CODE 1.2
USER MANUAL AND TUTORIALS

John Werth, Dwip Banerjee, J. C. Browne, Ravi Jain,
Steve Lin, Peter Newton, Ravi Rao, and Steve Sobek

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-90-35 November 1990

ABSTRACT

The CODE 1.2 Manual describes the use of the parallel programming environment CODE. This environment is a hierarchical, graphical, architecture independent system for the design and development of parallel programs.

Keywords: Software development environments, parallel programming, graphical programming.

Table of Contents

I. Introduction

II. SUC Nodes

III. Data Dependencies

IV. Switch Nodes

V. Exclusion Dependencies

VI. Subgraphs

VII. Replicated Dependencies and Nodes

VIII. Executing the CODE Program

IX. Miscellaneous

Appendix A Code Publications

Appendix B. Code Window and Menu

Appendix C. Command Menu

Appendix D. Exclusion Constraints

Appendix E. Developing CODE Programs for the IBM 3090

Tutorial - Example 1: SUCs, Data Dependencies, and Exclusion Dependencies

Tutorial - Example 2: Subgraphs

Tutorial Example 3: Replicated Structures

How to Use this Manual

1. Read the Introduction, Section I
2. Do the Tutorial Examples 1 and 2
3. Read the remainder of the manual

CODE 1.2

I. Introduction

A. Outline and purpose

The purpose of this document is to outline the model of computation behind the CODE system, and to describe its use. In this introduction we give a quick and simplified survey of the model and an easy example. In later sections we expand on the model and give detailed examples of its use.

Roughly, one can picture the CODE 1.2 model of computation as a dataflow graph which allows more flexible execution rules for some node types and which allows data sharing among the nodes. However, there are many other features of both the model and the implementation. These are further discussed in this introduction.

The implementation is distinguished by the software engineering features which have been incorporated to facilitate its practical use. These features encompass the user interface, provisions for reuse of program fragments, and facilities for structuring programs. The system has been through several versions and has had substantial use by graduate and undergraduate students in classes.

A bibliography of CODE related documents appears in Appendix A.

B. A simplified discussion

CODE is a program development system for parallel programs. In CODE, programs are organized as graphs with three possible types of nodes and two possible types of arcs. The nodes are associated with computations, and the arcs are associated with data.

1. **Directed arcs** (denoted by arrows, also called data dependencies) indicate data being generated by the source node, and then flowing to the sink node.
2. **Hyperarcs** (undirected arcs potentially joining more than two nodes, denoted by dotted lines and also called exclusion dependencies) indicate data which is shared by the computations represented by the nodes joined by the hyperarc. The hyperarcs have associated constraints that control access to the data; this is the basic mechanism for preventing race conditions.
3. **Schedulable Units of Computation (SUC)** nodes (denoted by circles) are associated with some computation. They are distinguished by only being able to execute when data is present on all incoming directed arcs. They place data on all of their outgoing directed arcs at the end of their execution.
4. **Switch nodes** (denoted by diamonds) perform specialized computations associated with making choices as well as merging and distributing data. They are enabled for execution if data is present on any input arc. These nodes may also place data on any subset of their outgoing directed arcs after execution.
5. **Subgraph nodes** (denoted by boxes) encapsulate computations performed by entire graphs.

Program development with CODE requires first specifying the graph (which represents the overall organization of the computation) and then providing details about each graph element:

For SUC nodes,
the user supplies a computation in the form of a subprogram which may come from a library or be written from scratch.

For switch nodes, the user supplies a condition on each input arc which describes the conditions under which data on that arc are to be passed through the node and on to the destination node(s) to which it is routed.

For directed arcs, the user supplies a data name and data type.

For hyperarcs, the user supplies a data name, a data type and a data sharing constraint to be preserved by the system among the nodes sharing the data . (We will see later that this condition is actually specified by annotating the nodes)

Once program development is complete, the user is able to request translation of the program to any of several executable forms. Each executable form is targeted to the specific hardware and software environment in which the program will be executed. By analogy with the compilation process, the system specific portion of CODE which creates these executables is called a backend. The current version of the software supports backends for Ada and Fortran on a variety of architectures.

C. Two easy examples

These examples are meant to give a quick feel for the CODE model; for the experienced programmer of parallel systems, they may raise as many questions as they answer. Beginning in Section II, we answer these questions in detail. For now we employ a mix of real and idealized notation to illustrate all of the major elements of CODE. These two examples are used as Tutorial Examples 1 and 2 presented later in this document.

i. Example A. (Illustrates SUC nodes, directed arcs and hyperarcs)

Consider a program to add a vector of numbers V = (v_i: i =1, N). The computation proceeds by splitting the vector in half, then forming the partial sums of the halves, and finally adding the partial sums.

There are four SUC nodes:

Init
- Initializes the vector V
- Initializes the variable Sum to 0
- Divides V into V1 = (v_i: i =1, N/2) and V2 = (v_i: i =(N/2 + 1), N).

