CODE 2.0 User and Reference Manual

March 24, 1993

Peter Newton

Seema Y. Khedekar

1.0 Introduction

CODE is a graphical parallel programming environment. The idea behind it is that users can create parallel programs by drawing and then annotating a picture. This picture is then automatically translated by the CODE system into a parallel program that can be compiled and run on a target parallel machine such as the Sequent Symmetry.

CODE programs are graphs (of the nodes and arcs variety). Graphs are an excellent vehicle for parallel programming because they directly display both parallel operations and communications structure. Nodes represent operations and arcs represent relationships among them, primarily showing dataflow. The next chapter provides an overview of CODE.

1.1 CODE's Goals

Wide acceptance of parallel architectures has been hindered by the lack of effective tools for programming them. CODE represents an attempt to ameliorate this problem. Parallel programming environments must satisfy three practical goals in order to be effective. They must be easy to use so that non-specialists can write programs. They must permit a wide variety of programs to be expressed in a reasonably portable notation so that programs are not bound to a single architecture, and they must produce efficient executable structures. The purpose of parallel computing is speed, after all. CODE integrates many technologies and ideas in order to satisfy these goals.

• <u>Ease of Use</u>

CODE simplifies parallel programming by giving users a graphical interface in which they can express programs using a very abstract computational model. Programming is reduced to drawing pictures and then declaratively annotating them. There are no low-level primitives to learn, only declarative annotations. The CODE programming model combines aspects of both dataflow and shared-memory programming styles so algorithms that are biased towards either can be expressed directly. The model is also designed to support reuse of program components at all levels.

• <u>Portability</u>

The CODE model is the result of compromise. It is as expressive as its designers could make it while containing no features that preclude implementation on either shared or partitioned memory address space architecures. The model is designed to be implementable on all common MIMD machines. Of course, there is no magic bullet to solve the portability problem. Some algorithms are inherently biased towards a particular architecture. CODE reduces this problem by allowing programs to be expressed in an abstract notation that is not closely bound to any one architecture.

• <u>Efficiency</u>

The CODE computational model is designed to be as expressive as possible without being difficult to compile. Traditional sequential compilers have utilized optimization technology for decades. CODE attempts to use similar technologies but applies them at a higher level of abstraction. CODE is implemented using object-oriented programming techniques that allow optimizations to be added easily. CODE's designers hope that performance-critical segments will be idiomatic. Optimizations can be added over time to cover common programming paradigms.

1.2 Overview of this Manual

This document consists of two major pieces, the User Manual chapters and a Reference Manual that is included as a substantial appendix. The User Manual sections are intended to be introductory. They attempt to present the most important aspects of CODE clearly and with many examples. There is no attempt to be complete-- only the most important points are covered. The Reference Manual defines the complete language. It is intended for those who already grasp the essential elements of CODE.

Those who wish to learn the CODE system may do well to adopt the following plan.

1. Read chapters 1 and 2 and to get an overall view of how CODE works. Then read appendix 1. (Do not skip this appendix!)

2. Sit down at a workstation and actually run CODE while reading chapter 4. When a user interface feature is mentioned, try it. The goal in chapter 4 is not to create programs, it is to manipulate the user interface and draw pictures. When you can draw graphs, and open and fill out attribute forms with random text and menu choices, you are ready to move on to the next step.

3. Remain at your computer and work through the tutorial examples in chapter 5. These will guide you through every step in creating and running CODE programs. You will want to exit CODE and restart it when you begin chapter 5.

4. Skim through the rest of the User Manual chapters.

5. Start programming for real, refering to the Reference Manual sections in the appendix when you get stuck.

1.3 CODE Distribution and Support

At this time, there are no fixed mechanisms or policies for the distribution and support of CODE. You may try sending email (in preference order) to:

code@pompadour.csres.utexas.edu
newton@cs.utexas.edu (Peter Newton)
bwest@cs.utexas.edu (Brian West)
browne@cs.utexas.edu (James C. Browne)

2.0 Overview of the CODE Language

CODE is a graphical parallel programming language. You create parallel programs with it by drawing and annotating pictures. When you first run CODE, an empty window will appear on your workstation screen. You draw pictures into this window using a mouse.

2.1 An Example Program

A graph is a type of diagram that consists of nodes (represented by icons in CODE) and arcs that interconnect the nodes. CODE pictures are graphs in which there are several node types with different meanings and uses. Each different node type is represented by a different icon. The most important type of node is the Unit of Computation (UC) node. These represent sequential computations that can be composed into parallel programs by drawing arcs that show dataflow among them. Figure 2.1 shows a complete CODE program that has been created using only UC nodes and dataflow arcs.

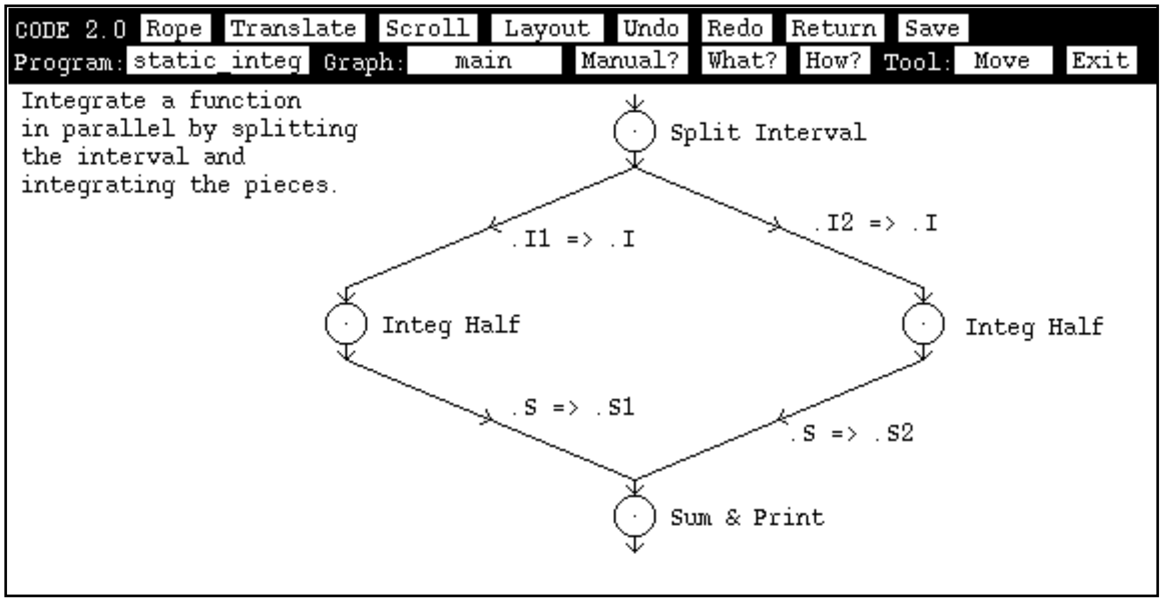
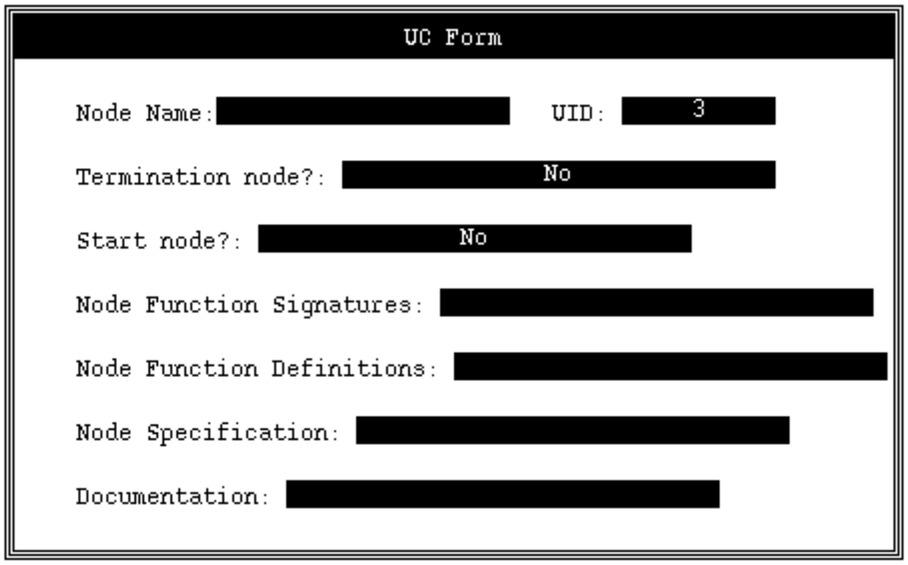


Figure 2.1: A CODE Program

The program integrates a function over an interval by dividing the interval in half and integrating over each piece separately. The results are summed to form the final answer. This program consists of four UC nodes. The arcs represent information that is created by one UC and is needed before another UC can begin its sequential computation. When the program runs, it first executes (or fires) the UC called “Split Interval” which divides the interval in half and sends the endpoints of the sub-intervals to the “Integ Half” nodes. These do the integration and can run in parallel since there is no arc from one to the other. Node “Sum & Print” waits for results from the parallel integrations and then adds them together and prints this sum. The arcs represent unbounded FIFO queues of data.

Drawing the graph is only one part of creating a CODE program. You also have to annotate it.

These annotations define many aspects of the program-- what sequential computation a UC node will perform, under what conditions it can fire, what data types are defined for the program, etc. Annotatations are performed by filling out attribute forms associated with the object being defined. Figure 2.2 shows the attribute form associated with a UC node.


UC Form

Node Name: UID: 3

Termination node?: No

Start node?: No

Node Function Signatures:

Node Function Definitions:

Node Specification:

Documentation:

Figure 2.2: UC Node Attribute Form

Notice the fields for “Terminate node?” and “Start node?”. The programmer must designate exactly one UC node in the program to be the Start Node and exactly one node to be the Terminate node. When a CODE program is run the system first executes the Start Node to get the computation started. The firing of the Terminate Node signals the end of the computation to the CODE system.

Most of the attributes of a UC node are supplied by filling in stanzas of text in the Node Specification field of the attribute form. These stanzas are written in a small mostly declarative textual programming langauge. The annotations of the four nodes in the integration program are described below.

```
// ***** Node Split Interval ***********
output_ports { IntegInfo I1; IntegInfo I2; }
vars { real a; real b; int n; IntegInfo i1; IntegInfo i2; }
comp {
   a = ReadReal();
   b = ReadReal();
   n = ReadInt();
   i1.a = a;
   i1.b = (b - a)/2.0;
   i1.n = n/2;
   i2.a = i1.b;
   i2.b = b;
   i2.n = n - i1.n;
}
routing_rules {
   TRUE => I1 <- i1; I2 <- i2;
}
```

Consider Split Interval's annotations. Output ports are a node's private names for arcs that leave it. Input ports are a node's private names for arcs that enter it. More on this later. For now, it is enough to realize that two arcs that it calls I1 and I2 leave Split Interval. The type name of the data that will be placed on these arcs is IntegInfo, a structure that contains the endpoints of an interval and the number of points to use in the integration.

The Vars stanza of a node defines local variables that are inside the node and are used by its sequential computation.

The Comp stanza defines the sequential compuation. Note the calls to functions ReadReal and ReadInt. These functions are ordinary sequential functions written in C. It is expected that all substantial sequential computations will be encapsulated in such ordinary functions that are defined outside of CODE (although CODE has facilities to help in managing, placing, and compiling them).

Since Split Interval is the Start Node, its computation will be run immediately when the program is executed. After it is complete, the routing rules are evaluated. Routing rules determine what values will be placed on what outgoing arcs and have the form shown.

```
   Guard, Guard, ..., Guard => Binding; Binding; ...; Binding;
```

Guards are Boolean expressions and Bindings are either assignment statements or "Arc Outputs" that use a "<-" operator to place a value onto an arc. All bindings are performed from left to right on all routing rules whose guards all evaluate to TRUE. Node Split Interval places struct i1 onto arc I1 and struct i2 onto arc I2. These define the intervals that the Integ Half nodes will integrate.

Both of the Integ Half nodes have identical attribute sets. Only one of them must be discussed, and the only new stanza is the Firing Rules Stanza.

```
   // ******* Both Node Integ Half are identical *********
   input_ports { IntegInfo I; }
   output_ports { real S; }
   vars { IntegInfo i; real val; }
   firing_rules {
      I -> i =>
   }
   comp {
      val = simp(i.a, i.b, i.n);
   }
   routing_rules {
      TRUE => S <- val;
   }
```

Firing rules serve two purposes. They define the conditions under which a UC node is allowed to fire, and they describe what arcs have values removed from them and placed into local variables for use by the node's sequential computation. Firing rules have the same general form as routing rules.

```
        Guard, Guard, ..., Guard => assign_stmt; assign_stmt; ...assign_stmt; ;
```

However, guards represent either Boolean expressions or “Arc Inputs” that extract information from arcs using the “->” operator. Only assignment statements appear on the left hand side.

Arc inputs represent both a condition and a binding. The notation “I -> i” represents the condition or guard “there is a value on arc I” and the binding “remove a value from arc I and place it into variable i”. A UC node may execute whenever all of the guards on any of its firing rules are TRUE. Such a rules is said to be “satisfied”. When the node fires, the bindings and assignment statements associatedwith the satisfied rule are performed. Hence, Integ Half may fire whenever there is a value on arc I. It extracts a value from I and places it into variable i for use by the nodes’ computation.

It is possible for a node to have many firing rules. As indicated above, the node may fire when any of them are satisfied. If more than one rule is satisifed, the CODE system chooses one of the satisfied rules arbitrarily and performs its bindings before firing the node.

So, the nodes Integ Half wait for a value on arc I, perform an integration described by the interval received, and then pass the result out on arc S.

Node Sum & Print has a firing rule that requires it to wait for values on both arcs S1 and S2 before it can fire. When it fires, it prints the sum of the values received and, since it is the Terminate Node, signals the end of the computation.

```
        // ******* Node Sum & Print **********
        input_ports { real S1; real S2; }
        vars { real s1; real s2; }
        firing_rules {
           S1 -> s1, S2 -> s2 =>
        }
        comp {
           PrintReal(s1+s2);
        }
```

The last issue in the Integration program is the annotation of arcs. The annotations are shown in figure 2.1 and are called “arc topology specifications”. They serve to bind names between nodes. As mentioned above, nodes use ports as local names for arcs. Arc topology specifications bind port names together. For example, the specification

```
        .I1 => .I
```

indicates that output port I1 is to be bound to input port I. When you draw an arc between UC nodes, you must specify what pair of ports the arc binds.

It is reasonable to think of UC nodes as being analagous to integrated circuits. The port names of the UC serve the same purpose as the pin names on the IC. You place UCs into a graph in any way you like and connect them with arcs rather than wires. The arc topology specification describes what “pins” have been connected.

2.2 CODE Objects Are Actually Templates

It is time to confess that the description of CODE so far, although completely correct, has been oversimplified. The notations as they have been described are inadequate as the basis for a programming system because they are static. The graph drawn represents a completely fixed computation and communication structure. Many real-world algorithms do not fit into this view. Aspects of their structure depend on runtime information. As a simple case, one may wish to prepare a program that will utilize as many processors as happen to be available at a particular moment to perform a certain task. Perhaps the desired structure is as shown in figure 2.3 where there are N

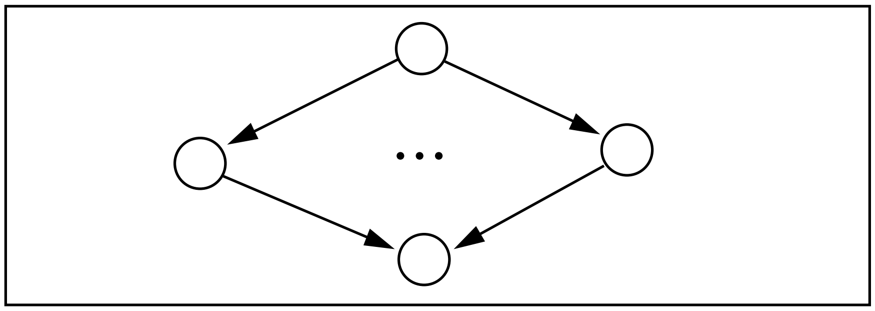


Figure 2.3: A Dynamic Computation Structure

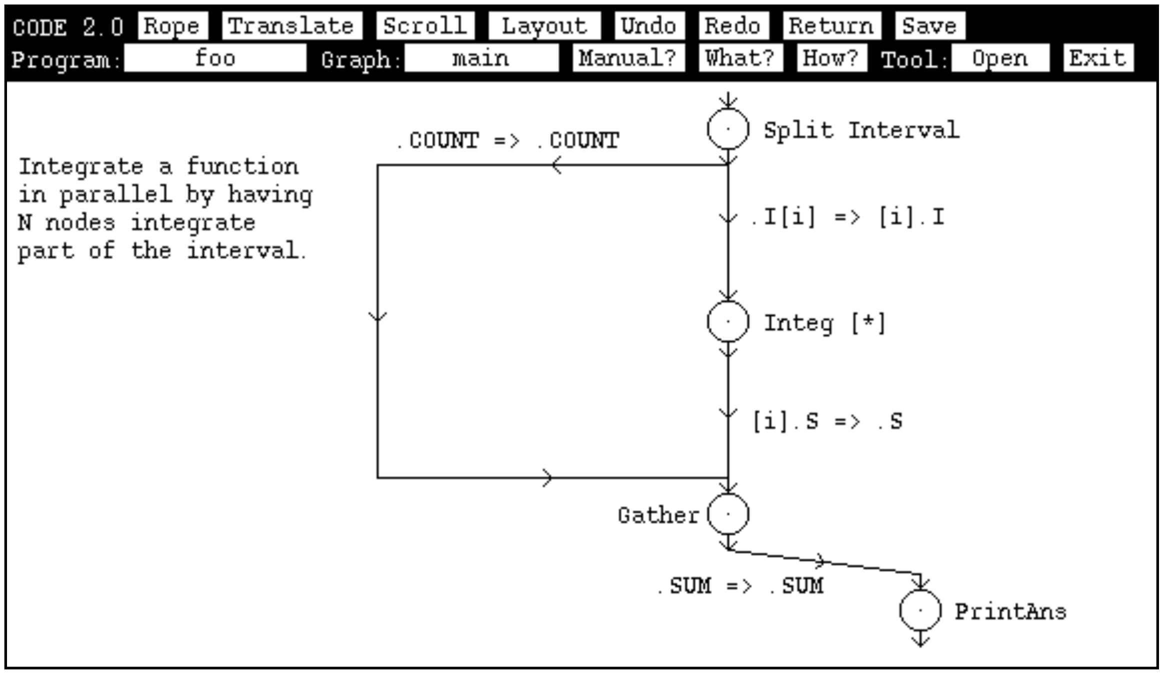
replicated nodes in the center, where N is a runtime determined value. Structures that are dictated by runtime parameters are called “dynamic”.

CODE directly supports the specification of dynamic structures. Rather than being static, every node or arc users draw in CODE can be instantiated any number of times at runtime. The instantiations are named by integer valued indices. Hence, it is more accurate to sat that you are creating a *template* for a node rather than a node itself when you draw and annotate a UC node as described above.

The simplest way to use the template is to use the instantiation of it that has no indices. This is what was done in the Integration program.

The Integration program is limited to two-way parallelism. There are only two Integ Half nodes to run in parallel. An N-way parallel program can easily be envisioned that would use N “Integ” nodes each of which integrates a fraction of the interval of size (b - a)/N. A CODE program that implements this scheme is shown in figure 2.4. The node Integ is instantiated N times using the structure shown in figure 2.3.

The “[*]” by the name is a comment reminding the viewer that the node is instianted multiple times. In fact, all parts of UC node name serve only as comments and are optional attributes.



Integrate a function
in parallel by having
N nodes integrate
part of the interval.

Figure 2.4: Another Integration Program

The multiple instantiation is specified by annotations that will be described in later chapers. The important idea to remember now is that all CODE objects you draw are in reality templates that may be instantiated many times, with the different instantiations named by zero to seven integer valued indices. The graph depicts static aspects of the program. Annotations capture dynamic aspects.

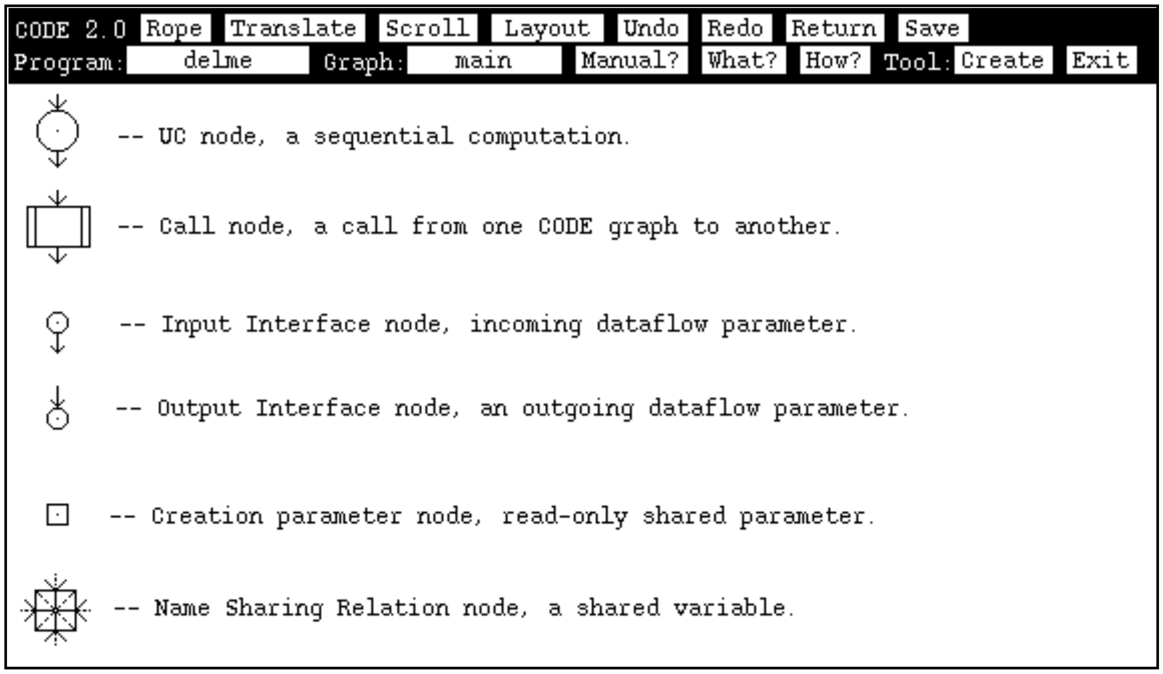
2.3 Other CODE Nodes

There are other node types than UC nodes in CODE. Many of these serve to heirarchically structure programs, exactly as subprograms and Call statements do in conventional languages such as C or Fortran. Figure 2.5 shows all of CODE's node types.

CODE graphs play the role of subprograms. In general, CODE programs consist of many graphs that interact by means of Call nodes. The Integration program is simply a single graph program.

