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 one 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. Enter function body definitions. 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 it in. 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. 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 and Firing and Routing Rules

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. See also the User Manual chapters on Call nodes and Name Sharing Relations.

7.1 Arcs that Bind one UC Node to Another

Arcs that run from one 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. Figure 7.1 shows a complete example of a program that sends a two dimensional array from UC node S to UC node R.

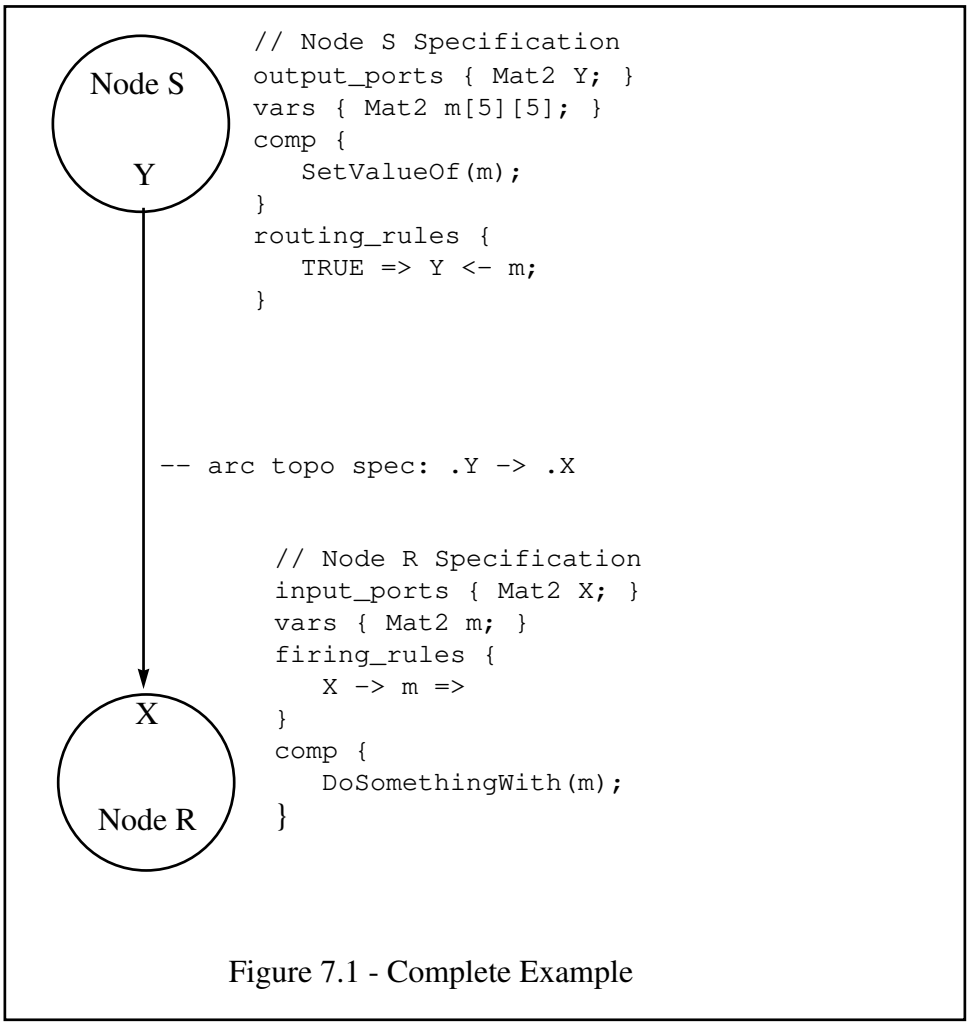


Figure 7.1 - Complete Example

Notice that node S has an output port called Y, and node R has an input port X. An arc is drawn

from S to R. The user 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. They are both of type Mat2 in this case.

When S fires it places a two dimensional array onto its port Y. The array is placed into a FIFO queue to be extracted later when Node R fires, according to its firing rule that states that it can execute when there is data on the arc to associates with its input port X.

So, to send data from one node to another you must do the following.

1. Define an output port on the sending UC node.
2. Place data onto that port in a routing rule.
3. Define an input port in the receiving UC node.
4. Extract data from that port in a firing rule.
5. Use an arc topology specification to bind the output port to the input port.