Add1
- Adds the elements of V1 to produce Sum1
- Updates Sum to Sum + Sum1

Add2
- Adds the elements of V2 to produce Sum2
- Updates Sum to Sum + Sum2

PrintSum
- Prints the value of Sum1
- Prints the value of Sum2
- Prints the value of Sum

There are four directed arcs:

(Init, Add1)	data item is V1
(Init, Add2)	data item is V2
(Add1, PrintSum)	data item is Sum1
(Add2, PrintSum)	data item is Sum2

There is one hyperarc

{Init, Add1, Add2, PrintSum} - data item is Sum and exclusion condition is Init mutex Add1mutex Add2 mutex PrintSum.

That is, at most one of Init, Add1, Add2, and PrintSum may simultaneously access Sum. We specify the nodes of a hyperarc using set notation since no ordering of nodes is implied by the hyperarc.

Figure 1 is a drawing (rather than a screen dump) of the CODE graph specifying this computation. This drawing includes detail about the nodes and the exclusion constraint that is suppressed in a real drawing from CODE. Figure 2 is a screen dump of the actual graph from the CODE system. In CODE, the information about the nodes, arcs and constraints is entered using a mix of graphics and menus.

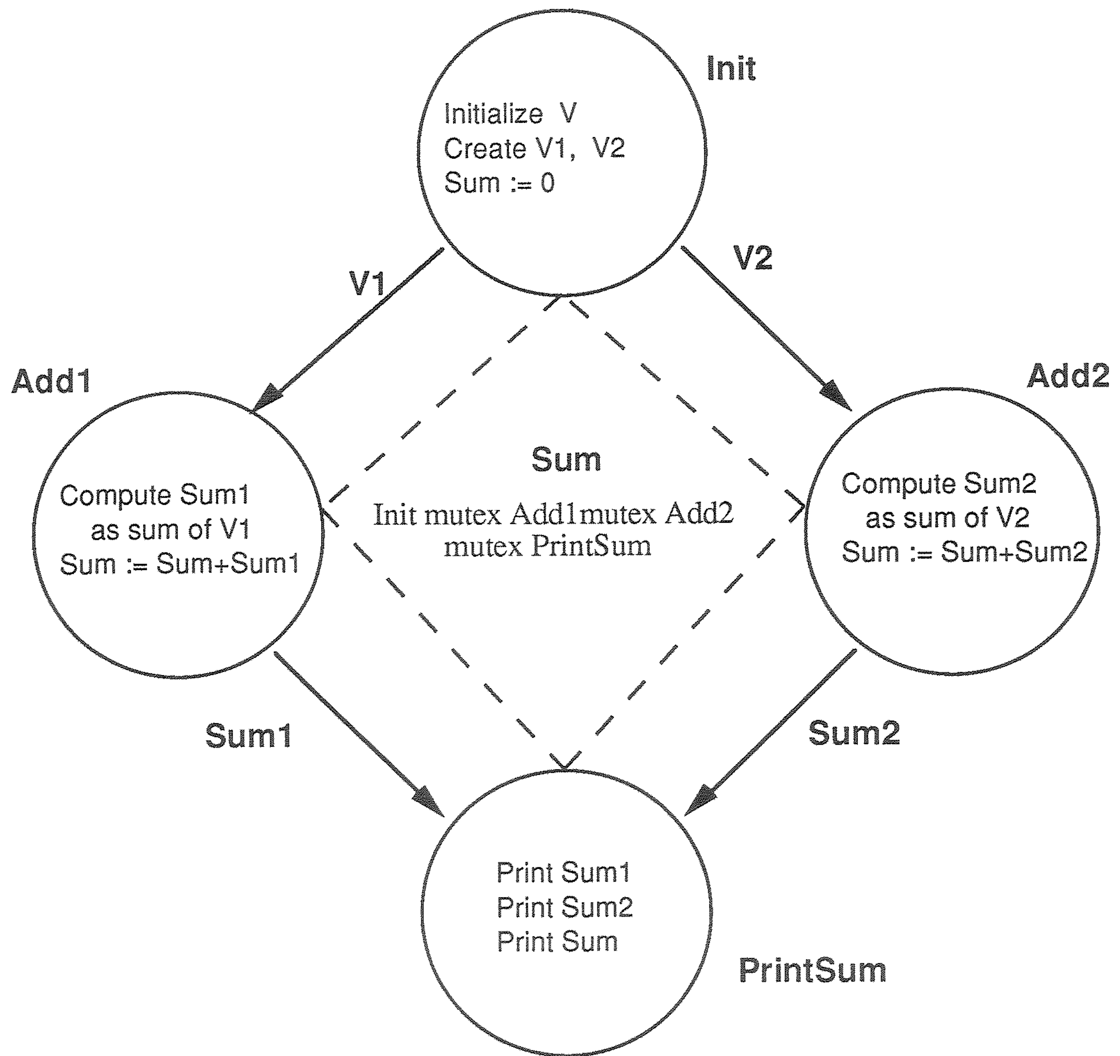


Figure 1
CODE Graph for Example A

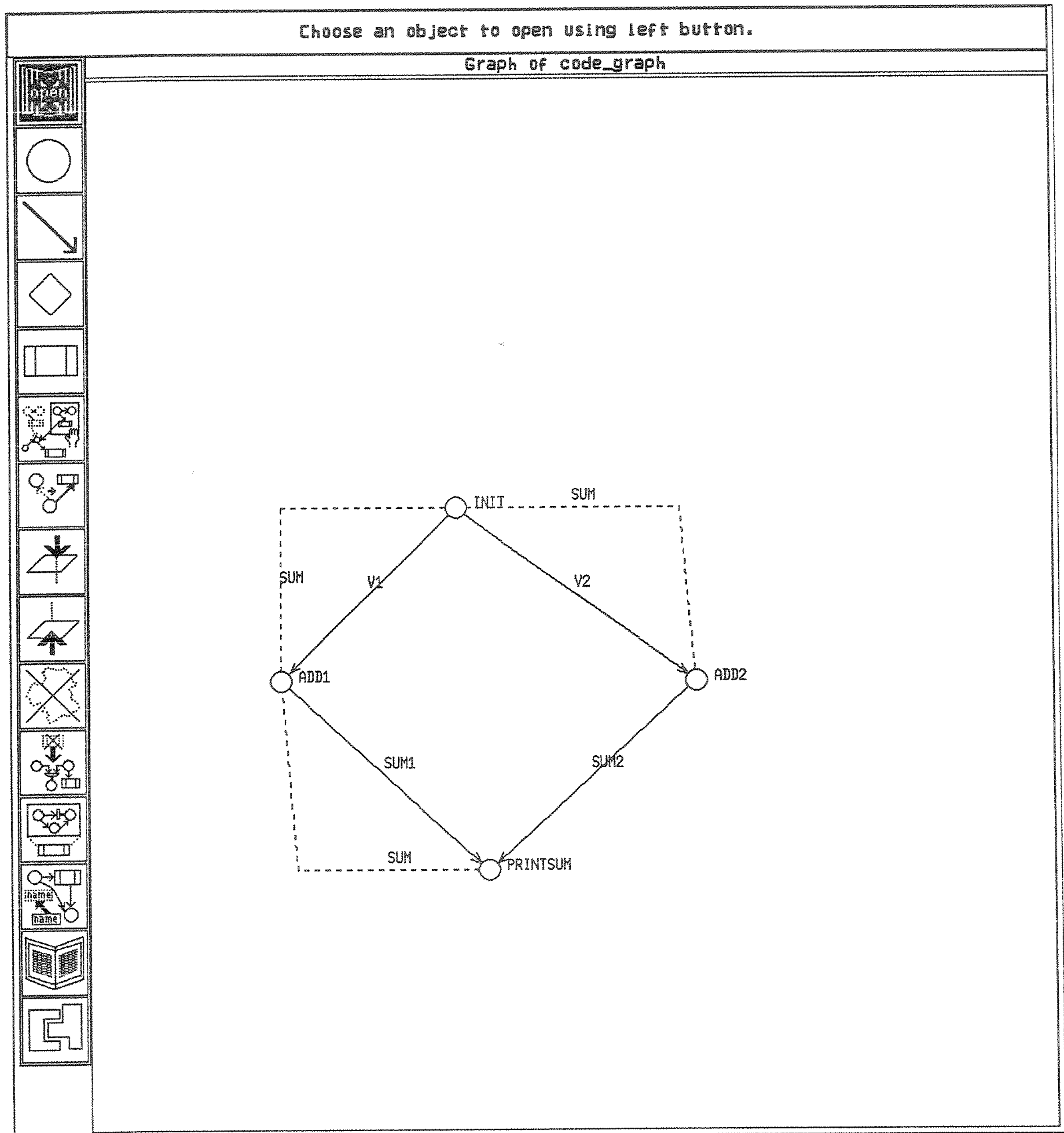


Figure 2
Actual CODE System Window for Example A

ii. Example B. (Illustrates switch nodes and subgraphs) The previous example illustrated the dataflow and data sharing possibilities of CODE; this example illustrates switch nodes (which are used for making choices) and the subgraph node (which is used for hierarchical structuring).

Consider a program to read a number N and then read N vectors, forming the sum of the elements of each vector and printing the sum of each vector. After finding the last sum, a closing message is printed.

There are four nodes in the computation graph:

Read_N SUC
Reads N, the number of vectors to process

Wrap_Up SUC
Prints a closing message

Sum_Vector Subgraph
Reads a vector, adds its elements, prints its sum, and decrements the loop control variable. This subgraph is the obvious modification of Example A.

Loop Switch
Controls the execution of the loop
The switch condition specifies two tests.