Just as with subprograms in conventional languages, CODE graphs have formal parameters. There are defined by Interface nodes and Creation Parameter nodes. Actual parameters are arcs that enter or leave Call nodes. These arcs are bound (by means of arc topology specifications) to Interface or Creation Parameter nodes within the called graph. Graph calling and parameter binding will be fully described in a later chapter.

Name Sharing Relation nodes introduce shared variables. UC nodes that access them must declare themselves to be either readers or writers of the variable. This too will be described later.



-- UC node, a sequential computation.

-- Call node, a call from one CODE graph to another.

-- Input Interface node, incoming dataflow parameter.

-- Output Interface node, an outgoing dataflow parameter.

-- Creation parameter node, read-only shared parameter.

-- Name Sharing Relation node, a shared variable.

Figure 2.5: All CODE Node Types

2.4 Translating and Running CODE Programs

Once a CODE program has been drawn and annotated, the CODE system can translate it into a program that can be compiled and run on a parallel machine. This process will be described in a later chapter but the general idea is that the user clicks on the translate button at the top of the CODE window and picks a target machine type, such as the Sequent Symmetry. CODE will produce a program in C that realizes the computation and communications structures expressed by the graph. The user may then compile and run this program on the parallel machine.

Users are offered choices on how CODE will optimize and instrument the program. For example, one can ask CODE to automatically have the program be instrumented to measure how long each UC takes to fire and how often it fires.

3.0 Hardware Requirements

CODE runs on Sun 4 (Sparc) workstations, either color or monochrome. It requires approximately 3 MB of disk space. You must have X-windows installed since CODE runs under it. You may use any window management system including OpenWindows and Motif.

At this time, CODE has a translator only for the Sequent Symmetry parallel processor. Hence, you will need access to such a machine to run any CODE programs you create.

It is most convenient if you can store your programs on a disk that is mounted on both the parallel machine and the Sun workstation. In this case, you can simply switch to a window on the workstation in which you have logged into the parallel machine and type “make” to compile the parallel program generated by CODE. It is better but not necessary to have the disk physically connected to the parallel machine rather than the Sun workstation. This configuration maximizes I/O performance for running parallel programs.

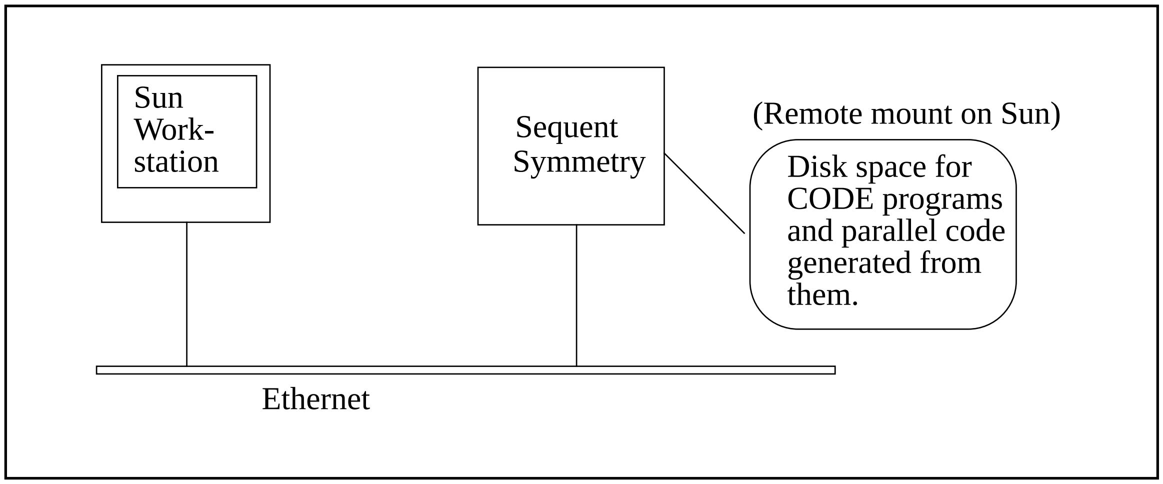

Sun Work-station
Sequent Symmetry
(Remote mount on Sun)
Disk space for CODE programs and parallel code generated from them.
Ethernet

Figure 3.1 - Ideal Hardware Configuration

Installation instructions are distributed with the CODE software.

4.0 The CODE User Interface

This chapter describes how to use CODE's graphical interface. It explains how to perform the tasks you must know how to do in order to program in CODE. For example, you must be able to draw nodes and arcs and fill out attribute forms. Perhaps the best way to learn is to run CODE and try the operations described below as you read about them. To run CODE, enter

```
code2 progname.grf
```

at the UNIX prompt. This will run CODE and open a program called "progname", creating it if necessary. Of course, code2 must be on your UNIX path. Also, it is possible that you will need a configuration file called ".coderc" in your home directory in order to set a path variable that CODE requires. See the code2 UNIX man page and installation instructions. You must be running some form of X-windows to use CODE. We have run it under X11R4, X11R5, and OpenWindows.

When CODE starts, it opens a single window. All activity will take place within it. The window is divided into two parts, a black banner line full of buttons and labels at the top, and an area of empty space in which you will draw graphs. Figure 4.1 shows the CODE window. Several of the buttons and the banner line have been labeled. All operations involve clicking (press the mouse button and release without moving the mouse) on some part of the window. Only a single mouse button is needed. Use the left button on multiple button mice. As a note to Macintosh users, CODE does not use double clicking or dragging.

CODE's user interface can be considered as two separate components. There are facilities for drawing graphs and facilities for annotating graphs by filling out forms. They will be discussed separately.

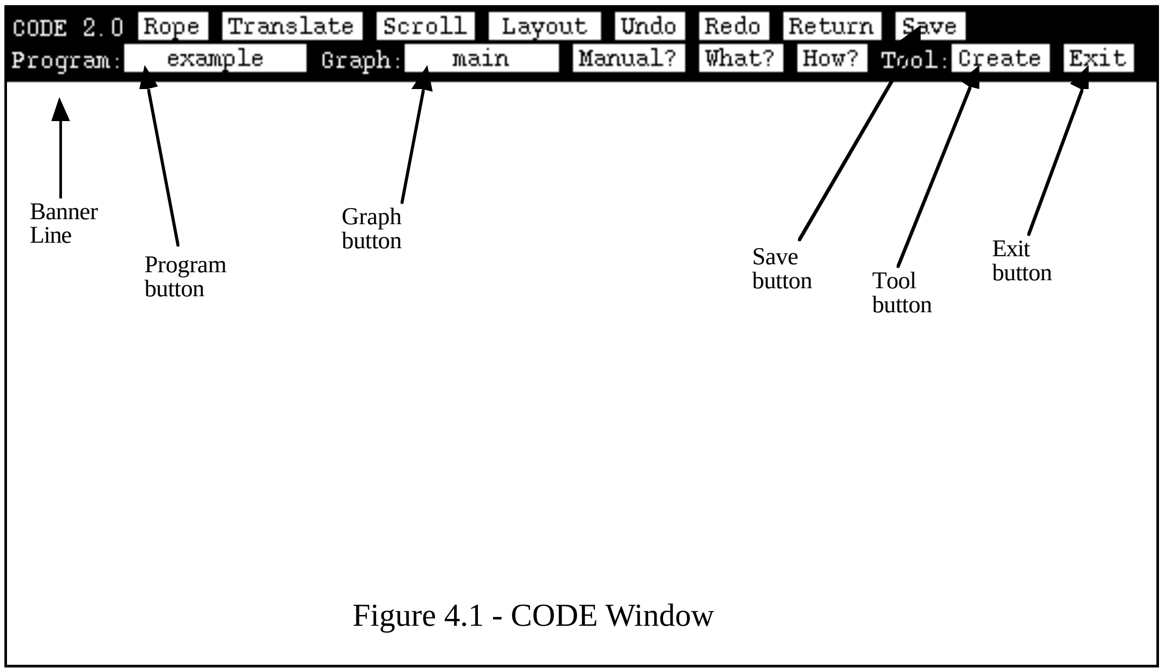


Figure 4.1 - CODE Window

4.1 Drawing Graphs

CODE provides a numbers of “tools” for drawing graphs. There is a tool for creating nodes and arcs, a tool for moving them, a tool for deleting them, etc. Each tool has a cursor associated with it. Figure 4.2 shows the CODE cursors. Their names and uses are as follows.

• Create	Creates objects such as nodes, arcs, and graphs.
• Open	Opens attribute forms of objects.
• Copy	Permits nodes, arcs, and graphs to be copied.
• Travel	Permits display of different graphs and programs.
• Move	Moves nodes, arcs, etc. on the screen.
• Erase	Deletes objects. Use the Undo button to bring them back.
• Square	Changes arcs to run only vertically and horizontally. Click on an arc to square it.

The default cursor is the Create cursor. When you first run CODE, this cursor will be active. It is shaped like a pencil and is used to create (meaning draw) things.

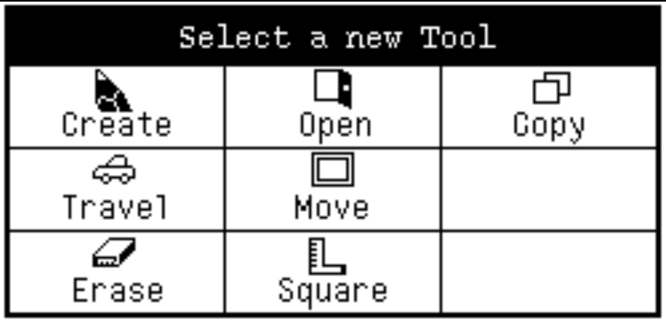


Figure 4.2 - CODE Cursor Tools

You change from one cursor to another by clicking (left button) on the Tool button in the banner line. A menu of tools will appear. You click once to make the menu appear and then click again to make a choice from it.

There is also an optional shortcut for changing tools. Click the right mouse button. A graphical menu of tools will appear. This is the only use CODE makes of the right button. The middle button is never used.

You draw a node by clicking on empty space with the Create cursor. A menu of node types will pop up. Pick one by clicking on it. A node will appear where you first clicked. Remeber to click with the left mouse button.

Draw an arc by clicking the create cursor on the tip of an outgoing arrow on a node. Click again on the end of an incoming arrow to complete the arc. See “Drawing an Arc” in section 4.3.

Click the Move cursor on an object to move it. Click the Erase cursor on an object to delete it, and so on.

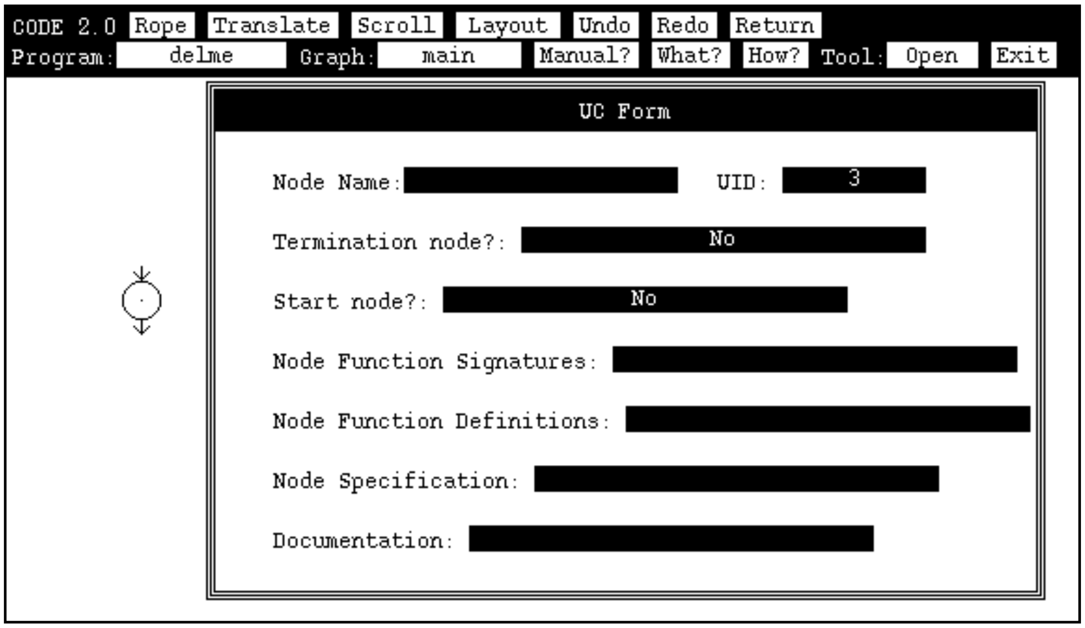
CODE programs actually consist of a set of graphs, not just one. You draw and work on them one at a time. When you first run it, CODE creates a graph with the default name “main.” You can create a new graph by clicking the Create cursor on the Graph button and entering a name for the new graph. You can view a different graph in the set of graphs you have created by clicking the Travel cursor on the Graph button. A menu (with scroll bars in case you have many graphs) will appear.

When you click the Open cursor on an object, its attribute form appears. These are discussed in the next section. For now, note that attributes that apply to the entire program are (logically enough) in the Program attribute form. Open it by clicking the Open cursor on the Program button. Open the current graph’s attribute form by clicking the Open cursor on the Graph button.

Notice that the tools have a faintly object-oriented flavor. They do the right thing when you click them on different parts of the screen. The program button represents the entire problem. You open the program’s attribute form by clicking the Open cursor on the Program button. Make a form go away by clicking on empty space outside of it. You can travel to (load) another CODE program by clicking the Travel cursor on the Program button. The Graph button represents the currently displayed graph. Click the Open cursor on it to open the graph’s attribute form. Click the travel cursor on it to display a different graph. Click the Create cursor on it to make a new graph, and so on.

4.2 Annotating Graphs Via Forms

Once you have drawn a graph, you must annotate it. As has been mentioned, you click the Open cursor on an object to open its attribute form. This works for nodes, arcs, the Program button, and the Graph button. Figure 4.3 show a UC node’s attribute form.



Node Name:

UID: 3

Termination node?: No

Start node?: No

Node Function Signatures:

Node Function Definitions:

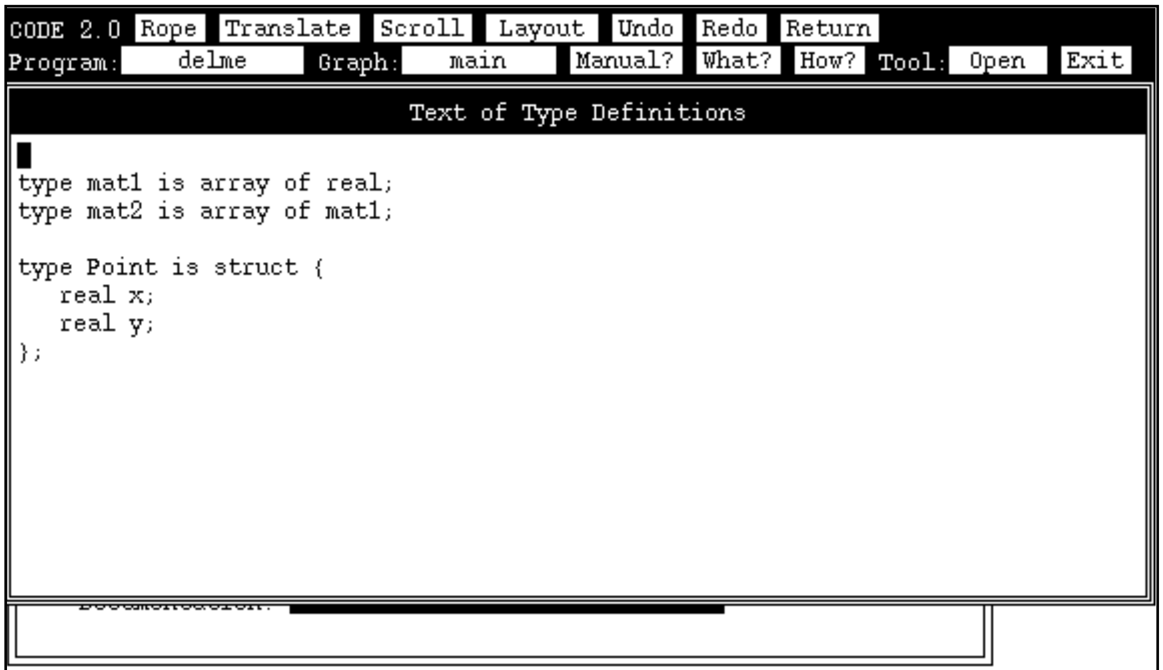
Node Specification:

Documentation:

Figure 4.3 - UC Node Attribute Form

Forms are hierarchical. They contain fields that are references to other things called entry methods. These include forms, menus, scrolling tables, and text editor boxes. For example, the Node Name field is a small text editor box. Click on it, and a text box will appear. You can enter a name for the node in this box. The name will be displayed next to the node, but you can move it with the Move cursor. You make forms, menus, text boxes, etc. go away by clicking on something else. You go to what you clicked on. After you finish entering a node name, click on white space in the UC form to return to it. Click on empty space outside the form to exit it altogether.

Figure 4.4 shows a large text box open on top of a form. You may move an entry method by clicking in its title bar and dragging. This is useful in case you need to slide something out of the way. CODE’s text editor boxes use emacs key bindings, although only a small subset are supported. For large editing tasks, press ESC-e (the escape key followed by e) to bring up a window running the editor defined by your EDITOR environment variable. You may also define this editor in your “.coderc” file using the “editor” variable.



```
type mat1 is array of real;
type mat2 is array of mat1;

type Point is struct {
   real x;
   real y;
};
```

Figure 4.4 - A Text Box

Here is a summary of some of the important key bindings used in text editor boxes. The symbol “^” means to hold down the control key. A “region” is the text between the mark and the current location of the cursor.

^f	Forward character	^b	Backwards character
^n	Next line	^p	Previous line
^a	Beginning of line	^e	End of line
^v	Next page	ESC-v	Previous page
^z	Scroll forward one line	ESC-z	Scroll back one lne
^d	Delete character	DEL	Delete character to the left.

^k	Delete to end of line^@Set mark at cursor
^w	Delete to region to yank bufferESC-wCopy region to yank buffer
^y	Paste at cursor

Click the left mouse button to position the cursor.
Click outside the form to exit it.

4.3 Specific Operations

These section describes many of the important operations you will have to be able to do in order to use CODE. It also summarizes some of the information above.

Leaving CODE

To leave CODE, click any cursor on the Exit button. You will be asked if you wish to save changes.

Saving Changes Without Leaving CODE

Click the Create cursor on the Program button. A menu will appear that has save as an option. Do this often as protection against crashes.

Drawing a Node

Click the Create cursor on empty space. Pick a node type from the menu that will appear. Figure X.X shows the different node types. Figure 4.5 shows the menu.

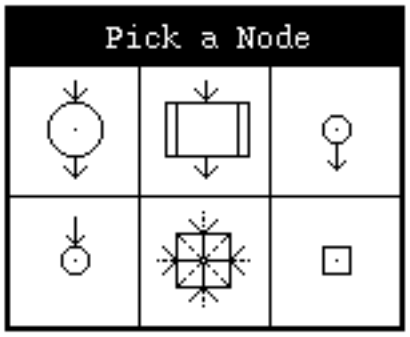


Figure 4.5 - Node Menu

Drawing an Arc

To create an arc, begin by clicking (press and release!) the Create cursor on the tip of an arrow leaving a node. You are now in arc drawing mode. Click on empty space to create intermediate points (elbows). Click on the beginning on an incoming arrow on a node to complete the arc. Click on the point at which you started the arc to cancel drawing it, or you can finish it and then delete it with the Erase cursor. It is necessary to aim the mouse with some care when you are drawing arcs, but it is easy after a little practice. Figure 4.6 show points on a node at with an arc can be started and finished.

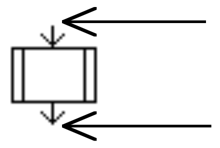

Click at the beginning of an in-arrow to complete an arc.
Click at the end of an out-arrow to start an arc.

Figure 4.6 - Arc Drawing Points

Creating a New Graph

Click the Create cursor on the Graph button. Select “create” from the menu and enter a name for the graph.

Creating a New Program

Exit CODE and start it again using the program name as the command line argument.

Opening an Attribute Form

Click the Open cursor on the object-- node, arc, Program button, or Graph button. When clicking on an arc, avoid clicking on its intermediate points. Click outside of the form to close it. There will be a beep when you close a the form if you have made a syntax error within it. Semantic errors are not caught until you translate your program.

Changing Cursor Tools

Click any Cursor on the Tool button or click the right mouse button.

Scrolling the Graph View Area

What do you do if your graph is too large for the CODE window? The best advice is to use more graphs and Call nodes to hierarchically structure your program. You can also use your X window manager to resize the CODE window, or you can scroll by clicking any cursor on the Scroll button. A scroll window will appear. Use its scroll bars to scroll the graph view one node at a time. The scroll window itself shows a zoomed-out view of the CODE graph. Every node is represented by a single pixel. Click in this area to jump to a particular part of the graph. Figure 4.7 shows the scroll window.

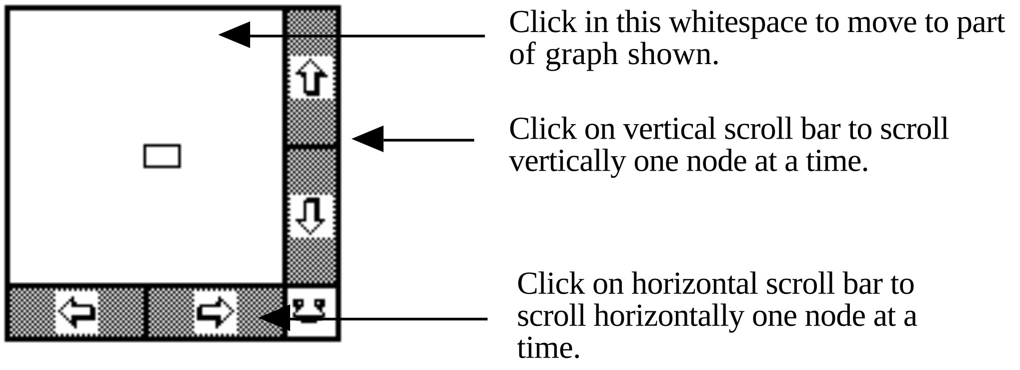

Click in this whitespace to move to part of graph shown.
Click on vertical scroll bar to scroll vertically one node at a time.
Click on horizontal scroll bar to scroll horizontally one node at a time.

Figure 4.7 - The Scroll Window

Viewing a Different Graph

CODE programs consist of a set of graphs that are drawn independently. There is no way to view more than one graph at a time. To view a particular graph, click the Travel cursor on the graph button. A scrolling menu of graphs will appear. Click to pick one. Figure 4.8 shows this scrolling menu.

Once a Call node has been annotated with the name of the graph that it calls, you can travel to that graph by clicking the travel cursor on the Call node. Click on the Return button to return to the Calling graph.

