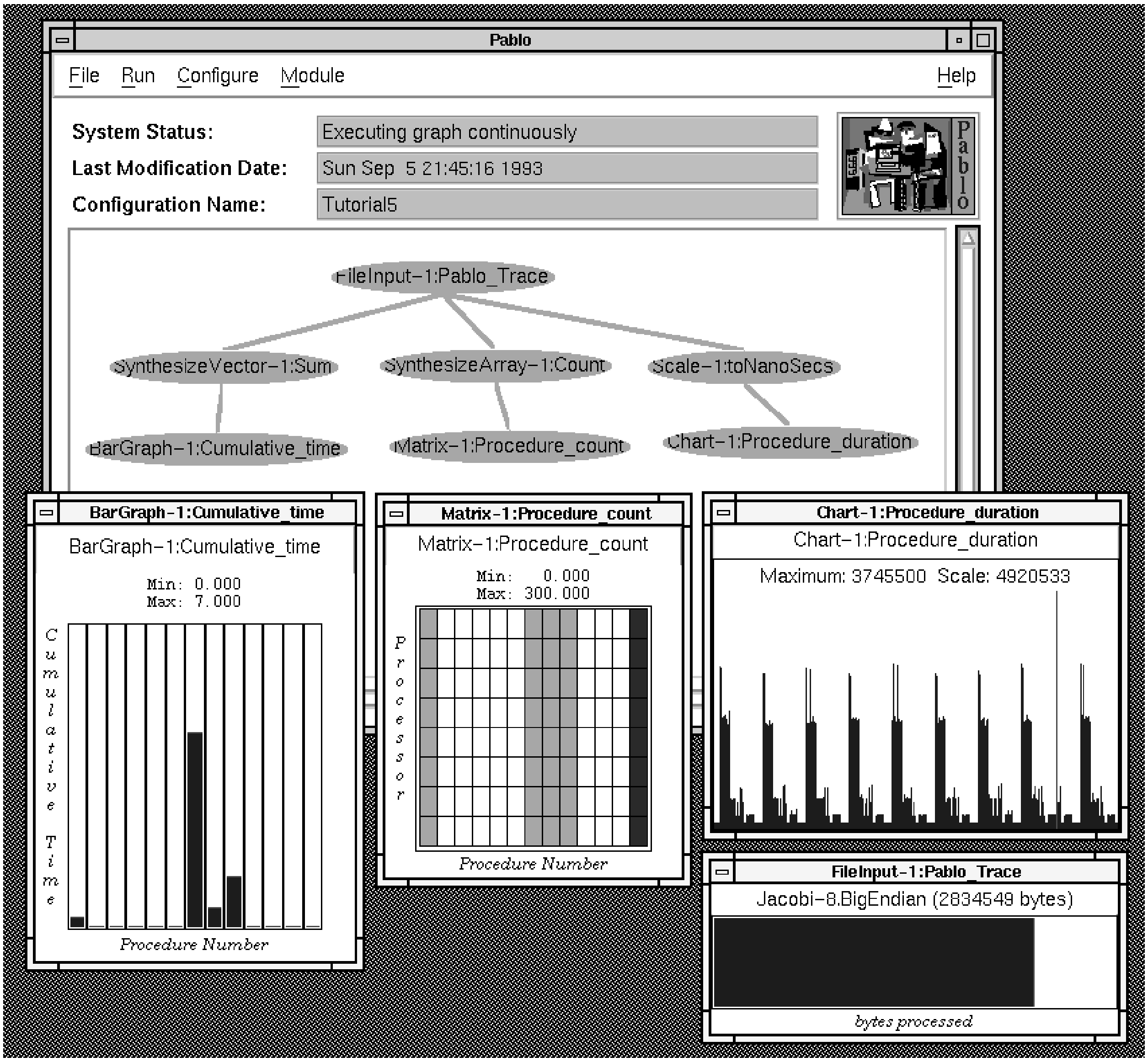

Pablo
File Run Configure Module Help
System Status: Executing graph continuously
Last Modification Date: Sun Sep 5 21:45:16 1993
Configuration Name: Tutorial5
Pablo
FileInput-1:Pablo_Trace
SynthesizeVector-1:Sum
SynthesizeArray-1:Count
Scale-1:toNanoSecs
BarGraph-1:Cumulative_time
Matrix-1:Procedure_count
Chart-1:Procedure_duration
BarGraph-1:Cumulative_time
BarGraph-1:Cumulative_time
Min: 0.000
Max: 7.000
Cumulative Time
Procedure Number
Matrix-1:Procedure_count
Matrix-1:Procedure_count
Min: 0.000
Max: 300.000
Processor
Procedure Number
Chart-1:Procedure_duration
Chart-1:Procedure_duration
Maximum: 3745500 Scale: 4920533
FileInput-1:Pablo_Trace
Jacobi-8.BigEndian (2834549 bytes)
bytes processed

FIGURE 5.2

E.2 Configuration Parameters for Individual Modules

E.2.1 FileInput-1:Pablo_Trace

Module Name:	FileInput-1:Pablo_Trace
Description:	File input module.
Selected Input File:	Jacobi-8.BigEndian

E.2.2 SynthesizeVector-1:Sum

Module Name: SynthesizeVector-1:Sum

Description: SynthesizeVector synthesis module.
Build vector of time spent in each procedure;
indexed by procedure number.