7.2 Specification of Arcs between Replicated UC Nodes, an Example

There is another role that arc topology specifications play. They permit multiple instantiations of UC nodes to be created. To understand this you may want to think of arcs as being rules that describe where data will go to, given where it comes from. For example, the rule above (.Y -> .X) could be read as follows.

> If data come from the sending node's port Y,
> they must be sent to the receiving node's port X.

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 7.2 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.

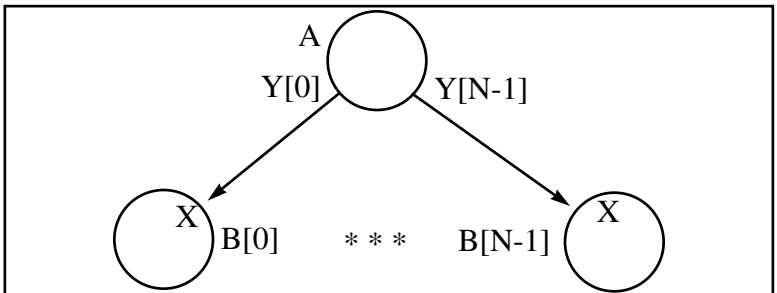


Figure 7.2 - Example Communication Structure

This structure can be created with the CODE program shown in figure 7.3. Consider UC node A's routing rule. The syntax { <thing> : (ind N)} replicates <thing> for ind taking on values from 0 to N-1. Hence, node A sends data out on ports Y[ind] for ind with values 0 to N-1. The index (ind) must be a legal CODE identifier. It need not be declared in the node's vars stanza.

It is vital to understand that the index on port Y has absolutely nothing do to with the array being sent. An entire 2 dimensional array is being placed onto each of the ports Y[0], ..., Y[N-1]. The type of the data sent on a port is determined completely by the port's type, not the ports indices. Y is of type Mat2. A port index defines which port data will be sent on.

In fact, it is necessary to send a copy of the array on each port since sending an array on a port consumes the array. If the routing rule stated "Y[ind] <- m;" instead of "Y[ind] <- copy(m);", the array m would be sent on Y[0] and empty arrays (with no storage allocated for them) would be sent on Y[1], ..., Y[N-1].