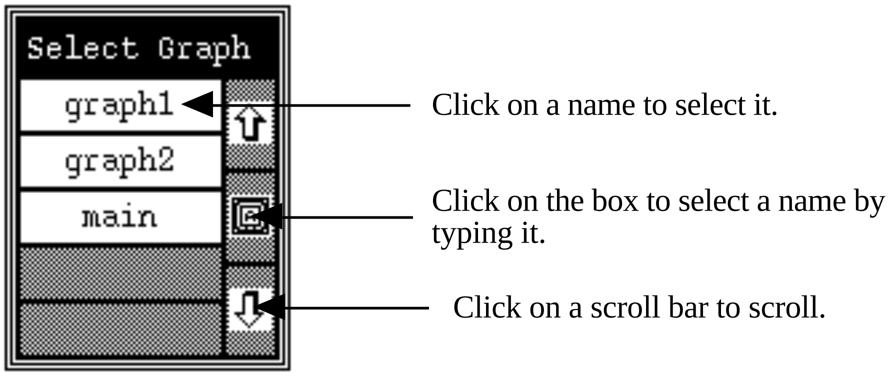

Select Graph
graph1
graph2
main
Click on a name to select it.
Click on the box to select a name by typing it.
Click on a scroll bar to scroll.

Figure 4.8 - Scrolling Menu of Graphs

Translating a Program for Execution

Once you drawn and properly annotated a CODE program, you can translate it into parallel C source code for a particular architecture by clicking any cursor on the Translate button and selecting an architecture. Error messages will be written onto the terminal window from which you ran CODE. See the chapter on translation in this manual. The “List” menu pick dumps a text-form of your program to the terminal window. It is primarily a debugging tool for us.

Deleting Things

Delete a node arc, etc. by clicking on it with the Erase cursor. Undo undoes this. Redo undoes the undo!

Moving Things on the Screen

Clink on a node, label, arc intermediate point, etc. with the Move cursor. Move the mouse to the desired location and click again.

Writing on the Graph

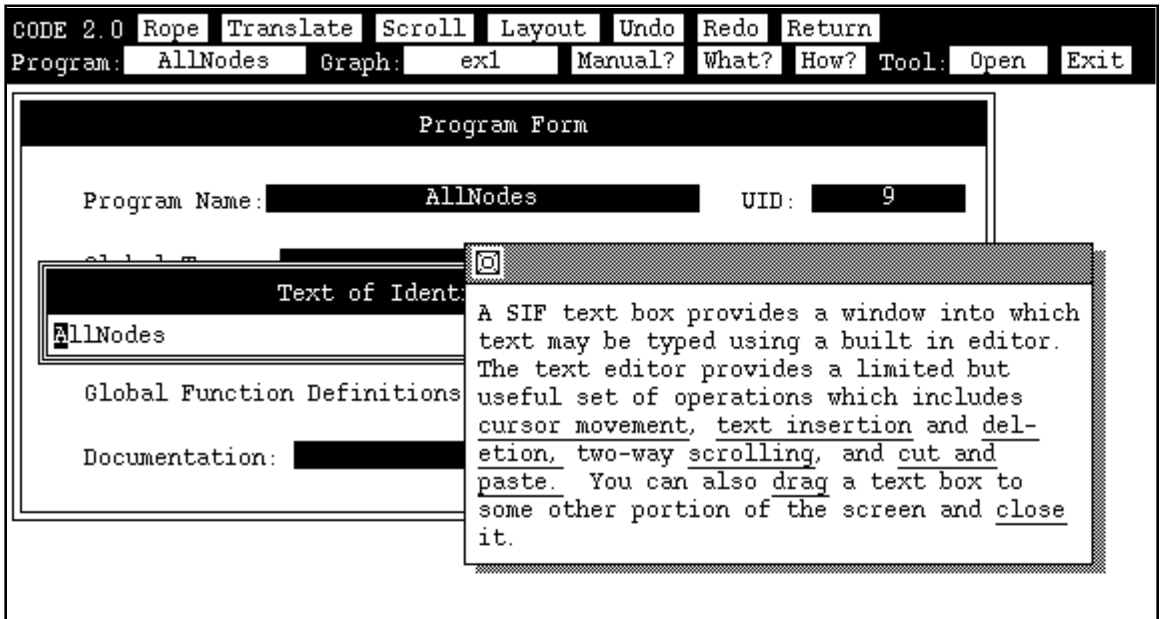
You can write comments right on the graph by clicking the Open cursor on empty space. A text box will appear. Enter your comment into it and click outside it to make it go away. The comment can be moved with the Move cursor, edited with the Open cursor, and deleted with the Erase cursor.

Copying Things

Use the Copy cursor to copy things, along with their attributes. For example, to copy a node, click on it with the Copy cursor. Click again on empty space, and the copy will appear at that point.

4.4 Online Help

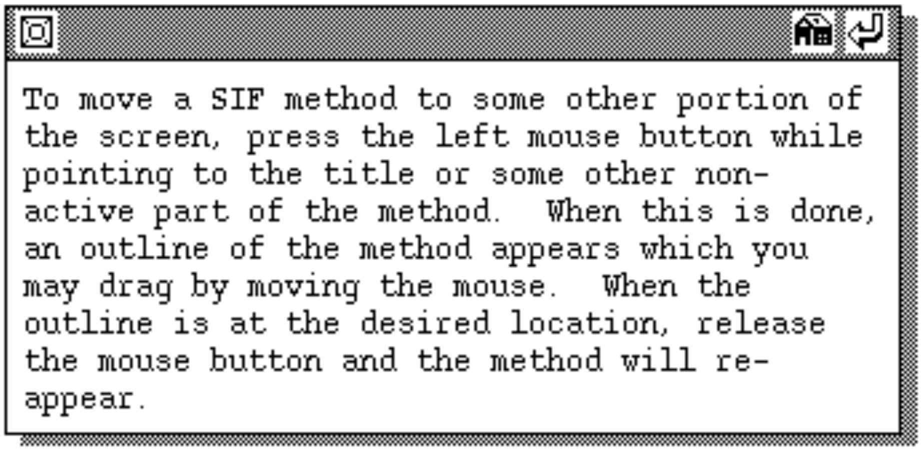
CODE has an elaborate online help system that uses hypertext. Click any cursor on the Manual button to bring up the online manual. The What and How buttons enable two kinds of context-sensitive help. The How button describes how to use the user interface. For example, if you click on it while a UC attribute form it open, it will tell you how to manipulate forms. The What button tells you what you are supposed to do with the user interface at a given point. If you click on it while a UC attribute form it open, it will describe the purpose and meaning of the various fields in the form. Figure 4.9 shows the help box that would appear if you clicked on the How button while a text editor box is active. Click on the underlined (or highlighted on color monitors) text for more help on that subject.



A SIF text box provides a window into which text may be typed using a built in editor. The text editor provides a limited but useful set of operations which includes cursor movement, text insertion and deletion, two-way scrolling, and cut and paste. You can also drag a text box to some other portion of the screen and close it.

Figure 4.9 - A Help Box

Figure 4.10 shows the form that would appear if you clicked on “drag” in the help box. Click on the hooked arrow icon to return to the previous help box. Click on the house icon to return to the home (or root) help box. Click on the little square icon at the upper left of the help box to turn help mode off.



To move a SIF method to some other portion of the screen, press the left mouse button while pointing to the title or some other non-active part of the method. When this is done, an outline of the method appears which you may drag by moving the mouse. When the outline is at the desired location, release the mouse button and the method will re-appear.

Figure 4.10 - Help Box Icons

5.0 Tutorial Example

The following tutorials assume that the user has read Chapters 1,2 and 4 of this manual and has a basic familiarity with graphical interfaces which employ mice and menus to communicate with users, with X windows, and with UNIX commands and editors. These tutorials will guide you, step by step, through the process of creating and running CODE programs.

5.1 TUTORIAL : EXAMPLE 1

In this example we will be creating a CODE program that sums the elements of a 100 element vector. We first initialize two vectors of 50 elements each and then add the two vectors seperately, in parallel. The resulting partial sums are then added to produce the final sum.

Contents of the Tutorial

I. Preliminaries
II. Entering Example 1
  Step 1. Draw all the nodes and arcs in a graph.
  Step 2. Use attribute forms to enter all information about the nodes, arcs, graphs and programs.
III. Executing Example 1 on the Sequent.
  Step 1. Save and compile the program.
  Step 2. Create the files which record the node function definitions.
  Step 3. Create the executable and run it.

I. Preliminaries

Get the X-Windows system running. Create a directory to hold the files associated with the program and *cd* to that directory. Before runnng CODE, ensure that it is present on your UNIX path (or be prepared to type a complete path name). Then enter

```
code2 sumvector.grf
```

at the UNIX prompt. This will run CODE and create a program called “sumvector.” A single window will be opened and all activity will take place within it. Figure 4.1 shows the CODE window.

II. Entering Example 1

Our goal is to draw the following graph:

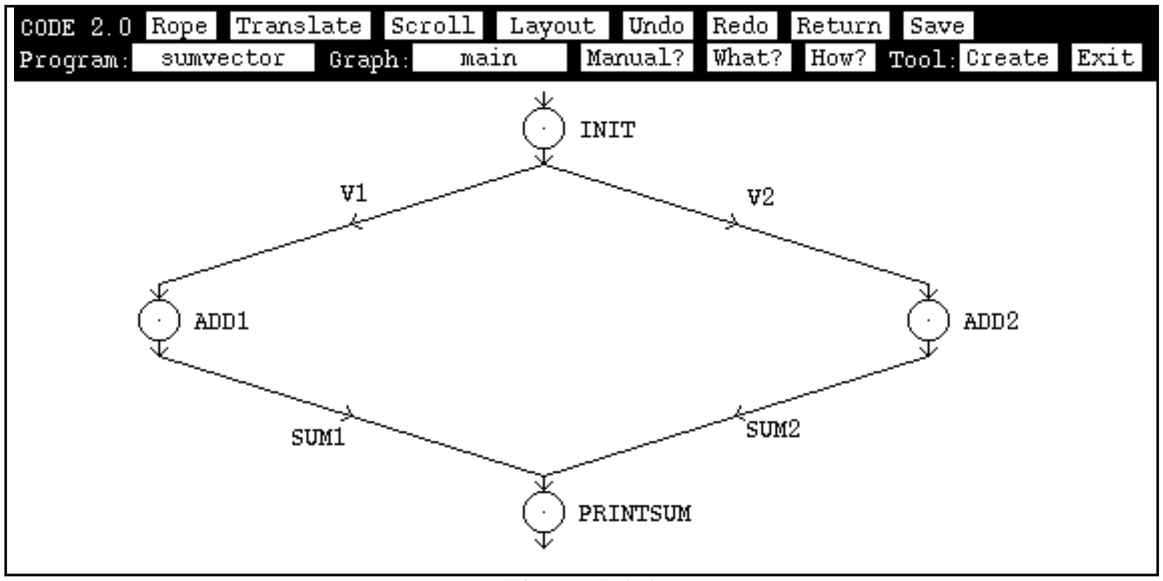


Figure 5.1.1

You may enter and annotate the nodes and arcs of your program in any order, but we will adopt the following sequence of operations:

Step 1. Draw all the nodes and arcs in a graph.
Step 2. Use attribute forms to enter all information about the nodes, arcs, graphs and programs.

Before we start, it is important to point out that you should save your work periodically. Then, in the (unlikely of course!) event that the CODE system crashes, your work will not be lost.

To saving your work as you go:

With create cursor in the CODE window, click on the program button. A menu of program options will pop up. Pick the save option by clicking on it. By doing this regularly you can avoid losing your work in case of a system crash.

Correcting mistakes :

a. Finish, drawing the node or arc.
b. Select the erase cursor from the tool window.
c. Place the erase cursor on the object to be deleted and click the left mouse button.
d. If you want to undo the deletion, click on the 'Undo' button in the upper part of the CODE window.
e. Clicking on the 'Redo' button undoes the undo !
f. If you make any syntax errors while entering, CODE will beep to let you know that you have made a mistake.

STEP 1. Draw all the Nodes and Arcs in a Graph

A. Draw all the nodes.

1.Draw the UC node, INIT by

a. Position the create cursor in the CODE window, where the UC INIT is to be located.
b. Click with the left mouse button, to bring up the menu of node types.
c. Pick the Unit of Computation (UC) node by clicking on it.

2. Draw the UC's ADD1, ADD2, and PRINTSUM.

.

B. Draw all the arcs.

1. Draw the arc V1 by,

a. With the create cursor, click on the lower arrow of UC, INIT.
b. Also, click on the upper arrow of UC, ADD1.

2. Draw the arcs V2, SUM1 and SUM2.

At this point the graph looks like Fig. 5.1.2.

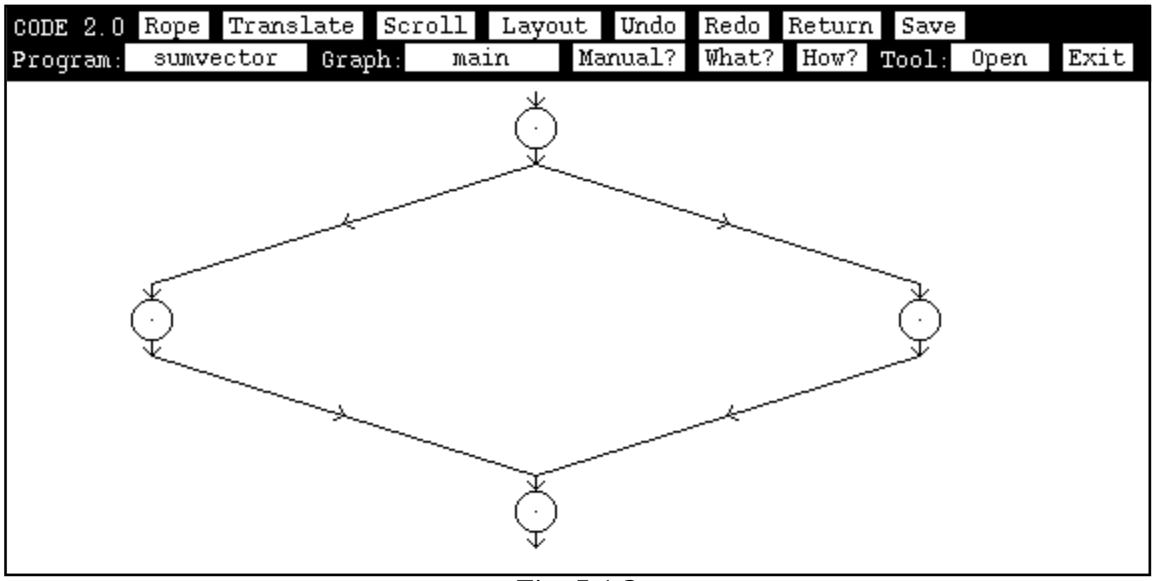


Fig. 5.1.2

STEP 2. Use Attribute Forms to enter all Information about Nodes, Arcs Graphs and Programs

A. Enter information about the nodes

1. Enter information about INIT

a. Select the open cursor from the tool window.

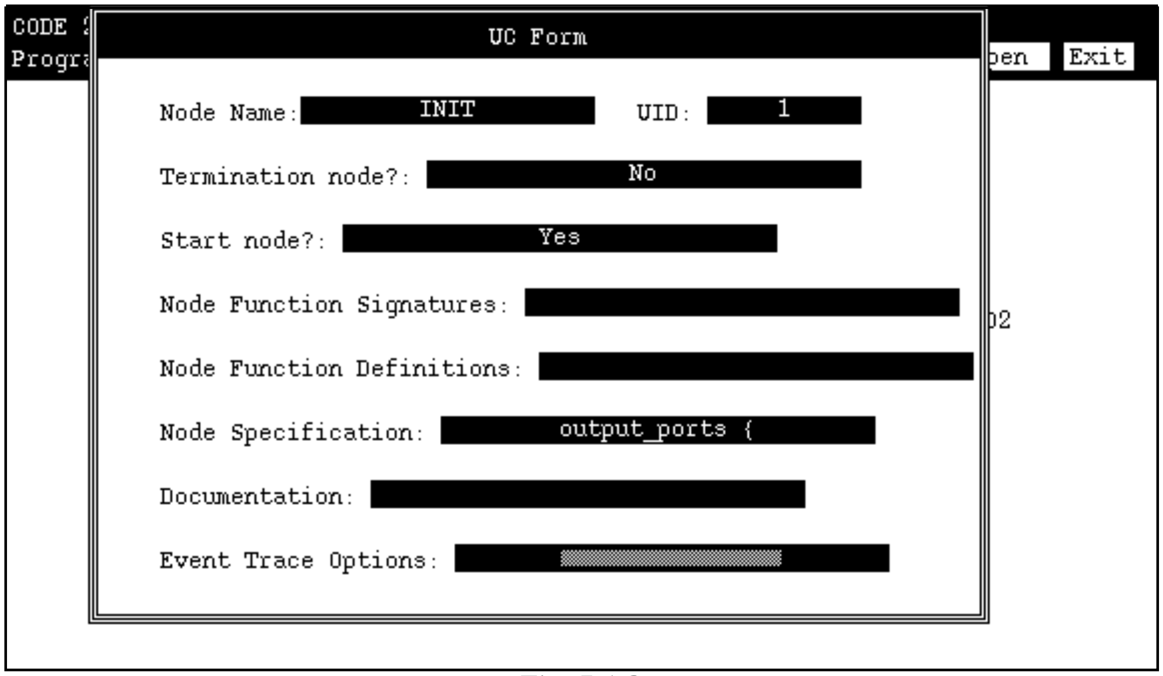
b. Click the open cursor on the UC INIT to open its attribute form. It looks like Fig.5.1.3

c. Click on the node name field and a text box will appear. Enter the name 'INIT' in this box. You can make the text box go away by clicking on the white space in the UC form to return to it.

d. The value in the UID field is automatically generated by CODE. It is a debugging aid which you can ignore for now.

e. The termination node and start node fields will have a default value of 'No'. Since INIT is a start node, update the Start node field to 'Yes'.

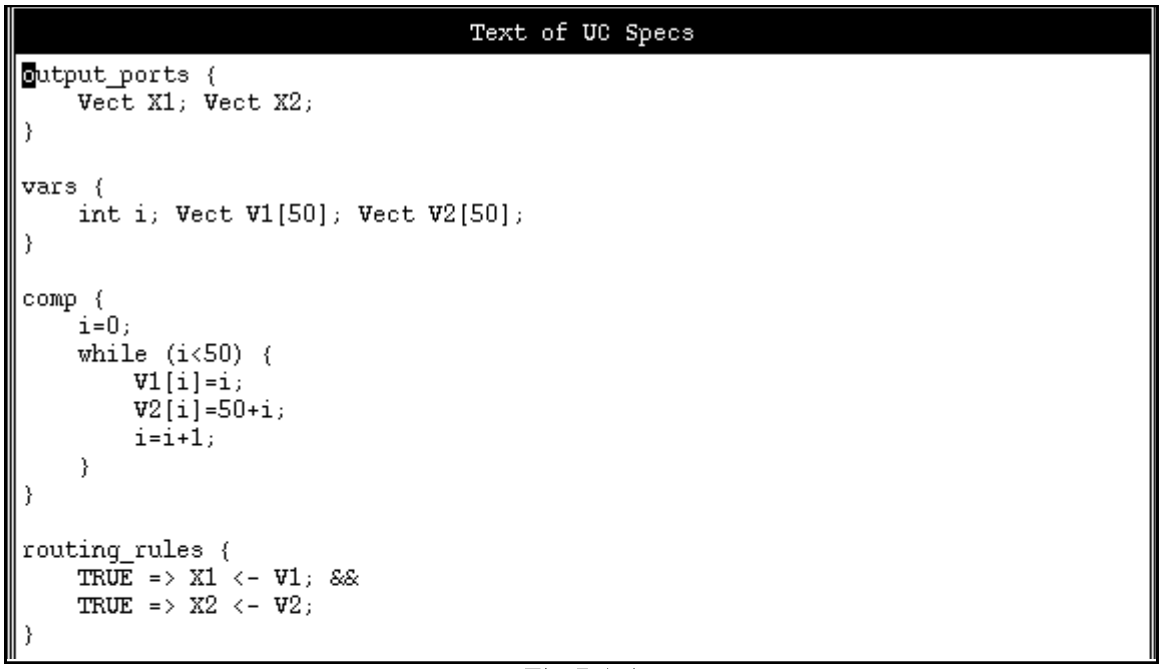
f. The Node Function Signatures and Node Function Definitions fields are to be left blank.



Node Name:	INIT
UID:	1
Termination node?:	No
Start node?:	Yes
Node Function Signatures:	
Node Function Definitions:	
Node Specification:	output_ports {
Documentation:	
Event Trace Options:	▨

Fig. 5.1.3

g. Click on the Node Specification field. A text box labelled 'Text of UC Specs' will appear. Fig. 5.1.4 shows the code to be entered in this text box. This code can also be entered via a window running the editor defined by your EDITOR environment variable. To bring up this window press ESC-e(the escape key followed by e). After entering the code, you can exit from the window, the way you would normally exit from your editor.



Text of UC Specs

```
output_ports {
    Vect X1; Vect X2;
}

vars {
    int i; Vect V1[50]; Vect V2[50];
}

comp {
    i=0;
    while (i<50) {
        V1[i]=i;
        V2[i]=50+i;
        i=i+1;
    }
}

routing_rules {
    TRUE => X1 <- V1; &&
    TRUE => X2 <- V2;
}
```

Fig.5.1.4

h. The documentation field is to be left blank.

i. To exit the UC INIT attribute form, click on the white space outside the form in the CODE window.

The name 'INIT' will be displayed next to the node, but you can move it with the Move cursor.