Test 1: If N (the initial value read) <= 0 then
Transfer to Wrap_Up
Else
Pass N to Sum_Vector
Execute Sum_Vector

Test 2: If at any point Loop_Control_Out <= 0 then
Set Final_N to Loop_Control_Out
Transfer to Wrap_Up
Else
Pass Loop_Control_Out to Sum_Vector
Execute Sum_Vector

There are four directed arcs

(Read_N, Loop) data item is N
(Loop, Sum_Vector) data item is Loop_Control_In, the loop control value
(Sum_Vector, Loop) data item is Loop_Control_Out, the new loop control value
(Loop, Wrap_Up) data item is Final_N

Figure 3 is a drawing (rather than a screen dump) of the CODE graph specifying this computation. This drawing includes a good bit of detail about the nodes and the subgraph that is not explicitly displayed in a real drawing from CODE (note the substitution of LCO and LCI for Loop_Control_Out and Loop_Control_In respectively). Figure 4 is a screen dump of the actual graph from the CODE system.

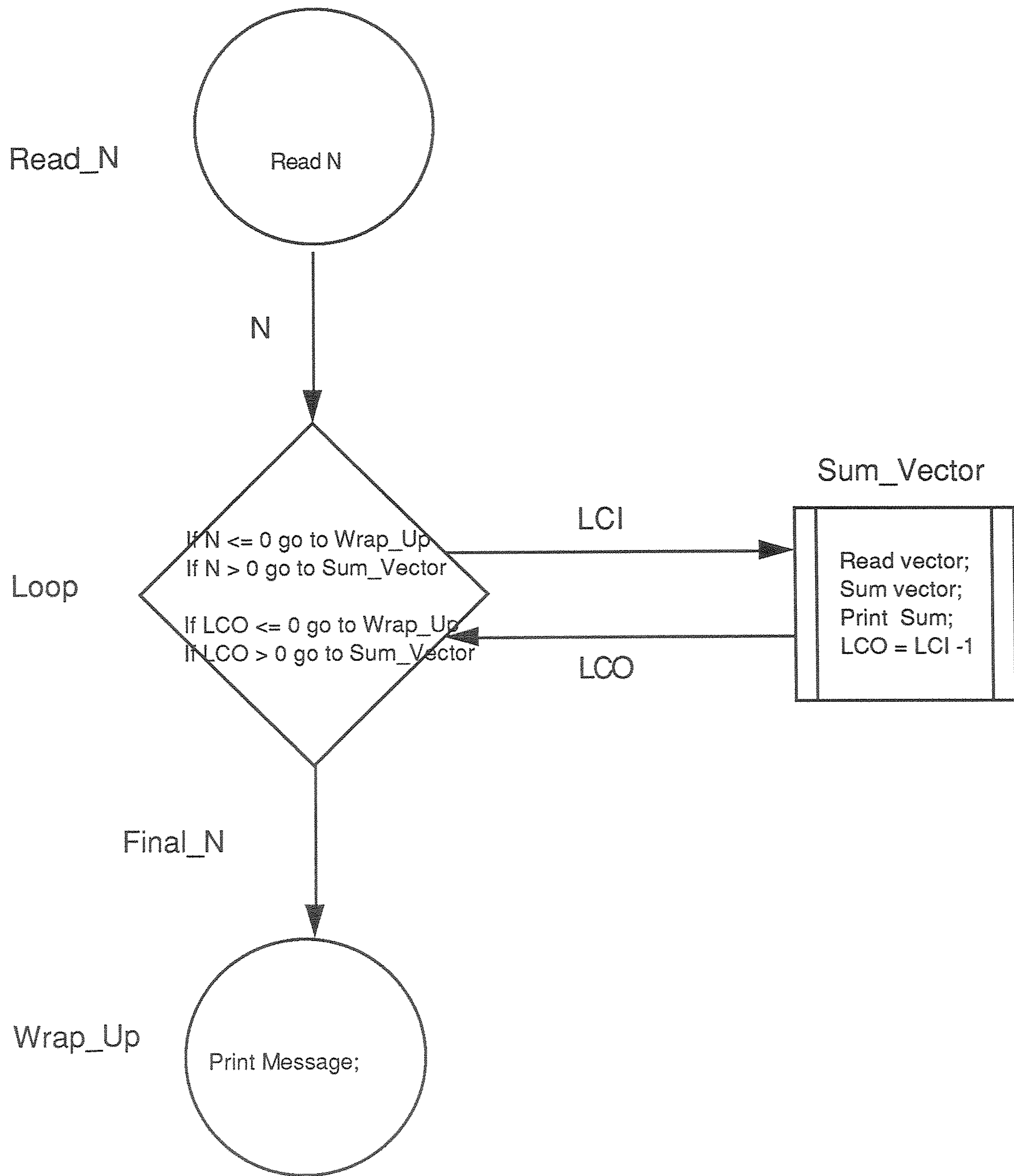


Figure 3