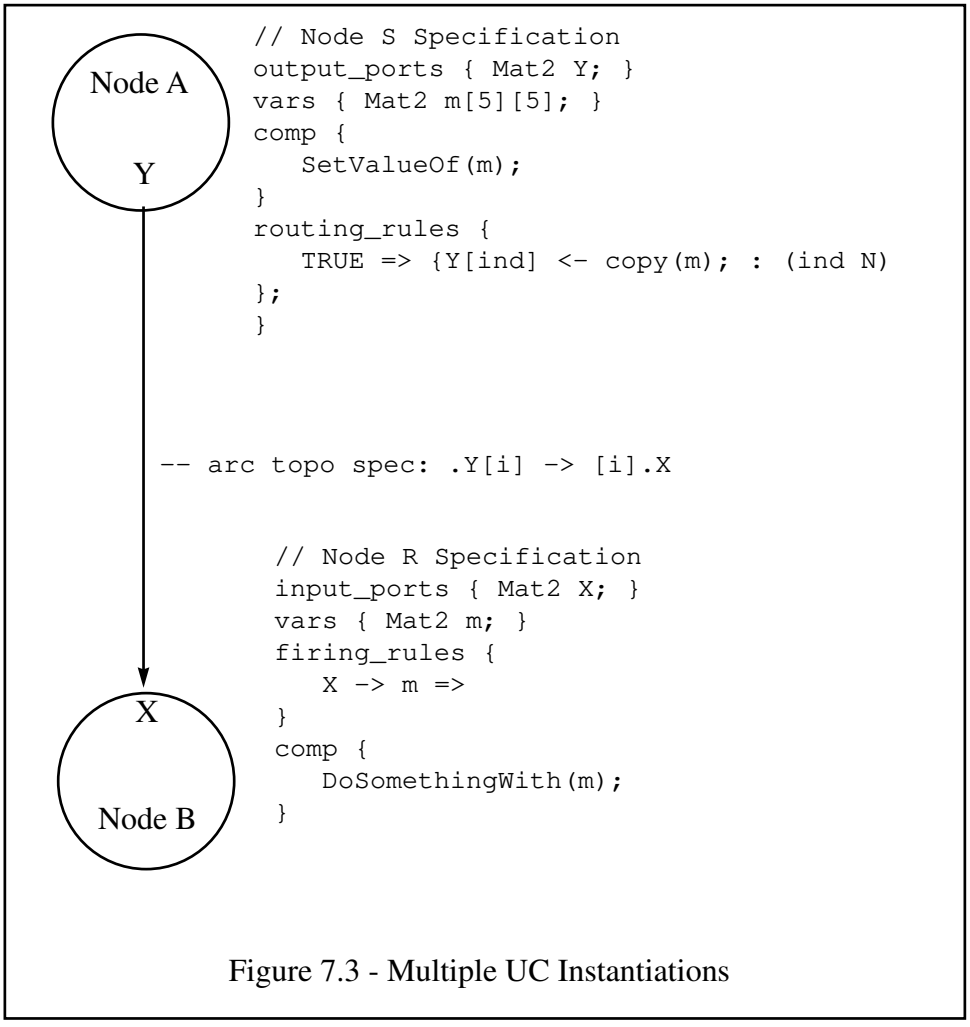


Figure 7.3 - Multiple UC Instantiations

Now, let's consider the topology specification of the arc from node A to node B. It is shown below.

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

Again, read the specification as a rule that describes where data go to given where they come from.

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

Node A places data on its output ports Y[0], ..., Y[N-1]. The arc topology specification causes data placed on them to be sent to port X on nodes B[0], ..., B[N-1].

Hence, the value of "ind" in A's routing rule determines the index of the receiving node. The index "i" on the left on the topology specification is a free variable. It's value is determined by the index of the port on which the sending node places data. If A placed data onto Y as shown.

```
routing_rule {               // a strange but legal routing rule for node A.
   TRUE => Y[g+h/2] <- m;    // Assume g and h are local variables in A.
}                            // They could also be creation parameters.
```

Then "i" would have the value g+h/2.

Indices on the left of the "=>" in an arc topology specification must be a simple uniquely named identifiers. They are holders for values of port indices from the sending node. The indices on the right of the "=>" can be expressions. Here are some examples of legal and illegal arc topology specifications.

```
.Y[i+1] => [i].X       // Illegal: no expressions allowed on the left.
.Y[i][i] => [i].X      // Illegal: "i" appears twice on the left of the =>.
.Y[i][j] => [i][j].X   // Legal, but sending node should use two port indices.
.Y[i] => [i+1].X       // Legal.
```

Note that you do not need to declare any bound on the number of receiving nodes. They are created as needed. If node A from figure 7.3 puts a value onto port Y[573] then node B[573] will be automatically created.

Also notice that port X on node B has no index. Each instance of B has its own port X so no port index is required in its firing rule. Only a single arc goes to each instance of node B.

7.3 General Specification of Arcs between Replicated UC Nodes

Leaving our example, the general form of an arc topology specification for an arc that runs from a UC node to a UC node 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. There may never be more than 7 indices in single index list. This is an arbitrary limitation imposed by CODE's implementation.

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.