2. Enter information about ADD1 as in 1.

The one difference is that ADD1 is not a Start node. The code to be entered in the Node Specification field of ADD1 is as shown below,

```
input_ports {
 Vect Y1;
}

output_ports {
 int P1;
}

vars {
 Vect V1[50]; int i; int sum1;
}

firing_rules {
 Y1 -> V1 =>
}
```

```
comp {
 i=0;
 sum1=0;
 while (i<50) {
 sum1=sum1+V1[i];
 i=i+1;
 }
}

routing_rules {
 TRUE => P1 <- sum1;
}
```

3. Enter information about ADD2 as in 1.

The one difference is that ADD1 is not a Start node. The code to be entered in the Node Specification field of ADD2 is as shown below,

```
input_ports {
 Vect Y2;
}

output_ports {
 int P2;
}

vars {
 Vect V2[50]; int i; int sum2;
}

firing_rules {
 Y2 -> V2 =>
}

comp {
 i=0;
 sum2=0;
 while (i<50) {
 sum2=sum2+V2[i];
 i=i+1;
 }
}

routing_rules {
 TRUE => P2 <- sum2;
}
```

4. Enter information about PRINTSUM as in 1.

The one difference is that PRINTSUM must be designated as a termination node. The code to be entered in the Node Specification field of PRINTSUM is as shown below,

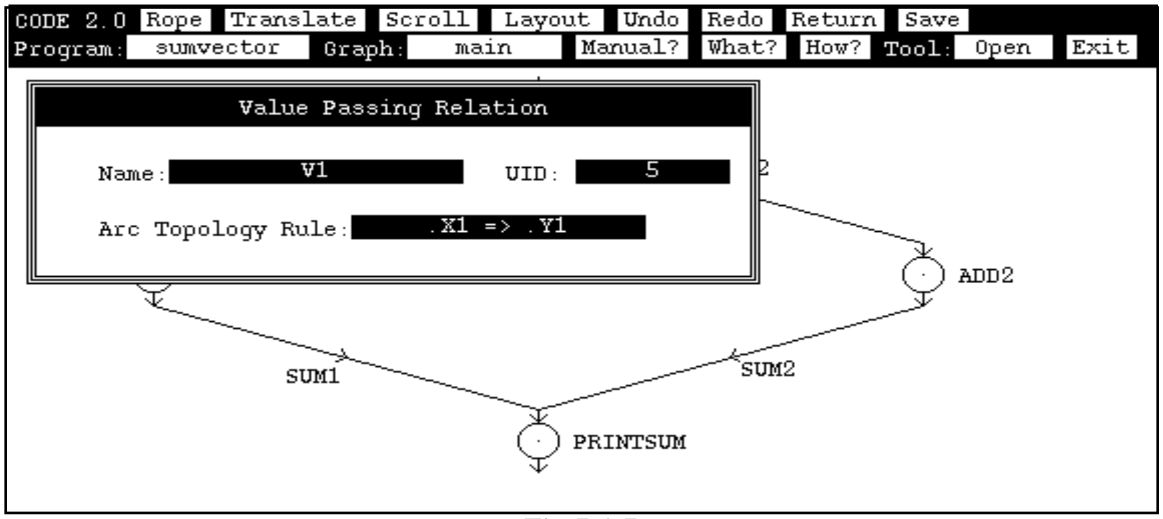
```
input_ports {
 int Q1; int Q2;
}
```

```
vars {
 int sum1; int sum2; int sum;
}

firing_rules {
 Q1 -> sum1, Q2 -> sum2 =>
}

comp {
 sum=sum1+sum2;
 print(sum1,sum2,sum);
}
```

B. Enter information about the arcs

1. Enter information about V1

   a. Click the open cursor on the arc V1 to open its attribute form. It looks like Fig. 5.1.5.



Value Passing Relation

Name: V1 UID: 5

Arc Topology Rule: .X1 => .Y1

Fig.5.1.5

   b. Click on the name field and a text box will appear. Enter the name 'V1' in this box. You can make the text box go away by clicking on the white space in the Value Passing Relation form to return to it

   c. The value in the UID field is automatically generated by CODE.

   d. Click on the Arc Topology Rule field and another text box will appear. The code to be entered in this text box is shown below,

```
.X1 => .Y1
```

   e. Exit from both the text box and the form by clicking on the white space outside the two in the CODE window.

   The name 'V1' will be displayed next to the arc, but you can move it with the Move cursor.

2. Enter information about V2, SUM1 and SUM2 as in 1.

The Arc Topology Rule field of V2 is,

```
.X2 => .Y2
```

The Arc Topology Rule for SUM1 is,

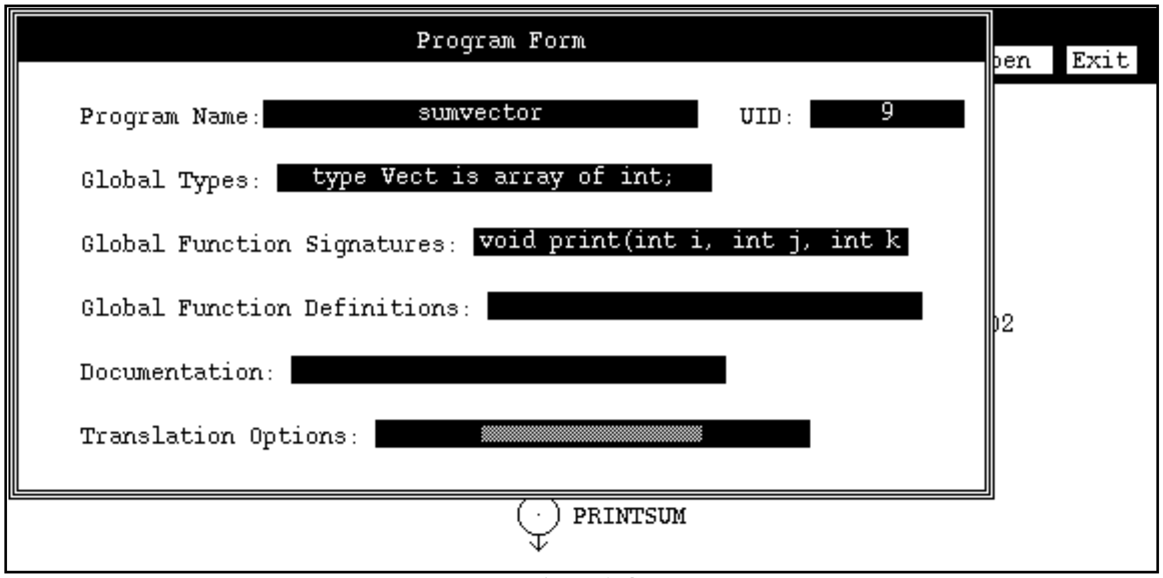
```
.P1 => .Q1
```

The Arc Topology Rule for SUM2 is,

```
.P2 => .Q2
```

C. Enter information about the program

a. Click the open cursor on the Program button in the CODE window, to open its attribute form. It looks like Fig. 5.1.6.


Program Form

Program Name:	sumvector
UID:	9
Global Types:	type Vect is array of int;
Global Function Signatures:	void print(int i, int j, int k
Global Function Definitions:	
Documentation:	
Translation Options:	▨

Fig.5.1.6

b. The values in the program name and UID fields are automatically generated by CODE.

c. Click on the Global Types field and a text box will appear. The code to be entered in this text box is shown below,

```
type Vect is array of int;
```

d. Click on the Global Function Signatures field and another text box will appear. The code to be entered in this text box is shown below,

```
void print(int i, int j, int k);
```

e. The Global Function Definitions and Documentation fields are to be left blank.

f. Click on the Translation Options field, to bring up it's attribute form. Now, click on the 'Object Files to Link' field. A textbox will appear. Enter the name of the file in which node function definitions will be specified, in this textbox.

```
filename.o
```

g. Exit from the program form by clicking on the white space outside the form in the CODE window.

III. Executing Example1 on the Sequent

We will now compile and run the program in the following order:

Step 1. Save and compile the program.
Step 2. Create the files which record the node function definitions.
Step 3. Create the executable and run it.

STEP 1. Save and Compile the Program

a. Click the create cursor on the Program button and select the 'save' option.
b. Click the create cursor on the Translate button and select the 'Sequent' option.
c. If there are any errors, these are displayed in the window from where CODE was run.
d. Eliminate the errors, if any.
e. After the program has been successfully compiled and saved, exit the program by clicking on the Exit button. Select the 'save' option and quit.

At this stage, the following files and directories will be present under the directory example1.

README	File
sumvector.grf	File
sumvector.sequent	Directory

STEP 2. Create the Files which record the Node Function Definitions

$cd ./sumvector.sequent   The following files will be present in this directory.
Makefile
c2_globtype.h
c2_main.h
c2_main.c
main.c

$vi filename.c   Record all node function definitions inthis file. Give it the same name as the object file to be linked. The text to be entered in this file is as shown below. This file "routines.c" can also be picked up from the directory,
`examples/tutorial1/sumvector.sequent.`
These will be present in the directory in which CODE2 has been installed.

```
print( i, j, k)
int i,j,k;
{
printf(“The first partial sum is %d\n”,i);
printf(“The second partial sum is %d\n”,j);
printf(“The total sum is %d\n”,k);
}
```

STEP 3. Create the executable and run it

These operations will depend on how your workstation is connected to the Sequent. Before giving the 'make' command, kindly ensure that you are logged in on the Sequent. Please refer to the Chapter on ‘Translating & Running Programs’ for the details. At Dept. of Computer Sc, Univ. of Texas, the following command should be given to login onto the Sequent.

```
$rlogin qt
```

After having logged in on the Sequent, give the following commands,

$cc -c filename.c	Compile the file specifying node function definitions.
$make	This will create the executable 'sumvector'.
$./sumvector	Run the executable. The output displayed should be that shown below,

```
The first partial sum is 1225
The second partial sum is 3725
The total sum is 4950
```

5.2 TUTORIAL EXAMPLE 2

The following aasumes that the user has completed Tutorial Example 1.

This example is an altered version of Example 1. There are now ten adder nodes instead of two and each adder prints it’s number and the sum it has calculated when it completes. Replication is used to produce the ten adders.

Contents of the Tutorial

I. Preliminaries

Get the X-Windows system running. Create a directory to hold the files associated with the program and move to that directory. Before runnng CODE, ensure that it is present on your UNIX path. Then enter

```
code2 rsum.grf
```

at the UNIX prompt. This will run CODE and open a program called "rsum". A single window will be opened and all activity will take place within it. Fig. 4.1 shows the CODE window.

II. Entering Example 2

Our goal is to draw the following graph:

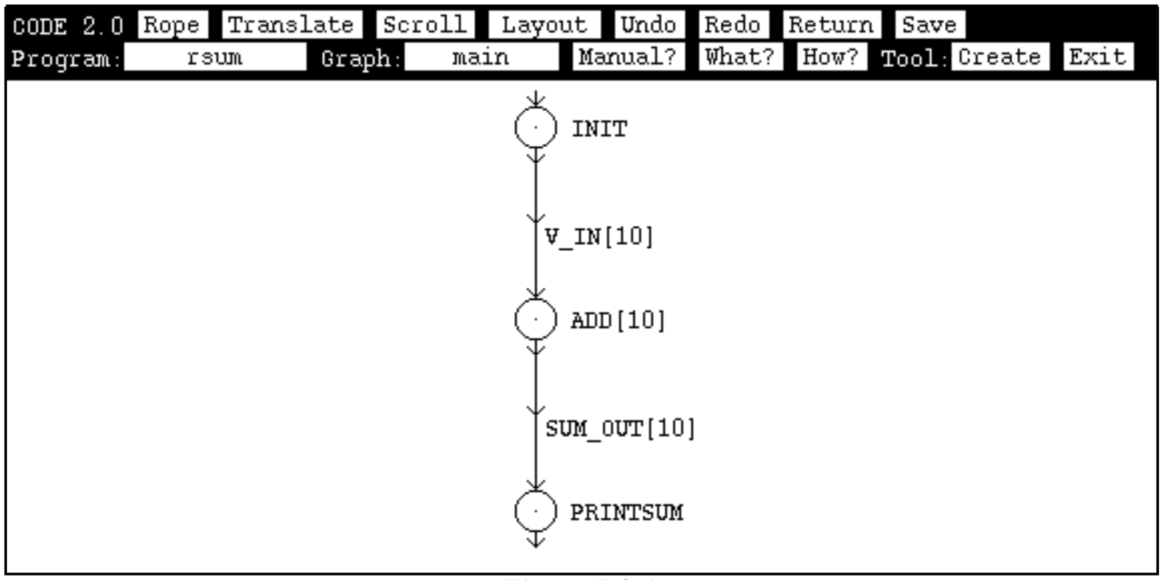


Figure 5.2.1

You may enter and annotate the nodes and arcs of your program in any order, but we will adopt the following sequence of operations:

Step 1. Draw all the nodes and arcs in a graph.
Step 2. Use attribute forms to enter all information about the nodes, arcs, graphs and programs.

Before we start

Saving your work as you go:

With create cursor in the CODE window, click on the program button. A menu of program options will pop up. Pick the save option by clicking on it. By doing this regularly you can avoid losing your work in case of a system crash.

Correcting mistakes :

a. Finish, drawing the node or arc.
b. Select the erase cursor from the tool window.
c. Place the erase cursor on the object to be deleted and click the left mouse button.
d. If you want to undo the deletion, click on the 'Undo' button in the upper part of the CODE window.
e. Clicking on the 'Redo' button undoes the undo !
f. If you make any syntax errors while entering, CODE will beep to let you know that you have made a mistake.

STEP 1. Draw all the Nodes and Arcs in a Graph

A. Draw all the nodes.

1.Draw the UC node, INIT by

a. Position the create cursor in the CODE window, where the UC INIT is to be located.
b. Click with the left mouse button, to bring up the menu of node types.
c. Pick the Unit of Computation (UC) node by clicking on it.

2. Draw the UC's ADD[10] and PRINTSUM.

.

B. Draw all the arcs.

1. Draw the arc V_IN[10] by,

a. With the create cursor, click on the lower arrow of UC, INIT.
b. Also, click on the upper arrow of UC, ADD[10].

2. Draw the arc SUM_OUT[10].

At this point the graph looks like Fig. 5.2.2.

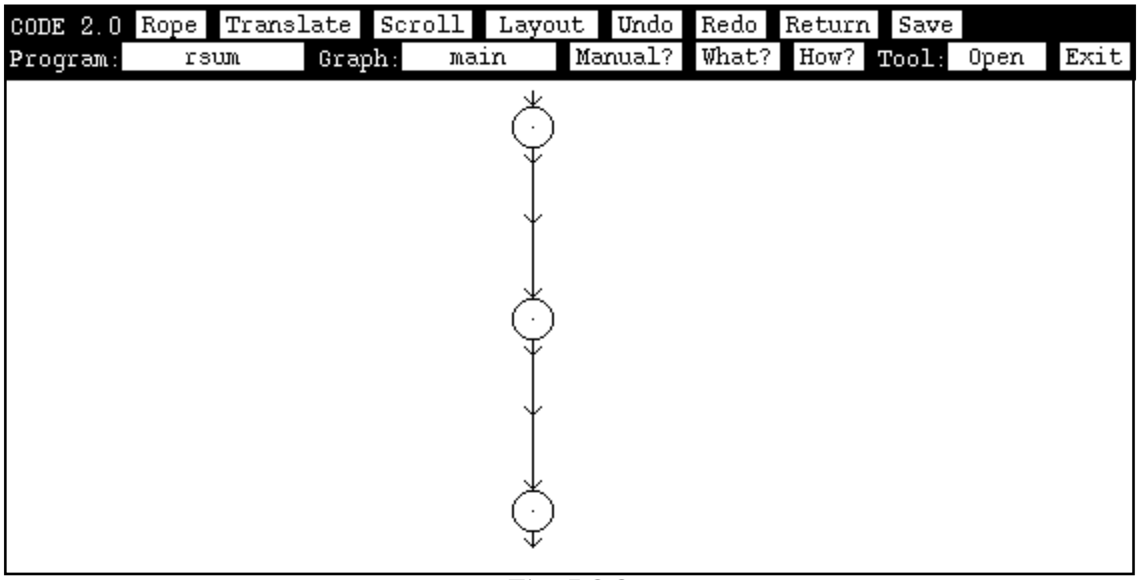


Fig. 5.2.2

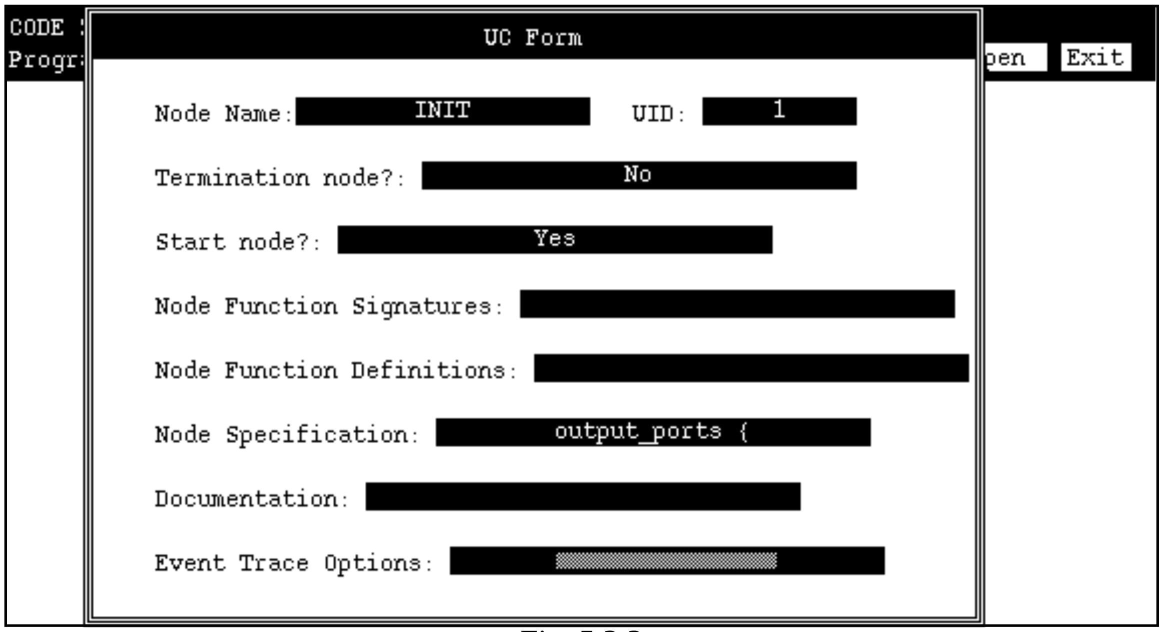
STEP 2. Use Attribute Forms to enter all Information about Nodes, Arcs Graphs and Programs

A. Enter information about the nodes

1. Enter information about INIT

a. Select the open cursor from the tool window.

b. Click the open cursor on the UC INIT to open its attribute form. It looks like Fig.5.2.3



UC Form

Node Name: INIT UID: 1

Termination node?: No

Start node?: Yes

Node Function Signatures:

Node Function Definitions:

Node Specification: output_ports {

Documentation:

Event Trace Options:

Fig. 5.2.3

c. Click on the node name field and a text box will appear. Enter the name 'INIT' in this box. You can make the text box go away by clicking on the white space in the UC form to return to it.

d. The value in the UID field is automatically generated by CODE. It is a debugging aid which you can ignore for now.

e. The termination node and start node fields will have a default value of 'No'. Since INIT is a start node, update the Start node field to 'Yes'.

f. The Node Function Signatures and Node Function Definitions fields are to be left blank.

g. Click on the Node Specification field. A text box labelled 'Text of UC Specs' will appear. The code to be entered in this text box is as shown below. This code can also be entered via a window running the editor defined by your EDITOR environment variable. To bring up this window press ESC-e(the escape key followed by e). After entering the code, you can exit from the window, the way you would normally exit from your editor.

```
output_ports {
 Vect OV;
}

vars {
 Mat V[10][10]; int i; int j; int k;
}

comp {
 k=0;
 j=0;
 while (k<10)
 {
 i=k*10;
 while (j<10)
 {
 V[k][j]=i;
 j=j+1;
 i=i+1;
 }
 k=k+1;
 j=0;
 }
}

routing_rules {
 TRUE => { OV[i] <- V[i]; : (i 10) };
}
```

h. The documentation field is to be left blank.

i. To exit the UC INIT attribute form, click on the white space outside the form in the

CODE window.

The name 'INIT' will be displayed next to the node, but you can move it with the Move cursor.

2. Enter information about ADD[10] as in 1.

The one difference is that ADD[10] is not a Start node. The code to be entered in the Node Specification field of ADD[10] is as shown below,

```
input_ports {
 Vect IV;
}

output_ports {
 int SUMS;
}

vars {
 Vect C[10]; int sum; int i;
}

firing_rules {
 IV -> C =>
}

comp {
 i=0;
 sum=0;
 while (i<10)
 {
 sum=sum+C[i];
 i=i+1;
 }
}

routing_rules {
 TRUE => SUMS <- sum;
}
```

3. Enter information about PRINTSUM as in 1.

The one difference is that PRINTSUM must be designated as a termination node. The code to be entered in the Node Specification field of PRINTSUM is as shown below,

```
input_ports {
 int I_SUM;
}

vars {
 Vect p_sum[10]; int i; int t_sum;
}

firing_rules {
 { I_SUM[i] -> p_sum[i] : (i 10) } =>
}

comp {
 routine1(p_sum);
}
```

B. Enter information about the arcs

1. Enter information about V_IN[10]

   a. Click the open cursor on the arc V_IN[10] to open its attribute form. It looks like Fig. 5.2.4.

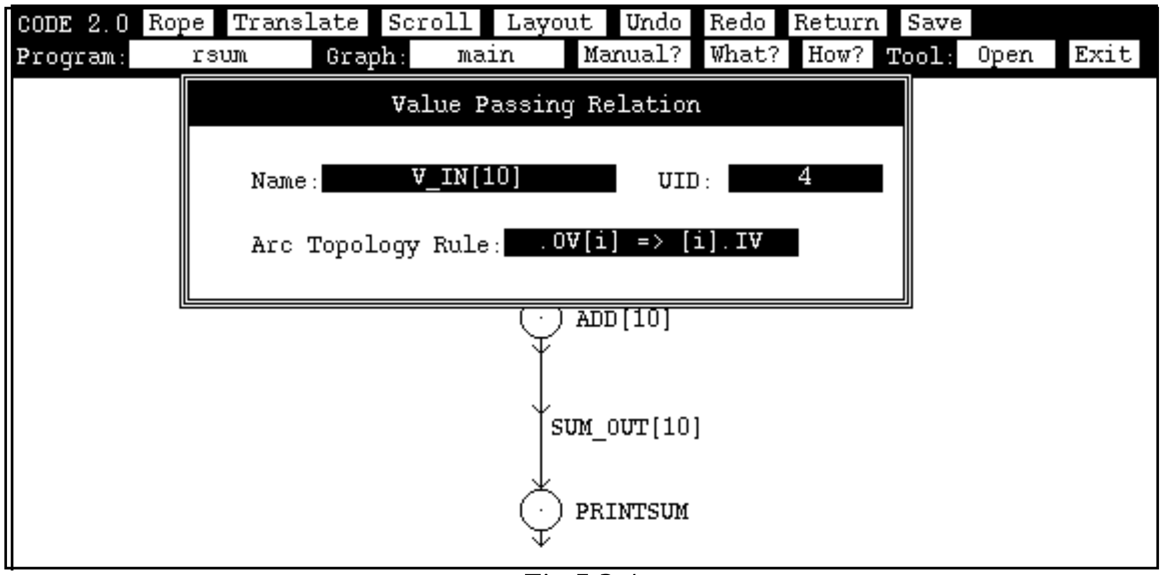
   b. Click on the name field and a text box will appear. Enter the name 'V_IN[10]' in this box. You can make the text box go away by clicking on the white space in the Value Passing Relation form to return to it

   c. The value in the UID field is automatically generated by CODE.

   d. Click on the Arc Topology Rule field and another text box will appear. The code to be entered in this text box is shown below,

   ```
   .OV[i] => [i].IV
   ```

   e. Exit from both the text box and the form by clicking on the white space outside the two in the CODE window.



Name: V_IN[10]
UID: 4
Arc Topology Rule: .OV[i] => [i].IV

Fig.5.2.4

The name 'V_IN[10]' will be displayed next to the arc, but you can move it with the Move cursor.