Operation Selection: Update/Addition

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FileInput-1:Pablo_Trace	Index Cell Value

Specify Output Record Fields

Output Record Name: Cumulative Procedure Time	
Input Source	**Output Field**
Module Result	Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Pablo_Trace			
Action	**Record**	**Field**	**Bound To**
Extract	Procedure Exit Trace	Procedure Index	Index
		Exclusive Seconds	Cell Value
Discard	*All Others*		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Vector	Vector*

Parameter Configuration

Parameter	Value
Initial Vector Size	8
Maximum Vector Size	64
Initial Cell Value	0.00

E.2.3 BarGraph-1:Cumulative_time

Module Name: BarGraph-1:Cumulative_time
Description: BarGraph display module.
Shows cumulative time in each procedure.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeVector-1:Sum

Action	Record	Field	Bound To
Extract	Cumulative Procedure Time	Vector	Input

Parameter Configuration

Parameter	Value
Historical marker interval	5
Minimum value	0.00
Maximum value	7.00
Vertical axis label	Cumulative Time
Horizontal axis label	Procedure Number

E.2.4 SynthesizeArray-1:Count

Module Name: SynthesizeArray-1:Count
Description: SynthesizeArray synthesis module.
Build array of procedure entry counts;
indexed by processor number and procedure.

Operation Selection: Update/Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FileInput-1:Pablo_Trace	Index 0
	Index 1
	Cell Value

Specify Output Record Fields

Output Record Name: Procedure Count by PE, Proc	
Input Source	**Output Field**
Module Result	Array

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Pablo_Trace			
Action	**Record**	**Field**	**Bound To**
Extract	Procedure Entry Trace	Processor Number	Index 0
		Procedure Index	Index 1
		Processor Number	Cell Value
Discard	*All Others*		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Array	Array*

Parameter Configuration

Parameter	Value
Initial Array Size (Dimension 0)	1
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	1
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

E.2.5 Matrix-1:Procedure_count

Module Name: Matrix-1:Procedure_count

Description: Matrix display unit.
Displays number of times each procedure entered.

Data Input Type: Array or Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SynthesizeArray-1:Count			
Action	**Record**	**Field**	**Bound To**
Extract	Procedure Count by PE,Proc	Array	Input

Parameter Configuration

Parameter	Value
Initial Number of Rows	1
Initial Number of Columns	1
Minimum value	0.00
Maximum value	300.00
Vertical axis label	Processor
Horizontal axis label	Procedure Number

E.2.6 Scale-1:toNanoSecs

Module Name:	Scale-1:toNanoSecs
Description:	Scale math unit. Scales procedure duration seconds to nanoseconds for input to ChartFU.

Parameter Configuration

Parameter	**Value**
Scaling Option	Static
Offset	0.00
Scale Factor	1000000000.00

Specify Output Record Fields

Output Record Name: Procedure Duration (ns)	
Input Source	**Output Field**
Module Result	Result

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Pablo_Trace

Action	**Record**	**Field**	**Bound To**
Extract	Procedure Exit Trace	Exclusive Seconds	Operand
Discard	*All Others*		

E.2.7 Chart-1:Procedure_duration

Module Name: Chart-1:Procedure_duration
Description: Chart display unit.
Shows duration spent in each procedure using strip-chart format.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: Scale-1:toNanoSecs			
Action	**Record**	**Field**	**Bound To**
Extract	Procedure Duration (ns)	Result	Input