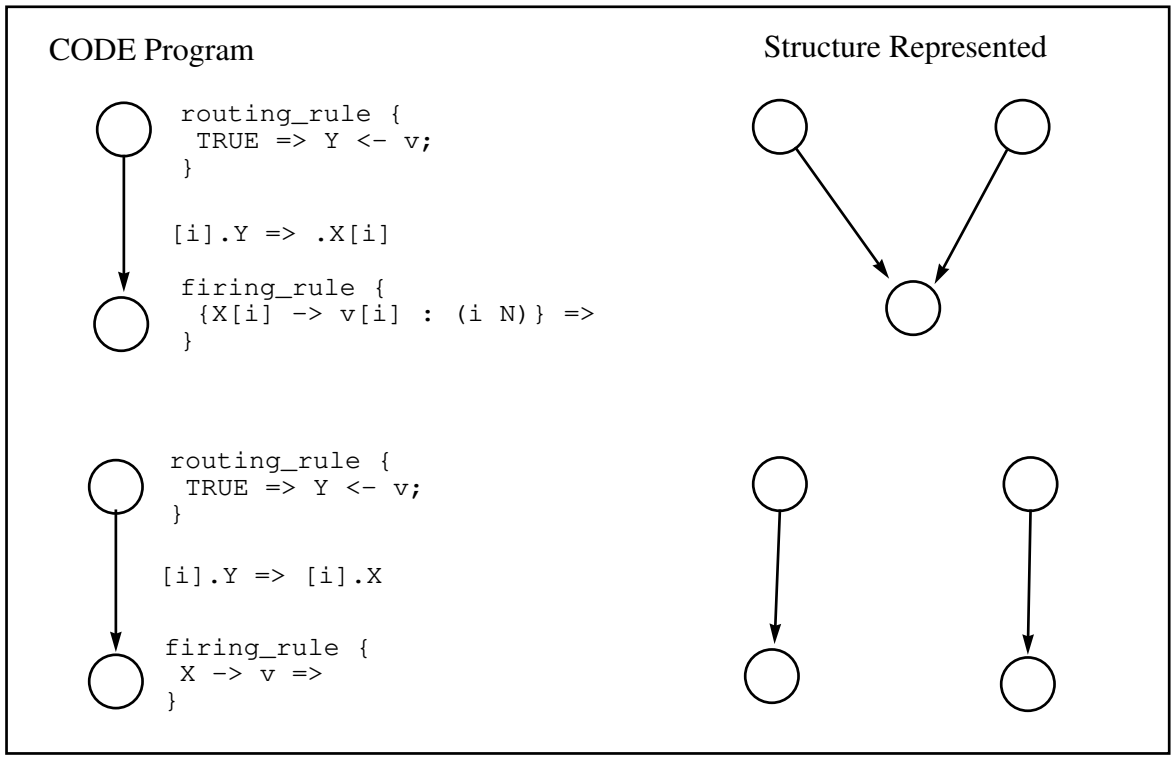
Consider an example specification.

```
[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 sending node with indices [i][j] and port Y with index [k], they are sent to the receiving node with index [i+k] and port X with index [j-1].

The following figures show some other common cases. Only, the the most relevant parts of the UC node specifications are shown.



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.

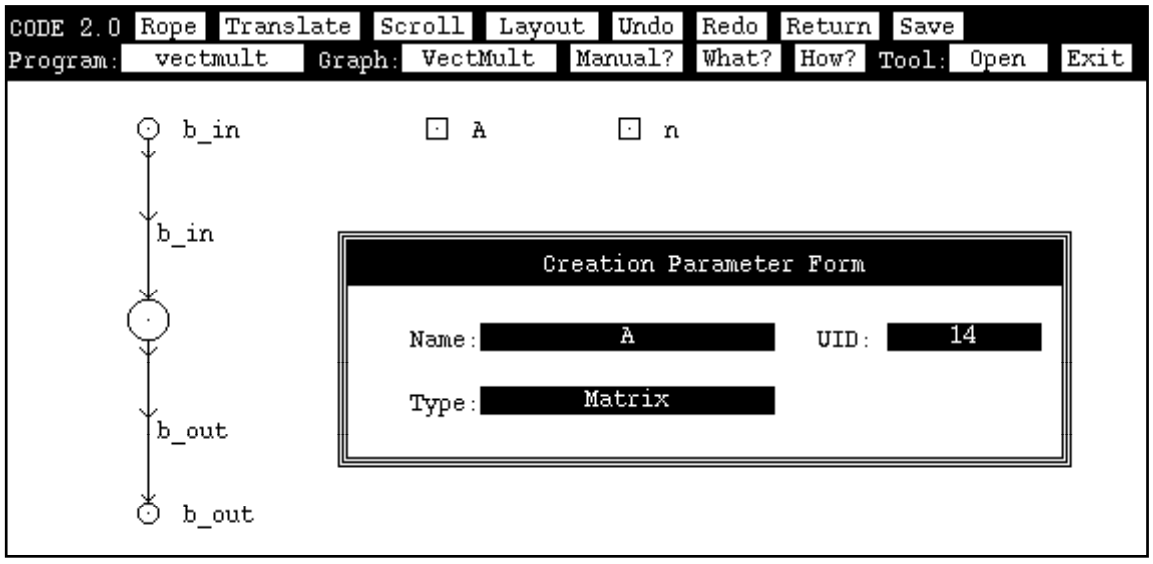


Figure 8.1 - VectMult Graph

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 Interface 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" (which is not hown in th figure), 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. A Creation Parameter is a kind of shared variable that is given a value when the graph that contains it is created. From then on, it is a read-only variable.