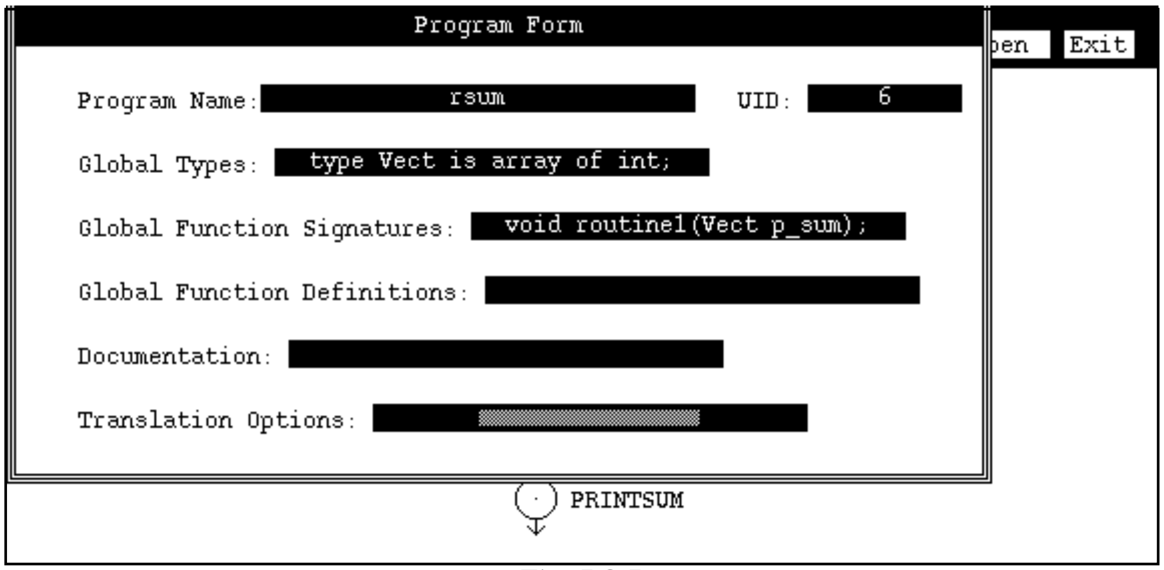
2. Enter information about SUM_OUT[10] as in 1.

The Arc Topology Rule field of SUM_OUT[10] is,

```
[i].SUMS => .I_SUM[i]
```

C. Enter information about the program

a. Click the open cursor on the Program button in the CODE window, to open its attribute form. It looks like Fig. 5.2.5.


Program Form

Program Name: rsum UID: 6

Global Types: type Vect is array of int;

Global Function Signatures: void routine1(Vect p_sum);

Global Function Definitions:

Documentation:

Translation Options:

Fig. 5.2.5

b. The values in the program name and UID fields are automatically generated by CODE.

c. Click on the Global Types field and a text box will appear. The code to be entered in this text box is shown below,

```
type Vect is array of int;
type Mat is array of Vect;
```

d. Click on the Global Function Signatures field and another text box will appear. The code to be entered in this text box is shown below,

```
void routine1(Vect p_sum);
```

e. The Global Function Definitions and Documentation fields are to be left blank.

f. Click on the Translation Options field, to bring up it's attribute form. Now, click on the 'Object Files to Link' field. A textbox will appear. Enter the name of the file in which node function definitions will be specified, in this textbox.

```
filename.o
```

g. Exit from the program form by clicking on the white space outside the form in the CODE window.

III. Executing Example2 on the Sequent

We will now compile and run the program in the following order:

Step 1. Save and compile the program.
Step 2. Create the files which record the node function definitions.
Step 3. Create the executable and run it.

STEP 1. Save and Compile the Program

a. Click the create cursor on the Program button and select the 'save' option.
b. Click the create cursor on the Translate button and select the 'Sequent' option.
c. If there are any errors, these are displayed in the window from where CODE was run.
d. Eliminate the errors, if any.
e. After the program has been successfully compiled and saved, exit the program by clicking on the Exit button. Select the 'save' option and quit.

At this stage, the following files and directories will be present under the directory example1.

README	File
rsum.grf	File
rsum.sequent	Directory

STEP 2. Create the Files which record the Node Function Definitions

$cd ./rsum.sequent — The following files will be present in this directory.
Makefile
c2_globtype.h
c2_main.h
c2_main.c
main.c

$vi filename.c — Record all node function definitions inthis file. Give it the same name as the object file to be linked.The text to be entered in this file is as shown below. This file "routines.c" can also be picked up from the directory,
`examples/tutorial2/rsum.sequent.`
These directories will be present under the directory in which CODE2 has been installed.

```
routine1(p_sum)
int p_sum[];
{
 int i,t_sum;

 i=0;
 t_sum=0;
 while (i<10)
 {
 printf(“Partial sum %d is %d\n”,i,p_sum[i]);
 t_sum=t_sum+p_sum[i];
 i=i+1;
 }
 printf(“Total sum is %d\n”,t_sum);
}
```

STEP 3. Create the executable and run it

These operations will depend on how your workstation is connected to the Sequent. Before giving the 'make' command, kindly ensure that you are logged in on the Sequent. Please refer to the Chapter on ‘Translating & Running Programs’ for the details. At Dept. of Computer Sc, Univ. of Texas, the following command should be given to login onto the Sequent.

```
$rlogin qt
```

After having logged in on the Sequent, give the following commands,

$cc -c filename.c	Compile the file specifying node function definitions.
$make	This will create the executable 'rsum'.
$./rsum	Run the executable. The output displayed should be that shown below,

```
Partial sum 0 is 45
Partial sum 1 is 145
Partial sum 2 is 245
Partial sum 3 is 345
Partial sum 4 is 445
Partial sum 5 is 545
Partial sum 6 is 645
Partial sum 7 is 745
Partial sum 8 is 845
Partial sum 9 is 945
Total sum is 4950
```

6.0 CODE Programming Checklist

This chapter summarizes the steps involved in creating a CODE program. You do not necessary need to perform the steps in the order below, but it is an organized approach. Also, there is sometimes more than way to accomplish something. Alternatives are not mentioned here.

1. Draw all of the nodes and arcs in your program. Make exactly one UC node the Start node and one UC node the Terminate node by making the appropriate menu pick in the UC attribute forms.

2. Create all necessary type definitions. It is easiest to put them all into the Type field of the Program attribute form. You will need a type definition for every array and structure type you use in your program. Here are some examples.

```
type mat1 is array of real;
type mat2 is array of mat1; // This is how to make a 2D array.
type Point is struct {
       real x;
       real y;
};
```

3. Create function signatures (also called function prototypes) for all sequential functions you will call directly from CODE. You can put all of these into the Function Signatures field of the Program attribute form. You will need sequential functions to perform all I/O in addition to any other use you have. Here are some examples.

```
void PrintTheAnswer(int i, real x); // void for procedures
int ReadInt();
void ReadReal(real *x); // * means call by reference
void PrintVect(mat1 v, int n);
```

4. Either enter function body definitons. If you put the signature for a function into the Program form, you must either put its body into the Program attribute form's Function Definition field or put it into a separate file. In the latter case, you will need to supply the name of an object (.o) file in the Translation Options field of the Program form. If you put the signature into a node or graph form, the body definition should go into the Function Definition field of the same form.

5. Open the attribute form of every node and fill in its specification field. Here is an example UC node specification to remind you of the syntax and order of the stanzas. If you have nothing to put into a stanza, you may omit it.

```
input_ports { // Name and type for incoming arc ports.
      Vect V1; Vect V2;
}
output_ports { // Name and type for outgoing arc ports.
      Vect OV1;
}
shared_vars { // Names for shared variables. See Name
      Mat2 m writer; // sharing relations.
```

```
        }
        vars { // Local variables.
              Vect v;
        }
        init_comp {
              initmat(m); // Computation done when node is created.
        }
        firing_rules { // Fire when there is data on either V1 or
              V1 -> v => || // V2. Store it in v.
              V2 -> v =>
        }
        comp { // Computation done when node fires.
              if (v[0] == 0) mycomp(v, m); // Call to sequential procedure
        }
        routing_rules { // If comp is 0, send v on OV1 else OV2.
              TRUE => OV1 <- v;
        }
```

6. Open the attribute form for every arc and fill it in. What you must enter varies according to the kinds of nodes the arcs connect. Every arc except one that connects an Input Interface node to an Output Interface node requires some form of annotation. See the section on arcs in the Reference Manual appendix. Here is an example of an arc topology specification on an arc that runs from one UC node to another.

```
.X => .Y // Binds port X to port Y
```

7. Save and translate your graph.

8. Some notes on names might be helpful. All objects (nodes, arcs, etc.) in CODE can have names, but few require them. Only nodes that form the interface for a graph must be given names. These nodes are Input Interface, Output Interface, and Creation Parameter nodes. Their names must be valid CODE identifiers and must be unique within the graph.

7.0 Arc Topology Specifications

This chapter discusses the annotation of arcs that run from one UC node to another. This is the most important case. Other cases are discussed in the arcs section of the Reference Manual appendix. See also the chapters on Call nodes and Name Sharing Relations.

Arcs that run from on UC to another are dataflow arcs. They represent the flow of data from the computation of one UC to the computation of another. They are FIFO queues of data. The annotation of such an arc is called an “arc topology specification.”

UC nodes use ports to reference these arcs. You can think of a port name as being a UC node’s local name for an arc that is incident upon it. Hence, the primary job of an arc topology specification is to bind an output port on one UC to an input port on another.

Suppose a UC node has an output port called Y and another UC node has an input port X. You can draw an arc from the first UC to the second. You must then provide an arc topology specification to state which ports the arc binds together. The specification

```
.Y => .X
```

binds port Y to port X. Note the dots-- they are required. The types of the ports bound together must match.

There is another role that arc topology specifications play. To understand this you may want to think of arcs as being rules that describe where data will go to, given where it come from. For example, the rule above could be read as follows.

> If data come from the “from” node’s port Y,
> they must be sent to the “to” node’s port X.

Under this view, the addition of indices to arc topology specifications allows them to specify runtime determined communication structures and greatly extends the representational power of CODE.

An example is in order. Recall that any CODE node or arc can be instantiated any number of times at runtime and that the instantiations are named by integer valued indices. Suppose one wishes to create the structure shown in figure 7.1 in which values placed onto node A’s port Y[i] are sent to the instance of node B with index [i], for i taking on values from 0 to N-1.

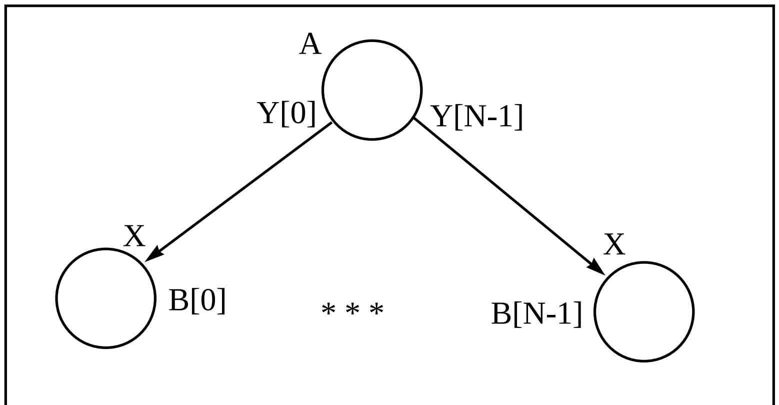
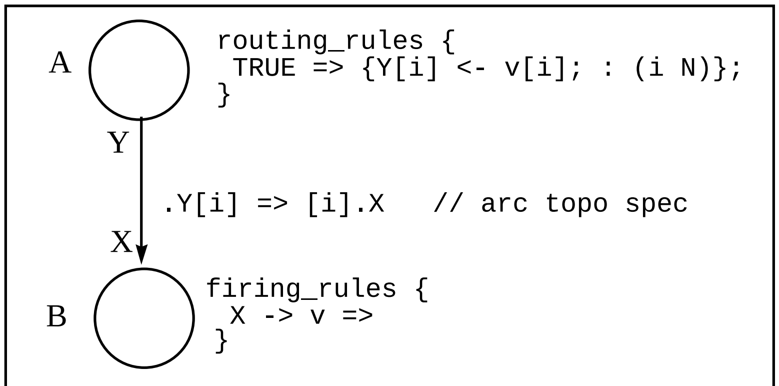


Figure 7.1 - Example Communication Structure

This structure can be created with the CODE program shown in figure 7.2. Consider node A's routing rule. The syntax { <thing> : (i N)} replicates <thing> for i taking on values from 0 to N-1. Hence, A sends data out on the desired ports Y[i].



```
routing_rules {
 TRUE => {Y[i] <- v[i]; : (i N)};
}
```

```
.Y[i] => [i].X   // arc topo spec
```

```
firing_rules {
 X -> v =>
}
```

Figure 7.2 - CODE Program for Example

The arc topology specification may read as follows.

> if data come from the "from" node's port Y[i],
> they must be sent to the "to" node with index [i]'s port X.

Hence, the value value of "i" in A's routing rule determines the index of the receiving node. You do not need to declare any bound on the number of receiving nodes. They are created as needed. Also notice that port X on node B has no index. Each instance of B has its own port X so no index is required.

For arcs running from UC nodes to UC nodes, the general form of an arc topology specification is as follows.

```
[ident]•••[ident].FROMPORTNAME[ident]•••[ident] =>
[expr]•••[expr].TOPORTNAME[expr]•••[expr]
```

The indices on the left of the “=>” must be unique variable names. Their values are determined by the indices of the node and port the arc comes from. The indices before the dot refer to the node. Those after the port name refer to the port.

The indices on the right of the “=>” may be general integer valued expressions using the variable names from left as well as creation parameters and function calls.

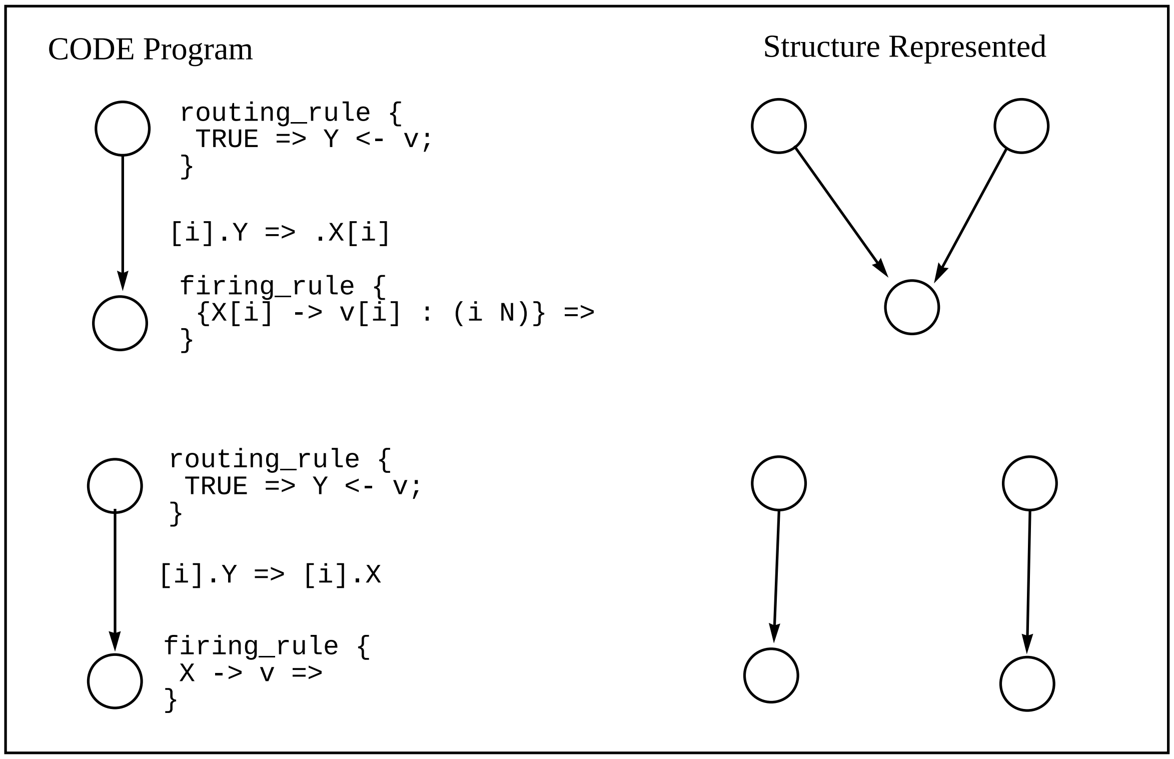
Consider an example specification.

```
[i][j].Y[k] => [i+k].X[j-1]
```

This is a strange but perfectly legal arc topology specification. It may be read as follows.

> If data come from the “from” node with indices [i][j] and port Y with index [k], they are sent to the “to” node with index [i+k] and port X with index [j-1].

The following figures show some other common cases.



```
routing_rule {
 TRUE => Y <- v;
}

[i].Y => .X[i]

firing_rule {
 {X[i] -> v[i] : (i N)} =>
}
```

```
routing_rule {
 TRUE => Y <- v;
}

[i].Y => [i].X

firing_rule {
 X -> v =>
}
```

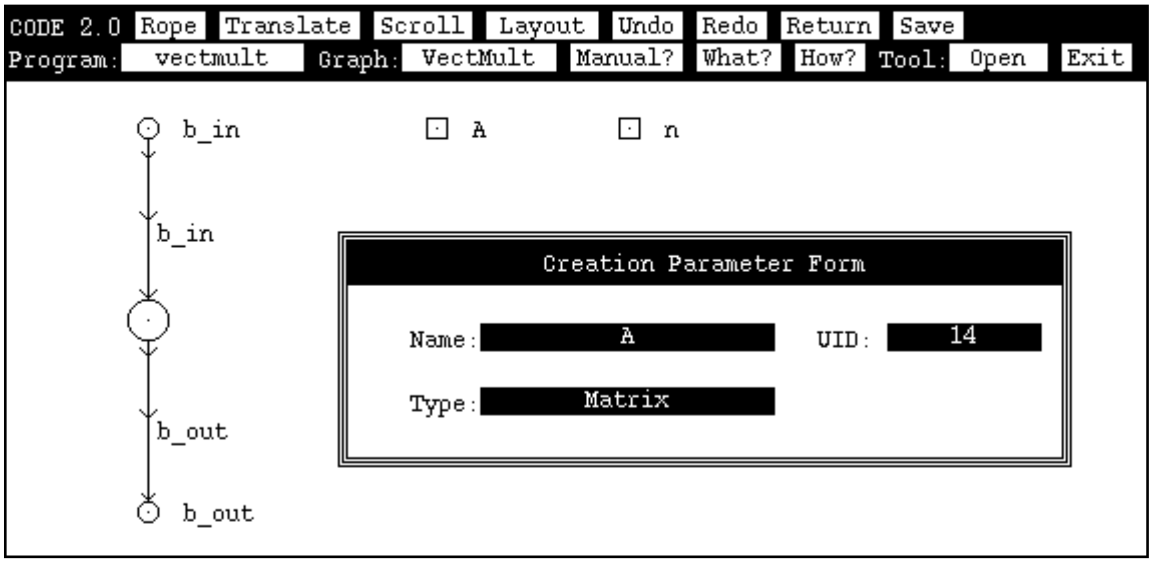
8.0 Call Nodes and Hierarchical Structuring

No useful programming language can be without facilities for hierarchically structuring programs. Conventional programming languages such as C and Fortran use Call statements and procedures for this purpose. A program is made up of a number of procedures that interact via Call statement. CODE has facilities that are entirely analogous. A CODE program is made up of a set of graph instances that interact via Call nodes. Graphs are like procedures in that they are the basic unit of hierarchical structuring. Call nodes are like Call statements.

8.1 Creating a Graph

When CODE is first run and given the name of a new graph (.grf) file, it creates the file and a graph called “main” within it. It is intended (but not required) that your Start node be in this graph. You can create another graph which can be called from “main” or any other graph that you create. Graphs can even be recursive. To create a graph, click the create cursor on the Graph button and enter a name for the new graph. The name must be a legal CODE identifier.

Let us consider the elements of a CODE graph. Just as Fortran procedures have formal parameter lists to define their interface, CODE graphs have Interface and Creation Parameters nodes to define theirs. This is perhaps best explained by an example. Suppose we wish to create a graph that could be called multiple times to multiply a series of vectors by a fixed matrix to produce a vector result. Figure 8.1 shows such a graph.



Creation Parameter Form

Name: A UID: 14

Type: Matrix

Figure 8.1 - VectMult Graph

This graph has three input formal parameters and a single output formal parameter. All formal parameters to graphs are either interface nodes or creation parameters. These nodes must be named (with unique names that are legal CODE identifiers) since their names form the graph’s interface.

Parameter	Type	Node Type
b_in	Vector	Input Inteface node
A	Matrix	Creation Parameter node
n	int	Creation Parameter node
b_out	Vector	Output Interface node

Let us first discuss the Interface nodes. They have only two attributes, a name and a type name. These nodes exist because arcs in the calling graph must be bound to ports or shared variables in the called graph, but one would not want such things to form a graph's interface because they are too "local"-- they are defined within nodes. Interface nodes serve as aliases for ports in UC nodes or shared variables in Name Sharing Relation nodes.

For example, the UC node in our example graph has an input port called "b_in," and we wish the calling graph to send data to it. So, we create an interface node and give it a name. The example uses name "b_in." It is the same as the port name in this case, but need not be. Next, an arc is drawn from the Input Interface node to the UC. Its attribute form contains a "Port to Connect to" field. It's value should be the name of the port the Interface node aliases, "b_in" in this example. All of these annotations are shown in figure 8.1.

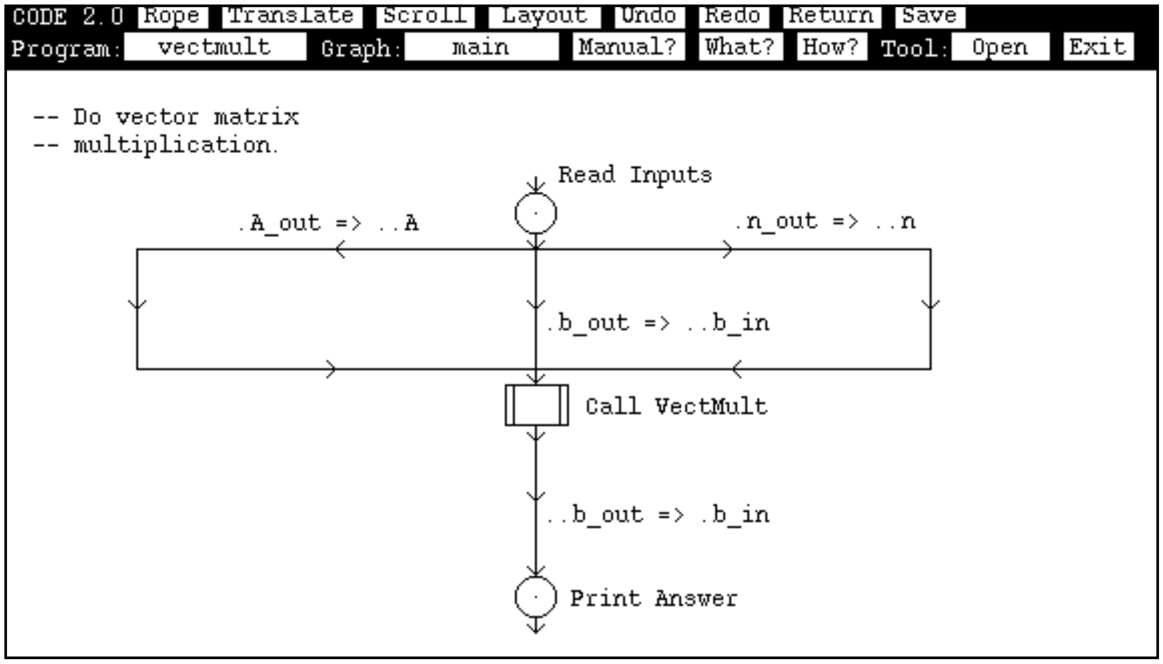
The Output Interface node works in the same way. It aliases a port in the UC that happens to be called "b_out."