CODE Graph for Example B

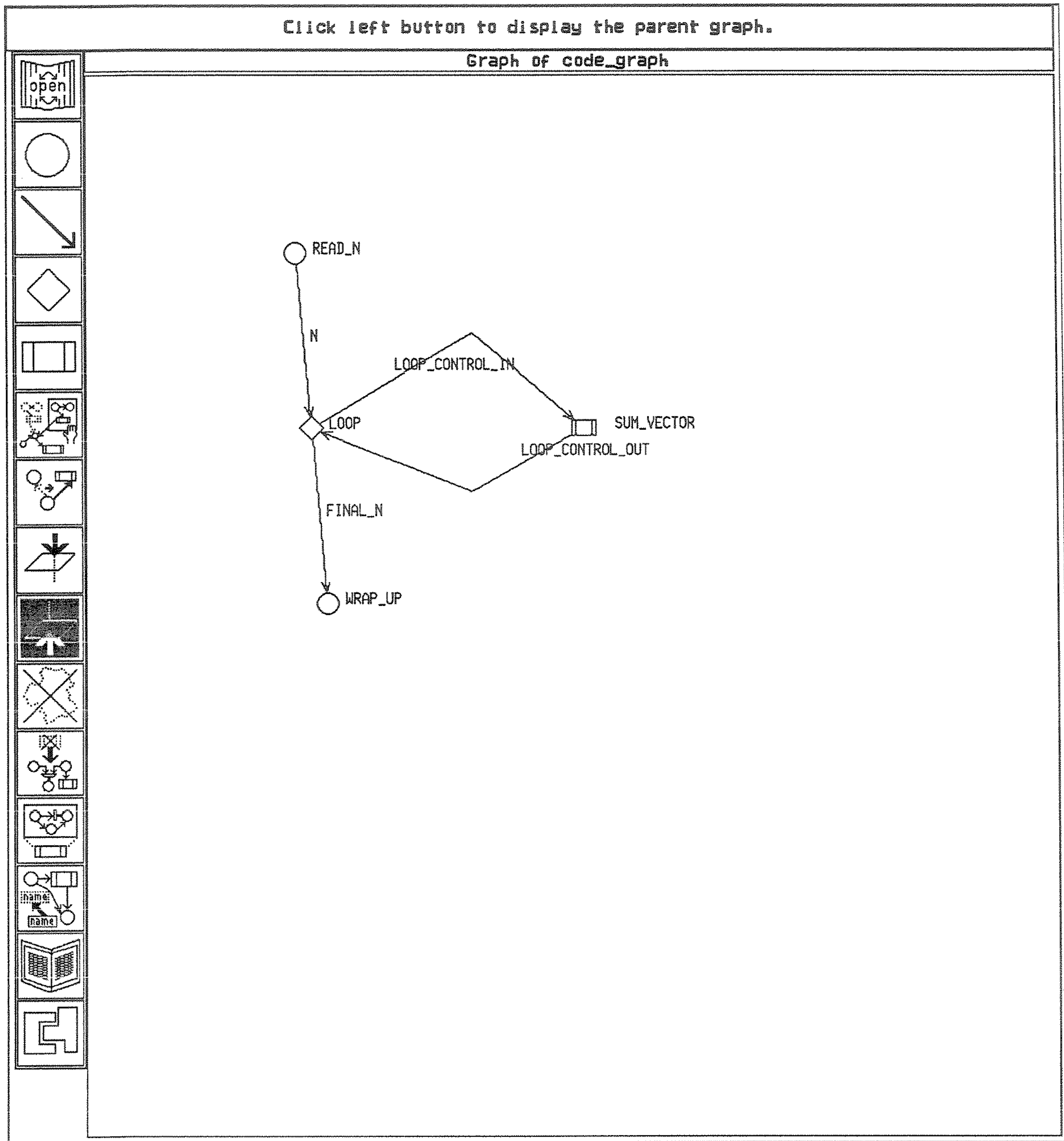


Figure 4
Actual CODE System Window for Example B

D. Multiple identical objects (Replication dependencies and nodes)

Graphs often contain repeated nodes and dependencies. These reflect parallelism achieved by allowing multiple copies of the same program to work on data objects of the same type. Such a graph might look like Figure 5, which represents the process of dividing a data structure into substructures, passing these substructures to copies of the same routine to compute partial results, and then assembling the partial results. This same information is captured using less screen area with the notation of Figure 6, in which concentric circles and brackets are used to indicate multiple copies of a SUC, and brackets on the data name indicate multiple instances of a data type. This style could have been used in the first example, rather than having the two (nearly) identical routines Add1 and Add2.