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.

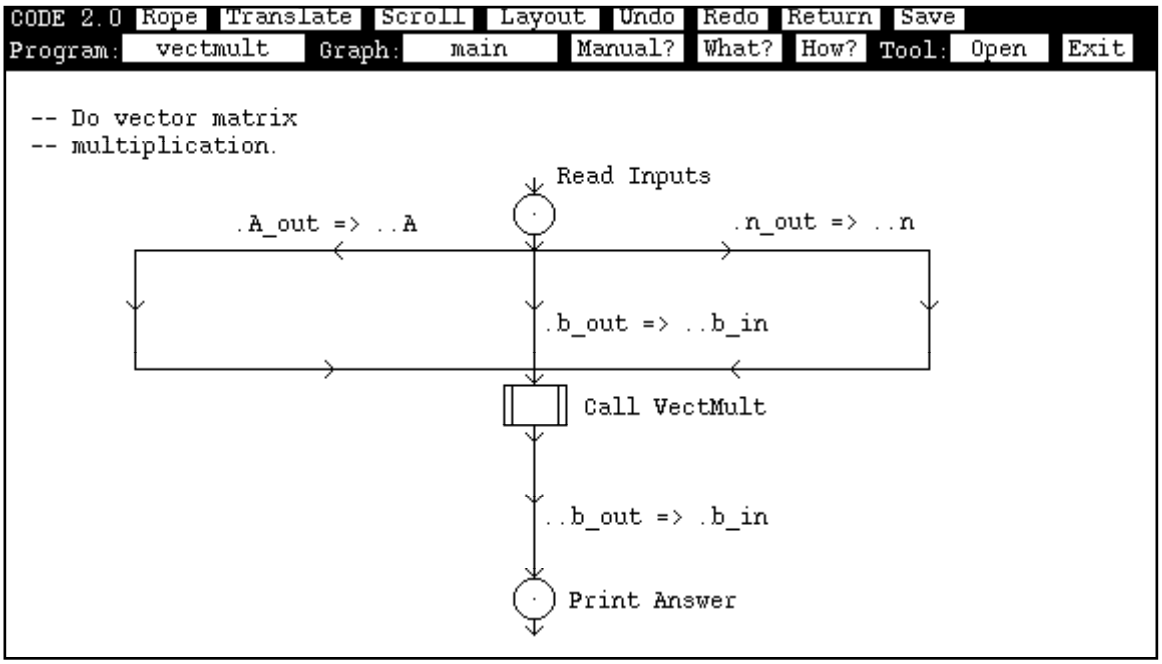


Figure 8.2 - Main Graph

All arc topology specifications are shown as comments written on the graph. They are not automatically displayed. 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.

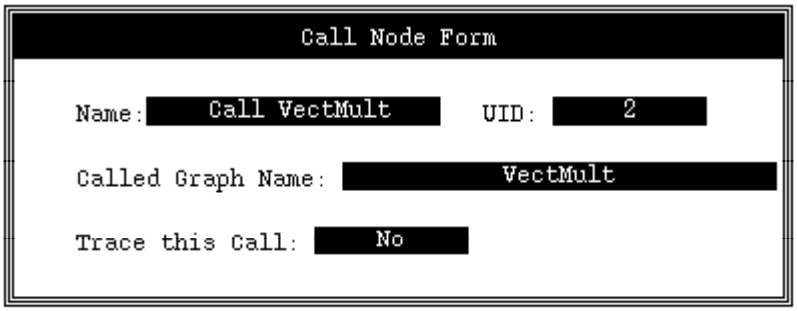


Figure 8.3 - Call Node Attribute Form

This is also true for graphs called from different instances of the same Call node. Note that 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 expression 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.

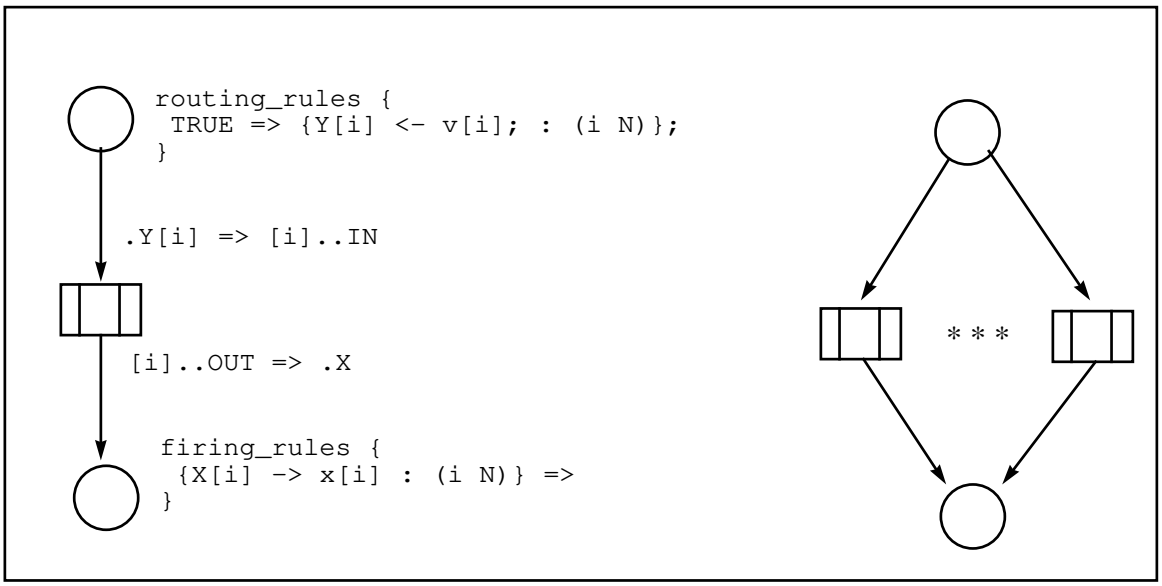


Figure 8.4 - Example Replicated Call

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.

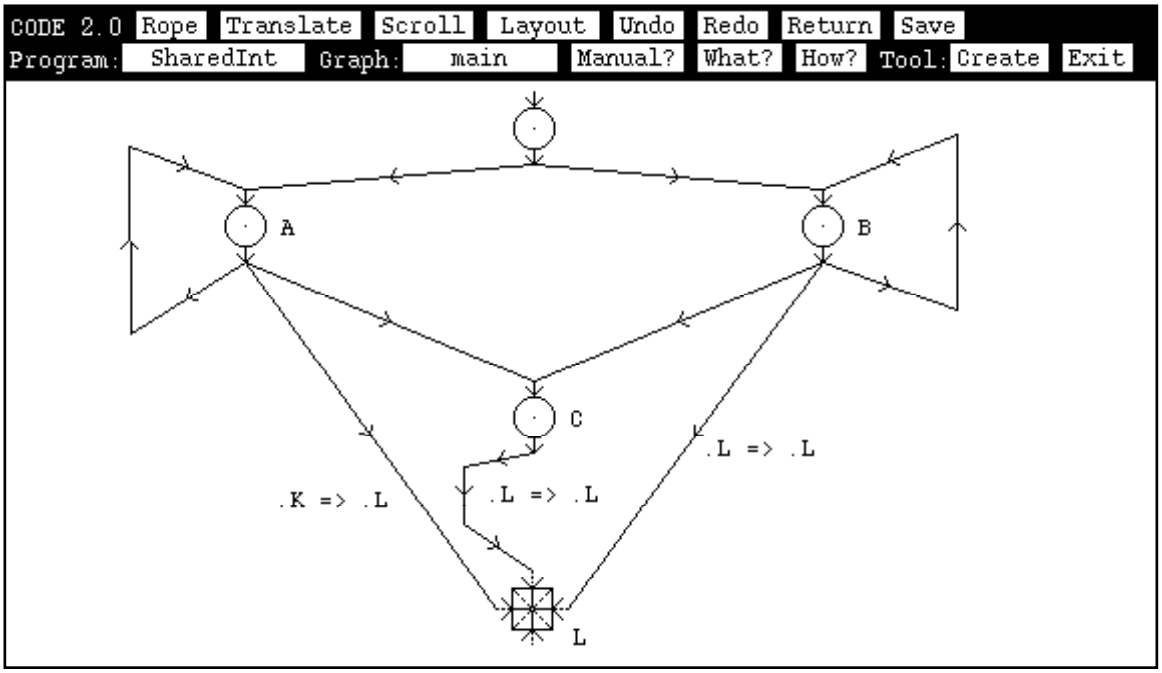


Figure 9.1 - Name Sharing Relation Example

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 for other cases involving Name Sharing Relation nodes.

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

One always draws arcs to 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 to be a reader.

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 CODE Arrays, Structures and Storage Allocation

Since CODE users must write sequential routines that operate on CODE arrays and structures, it is necessary to describe their implementation and allocation. Sequential routines must be written to operate on the types described below.

10.1 CODE Arrays: Representation

CODE multidimensional arrays do not necessarily consist of contiguous storage. Instead they are implemented as an array of pointers to objects of the array's base type. For example, consider the CODE types shown below.