F Figures and Configuration Parameters for Tutorial 6

F.1 Figures 6.1 through 6.2

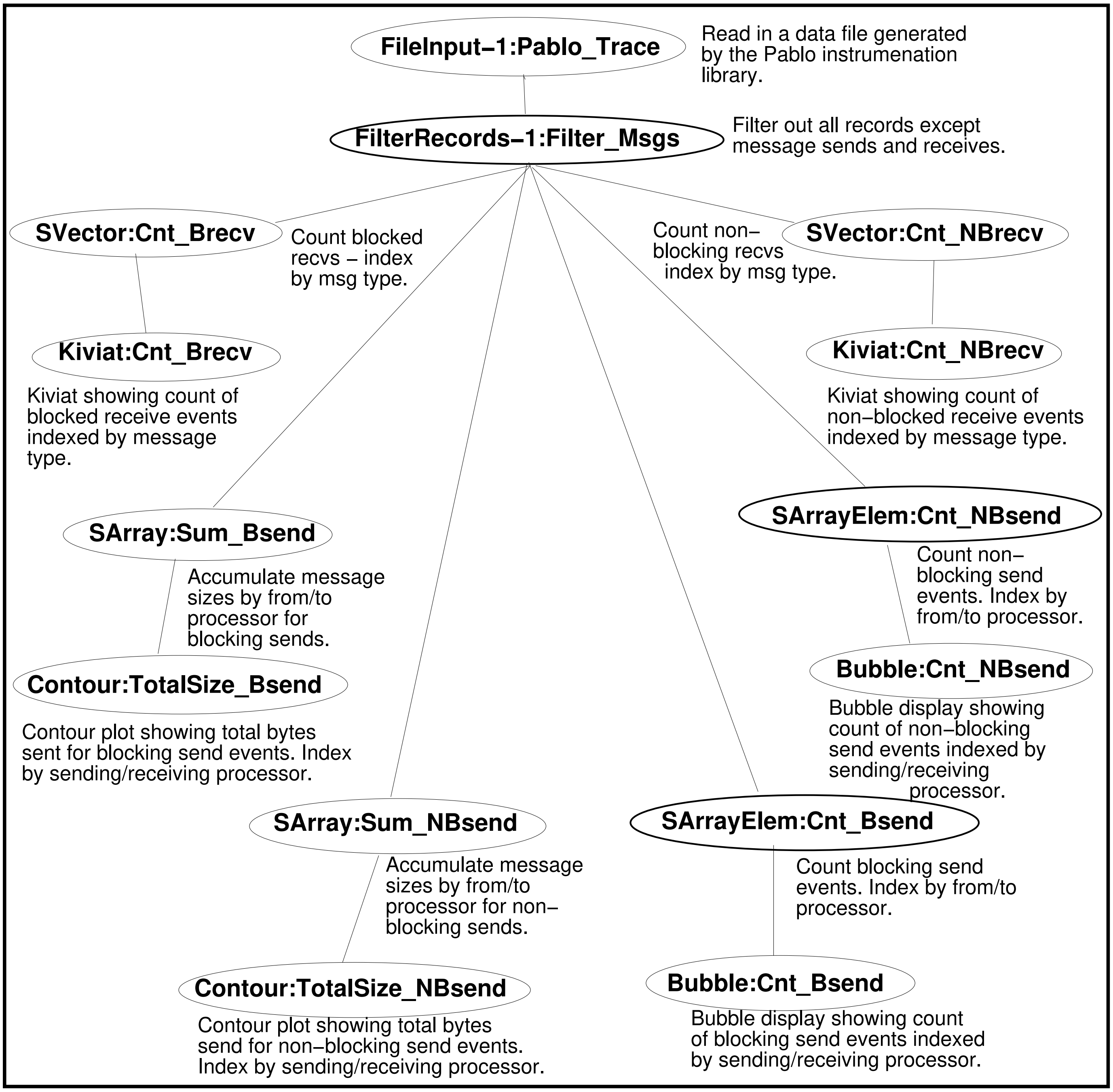


FIGURE 6.1

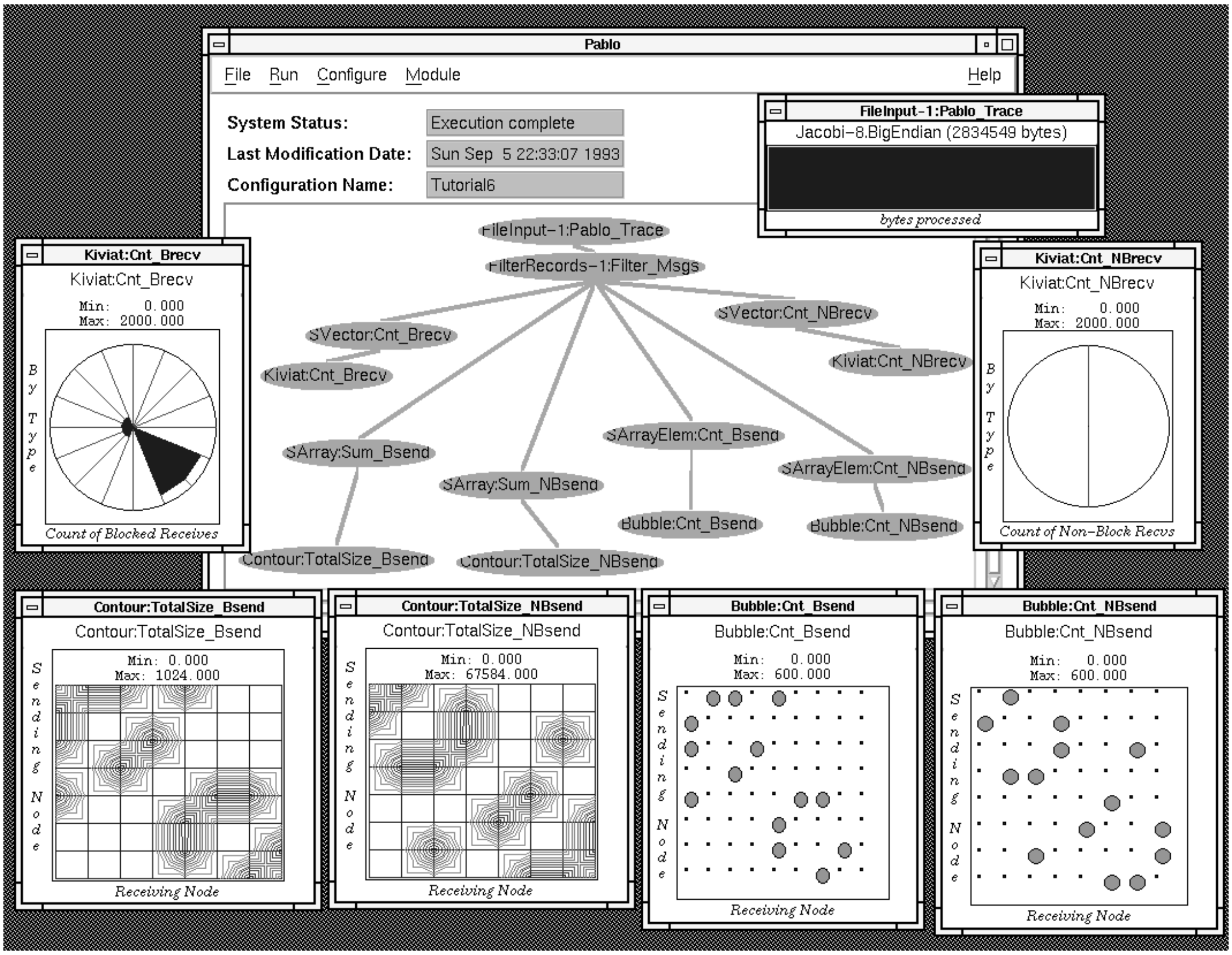

Pablo
File
Run
Configure
Module
Help
System Status:
Execution complete
Last Modification Date:
Sun Sep 5 22:33:07 1993
Configuration Name:
Tutorial6
FileInput-1:Pablo_Trace
Jacobi-8.BigEndian (2834549 bytes)
bytes processed
Kiviat:Cnt_Brecv
Min: 0.000
Max: 2000.000
By Type
Count of Blocked Receives
Kiviat:Cnt_NBrecv
Count of Non-Block Recvs
FilterRecords-1:Filter_Msgs
SVector:Cnt_Brecv
SVector:Cnt_NBrecv
Kiviat:Cnt_NBrecv
SArray:Sum_Bsend
SArrayElem:Cnt_Bsend
SArrayElem:Cnt_NBsend
SArray:Sum_NBsend
Bubble:Cnt_Bsend
Bubble:Cnt_NBsend
Contour:TotalSize_Bsend
Contour:TotalSize_NBsend
Max: 1024.000
Max: 67584.000
Max: 600.000
Sending Node
Receiving Node

FIGURE 6.2

F.2 Configuration Parameters for Individual Modules

F.2.1 FileInput-1:Pablo_Trace

Module Name:	FileInput-1:Pablo_Trace
Description:	File input module
Selected Input File:	Jacobi-8.BigEndian

F.2.2 FilterRecords-1:Filter_Msgs

Module Name:	FilterRecords-1:Filter_Msgs
Description:	FilterRecords module. Used as filter so that only records for message send and receive "begin" events are passed to lower nodes in the graph.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FileInput-1:Pablo_Trace

Action	Record	Field	Bound To
CopyThru	Blocking Send Begin		
CopyThru	Blocking Receive Begin		
CopyThru	Nonblocking Send Begin		
CopyThru	Nonblocking Receive Begin		
Discard	*All Others*		

F.2.3 SVector:Cnt_Brecv