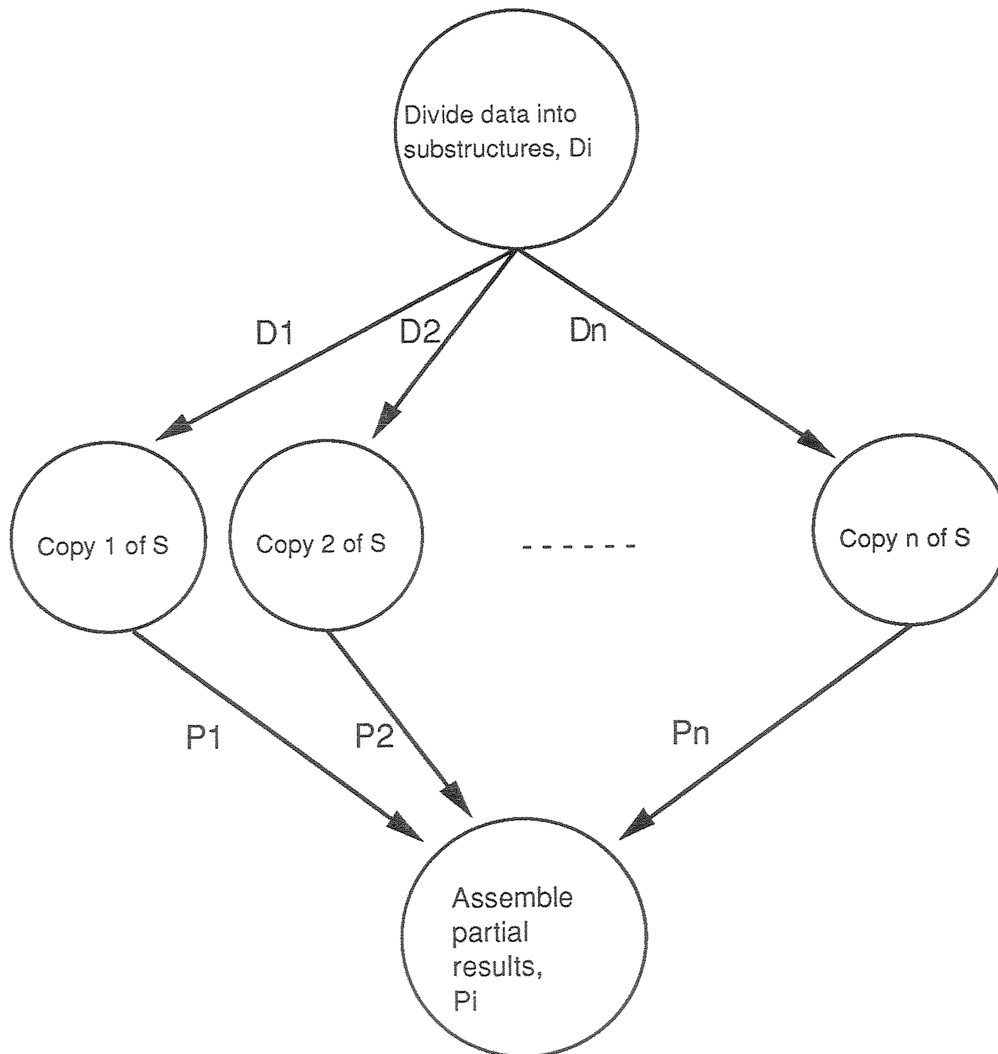


Figure 5
Multiple Identical Objects

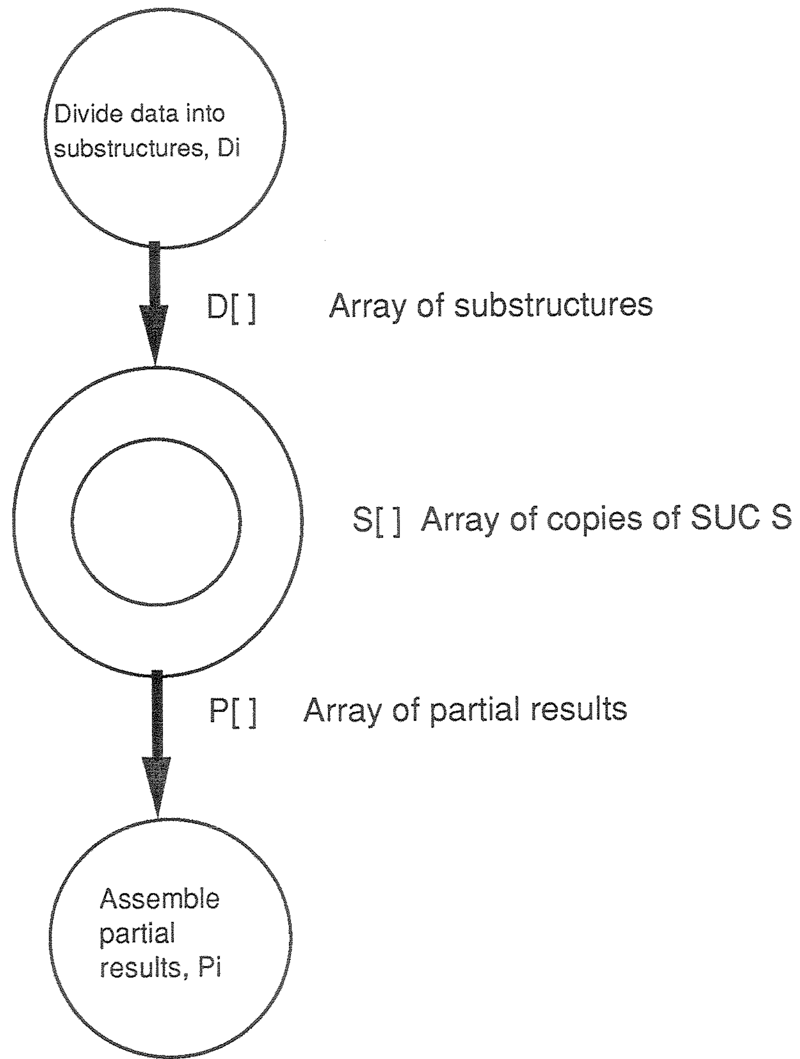


Figure 6
Replication Node

E. Some limitations of CODE 1.2

1. Use of names

a. The name space of the graph is entirely global, so scoping effects cannot be achieved at the graph level. In particular, subgraphs are not a scoping mechanism. Two arcs whose sources and sinks are disjoint are allowed to have the same label. Of course, the name spaces of the individual nodes are completely private except for those variables shared with other nodes by using the exclusion dependency mechanism.

b. Naming is clumsy in some instances since CODE 1.2 uses the name of data flowing on a directed arc to refer to the arc. This is handy in many ways, but since there is no separate arc name, there is no way to allow the same data name to be associated with different arcs impinging on the same SUC. This causes problems when one wishes to pass a data name or item through a node since a new name will have to be used for the output value even though the data may be unchanged

2. Subgraphs

Replicated subgraphs are not allowed.

3. Exclusion dependencies

A node may participate in at most one exclusion dependency. This dependency may involve more than one data item but not more than one constraint.

4. Replication Nodes

The structure does not allow arbitrary mapping of replicated dependencies to replicated nodes.

5. Dynamic Graphs

There are only limited facilities for dynamic graphs. The notation used for multiple SUCs allows the input of a parameter denoting the number of copies to make. See Section IX.

6. Switch Nodes

a. There are clumsy constructs needed to handle some frequent cases. Perhaps the worst is when the data on one arc must be tested to see if the data on another arc will be passed forward. See Section IV.

You should now do the Tutorial Examples 1 and 2

II. SUC Nodes

This chapter assumes that the reader has read the introduction and has worked through Example 1 and Example 2.

A. Introduction

The idea behind CODE graphs is that the program is described by two components: a collection of units of computation (the SUC nodes) which do the actual work and a high level graph which describes the parallelism and the interaction among these components.

At program construction time SUC computations are programs structured as a template generated by the system, filled in with user inserted code.

At execution time the two defining characteristics of the SUC nodes are:

1. They interact with other nodes in the graph only at the following times:
   a. At the beginning of execution they consume a unit of data from each input data dependency