```
type Mat1 is array of real;
type Mat2 is array of Mat1;
```

To declare a variable of type Mat2 in C, one would use the following.

```
double **m;
```

A CODE 3-dimensional array would be a C "double ***", and so on. Other array types such as integer arrays are similar.

10.2 Storage Allocation

Notice that the type Mat2 is actually a pointer to an array. No array storage is allocated by the C variable declaration. It is assumed that storage has been allocated on the heap. In fact, all CODE arrays and structures are allocated on the heap. For example, the following CODE variable declaration and CODE "new" expression both allocate the storage shown in figure 10.1 (a).

```
vars { Mat2 m[4][3]; }      // One way to allocate array storage in CODE.

m = new Mat2[4][3];   // Another way. (An expression that may be used wherever
                      // an expression is legal.
```

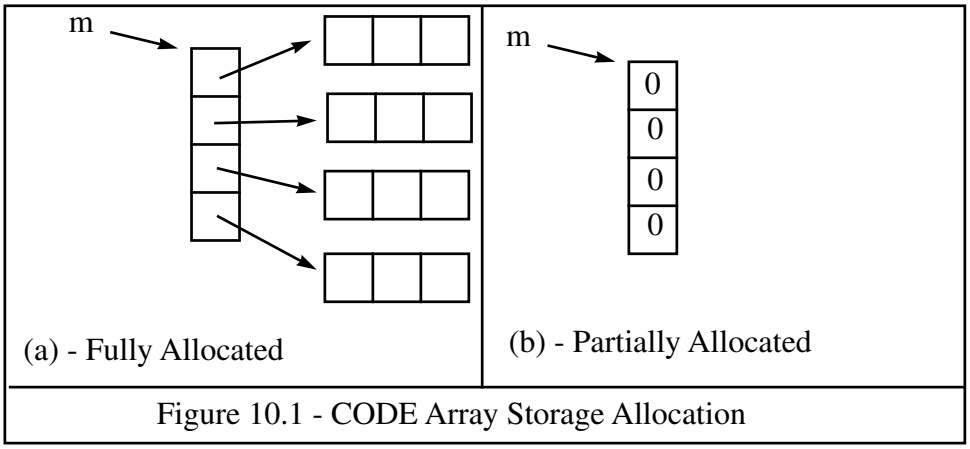


Figure 10.1 - CODE Array Storage Allocation

It is also possible to "partially" allocate a CODE array as depicted in figure 10.1 (b). A variable

declaration and “new” expression that do this are shown below.

```
vars { Mat2 m[4]; }

m = new Mat2[4];
```

This can be useful in a firing rule that will gather vectors, for example, on incoming arcs to form an array. Notice that the type of the expression m[i] is Mat1.