Module Name: SVector:Cnt_Brecv

Description: SynthesizeVector synthesis module.
Builds vector with count of blocking receive events;
indexed by message type

Operation Selection: Update/Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FilterRecords-1:Filter_Msgs	Index Cell Value

Specify Output Record Fields

Output Record Name: Count Brecv by Msg Type	
Input Source	**Output Field**
Module Result	Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecords-1:Filter_Msgs			
Action	**Record**	**Field**	**Bound To**
Discard	Blocking Send Begin		
Extract	Blocking Receive Begin	Message Type	Index
		Message Length	Cell Value
Discard	Nonblocking Send Begin		
Discard	Nonblocking Receive Begin		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Vector	Vector*

Parameter Configuration

Parameter	Value
Initial Vector Size	2
Maximum Vector Size	64
Initial Cell Value	0.00

F.2.4 Kiviat:Cnt_Brecv

Module Name: Kiviat:Cnt_Brecv

Description: Kiviat display module.
Displays number of times blocking receive
event seen for each message type.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SVector:Cnt_Brecv			
Action	**Record**	**Field**	**Bound To**
Extract	Count Brecv by Msg Type	Vector	Input

Parameter Configuration

Parameter	**Value**
Number of kiviat axes	2
Minimum value	0.00
Maximum value	2000.00
Vertical axis label	By Type
Horizontal axis label	Count of Blocked Receives

F.2.5 SArray:Sum_Bsend

Module Name: SArray:Sum_Bsend
Description: SynthesizeArray synthesis module.
Build array of total bytes sent by blocking send events.
Indexed by sending node and receiving node.