Creation Parameter nodes represent a special kind of shared variable. They receive exactly one value from the calling graph-- at the moment that the graph is created. This variable can then be read from anywhere within the graph containing the Creation Parameter node.

Our example has a Creation Parameter called "A." Its attribute form is shown in figure 8.1. It has bound to it the matrix that will be used in all of the vector-matrix multiplications. Variable "A" is of type Matrix and can be read from any point in the graph (including the UC node that does the multiplication) just as though it were a local variable. It is illegal to write to "A" however. Also, no arcs may be drawn to or from Creation Parameter nodes. There is no need in any case.

8.2 Calling a Graph

Call nodes are use to call one graph from another. Arcs that enter or leave a Call node are actual parameters to the called graph. They are bound to formal parameters in the called graph by means of their arc topology specifications. Figure 8.2 shows a graph with a Call node that calls the example graph defined above.



```
-- Do vector matrix
-- multiplication.
```

Figure 8.2 - Main Graph

All arc topology specifications are shown. Notice that names of Interface and Creation Parameter nodes appear as well as two dots instead of the usual one. The specification

```
.A_out => ..A
```

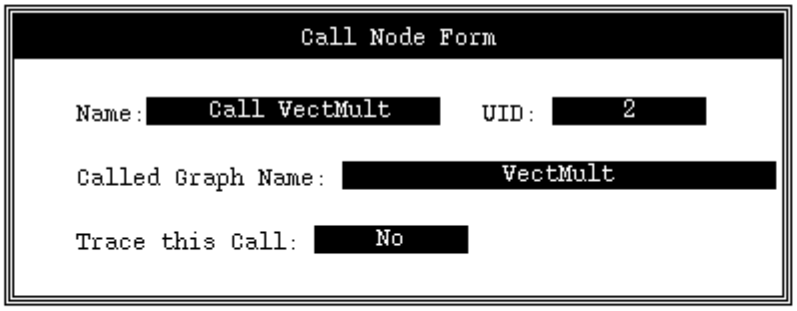
binds an output port in UC “Read Inputs” with name “A_out” to the Creation Parameter A in the graph called. The specification

```
..b_out => .b_in
```

binds the Output Interface node called “b_out” in the called graph to port “A” in the UC with name “Print Answer.” Recall that “b_out” was an alias for port “b_out” in the UC in the called graph. Hence, that port is bound to port “b_in.”

The Call node’s attribute form contains a field for the name of the graph called. This form is shown in figure 8.3.

One very important fact is that a different instance of a graph is associated with every different instance of a Call node. Hence, if a program contains two Call nodes that are set to call the same graph name, they will call different instances of the graph.


Call Node Form

Name:	Call VectMult
UID:	2
Called Graph Name:	VectMult
Trace this Call:	No

Figure 8.3 - Call Node Attribute Form

This is also true for graphs called from different instances of the same Call node. Call nodes can be replicated like other nodes in CODE. Instances are named by integer-valued indices as always. This, if fact, is the reason for using two dots instead of one on arc topology specification that go to or from Call nodes. The general form for the topology specification of an arc that goes from a UC to a Call node is as follows.

```
[ident]•••[ident].NAME[ident]•••[ident] =>
[expr]•••[expr].[expr]•••[expr].NAME[expr]•••[expr]
```

The left hand side has only one dot. The identifiers before it refer to the indices of the UC node and identifiers after the port name refer to the indices of the port on the UC. The right hand side has two dots. The first espression list refers to the indices of the Call node. The second refers to the indices of the UC or Name Sharing Relation node within the graph called, and the expression list after the port name refers to port indices.

The general form for an arc going from a Call node to a UC node is shown below. Notice that the two dots are now on the left hand side. The first list of identifiers refers to the indices of the Call node and so on.

```
[expr]•••[expr].[expr]•••[expr].NAME[expr]•••[expr] =>
[ident]•••[ident].NAME[ident]•••[ident]
```

As you might guess, the general form for an arc that goes from a Call node to a Call node has two dots on both sides.

```
[expr]•••[expr].[expr]•••[expr].NAME[expr]•••[expr] =>
[expr]•••[expr].[expr]•••[expr].NAME[expr]•••[expr]
```

An example might be useful at this time. Figure 8.4 shows a CODE program and the communication structure defined by it. Remember that a separate graph instance is called from each call node instance.

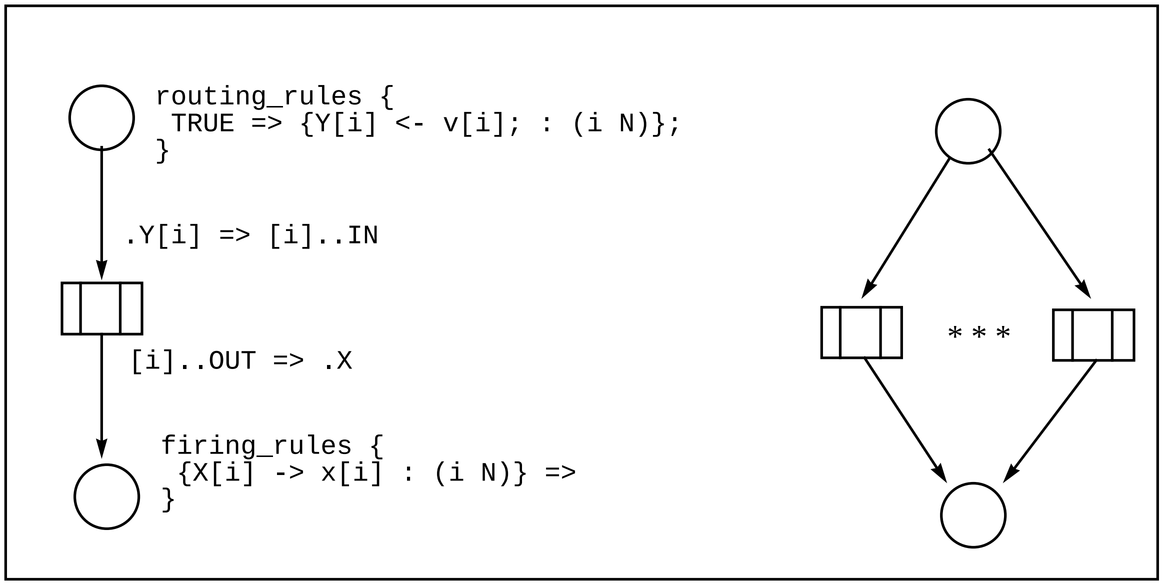


Figure 8.4 - Example Replicated Call

We assume that the called has an Input Interface node called “IN” and an Output Interface node called “OUT.” The number of calls is determined by “N” in the top UC’s routing rules.

9.0 Name Sharing Relations

Name Sharing Relation nodes are a mechanism for declaring variables that will be shared among a set of UC nodes. Theses nodes contain definitions and initializations of shared variables but do not in themselves specify which UC nodes will access which variables and in which way. That is done by shared variable declarations in UC nodes and by arcs that bind UC shared variables to shared variables in a Name Sharing Relation node.

A Name Sharing Relation node's attribute form contains a specification field in which stanzas must be supplied in much the same manner as for a UC node. However, the Name Sharing Relation node's specification is substantially simpler. Here is an example showing all of the possible stanzas in the required order.

```
shared_vars {// These may be accessed by UCs.
        int k;
        Mat2 M;
        Vector b[n]; // storage allocated-- n must be a creation parameter.
        real x;
}
vars { // These are for use by the initialization computation.
        int i; // vars section is exactly like a UC's.
}
init_comp { // init_comp section is exactly like a UC's.
        i = 0;
        while (i < n) {
                b[i] = 0.0;
                i = i + 1;
        }
}
```

The vars and init_comp stanzas are exactly like a UC node's. The initialization computation is run when the Name Sharing node is created, before any of the shared variables can be access by UC nodes.

The shared_vars stanza is not exactly like a UC's. It is illegal to specify "reader" or "writer", but it is legal to allocate storage.

We have discussed how to create shared variables. Let us now discuss how to use them from UC nodes. The first step is to declare one or more shared variables in a UC's shared_vars stanza. Here it is illegal to allocate storage. That was done in the Name Sharing Relation node. However, a use specification, either "reader" or "writer" must be provided. This permits the CODE system to automatically synchronize access to the shared variable. There are no explicit locking primitives. Access control is automatic.

Once a shared variable has been declared in a UC, it must be bound to a shared variable in a Name Sharing Relation node by means of an arc with an arc topology specification. The specification is much like that for a dataflow arc, but no port indices are permitted.

One should notice that a UC shared variable is much like a port. It is a name that is to be bound to something outside of the UC node itself. The arc topology specification provides this binding.

An example may be useful. Figure 9.1 shows the graph for a simple program. In it, nodes A and B have firing rules and dataflow arcs that cause them to fire repeatedly. Each time they fire, they increment a shared integer.

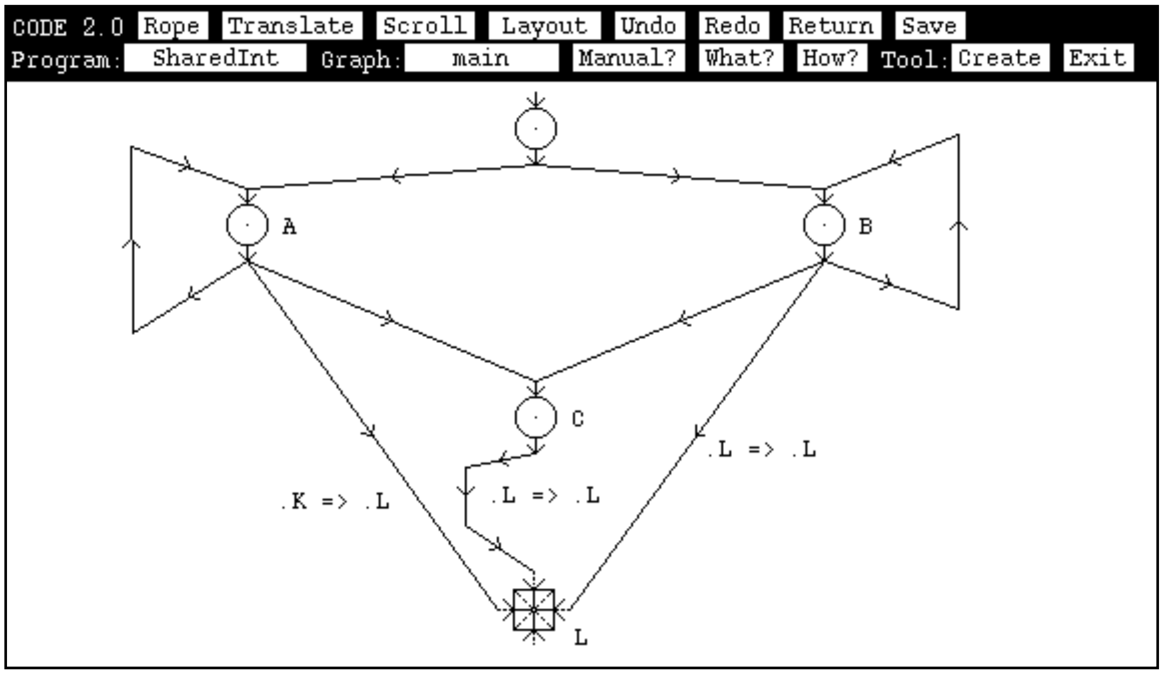


Figure 9.1 - Name Sharing Relation Example

After A and B fire the specified number of times, they send dummy values to C which cause it to fire. It prints the value of the shared integer. A and B are writers of the integer. C is a reader. We will ignore all aspects of this program but the use of the shared variable.

Node A's specification is shown below. Notice that it uses name "K" for the shared variable and declares itself to be a writer of it.

```
input_ports { int X; }
output_ports { int Y; int AGAIN; }
shared_vars {
 int K writer;
}
vars { int v; }
firing_rules {
 X -> v => v = v - 1;
}
comp {
 K = K + 1; // Increment shared int
}
routing_rules {
 v == 0 => Y <- v; &&
 v > 0 => AGAIN <- v;
}
```

Node B is the same except that it uses name L.

The specification of the Name Sharing Relation node follows.

```
        shared_vars {
                int L;
        }
        init_comp {
                L = 0;
        }
```

Node A's shared variable name K is bound to shared variable L by means of the arc drawn from UC node A to the Name Sharing Relation node. The arc topology specification is as shown below.

```
        .K => .L
```

The general form for an arc connecting a UC node to a Name Sharing Relation node does not permit port indices. See the Arcs section of the Reference Manual appendix for other cases involving Name Sharing Relation nodes.

```
        [ident]•••[ident].UCSHVARNAME => [expr]•••[expr].NSRELSHVARNAME
```

One always draws arcs <u>to</u> Name Sharing Relation nodes, even if the arc must "cross" Call boundaries.

There are some limitations on the use of shared variables within a UC node. They may be used only within the UC's comp and routing_rules stanzas, and it is of course illegal to write to a shared variable that has been declared read-only.

In addition no two shared variables within one UC node may be bound to the same shared variable in a Name Sharing Relation node-- even when different instances (meaning nodes with indices) of the Name Sharing Node are involved.

10.0 Translating, Running, and Measuring CODE Programs

Before a CODE program can be run, it must be translated into a form that can be compiled by some native compiler on the parallel machine you wish to use. You do this translation by clicking on the Translate button and selecting a target machine. The options and form of the results vary from one selection to another. The basic step is to click on the Translate button and select an architecture. At this time, only “Sequent” is implemented.

If CODE detects either a syntactic or semantic error in your program, it will display an error message and refuse to generate a parallel program. Error messages will appear as text in the xterm window from which you ran code. The text will attempt to state the location of the error, giving the name of the graph and node or arc provided you named them. It also gives an integer UID that uniquely identifies the attribute form containing the error. If you look at a node or arc attribute form, you will notice that its UID is shown.

Unfortunately, there is no feature in CODE to open the form associated with a given UID. You have to find it yourself. Clearly, this is a weakness in CODE’s user interface that ought to be fixed.

Also, recall that CODE beeps whenever you close an attribute form in which you have made a syntax (not semantic) error.

There are some attributes that control the translation process. In the Program attribute form, there is a sub-form called Translation Options. This form has the following fields.

Summary Statistics: if set to “Yes” causes CODE to record execution times and count for all nodes. (Not implemented).

Optimize: If set to “Yes” the program is optimized for execution speed by both the CODE system and the native compiler.

Object Files To Link: Form in which you can enter the names of object (.o) files that you wish to have linked with your CODE program. Separate the names with spaces. This option is convenient if you wish to enter sequential functions into files outside of the CODE system. You must place a function signature into the Program form for all such external functions that are called directly from CODE. File names defined in this form are placed into the Makefile that CODE generates.

Trace What Nodes: Menu that selects what nodes will be traced when the debugging environment that supports them is complete. For now, if you pick “All Nodes”, a message will be written whenever a node fires. “Selected Nodes” does nothing.

Buffer Even Traces: (Not implemented.)

There are also trace options mentions in Call and UC node attribute forms. These are not yet implemented.

One of the choices in the Translate menu is “List.” This is not an architecture. Rather, List causes a text representation for your program to be written to standard output, provided your program contains no syntax errors. This facility is primarlily a debugging tool used by the CODE system’s developers, but it may be useful to general users as well.

10.1 • Sequent Symmetry Translation

Suppose your program is called MyProg and hence is stored in file MyProg.grf. The Sequent translation process creates a directory called MyProg.sequent that contains a C version of your program. It uses routines from the FastThreads package from the University of Washington to create and manipulate parallel structures.

To run your program, you must first transfer the contents of MyProg.sequent to a Sequent Symmetry. It is most convenient if you store your program on a disk that is mounted both on the workstation on which you are running CODE and on the Sequent. It that case, you need only “cd” to MyProg.sequent.

CODE automatically creates a Makefile to build the executable version of your program. Just type “make” while in directory MyProg.sequent. Assuming no bugs in the CODE system, any compilation errors will be caused by mistakes you made in the definitions of your sequential functions. CODE does not parse these definitions so it cannot check them for you. If there is an error, you must return to CODE and fix it. Use the native compiler’s messages (and look at CODE’s output files) for clues.

If the make succeeds, the executable image will be stored in MyProg. Run it by entering

```
MyProg -n#
```

at the Sequent UNIX prompt. The # is an integer that determines how many processors (really UNIX processes) will be allocated to running your program. For example, “MyProg -n6” will run your program using 6 processes.

There are a few things you should keep in mind when translating for the Sequent. CODE overwrites all of the files it created in MyProg.sequent whenever you translate. Hence, any changes you make in the files CODE produces will be lost if you translate a second time. In general, there is no reason to change files that CODE generates anyway.

Also, it is best if you run your parallel programs from a disk that is directly attached to the Sequent rather than remoted mounted using NFS. The reason has to do with an interaction bewteen NFS and how the Sequent implements shared memory. If you run from a remote-mounted disk, you will notice potentially long delays between the terminate node firing and the time you get the UNIX prompt back.

11.0 Input/Output in CODE

CODE does not at this time include I/O in its model of computation. This is, of course, a serious flaw since facilities for I/O vary greatly from one parallel machine to another. Until CODE includes I/O capabilities, it is not possible to use it to write truly portable programs that do I/O. For now, you must do I/O from your sequential functions. Whatever I/O you include must work with the parallel structures CODE creates for a particular architecture.

Be aware that I/O to the same file or device from nodes running in parallel with each other can cause unintended results. For example, if two nodes that run in parallel write to the same file, there is no way to know which will write first. They may even interleave their writing. The behavior in this regard can also be machine-dependent.

11.1 • Sequent Symmetry I/O

You may use standard UNIX I/O facilities from any node when your target is the Sequent Symmetry. For example, C library routines printf and scanf work. There does appear to be a possible problem with scanf when you redirect stdin to come from a file or pipe. You may be better off doing direct file I/O.

12.0 Common CODE Conceptual and Usage Errors

Port indices are not array indices.

Do not confuse port indices with indices of arrays that will be sent on the port. They are unrelated concepts. They type of data that a port will carry is determined by the port's type. For example, suppose we have the following type and variable defintions.

```
type mat1 is array of real; // These go in the program form
type mat2 is array of mat1;

output_ports { mat2 Y; }
vars { mat2 M[10][5]; }
```

The following routing rule sends all of M out on port Y[i] for i in 0..N.

```
routing_rules {
 TRUE => { Y[i] <- M; : (i N) };
}
```

The binding "Y[i] <- M[i];" is illegal. It asks that a value of type mat1 be placed on a port of type mat2. You would want to use a port of type mat1 in this case.

Arrays and structs are consumed when sent out on arcs.

Do not forget that arrays and structs are consumed when they are sent out on arcs. Storage for them is lost in the sending node and must be reallocated if the node is to be fired again. You can use the operator "new" to allocate storage.

Do not assume that rules or guards within rules are evaulated in any particular order.

Also, all binding are done after all conditions are checked. Hence the following rules are not very likely to be what you intend.

```
firing_rules {
       N -> n, { X[i] -> v[i] : (i n) } => || // n uses old value!
       N -> n, Y -> v[n] => // Cannot depend on order!
}
```

You <u>can</u> assume that bindings (on the right of a =>) are done in order from left to right.

Do not mismatch index counts.

If a routing rule refers to a port with (for example) two indices such as X[i][j], the arc bound to that port should have two indices associated with the port.

```
.X[i1][i2] => .... // port indices match routing rule in number (names can differ).
```

Make sure that function signatures match function defintions.

The CODE system has no way to perform any syntactic or semantic checks on function bodies. It is up to you to get them right. Messages from the native compiler on the parallel system may help. This is one good reason to keep your sequential routines in a file outside of the CODE system and use the Object Files to Link field in the program's attribute form's Translation Option field.

In particular be sure that function signatures match function bodies. CODE will not flag a mismatch as an error. For example, if you have this CODE function signature

```
void foo(int *n);
```

you must have a function body (assuming you are using traditional C) like the this.

```
void foo (n)
int *n;
{
  • • •
}
```

Appendix 1: Known Bugs and Limitations

• You may never use more than seven indices on anything.

• Click on the Save button to save you work. Do this often as insurance against crashes.

• Run code via “code2 filename.grf” not “code2 dirname/filename.grf”. In other words, run code from the directory that contains your graph file.

• Use CODE’s Exit button to leave code. Using an X-Windows “close” or “quit” causes a crash.

• The Copy Graph function (click Copy Cursor on Graph buttons) causes strange error messages to be displayed. Use it at your own risk!

• “What” online help is not implemented. “How” help is implemented.

• The Manual button displays the manual for a program that has nothing to do with CODE!

• Put all function signatures and type definitions in the Program form. Graph and UC scope is not correctly implemented for these items.

• Error messages are often cryptic.

• It is often hard to find the graphical item an error message refers to.

• The CODE language does not contain string and character literals.

• Your graph file name should be somename.grf where “somename” is a legal CODE identifier.

Appendix 2: Language Reference

This appendix defines the CODE graphical parallel programming language. It is intended more as a reference guide than as a tutorial. If you know nothing about CODE, this is not the place to begin reading. Readers are assumed to be familiar with programming language concepts and terminology commonly used in describing programming languages.

2.1 Overview

CODE programs consist of a set of graphs that users draw using a mouse. Each graph consist of nodes and arcs. The user must annotate the nodes and arcs with extra information. This annotated graph may then be automatically translated into a parallel program for a variety of target architecures.

There are several types of nodes in graphs. The most important type is the Unit of Computation (or UC) Node. UC nodes represent sequential computations. Users annotate UC nodes to define their sequential computations, the conditions under which they are allowed to execute, and more.

Arcs represent the flow of data from one node to another.

Graphs in CODE 2 play the same role that subroutines play in sequential languages. They allow programs to be hierarchically structured. One graph calls another by means of Call nodes. Arcs incident on a Call node are actual parameters to the called graph.