   b. At the end of execution they output one unit of data to each output data dependency
   c. During execution they may read and write shared variables

2. The programmer may not assume that a SUC retains state between executions of the node. (This is discussed further below)

B. Entering SUC Nodes

SUC nodes are entered and named as in Tutorial Example 1. The programs at a SUC are made up of two parts:
   a. a template generated by the system
   b. user inserted code.

1. Template

The template is of the following form (exact syntax depends on the language):

Subprogram Header (with parameter list)
Declarations of parameters and shared variables
Comment Section describing parameters
Comment Section describing shared variables

(*User Written code*)

Return and/or End Statements

NOTE. An extremely important point is that the update operation will completely rewrite this template. Only the user written code between the template comments and the Return/End will be carried over into the new template. **Any other user written code will be deleted. So keep your code in the user area!!**

2. Example templates

Assuming a node ADD1 with input dependency V1, output dependency SUM1, and shared variable SUM the following are the templates generated in the different languages:

Ada

```
type yyty0 is array(0..49) of INTEGER ;
procedure ADD1( V1 : in yyty0; SUM1 : out INTEGER; SUM : in out INTEGER ) is
--  V1 IS INPUT ONLY
--  SUM1 IS OUTPUT ONLY
--  SUM IS SHARED (BOTH INPUT AND OUTPUT)
-- begin user-written code --

end ADD1;
```

C

```
#define BOOLEAN int
ADD1(V1,SUM1,SUM)
int       V1[50];
int       *SUM1;
int       *SUM;
/*  V1 IS INPUT ONLY  */
/*  SUM1 IS OUTPUT ONLY  */
/*  SUM IS SHARED (BOTH INPUT AND OUTPUT) */
{
/* User Written Code*/
}
```

Fortran

```
      subroutine ADD1(V1,SUM1)
       integer   V1(50)
       integer   SUM1
       COMMON /SUM/SUM
c*  V1 IS INPUT ONLY
c*  SUM1 IS OUTPUT ONLY
c*  SUM IS SHARED AS COMMON BLOCK
c*   begin user-written code

       return
       end
```

3. User written code

An extremely important point is that the update operation will completely rewrite the area belonging to the template. Only the user written code between the template comments and the Return/End will be carried over into the new template. **Any other user written code will be deleted. So keep your code in the user area!! (We know we said this twice but it's important)**

There are no limitations on user written code other that those imposed by the normal rules of the language and the requirement to respect the naming conventions imposed by the CODE generated template. Of course, it is best if users stick to ANSI standard/portable constructs unless they know their target system well. You should avoid process related system calls.