Operation Selection: Update/Addition

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FilterRecord-1:Filter_Msgs	Index 0
	Index 1
	Cell Value

Specify Output Record Fields

Output Record Name: Sum of BSend From/To	
Input Source	**Output Field**
Module Result	Array

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecord-1:Filter_Msgs			
Action	**Record**	**Field**	**Bound To**
Extract	Blocking Send Begin	Processor Number	Index 0
		Destination Node	Index 1
		Message Length	Cell Value
Discard	Blocking Receive Begin		
Discard	Nonblocking Send Begin		
Discard	Nonblocking Receive Begin		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Array	Array*

Parameter Configuration

Parameter	**Value**
Initial Array Size (Dimension 0)	8
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	8
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

F.2.6 Contour:TotalSize_Bsend

Module Name:	Contour:TotalSize_Bsend
Description:	Contour display unit. Displays total bytes sent by blocking send events. Indexed by sending and receiving nodes

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SArray:Sum_Bsend

Action	Record	Field	Bound To
Extract	Sum of BSend From/To	Array	Input

Parameter Configuration

Parameter	Value
Initial Number of Rows	8
Initial Number of Columns	8
Number of contours	10
Iterations between Updates	25
Vertical axis label	Sending Node
Horizontal axis lable	Receiving Node

F.2.7 SArray:Sum_NBsend

Module Name:	SArray:Sum_NBsend
Description:	SynthesizeArray synthesis module. Build array of total bytes sent by non-blocking send events. Indexed by sending node and receiving node.
Operation Selection:	Update/Addition

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FilterRecord-1:Filter_Msgs	Index 0 Index 1 Cell Value

Specify Output Record Fields

Output Record Name: Sum of NBsends From/To	
Input Source	**Output Field**
Module Result	Array

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecord-1:Filter_Msgs

Action	Record	Field	Bound To
Discard	Blocking Send Begin		
Discard	Blocking Receive Begin		
Extract	Nonblocking Send Begin	Processor Number	Index 0
		Destination Node	Index 1
		Message Length	Cell Value
Discard	Nonblocking Receive Begin		

Module Results

Action	Record	Field	Bound To
		Array	Array*

Parameter Configuration

Parameter	**Value**
Initial Array Size (Dimension 0)	8
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	8
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

F.2.8 Contour:TotalSize_NBsend

Module Name:	Contour:TotalSize_NBsend
Description:	Contour display unit. Displays total bytes sent by non-blocking send events. Indexed by sending and receiving nodes

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SArray:Sum_NBsend

Action	Record	Field	Bound To
Extract	Sum of NBSends From/To	Array	Input

Parameter Configuration

Parameter	Value
Initial Number of Rows	8
Initial Number of Columns	8
Number of contours	10
Iterations between Updates	25
Vertical axis label	Sending Node
Horizontal axis lable	Receiving Node

F.2.9 SArrayElem:Cnt_Bsend

Module Name:	SArrayElem:Cnt_Bsend
Description:	SynthesizeArrayElement synthesis module. Build array containing count of blocking send events. Indexed by sending node and receiving node. Output array indices and element.
Operation Selection:	Update/Count

Bind Input Pipes to Input Ports

Input Pipe	**Input Port**
FilterRecord-1:Filter_Msgs	Index 0 Index 1 Cell Value

Specify Output Record Fields

Output Record Name: Count of B sends From/To	
Input Source	**Output Field**
Module Result	Subscript 0 Subscript 1 Matrix Element

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecord-1:Filter_Msgs			
Action	**Record**	**Field**	**Bound To**
Extract	Blocking Send Begin	Processor Number Destination Node Message Length	Index 0 Index 1 Cell Value
Discard	Blocking Receive Begin		
Discard	Nonblocking Send Begin		
Discard	Nonblocking Receive Begin		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Subscript 0 Subscript 1 Matrix Element	Subscript 0* Subscript 1* Matrix Element*

Parameter Configuration

Parameter	**Value**
Initial Array Size (Dimension 0)	8
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	8
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

F.2.10 Bubble:Cnt_Bsend

Module Name:	Bubble:Cnt_Bsend
Description:	Bubble display module. Displays total number of blocking send events. Indexed by sending and receiving nodes.
Data Input Type:	Array Indices and Element

Bind Input Pipes to Input Ports

Input Pipe	Input Port
SArrayElem:Cnt_Bsend	Index 0
	Index 1
	Cell Value

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SArrayElem:Cnt_Bsend			
Action	**Record**	**Field**	**Bound To**
Extract	Count of B sends From/To	Subscript 0	Index0
		Subscript 1	Index1
		Matrix Element	Array Element

Parameter Configuration

Parameter	Value
Initial Number of Rows	8
Initial Number of Columns	8
Minimum value	0.00
Maximum value	600.00
Vertical axis label	Sending Node
Horizontal axis lable	Receiving Node

F.2.11 SArrayElem:Cnt_NBsend

Module Name: SArrayElem:Cnt_NBsend

Description: SynthesizeArrayElement synthesis module.
Build array containing count of non-blocking send events.
Indexed by sending node and receiving node.
Output array indices and element.