```
input_ports { Mat1 V; }   // Note the port type is Mat1, not Mat2.
vars { Mat2 m[4]; }       // Could use a creation parameter for the size.
firing_rules {
   { V[i] -> m[i] : (i 4) } =>    // Gather vectors into an array.
}
```

Remember that sending an array or structure out on a port, consumes the array or structure. This storage is lost to the sending node; it has been “sent” to another node. Hence, if a node allocates an array by means of a variable declaration and sends it out on an arc and the node is to fire a second time, storage for the array must be reallocated by means of a “new” expression, or storage must be received through a port from another node. Do not forget this. It is a common source of bugs in CODE programs.

Also, do not attempt to allocated storage for CODE arrays or structures by yourself from within your seqeuntial routines. It will not work.

10.3 Some Useful Predeclared Functions

There are some predeclared functions that help in managing CODE arrays and structures. The term “deep” means that the function applies to the entire variable. For example, if you free a 2-dimensional array, the 1-dimensional arrays that make it up are also freed.

```
copy(v) -- Returns a (deep) copy of an array or structure.
free(v) -- Frees (again deep) storage associated with an array or structure.
size(a) -- Returns the number of elements (not bytes!) in an array.
```

As an example, “size(m[2])” returns the number elements in row 2 of m, assuming m has type Mat2.

10.4 Structure Allocation

Structures in CODE, like arrays are allocated on the heap. If you declare a structure type, the equivalent C type is a pointer type.

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

typedef struct {          // C type
   double x;
   double y;
```

```
} *Point;
```

If a structure variable is declared in CODE, storage for it is automatically allocated. However, if that variable is sent out of the node on a port, that storage is lost to the sending node, just as for arrays. A "new" expression may be used to reallocate storage for the structure.

10.5 File c2_globtypes.h

CODE creates a file called "c2_globtypes.h" whenever a program is translated. This file contains C typedefs for all CODE types declared in program-level scope (in the Program attribute form). Examining this file may help you to understand the implementation of CODE types. You may also "#include" it in your externally defined sequentially routines. These typedefs are automatically made available to function bodies entered into the Program and other attribute forms.

11.0 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.

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 primarily a debugging tool used by the CODE system's developers, but it may be useful to general users as well.

11.1 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 remote mounted using NFS. The reason has to do with an interaction between 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.

12.0 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.

12.1 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.

13.0 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 definitions.

```
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. A variable declaration allocates storage for a variable only once, when the node is created. Here are some examples of possibly incorrect UC nodes.

Example 1:

```
// If this node fires twice, an array with *no* storage allocated to it will be
// sent out on Y on the second firing. Use a "m = new Mat2[5][6];" in the comp
// stanza if the node will fire more than once. In that case, there is no need to
// allocate storage in the variable declaration.

input_ports { int X; }
output_ports { Mat2 Y; }
vars { Mat2 m[5][6]; int v; }
firing_rules {
   X -> v =>
}
comp {
   GiveValueTo(m);
}
routing_rules {
   TRUE => Y <- m;
}
```

Example 2:

```
// The array m is consumed when sent out on port Y[0]. Ports Y[1], ... Y[N-1]
// get arrays with no storage allocated to them. Use "Y[i] <- copy(m)" to send
// copies of an array (or struct) to multiple destinations.

output_ports { Mat2 Y; }
vars { Mat2 m[5][6]; int v; }
firing_rules {
   X -> v =>
}
comp {
   GiveValueTo(m);
}
routing_rules {
   TRUE => Y <- m;
}
```

Do not assume that rules or guards within rules are evaluated 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 definitions.

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;
{
  • • •
}
```

Do not create aliases using you sequential routines.

It is illegal and very dangerous to create aliases (multiple names for the same storage). For example, the following function gets an alias for an array type called Mat2.

```
Mat2 GetAlias(m)
       Mat2 m;
{
       return m;
}
```

You could call it from a comp stanza and create an alias.

```
comp {
       m2 = GetAlias(m1);
}
```

This is very dangerous since CODE frees storage on its own. Your aliases could end up pointing to freed storage!

Do not call shmalloc to get storage.

CODE has allocation primitives built-in. It will not work to mix calls to shmalloc (or malloc) with CODE's storage allocation. On some targets, it is permissible to get storage from the heap in a sequential routines, but do not pass any reference to this storage out of the sequential routine.

Remember that CODE arrays need not be implemented using contiguous storage.

The equivalent type in C to a CODE two dimensional array of real is "double **", a three dimensional real array is equivalent to "double ***." Keep this is mind when you write your sequential functions. See the chapter on arrays in the Reference Manual.

14.0 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 or character literals.

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