Formal parameters are defined using special nodes in the called graph.

Name sharing relation nodes permit UC nodes to share variables in a controlled fashion. CODE is not based upon a pure data flow model.

All CODE 2 objects (nodes, arcs, etc.) may be instantiated any number of times while the parallel program is running. Different instantiations of the same object are distinguished by means of integer valued indices.

The purpose of this appendix is to define the graphical entities from which users build programs and to specify all attributes users can or must supply.

The next few sections describe the graphical entities that make up a CODE 2 program. The specification of their attributes is of particular interest. The following notation is used.

(O) - Optional attribute. It is possible to create a correct CODE 2 program without supplying a value.
(R) - Required attribute. You must supply a value.

2.2 Programs and their Attributes

Attributes of the entire program are set in the Program Attribute Form. Click the Open cursor on the Program button to open this form.

-- Program Attributes --

• Global Types (O) -- Type definitions that apply to the entire program. See "Types".

• Global Function Signatures (O) -- Functions signatures for functions or procedures that may be called from anywhere in the CODE program. See "Functions & Function Calling".

• Global Function Definitions (O) -- Bodies for functions whose signatures are in the Global Function Signatures text box.

• Documentation (O) -- Text box for comments.

• Translation Options (O) -- Sub-form with the following fields.

Summary Statistics (O) -- if set to "Yes" causes CODE to record execution times and count for all nodes. (Not implemented).

Optimize (O) -- If set to "Yes" the program is optimized for execution speed by both the CODE system and the native compiler.

Object Files To Link (O) -- Form in which you can enter the names of object (.o) files that you wish to have linked with your CODE program. Separate the names with spaces. File names defined in this form are placed into the Makefile that CODE generates.

Trace What Nodes (O) -- Menu that selects what nodes will be traced when the debugging environment that supports them is complete. For now, if you pick "All Nodes", a message will be written whenever a node fires. "Selected Nodes" does nothing.

Buffer Even Traces (O) -- (Not implemented.)

Every CODE 2 program must have a graph called "main". It is created automatically when a CODE graph file is created. Graph main must contain exactly one node that has its Start Node attribute set. (See UC Nodes).

2.3 Graphs and their Attributes

Graphs are CODE's equivalent of subroutines in conventional languages. They must have names, and these names must be legal CODE 2 identifers (see Identifiers & Contants). No two graphs in a program may have the same name. Graphs are created by applying the Create cursor to the Graph banner button.

Formal parameters for a graph are defined by its Interface Nodes (both input and output) and its Creation Parameter nodes. All of these nodes must have names that are legal CODE 2 Identifiers, and they must be unique within the graph.

Open the currently displayed graph’s attribute from by clicking the Open cursor on the Graph button.

-- Graph Attributes --

• Name (R) -- The graph’s name. It will be entered in Call nodes to identify the graph called. The name must be a legal CODE identifier. Graph names must be unique within the program.

• Graph Types (O) -- Type definitions that apply to the graph. See “Types”.

• Graph Function Signatures (O) -- Functions signatures for functions or procedures that may be called from anywhere within the graph. See “Functions & Function Calling”.

• Graph Function Definitions (O) -- Bodies for functions whose signatures are in the Graph Function Signatures text box.

• Documentation (O) -- Text box for comments.

2.4 Interface Nodes and their Attributes

Interface nodes (along with Creation Parameter nodes) define a graph’s interface. Interface and Creation Parameter nodes represent formal parameters and so must be given names that are unique within their graph. These names are used in formal/actual parameter binding as described in Graphs and Graph Calling. They appear on arc topology specifications of arcs incident upon a Call node that calls the graph in which the Interface nodes appear.

Only a single arc may be connected to an interface node. A separate node must be used for every formal parameter in a graph.

-- Interface Node Attributes --

• Name (R) -- The interfaces node’s name. It must be a legal CODE 2 identifier and unique among the graph’s interface and creation parameter nodes. This name will be used on arc topology specifications on arcs incident upon Call nodes that call this graph.

• Type (R) -- The name of the type (an identifier) of the data that will flow on the arc connected to the interface node.

2.5 Creation Parameter Nodes and their Attributes

Creation Parameter nodes define a formal parameter to a graph that will have exactly one value bound to it in the lifetime of a graph instance. (See Graphs and Graph Calling). The value will be bound before the graph instance is created and becomes available as a constant throughout the graph.

-- Creation Parameter Node Attributes --

• Name (R) -- The Creation Parameter node's name. It must be a legal CODE 2 identifier and unique among the graph's interface and creation parameter nodes. This name will be used on an arc topology specifications on an arc incident upon a Call node that calls this graph. The name is also used to access the value of the creation parameters as a constant anywhere within the graph.

• Type (R) -- The name of the type (an identifier) of the value that will be bound to the Creation Parameter.

2.6 Call Nodes and their Attributes

Call nodes are used to call one graph from another. Exactly one graph instance is bound to each instance of a Call node. Hence, if a graph contains two Call nodes, both of which call graph SomeGraph, the two Calls refer to *different* instances of SomeGraph. Arcs that are incident on a Call node are actual parameters to the graph being called. Their topology specifications may bind them to either an Interface node or a Creation Parameter node in the graph being called. See Graphs and Graph Calling.

-- Call Node Attribues --

• Name (O) -- The name of the Call node. A comment.

• Called Graph Name (R) -- The name of the graph to be called.

• Trace this Call (O) -- Not Implemented.

2.7 Unit of Computation (UC) Nodes and their Attributes

UC nodes are CODE 2's basic computational unit. They extract information from incoming arcs, perform a sequential computation upon it, and pass resulting information out on outgoing arcs. This process is called "firing" and takes place under conditions specified by user-supplied firing rules. Each UC has its own names for the arcs that are incident upon it. These names are called "ports" for data flow arcs and "shared variables" for arcs that connect the UC to a name sharing relation. Node's may have local variables, and these variables retain their state from one firing to the next.

-- UC Node Attributes --

• Name (O) -- The name of the UC node. A comment.

• Terminate Node (R) -- A Boolean attribute. Should be "yes" if and only if the computation is to terminate when this node fires. Only one UC node in the entire program may be a terminate node.

• Start Node (R) -- A Boolean attribute. Should be “yes” if and only if this is the node that should be instantiated to start the computation. Only one UC node in the entire program may be a start node. The system will automatically create and fire (regardless of firing rules) the instance of the start node node with no indices when the program begins.

• Node Function Signatures (O) -- Function signatures for functions or procedures that may be called from anywhere within the node. See “Functions & Function Calling”.

• Node Function Definitions (O) -- Bodies for functions whose signatures are in the node’s Function Signatures text box.

• Node Specification (R) -- Text box that permits a number of node attributes to be defined using a mostly declarative text language. All nodes in a non-trivial program will have definitions for one or more of these attributes. The attributes appear in a sequence of stanzas. Each stanza is optional, but any that are present must appear in a given order. See “UC Node Specifications” for a complete explanation. Here, we give only an example that shows all of the stanzas in the required order.

```
input_ports { // Name and type for incoming arc ports.
      Vect V1; Vect V2;
}
output_ports { // Name and type for outgoing arc ports.
      Vect OV1; Vect OV2;
}
shared_vars { // Names for shared variables. See Name
      Mat2 m reader; // sharing relations.
}
vars { // Local variables.
      Vect v; int code;
}
init_comp {
      init(code); // Computation done when node is created.
}
firing_rules { // Fire when there is data on either V1 or
      V1 -> v => || // V2. Store it in v.
      V2 -> v =>
}
comp { // Computation done when node fires.
      mycomp(v, m, code);
}
routing_rules { // If comp is 0, send v on OV1 else OV2.
      code == 0 => OV1 <- v; &&
      code != 0 => OV2 <- v;
}
```

• Documentation (O) -- Text box for comments.

• Event Trace Options (O) -- sub-form of options that have not been implemented.

2.8 Name Sharing Relation Nodes and their Attributes

Name sharing relations allow the controlled use of shared variables in CODE programs. Access to them is specified declaratively. There are no locking primitives to be used within computations.

Shared variables are defined at Name Sharing Relation nodes. UC nodes that share the variable must have an arc drawn from them to the Name Sharing Relation node. These arcs have a topology rule that binds the shared variable name used in the UC node to the shared variable name used in the Name Sharing Relation node.

-- Name Sharing Node Attributes --

• Name (O) -- The name of the Name Sharing Relation node. A comment.

• Node Function Signatures (O) -- Functions signatures for functions or procedures that may be called from anywhere within the node. See "Functions and Function Calling".

• Node Function Definitions (O) -- Bodies for functions whose signatures are in the node's Function Signatures text box.

• NSRel Specification (R) -- A text box that permits a number of attributes to be defined using a mostly declarative text language. Like UC specifications, Name Sharing Node specifications consist of a sequence of stanzas, all of which are optional. However, non-trivial nodes will always have a shared_vars stanza. Here is an example showing all stanzas. See "Name Sharing Relation Specifications" for a complete syntax.

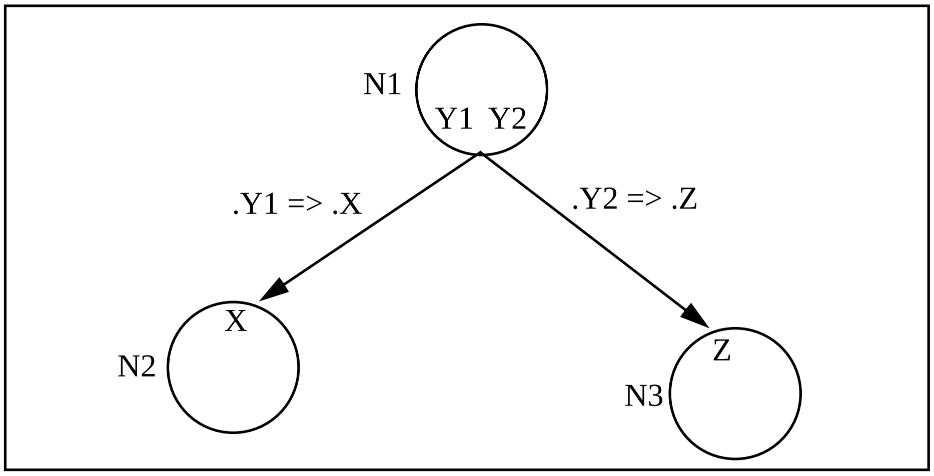
```
shared_vars { // These variables are shared.
      Matrix1 m[10];
      int MyInt;
}
vars { // Local vars for use by init_comp.
      int i;
}
init_comp { // Run when node is created.
      i = 0;
      while (i < 10) {
              m[i] = 2.5;
              i = i + 1;
      }
}
```

• Documentation (O) -- Text box for comments.

2.9 Arcs and their Attributes

Arcs in CODE bind nodes together. Their attributes depend upon the types of nodes they leave and enter. Arcs that are incident upon an interface node are the simplest. In most cases, one simply specifies the name of the port they are to bound to. See below.

Other arcs must be given arc topology specifications. The major purpose of these is to disambiguate arc conections. For example, suppose a graph has a UC node called N1with output ports Y1 and Y2, a UC node called N2 with input port X, and a UC node called N3 with input port Z.



If an arc is drawn from N1 to N2 it will be unclear whether it should be bound to output port Y1 or Y2. Arc topology specifications solve this problem. The specification

```
.Y1 => .X
```

says that the arc goes from N1’s output port Y1 to N2’s input port X. Read the rule in this way: if data come from node N1’s port Y1, then they must be sent to Node N2’s port X2. The node names need not be mentioned in the rule since the CODE system knows what nodes are involved by means of graphical context-- the user has already drawn the arc from one node to another.

Arc topology specifications may also involve indices. For example, suppose node N1 places data on its output port Y1[i] where i takes values from 0 to 9. The rule

```
.Y1[i] => [i].X
```

Says that data from N1’s port Y1[i] must be sent to node N2[i]’s port X.

Indices on the left of the “=>” are variable declarations. Hence, they must be unique and expressions are not allowed for them. Indices on the right may be expressions and, of course, need not be unique. For example, the following arc topology specifications are illegal.

```
.Y1[i][i] => .X2 // i is not unique on the LHS.
[i].Y1[i] => .X2 // i is still not unique on the LHS.
[i+1].Y1 => .X2 // i+1 is an expression.
```

The following are legal, although unusual.

```
[i].Y1[j] => [i % 2].X2[j+1]
[i][j].Y1[k] => .X2[i+j+k]
```

In the following discussion, let <varlist> stand for a list of variables in brackets-- like [i] or [ind1][j][k]. And let <exprlist> stand for a list of expressions in barckets-- like [i+4][j] or [i-j*foo(k)][g][6].

The syntax for an arc topology specification varies somewhat according to the type of nodes its arc connects. We will discuss this on a case-by-case basis, but there is a general rule. Use two dots for a Call node, otherwise use one.

- **UC node to UC node**

```
<varlist>.IDENT<varlist> => <exprlist>.IDENT<exprlist>
```

- **UC node to Call node**

```
<varlist>.IDENT<varlist> => <exprlist>.<exprlist>.IDENT<exprlist>
```

On the right of the “=>”, the first <exprlist> refers to the indices of the Call node, the second to the indices of the UC in the called graph, and the third refers to the indices of the port of the UC in the graph called. If the arc is bound to a creation parameter, the final <exprlist> is not allowed.

- **Call node to UC node**

```
 <varlist>.<varlist>.IDENT<varlist> => .<exprlist>.IDENT<exprlist>
```

This time, the extra <varlist> is on the left. It refers to the index of the Call node.

- **Call node to Call node**

```
 <varlist>.<varlist>.IDENT<varlist> => <exprlist>.<exprlist>.IDENT<exprlist>
```

- **UC node to Name Sharing Relation Node**

```
<varlist>.IDENT => <exprlist>.IDENT
```

- **Call node to Name Sharing Relation node**

```
<varlist>.<varlist>.IDENT => <exprlist>.IDENT
```

Arcs that are incident on Interface nodes do not have topology specifications. Instead, the user must supply the name of the entity the Interface node is connected to. One can think of the Interface node as aliasing an entity. For example, an Input Interface node connected to a UC node is bound to an input port in the UC. The Interface node serves as an alias for a particular port in the a particular UC. Thus, UC port names do not “leak” into a graph’s interface.

- **Input Interface node to UC node**

```
 Name of input port in UC that node aliases.
```

- **UC node to Output Interface node**

```
 Name of output port in UC that node aliases.
```

• Input interface node to Call node

```
Name of Input Interface Node in called graph that it aliased.
```

• Call node to output interface node

```
Name of output interface node in called graph.
```

• Input interface node to Name Sharing Relation node

```
Name of shared variable that is aliased.
```

• Input Interface node to Output Interface node

```
There is no arc specification. The nodes are aliases for each other.
```

2.10 Identifiers and Constants

Identifiers in CODE 2 consist of a letter or underscore followed by a sequence of zero or more letters, digits, and underscores. CODE is case sensitive so “k” and “K” are different identifiers.

Legal	Illegal
i	d-1
Abel	7Del
Point_X	hp:f
a9_	8U
A_8_y	9_a

CODE supports constants of type integer and real. Integer constants are a sequence of one or more digits. Real constants must include a decimal point. No scientific or exponential notation is allowed.

Example real constants: 1.0, 17., 0.666, .75, 123.450, 00.2300.

2.11 Predeclared Identifiers

There are several identifiers that are predeclared in in CODE.

TRUE	The integer value 1.
FALSE	The integer value 0.
GraphIndex	An integer array that contains graph index bounds. For example, if the current graph index is [3][4], GraphIndex[0] has value 3 and GraphIndex[1] is 4.
NodeIndex	An integer array that contains node index bounds.
copy	A function that returns a copy of an array ot struct. See CODE Arrays.
asize	A Function that returns the size of an array. See CODE Arrays.

2.12 Comments

Comments in CODE 2 are preceded by // and continue to the end of the line. Some examples follow.

```
comp {
      i = i + 1; // This is a comment.
      // So is this.
}
```

2.13 Scope in CODE

CODE 2 is a statically scoped language. Scopes are nested so a declaration in an inner scope can shadow a definition in an outer one. The major scopes in CODE 2 are as follows.

Program Scope - Declarations made in the program attribute form are usable anywhere in the program unless shadowed by a declaration in an inner scope.

Graph Scope - Declarations made in a graph's attribute form are usable anywhere in that graph unless shadowed by a declaration in an inner scope.

UC Node Scope - Declarations made in a Unit of Computation node attribute form are usable anywhere in that node unless shadowed by a declaration in an inner scope.

NSRel Node Scope - Declarations made in a Name Sharing Relation node attribute form are usable anywhere in that node unless shadowed by a declaration in an inner scope.

Index variables declared in replicators within rules or on the left hand side of an arc topology rule may shadow declarations in outer scopes.

2.14 Types in CODE

CODE supports integers, reals, characters, structures (records), and arrays. All types must have a name. Type names int, real, and char are predefined qnd may not be redefined.

Predefined Types

int
real
char

CODE does not have a Boolean type, but it does have Boolean operators and conditions. Like C, it uses the integer value 0 to represent FALSE. Any other integer value represents TRUE.

Array and structure types must be given a name by means of a type definition before they can be used. Type definitions may appear in the following places.

• Program attribute form - The type name is usable thoughout the entire program (program scope).
• Graph attribute form - The type name is usable thoughout the graph (graph scope).

The syntax of a type definition is as shown. NEWTYPE and OLDTYPE are identifiers. OLD-TYPE must be declared in the form before the type definition or in an outer scope.

```
<typedef> :=  type NEWTYPE is OLDTYPE ; |
              type NEWTYPE is <array_type_spec> ; |
              type NEWTYPE is <struct_type_spec> ;

<array_type_spec> := array of OLDTYPE ;

<struct_type_spec> := struct { <member_list> } ;

<member_list> :=     OLDTYPE IDENT ; <member_list> |
                     OLDTYPE IDENT ;
```

Array type defintions in CODE 2 are unusual in that they do not include dimensions. This is because CODE 2 arrays are dynamic. Array sizes are associated with variable instantiations. Here are some examples of type definition. Note that we are not declaring variables. We are defining types.

<u>Examples of Legal Type Definitions</u>

```
type foo is int;
type bar is foo; // foo must already be defined to be a type.

type mat1 is array of real; // Vector of reals.
type mat2 is array of mat1; // Only way to declare a 2D array.

type Point is struct {
       real x;
       real y;
};

type PolyLine is array of Point;

type line is struct {
       Point Start;
       Point End;
};
```

CODE is a statically typed language and is type-safe with the exception of ellipses (...) in function signatures. Despite the fact that all types must be named before that can be used, CODE uses type structure (rather than name) to determine when types are equivalent. For example, if we define the following types:

```
type mat1 is array of real;
type vect if array of real;
type diff is array of int;
```

Types mat1 and vect are equivalent. A variable of type mat1 may be used anywhere a variable of type vect may be used. Type diff is not equivalent to either type mat1 or type vect.

Just to be negative, here are some examples of illegal type definitions that you might wonder about.

Examples of Illegal Type Definitions

```
type bad is array of array of int; // Need to use 2 typedefs
type bad2 is int // Missing semicolon.
type bad3 is array of struct { // Need to use 2 typedefs
       int i;
       int j;
};
```

2.15 Expressions and Operators

Expressions in CODE may appear in UC Nodes, Name Sharing Relation Nodes, and Arcs. The operators allowed in CODE 2 are shown below in precedence order.

Operator	Associativity
[] .	left to right
! -(unary)	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right.

The following table shows the meaning of the operators, the types they are allowed to operate on and the types they return. Notice that there is no Boolen type. Integer constants 1 (for True) and 0 (for false) are used instead. Predeclared integer variables TRUE and FALSE have these values. Some instructions accept operands of different types. One of the operands will be promoted to the type of the other according to the following hierarchy.

char => int => real

The return type “mixed mode” means that the type of the operand whose type is highest in the hierarchy is returned.

Table 1: Operators and Types

Operator	Meaning	LHS	RHS	Returns
[]	array index	array	int	array base type
.	struct select	struct	(na)	member type
!	boolean not	(na)	int	int
-	unary minus	(na)	int, real	mixed mode
*	mult	int, real	int, real	mixed mode
/	division	int, real	int, real	mixed mode
%	mod	int, real	int, real	mixed mode
+	addition	int, real	int, real	mixed mode
-	subtraction	int, real	int, real	mixed mode
<	less than	int, real, char	int, real, char	int
<=	lt or equal	int, real, char	int, real, char	int
>	greater than	int, real, char	int, real, char	int
>=	gt or equal	int, real, char	int, real, char	int
==	equal	int, real, char	int, real, char	int
!=	not equal	int, real, char	int, real, char	int
&&	boolean and	int	int	int
||	boolean or	int	int	int

Function calls are also allowed in expression. Types of actual parameters must match types of formal paramaters. See “Functions and Function Calling”.

The operator new may be used to allocate new arrays or new arrays within structs. Here is an example.

```
new Mat2[10][10]
```

Mat2 is a type name. See “CODE Arrays”.

2.16 Statements

Statements may appear in initial computation (init_comp) and computation stanzas in UC Nodes and Name Sharing Relation Nodes. The set of statements offered is limited. It is intended that significant computations be performed by calls to externally defined functions.

The syntax of CODE statements is as follows.

```
<stmt> :=     <block_stmt> | <while_stmt> | <if_stmt> | <assign_stmt> |
              <null_stmt> | <call_stmt>

<block_stmt> := { <stmt> ... <stmt> }

<while_stmt> := while ( <expr> ) <stmt>

<null_stmt> := ;
```

The condition expression in a while statement must be of type int with zero representing FALSE and any other integer value representing TRUE.

```
<if_stmt> :=  if ( <expr> ) <stmt> |
              if ( <expr> ) <stmt> else <stmt>
```

The condition expression in an if statement must be of type int as with the while statement.

```
<assign_stmt> := <lhs_expr> = <expr> ;

<lhs_expr> := <ident> |
              <lhs_expr> . <ident> |
              <lhs_expr> [ <expr> ]
```

In an assignment statement, the lhs_expr and the expr be of any type. When assigning arrays and structs, types must match exactly. Types int, real, and char will be converted.

Call statements are much like void function calls in C. However, there is no need to use an address operator (&) when passing by reference. (See Functions and Function Calling).

```
<call_stmt> := <ident> ( <argument_list> ) ;
```

Here are some example CODE statements.

```
InitMat(M);
i = 0;
while (i < 10) {
       M[i] = p.x + i;
       if (i % 2) {
              i = i + 2;
              M[i] = M[i] + 1;
       } else
              i = i + i;
}
```

2.17 Functions and Function Calling

Sequential functions (and procedures but we will often just use the word function) are very much a part of CODE even though you cannot define them within it. Rather, they are written in a regular sequential language, most often C. It is expected that most significant UC computations will be specified using such external sequential functions. CODE supports their use in several ways.

1. You must provide function signatures (sometimes called function prototypes) so that CODE 2 can type check calls to your external functions.

2. CODE 2 provides function definition text boxes at program, graph, and node scope so that you can enter function defintions (function bodies) for C functions. These function bodies are not parsed by CODE 2. They are merely packaged for compiling by your system's native C compiler.