Operation Selection: Update/Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FilterRecord-1:Filter_Msgs	Index 0
	Index 1
	Cell Value

Specify Output Record Fields

Output Record Name: Count of NB Sends From/To	
Input Source	**Output Field**
Module Result	Subscript 0
	Subscript 1
	Matrix Element

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecord-1:Filter_Msgs			
Action	**Record**	**Field**	**Bound To**
Discard	Blocking Send Begin		
Discard	Blocking Receive Begin		
Extract	Nonblocking Send Begin	Processor Number	Index 0
		Destination Node	Index 1
		Message Length	Cell Value
Discard	Nonblocking Receive Begin		

Module Results			
Action	**Record**	**Field**	**Bound To**
		Subscript 0	Subscript 0*
		Subscript 1	Subscript 1*
		Matrix Element	Matrix Element*

Parameter Configuration

Parameter	Value
Initial Array Size (Dimension 0)	8
Maximum Array Size (Dimension 0)	64
Initial Array Size (Dimension 1)	8
Maximum Array Size (Dimension 1)	64
Initial Cell Value	0.00

F.2.12 Bubble:Cnt_NBsend

Module Name: Bubble:Cnt_NBsend

Description: Bubble display module.
Displays total number of non-blocking send events.
Indexed by sending and receiving nodes.

Data Input Type: Array Indices and Element

Bind Input Pipes to Input Ports

Input Pipe	**Input Port**
SArrayElem:Cnt_NBsend	Index 0
	Index 1
	Cell Value

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SArrayElem:Cnt_NBsend			
Action	**Record**	**Field**	**Bound To**
Extract	Count of NB Sends From/To	Subscript 0	Index0
		Subscript 1	Index1
		Matrix Element	Array Element

Parameter Configuration

Parameter	**Value**
Initial Number of Rows	8
Initial Number of Columns	8
Minimum value	0.00
Maximum value	600.00
Vertical axis label	Sending Node
Horizontal axis lable	Receiving Node

F.2.13 SVector:Cnt_NBrecv

Module Name: SVector:Cnt_NBrecv

Description: SynthesizeVector synthesis module.
Builds vector with count of non-blocking receive events;
indexed by message type

Operation Selection: Update/Count

Bind Input Pipes to Input Ports

Input Pipe	Input Port
FilterRecords-1:Filter_Msgs	Index
	Cell Value

Specify Output Record Fields

Output Record Name: Count of NB Recv by Msg Type	
Input Source	**Output Field**
Module Result	Vector

Bind Records To Input Ports and Output Record Fields*

Input Pipe: FilterRecords-1:Filter_Msgs			
Action	**Record**	**Field**	**Bound To**
Discard	Blocking Send Begin		
Discard	Blocking Receive Begin		
Discard	Nonblocking Send Begin		
Extract	Nonblocking Receive Begin	Message Type	Index
		Message Length	Cell Value

Module Results			
Action	**Record**	**Field**	**Bound To**
		Vector	Vector*

Parameter Configuration

Parameter	Value
Initial Vector Size	2
Maximum Vector Size	64
Initial Cell Value	0.00

F.2.14 Kiviat:Cnt_NBrecv

Module Name:	Kiviat:Cnt_NBrecv
Description:	Kiviat display module. Displays number of times non-blocking receive event seen for each message type.

Bind Records To Input Ports and Output Record Fields*

Input Pipe: SVector:Cnt_NBrecv

Action	Record	Field	Bound To
Extract	Count of NB Recv by Msg Type	Vector	Input

Parameter Configuration

Parameter	Value
Number of kiviat axes	2
Minimum value	0.00
Maximum value	2000.00
Vertical axis label	By Type
Horizontal axis label	Count of Non-Block Recvs

G Description of Parallel PDE Solver

This appendix contains a description of the parallel PDE solver that was executed on an eight-node Intel iPSC/860 to obtain the performance trace file used in tutorials five and six. The procedure call behavior and message communication patterns that show up in the Pablo displays of those tutorials are directly related to the algorithms used in the code. Thanks to Daniel A. Reed for contributing the remainder of this appendix.

G.1 Statement of Problem

Consider a simple elliptic partial differential equation (PDE) that describes the steady state temperature distribution on a metal plate:

$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = 0$$

$$u(x,y) = g(x,y) \qquad\qquad \forall x, y \in R$$

where R is the plate boundary and $g(x,y)$ is the temperature distribution on the boundary. Ideally, we would like to obtain a function describing the temperature distribution. Unfortunately, in the general case this is not possible, and one must settle for a specific, numerical solution at a fixed set of points. For simplicity's sake, assume the metal plate is square; we then seek to obtain the temperature at each of N^2 points in an $N \times N$ grid.

G.2 Sequential Algorithm

Intuitively, the value of each grid point must be related to that of its neighboring points (e.g., the average of the north, south, east, and west neighbors), and a temperature gradient must propagate from the boundaries across the grid until a steady state is reached. As an example, Figure 1 and Figure 2 respectively show the simple boundary conditions xe^y and the resulting steady state solution. This suggests the following simple iterative algorithm.

Given an initial guess for each of the N^2 points, substitute these guesses in each of the N^2 equations describing each point's value as the average of it's neighbors. Evaluate each equation, obtaining new guesses. Repeat this process, using the new guesses, until the difference between successive guesses for *all* points differs by less than a specified ϵ.[20] The core of a sequential implementation of this iterative algorithm is shown below.

```
Converged = TRUE;

for (i = 0; i < N; i++)
   for (j = 0; j < N; j++) {
      NewGrid[i][j] = 0.25 * (Grid[i-1][j] + Grid[i+1][j] +
                              grid[i][j-1] + Grid[i][j+1]);

      if (fabs(NewGrid[i][j] - Grid[i][j]) > EPSILON)
         Converged = FALSE;
   }
```

[20]This is a horribly inefficient way to solve this problem — there are much faster solution techniques. However, our goal is to illustrate Pablo's functionality, not efficient numerical techniques.

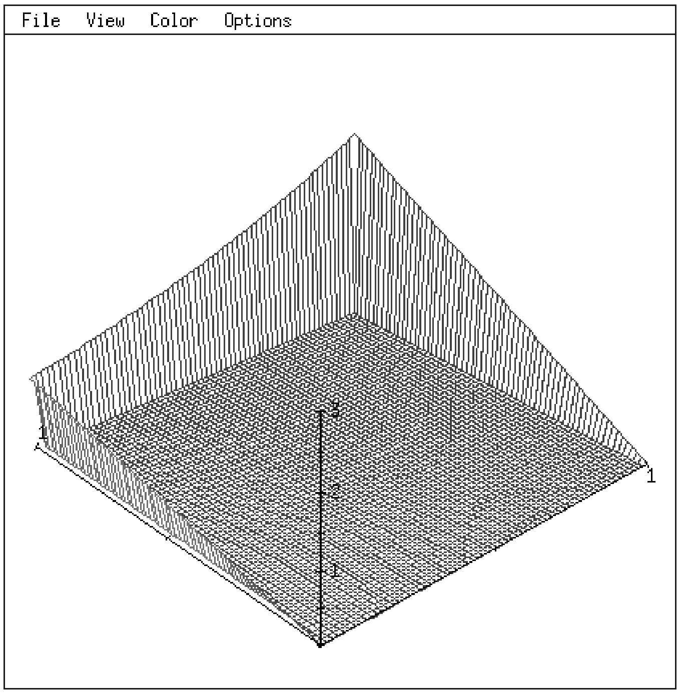


Figure 1: Boundary Conditions for Plate

G.3 Parallel Algorithm

The example program provided with the Pablo trace library includes a parallel version of this iterative solution technique for the Intel iPSC/860 distributed memory parallel system. Given P processors, this parallel algorithm partitions the grid of points into P horizontal strips ($P \leq N$) containing physically adjacent points, and each processor updates its subgrid, sharing grid rows with the two neighboring processors that contain logically adjacent strips.

On a parallel system, global convergence checking requires data sharing among the processors. Although there are many possible global convergence checking algorithms, the example program implements a simple, logarithmic time algorithm that relies on the hypercube topology of the Intel iPSC/860. First, each of the P processors computes the convergence of its local subgrid. Thereafter, the local convergence flags are combined in $\log P$ steps.

As Figure 3 shows for a three-dimensional cube, at each step, half the remaining processors send their convergence value to their neighbors. These processors "and" the just received convergence value with their local value and send the result to a neighbor in another dimension. After a logarithmic number of phases, processor 0 has the global convergence value, and it is distributed via an inverse process (i.e., a broadcast in logarithmic time).