3. CODE 2 produces file c2_globtypes.h that contains C typedefs equivalent to all of the CODE 2 types defined by the user at the program scope level. This file may be included in C files that are to be separately compiled and linked with the parallel code that CODE 2 produces. Signatures for these externally defined functions must be provided at the program scope level. Compiling and linking separate C files is a system-dependent process.

Sequential functions may be called anywhere expressions are allowed in CODE. Procedures may be called in computation and initial computation sections by means of Call statements (see Statements).

A function signature must be provided before any function (or procedure) can be called. This permits CODE 2 to type check arguments. CODE 2 function signatures are a subset of what is permitted in ANSI C.

```
<func_sig> := <typename> <ident> ( <formal_arg_list> ) ;
```

Use typename "void" for procedures. The identifier is the function name.

Formal arguments have the following syntax.

```
<formal_arg_list :=   <formal_arg> , ..., <formal_arg> |
                      ...
<formal_arg> := <typename> <ident> | <typename> * <ident>
```

The identifier is the argument name. The "*" indicates that the argument is to passed by reference. Arrays and structs are always passed by reference. A "..." argument list for a function specifies that CODE will not check arguments to the functions for correct type or number. Use this feature with caution. Note that CODE does not support function signatures that mix regular arguments with the "..." form.

```
foo(int i, ...); // This is illegal in CODE
```

Here are some example function signatures. Note the the formal parameter name is required in CODE, unlike C.

```
int foo(int i); // takes an int by value, returns an int.
int foo(int *i); // takes an int by reference, returns an int.

void foo(); // procedure that takes no arguments.

real VectSum(Vect v1, Vect v2); // Takes two args of type Vect.

real foo(int i, int *j, int k); // 2nd arg by reference.
```

CODE’s argument type checking may be bypassed by means of ellipses.

```
int foo(...); // Don’t type check arguments.
```

If you supply ellipses, you may supply no other arguments.

Function signatures may be entered into Function Signature text boxes in the program, graph, and node attribute forms.

Function calls in CODE are not quite like they are in ANSI C. If you pass an argument by reference you need not use an address operator. For example, assume there is a function foo with the following signature and that k in an integer variable.

```
int foo(int *i); // foo’s signature
```

In CODE, a call to foo would take the form

```
foo(k)
```

rather than

```
foo(&k) // illegal! CODE has no & operator.
```

as would be required in C.

Also, if an argument is passed by reference, an expression may not be used as a formal parameter. For example

```
foo(k+7) // Illegal to pass expression by reference.
```

is an illegal call.

2.18 UC Node Specifications

This section describes the text language that defines UC node specifications.

A UC specification consist of the following stanzas in the given order. Any stanza may be ommitted in a node that does not require it.

<input_ports> -- Local names and types for incoming arcs.

<output_ports> -- Local names and types for outgoing arcs.

<shared_vars> -- Local names and types for shared variable plus use rule.

<vars> -- Local variables defintions.

<init_comp> -- Executes when node is created at runtime.

<firing rules> -- States when node can fire and how to get data from incoming arcs.

<comp> -- Executes after a firing rule binding.

<routing_rules> -- States what data are to go on what outgoing arcs.

The syntax and purpose of each stanza will be presented in turn, but first we will introduce the syntax for replication. Guards, bindings, and rules in both firing rules and routing rules may be replicated. The syntax of replication is

```
<replication> :=

       { <thing_to_be_replicated> : <repl_list> } |

       { <thing_to_be_replicated> : <repl_list> : <expr> } |

<repl_list> := ( IDENT <expr>) ... ( IDENT <expr> )
```

For example, { <thing> : (i N) (j M+1) } replicates <thing> for values of i from 0 to N-1 and j from 0 to M. Here is another example.

```
{ m[i] = v[i]; : (i 2) } // same as m[0] = v[0]; m[1] = v[1];
```

The optional expression after the <repl_list> is a constraint. For example

```
{ <thing> : (i N) (j N) : i < j }
```

replicates <thing> for value of i and j from 0 to N-1 such that i < j. There may be at most one constraint expression. One could accoplish the same thing with

```
{ <thing> : (i N) (j i) }
```

We now present the stanzas that make up a UC node specification.

- **Input Ports Stanza**

```
<input_ports> :=
       input_ports { TYPENAME PORTNAME ; ... TYPENAME PORTNAME ; }
```

An input port is a node’s local name for an incoming dataflow arc. Input port names appear in arc topology specifications to bind an arc to a particular input port. See Arcs and their Specification.

Example:

```
        input_ports { int I; real x23; Matrix2d m; }
```

• Output Ports Stanza

```
        <output_ports> :=
                output_ports { TYPENAME PORTNAME ; ... TYPENAME PORTNAME ; }
```

An output port is a node’s local name for an outgoing dataflow arc. Output port names appear in arc topology specifications to bind an arc to a particular output port. See Arcs and their Specification. Example:

```
        output_ports { int J_index; real Y; Matrix2d m; }
```

• Shared Variables Stanza

```
        <shared_vars> :=
                shared_vars { TYPENAME VARNAME <use> ; ... TYPENAME VARNAME <use> ; }

        <use> := reader | writer
```

Variables declared in the shared variables stanza will be bound to shared variables that are declared and instantiated in a Name Sharing Relation node by means of an arc be drawn from the UC to the Name Sharing Relation node. The arc’s topology specification binds the UC’s shared variable name to the Name Sharing Relation’s shared variable name. Hence, UC’s have local names for the shared variables they use. Shared variables in UC nodes act somewhat like ports. There is no ”variable” directly associated with them. Instead they are bound to variables declared in Name Sharing Relation nodes. UC shared variables also have use rules that declare whether the UC will read or write the shared variable. Example:

```
        shared_vars {
                Matrix2D m reader;
                int Shared_counter writer;
        }
```

Shared variable names may be used only in the comp stanza of a UC (and in the init_comp stanza of the name sharing relation in which they are declared).

It is illegal to bind two shared variable names to two different instances of the same variable. For example, suppose a UC declares shared variable names s1 and s2 and that there is a Name Sharing Relation node that declared with shared variables x and y. A user could bind s1 to x and s2 to y be drawing two arcs from the UC node to the Name Sharing Relation node and giving them the following topology specifications.

.s1 => .x
.s2 => .y

This is legal. But the following arc topology specifications are not legal since they bind two shared variable name to two different instances of the same variable.

```
[i].s1 => [i].x
[i].s =>[i+1].x
```

• Variable Declaration Stanza

```
<vars> :=
       vars { <vardecl> ; ... <vardecl> ; }

<vardecl> := TYPENAME VARNAME | TYPENAME VARNAME <size_spec>

<sizespec> := <array_sizes> | <member_sizes> |
              <array_sizes> <member_sizes>

<array_sizes> := [ <expr> ] ... [ <expr> ]

<member_sizes> := { IDENT <sizespec> ; ... IDENT <sizespec> ; }
```

UC local variables are declared in the var section. These variables are used to store data extracted from arcs and may appear in computation sections. Their values are retained from one node firing to the next.

A <sizespec> is used to allocate arrays or arrays within structs. Recall that the size of an array is not included in a CODE 2 type definition. Arrays must be instantiated at a particular size before they can be used. A <sizespec> is one mechanism for doing that. See “CODE 2 Arrays”. Examples:

```
vars { int i; Mat2 m; }
```

Here we see a two dimensional array being allocated completely and another two dimensional array being only partly allocated. Partial allocation makes sense if the array’s rows will arrive on incoming arcs. See “CODE Arrays” for an example of this.

```
vars {
       Mat2 m1[n][m+1];
       Mat2 m2[n];
}
```

Suppose m is a two dimensional array that is a member of struct type mystruct. It may be allocated as follows.

```
vars {
       mystruct s { m[n][n]; };
}
```

• Initial Computation Stanza

```
<init_comp> :=
       init_comp { <stmt> ... <stmt> } (See “Statements”)
```

The initial computation is performed when the node is created at runtime-- before any firing rules are evaluated. Example:

```
init_comp {
       i = 0;
       foo(m);
}
```

- **Firing Rules Stanza**

```
<firing_rules> :=
       firing_rules { <frule> || ... <frule> }

<frule> := <fguard> , ... <fguard> => <fbind> ... <fbind>

<fguard> := <expr> | PORTNAME <indices> -> <lhs_expr> (See “Statements”)

<fbind> := <assign_stmt> (See “Statements”)
```

Firing rules serve two purposes. They specify conditions under which the node is allowed to fire, and they extract values from arcs and place them into local variables. In addition, they can set local variables. Firing rules consist of guards and bindings as shown.

```
Guard1, Guard2, ... GuardK => Bind1; Bind2; ... BindM
```

There need not be any bindings, but there must be at least one guard. The rule evaluates to TRUE and the node can fire when all of its guards evalaute to TRUE. You can read the commas as “and”.

There are two types of guards. Guards can be integer valued expressions. Such guards are FALSE when they have value 0 and TRUE otherwise. Guards can also be arc inputs. Every rule must have at least one arc input. The syntax is

```
PORT -> <lhs_exr> or PORT[<expr>]...[<expr>] -> <lhs_expr>
```

This type of guard is TRUE when there is a value on the arc named by the port. You may not assume that guards are evaluated in any specific order. Hence, the following rule is not reasonable since n need no be bound in time to be used in the replicator..

```
N -> n, { V[i] -> v[i] : (i n) } =>   // A rule that is not reasonable.
```

Bindings on the left of the => are assignment statements.

Entire rules as well as guards and bindings may be replicated.

If more than one rule is TRUE, one is chosen non-determinstically and its bindings are performed and the node fires.

Here are some example firing rule stanzas.

```
firing_rules {   // Fire when there is data on V.
```

```
        V -> n => // Store it in n.
}

firing_rules {               // Fire when there is data on V1 and V2.
        V1 -> n1, V2 -> n2 => // Store it in n1 and n2.
}

firing_rules {       // Fire when there is data on V1 or V2
        V1 -> n => || // Store it in n.
        V1 -> n =>
}

firing_rules {              // Fire when there is data on V1 or V2
        V1 -> n => s = 0; || // Store it in n and set s according to
        V1 -> n => s = 1;    // which rule fires.
}

firing_rules {           // Fire when there is data on V and s has
        s == 0, V -> n => // value 0. Store data in n.
}

firing_rules {                     // Fire when there is data on all
        { V[i] -> n[i] : (i N) } => // V[i] for i = 0 to N-1. Store
}                                  // it in n[i]. (Arc input is replicated.)

firing_rules {                  // Fire when there is data on any
        { V[i] -> n => : (i N) } // V[i] for i = 0 to N-1. Store
}                               // is in n. (Entire rule is replicated.)
```

• Computation Stanza

```
<comp> :=
        comp { <stmt> ... <stmt> } (See “Statements”)
```

The node’s computation fires afters a firing rule evaluates to TRUE, and its bindings are down. After the computation terminates, routing rules are evaluated. Example:

```
comp {
        i = 0;
        while (i < N) {
                foo(m);
                i = i + 1;
        }
}
```

• Routing Rules Stanza

```
<routing_rules> :=
        routing_rules { <rrule> && ... <rrule> }

<rrule> := <rguard> , ... <rguard> => <rbind> ... <rbind>

<rguard> := <expr>

<rbind> := <assign_stmt> | PORTNAME <indices> <- <expr> ;
```

Routing rules place data onto outgoing arcs. In addition, they can set local variables. Routing rules consist of guards and bindings as shown.

```
Guard1, Guard2, ... GuardK => Bind1; Bind2; ... BindM
```

There need not be any bindings, but there must be at least one guard. The rule evaluates to TRUE and its bindings will be done when all of its guards evalaute to TRUE. You can read the commas as “and”.

Guards must be integer valued expressions. They are FALSE when they have value 0 and are TRUE otherwise.

Bindings are assignment statements or arc outputs. The syntax of arc outputs is

```
PORT <- <expr> ; or PORT[<expr>]...[<expr>] <- <expr> ;
```

Bindings are performed from left to right. When an array is sent on an arc output, it is consumed and becomes null (See CODE Arrays). A null array has no storage associated with it. Hence, the following routing rule will send a null array M on Y.

```
TRUE => X <- M; Y <- M;
```

This problem can be avoided by using the copy function on the first (not the second!) arc output.

```
TRUE => X <- copy(M); Y <- M;
```

Entire rules as well as guards and bindings may be replicated.

Binding sections for all rules with TRUE guards are performed.

Here are some example routing rules stanzas.

```
routing_rules { // Place n+1 onto V.
       TRUE => V <- n+1;
}

routing_rules {             // Place n1 onto V1. and
       TRUE => V1 <- n1; && // n2 onto V2.
       TRUE => V2 <- n2;
}

routing_rules {              // Place n+1 onto V only if s > 0.
       s > 0 => V <- n+1; && // otherwise place n onto V.
       s <= 0 => V <- n;
}

routing_rules {                           // Place m[i] onto V[i]
       TRUE => { V[i] <- m[i]; : (i N) }; // for i = 0 to N-1.
}
```

2.19 Name Sharing Relation Specifications

Name Sharing Relation nodes allow variables to be shared among a set of UC nodes. These shared variables are declared in the node's specification text box using a simple text based language. The specification consists of three stanzas which must appear in the following order. All are optional, but all non-trivial Name Sharing Relation nodes will have at least the shared_var stanza.

<shared_vars> - Declares the name and type of all shared variables.

<vars> - Declares local variable which may be used in init_comp.

<init_comp> - A computation that is run when the node is created.

Each of the stanzas sytnax and purpose will be presented in turn.

- **Shared Variables Stanza**

```
<shared_vars> :=
        shared_vars { <vardecl> ; ... <vardecl> }

<vardecl> := (See <vardecl> under “UC Node Specification”.)
```

This stanza declares the name and type of variables to be shared. It also permits arrays and arrays within structs to be allocated at a particular size.

The variable name will appear on arc topology specifications that bind shared variables in the Name Sharing Relation to shared variable names in a UC. Here is an example:

```
shared_vars {
        int i;
        Vector v;
        Vector v[10];
        Matrix Mat[n][n]; // n is a creation parameter.
        Matrix Mat2[n]; // 2nd dimension unallocated.
        MyStruct s { m[n]; } ;
}
```

The shared_vars stanza in a Name Sharing Relation differs from the shared_vars stanza in a UC. In a Name Sharing Relation stanza, <sizespec>'s are allowed and use rules are not. The opposite is true in the UC.

- **Variable Declaration Stanza**

```
<vars> := (See the <vars> stanza under “UC Node Specification”.)
```

The <vars> stanza contains local variable declarations. These variables may be used in the initialization computation. The stanza is identical to the <vars> stanza in a UC node.

• Initial Computation Stanza

```
<init_comp> :=
       init_comp { <stmt> ; ... <stmt> ; }
```

The initialization computation is peformed when the Name Sharing relation is created. You may use it to give initial values to shared variables. UC nodes will not be granted access to the shared variables until initialization is complete.

2.20 Graphs and Graph Calling

Graphs are CODE's analogs to subroutines in conventional languages. They have parameters and they interact via call constructs, Call nodes in the case of CODE. The user creates a CODE program by drawing a set of graphs independently. Click the create cursor on the graph button to create a new graph. Call nodes within the graphs allow them to communication with one another.

Just as with conventional subroutines, there are formal and actual parameters to CODE graphs. Arcs that are incident upon a call node are actual parameters in a call to a graph. Formal parameters are defined within the called called graphs. Every interface node and creation parameter node defines a formal parameter. There must be a actual parameter (arc in the calling graph) bound to each formal parameter.

A separate instance of each graph is associated with every instance of a Call node. For example, suppose a user draws two graphs, main and foo. Suppose further that main contains two Call nodes, each calling foo. The two Call nodes refer to completely independent instantiations of foo. There is no way in CODE to refer to the same graph instance from two different Call node insta tiations.

Formal-actual parameter name binding involves the interaction of several annotations. Suppose an arc runs from a UC node with port X to a Call node with an input interface node named IN. X may be bound to IN by the following arc topology specification on the arc going from the UC to the Call.

```
.X => ..IN
```

The binding is still not complete. Suppose there is an arc running from the Input Interface node to a UC with a port Y in the called graph. This arc must be annotated with "Y", the name of the port the Interface node is to be bound to.

Topology specification on arcs incident upon Call node can be quite complex. Consider the right hand side of a topology specification of an arc that enters a Call node. The general form is as shown.

```
... => <call index list>.<port index list>.PORT<port index list>
```

Here are some examples.

```
        (ex. 1) ... => [i][j].[i+1].X[k]
        (ex. 2) ... => [i]..X[j]
        (ex. 3) ... => .[i].X[k]
```

The call indices (before the first dot) refer to the indices of the Call node the arc connects to. Recall that there is a separate Graph instance associated with every instance of a Call node. Hence, call indices are used to create a family of distinct graph instances. Consider example 2. If index variable i takes on values from 0 to 9, 10 separate graphs will be created.

Node indices refer to the index of a UC in the called graph. In example 3, if i has the value 5, and k the value 3, the arc is sending data to a UC with index 5 with a port with index 3. Port indices refer to the index of a port in a UC in the called graph. It is illegal to use node or port indices on arcs that are bound to creation parameters.

Arcs that leave Call nodes are similar to arcs that enter them, but this time the left hand side is of interest. The general form is as shown.

```
        <call var list>.<port var list>.PORT<port var list> => ...
```

Here are some examples.

```
        (ex. 1) [i][j].[k].X[l] => ...
        (ex. 2) [i]..X[j] => ...
        (ex. 3) .[i].X[j] => ...
```

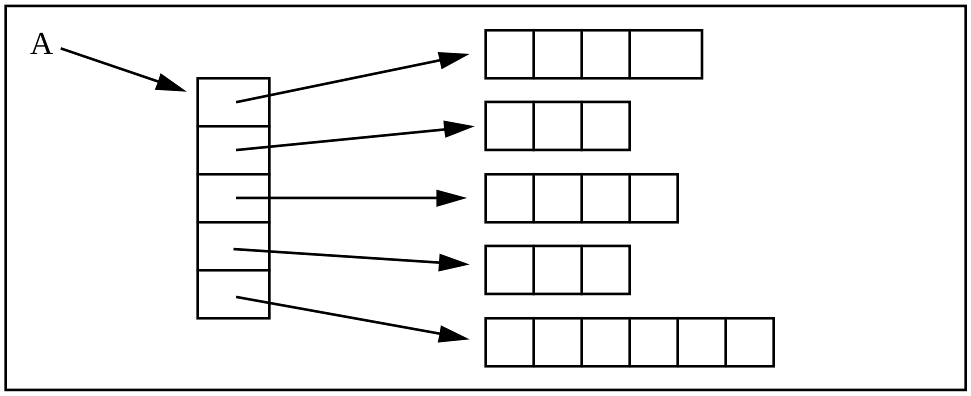
Just as with arcs going from UC to UC, only index variable are allowed on the left hand side, and the variables must be unique. They are place holders containing the indices of the entity that sent the data on the arc. The call index variables refer to the index of the Call node (and hence graph) the data come from. The node index variables refer to the index of the UC in the called the data come from, and the port index variables refer to the index of the port in that UC.

2.21 CODE Arrays

Arrays in CODE are somewhat unusual. They are not implemented using the customary contiguous memory scheme used by most programming languages. This implementation cannot be hidden from the CODE user since the user must be able to write sequential routines what deal with CODE arrays.

CODE arrays are implemented by means of pointers and vectors. A one dimensional real array is a pointer to a vector of reals. A two dimensional real array is a pointer to a vector of pointers to vectors of reals, and so on. There are several advantages to this scheme. It allows graphs and functions to be written that operate on arbitrarily sized arrays. Only the number of dimensions need be fixed. Also, it allows the efficient implementation of several important array partitioning operations on shared name space architectures. The performance and storage penalty is not great pr vided arrays are “skinny”. The size of the last dimension should be made the largest, whenever possible.

The following diagram show the layout of a two dimensional CODE array. The array has five rows, but the size of the rows (number of columns) varies. CODE allows this. Also, there is no implication that adjacent rows are in contiguous storage. Each row itself is, of course, in contiguous storage. Call this array A.



CODE provides a mechanism to help users keep track of the sizes of arrays. The asize function returns the length of an array. For example, asize(A) would return the value 4 since A is an array of 4 vectors. The value of asize(A[1]) is 3 since A[1] is a vector of length 3. CODE arrays are 0 based. There is no way to specify a non-zero lower bound. This matches the C language.

CODE users need not be concerned that pointers are used in implementing arrays. Arrays are used in a quite conventional way. For example, A[1][2] returns the value of the 3rd element in the 2nd row of A.

CODE array type definitions are unusual in that bounds on dimensions cannot be specified as part of the type. You define a “two dimensional” array, not an array type with fixed sizes like 5 x 10. All two dimensional arrays of the same base type are the same type.

Two and higher dimensional array types must be defined in the following way. There is no method for doing it directly.

```
type mat1 is array of real; // A 1-d array type.
type mat2 is array of mat1; // A 2-d array type.
type mat3 is array of mat2; // A 3-d array type.
```

Array types with more than 7 dimensions are not legal in code.

Since array sizes are not a part of the array type defintion, sizes must be allocated when an array is declared. One does this using size specifications. Here are some legal array variable declarations.

```
mat1 v[10]; // v is a vector of length 10
mat2 m[10][5]; // m is a matrix of size 10 x 5.
```

It is even possible to partially declare the size of an array.

```
mat2 M[10]; // m has 10 rows, but no storage for them.
```

This might be useful in a firing rule that will gather 10 vectors from an incoming arc into a 2 dimensional array.

```
firing_rules {
        PORT[i] -> m[i] =>
}
```

There is also an operator “new” that can assign storage. For example,

```
new mat2[10][3]
```

returns an array of size 10 x 3.

Function copy returns a copy of the array that is its argument.

Some comments on how to use CODE arrays from C might be useful. The situation is actually quite simple. An N dimensional CODE array array M of a given base type is declared in C as

```
basetype *...*M;
```

where the are N *s. For example a 3 dimensional int array is declared thus.

```
int ***M;
```

Elements of this type of array may be accessed from C just as they are in CODE.

```
h = M[i][j][k];
```

Of course, storage must be allocated to the array before it can be used.

As an example, here is a C function for summing the elements of a CODE 3 dimensional integer array. It could be called from CODE thus.

```
sum = SumArray(M, asize(M), asize(M[0]), asize(M[0][0]));
```

Assuming, of course that array is regularly sized.

```
int SumArray(M, i_size, j_size, k_size)
        int ***M;
        int i_size;
        int j_size;
        int k_size;
{
        int i, j, k;
        int *t1, **t2;
        int sum=0;

        for (i = 0; i < i_size; i++) {
                t1 = M[i];
                for (j = 0; j < j_size; j++) {
                        t2 = t1[j];
                        for (k = 0; k < k_size; k++) sum += t2[k];
                }
        }

        return sum;
}
```

The use of temporary variable t1 and t2 can substantially decrease the execution time of such functions.