The example program relies on the Intel iPSC/860's synchronous (`csend`/`crecv`) and asynchronous (`isend`/`irecv`) communication routines to maximize the overlap of grid update computation and data sharing. In particular, all sharing of grid partition boundaries is via asynchronous `isend` routines. This allows the update to the partition to occur in parallel with the transmission of partition boundaries to neighboring processors. In addition, the partitions are mapped onto processors using a gray coded ring. This ensures that all data sharing occurs between physically

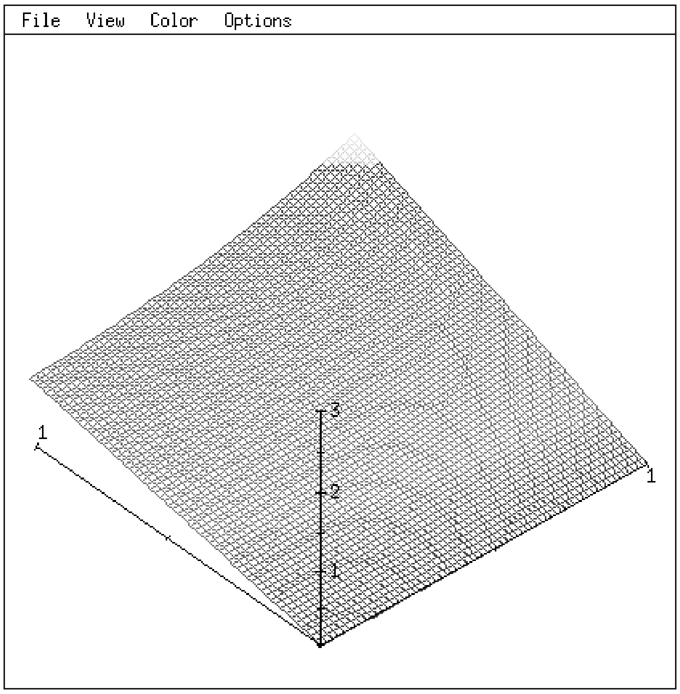


Figure 2: Steady State Solution for Plate

connected processors. For example, a gray code for an eight node, three-dimensional hypercube is `0 -> 1 -> 3 -> 2 -> 6 -> 7 -> 5 -> 4`; each adjacent node number differs by only one bit in its binary representation.[21] Similarly, the global convergence checking code uses the previously described logarithmic condensation algorithm. On an eight node, three-dimensional hypercube, the condensation pattern is that shown in Figure 3; this is also visible in the non-blocking communication pattern of the Pablo displays.

The need for data sharing and global convergence checking, create an interesting mix of computation and communication. The grid update messages are of size $O(N)$, and each processor must send (and receive) two such messages each iteration. Similarly, the grid update cost is $O(\frac{N^2}{P})$ and the cost of global convergence checking is $O(log_2 P)$ with unit size messages.

[21]This communication pattern will be visible in the Pablo blocking send message passing displays.

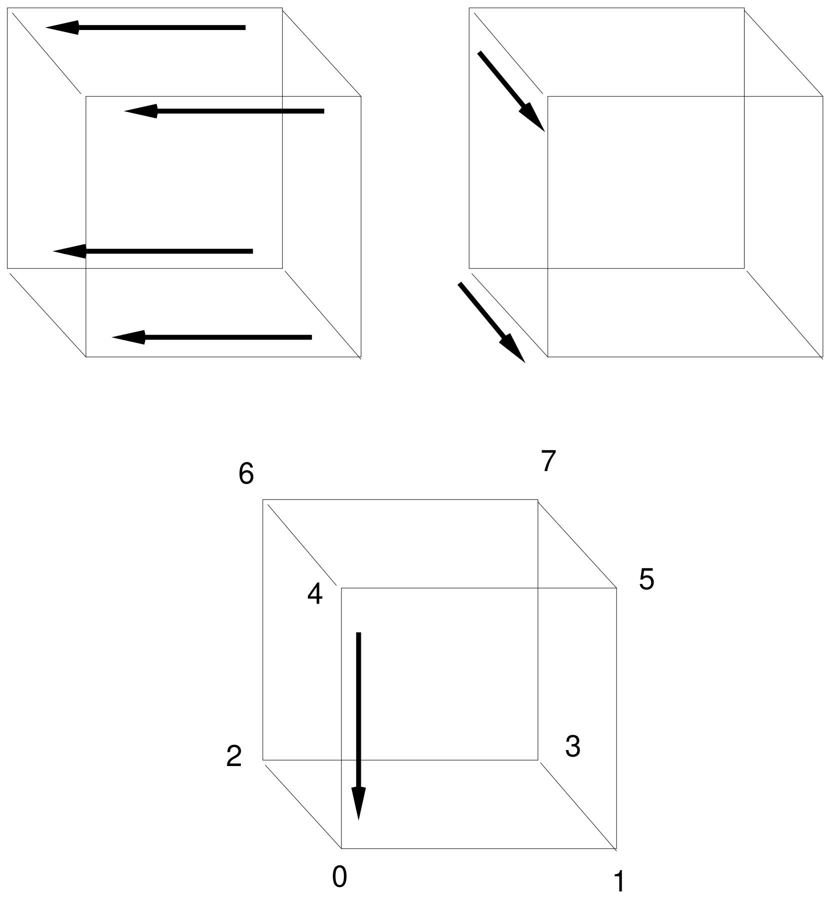


Figure 3: Global Convergence Algorithm