4. Replicated SUC nodes

See the discussion in Section VII.

C. Execution of SUC node programs

At execution time the two defining characteristics of the SUC nodes are:

1. They interact with other nodes in the graph only at the following times:
    a. At the beginning of execution they consume a unit of data from each input data dependency
    b. At the end of execution they output one unit of data from each output data dependency
    c. During execution they may read and write shared variables

2. The programmer may not assume that a SUC retains state between executions of the node

Each of these has some practical effects on the programmer's efforts.

Condition 1 implies that all input data is available immediately at the start of execution. It also implies that SUC nodes cannot be used to trigger execution selectively depending upon what data is available, nor can they be used to distribute data selectively to output nodes. Each of these tasks must be performed by a combination of switch node (to do the selecting) and SUC nodes (to do the computation).

Condition 2 implies that to achieve the effect of retained state (for example the seed of a random number generator) the programmer must either

1. use what amount to self loops in the style of Figure 3 (see discussion below) or

2. use exclusion constraints and shared variables or

3. adopt the dangerous tactic of programming outside the model (based on some special knowledge about the actual execution environment).

D. The SUC Form

Forms are just records containing information about a CODE object. A typical SUC form is shown in Figure 1; this form is for SUC INIT from Tutorial Example 1. The fields of a SUC form are listed and defined in Figure 2.

```
Suc:            [INIT                                   ]
Bounds:         [                                       ]
Termination Node?:      [■] No   [ ] Yes
Input   Dependencies: ** none **
Output   Dependencies: V1 V2
Exclusion Dependencies: SUM
Code File: [INIT                                   ]
Language:  [ ] Ada   [ ] C   [■] Fortran
Constraint Type?:  [■] Mutex   [ ] Shared
[ edit code ]   [ update code ]   [ quit ]
```

Form for SUC INIT
Figure 1

Field	Description
Suc:	;SUC name
Bounds:	;number of identical SUCs represented by this symbol
Termination Node:	;yes or no, does execution of this node imply termination of ;the program?
Input Dependencies:	;list of input dependencies for this SUC. Information ;supplied by system.
Output Dependencies:	;list of output dependencies for this SUC. Information ;supplied by system.
Exclusion Dependencies:	;list of exclusion dependencies in which the SUC participates ;(at most 1 in CODE 1.2). Supplied by the system.
Code File	;name of file holding actual code of SUC
Language:	;choice of Ada, C, Fortran
Constraint Type?	;type of participation in an exclusion dependence if there is ;one, meaningless otherwise. Default is mutex
Form Actions	
edit code:	;enter user code into the SUC
update code:	;rewrite template. This may be invoked repeatedly as the ;connecting dependencies are changed
quit:	;exit the form

Fields of a SUC Form
Figure 2

The fields of the SUC form (Figure 2) are generally self explanatory, but there are some potential confusions:

1. Termination Node

In a parallel program, an important issue is knowing when to terminate the program and kill all active processes which may have been spawned during execution. Designation of a node as a Termination Node is a signal from the programmer that the full program should be terminated when execution of the Termination Node is complete. More than one node may be so designated.

2. Constraint Type

A SUC may participate in an exclusion dependency in one of two ways: as a mutex node or as a share node. This is discussed in more detail in Section V, Exclusion Dependencies, but briefly, a node designated as share may access the exclusion dependency data simultaneously with any other share node but not with any mutex node. A node designated mutex may not share access with any other node, whether share or mutex.

3. Update Code - (re)generates the template to match the graph

This Form Action causes the template to be (re)written to match the graph. The system **does not** automatically maintain consistency between the graph and the SUC node program. Whenever the dependencies involving a node are changed, this Action must be invoked. A key point is that any user written code which appears outside the allocated user area will be deleted when this action is invoked.

4. Edit Code - allows user to enter code

This Form Action invokes the user's default editor on the complete node program. Note that this means the user can edit the template portion of the program, **a very dangerous action!**

E. Remarks

1. Using "Self Loops" to Retain State

Suppose that SUC N has a variable v whose state (value) is to be retained between executions. Suppose also that the value of v is to be provided by a external input variable v_initial on the first execution of the SUC. In addition, suppose that the SUC has input data dependency I and output data dependency O. Then the SUC of Figure 3 has the desired behavior, where the switch node is used to handle the two different cases of initialization and persisting value. Note that a simple self loop on N will not work since the SUC N cannot fire unless it has data on all input dependencies. If the dependencies v_persisting and v_initial were directly incident on N, then at initialization there would be no data on v_persisting, and on later executions there would be no data on v_initial.

An important point about this example is that three distinct names such as v, v_initial, v_persisting must be used. The naming conventions of CODE prevent the use of v for all three or even any two.

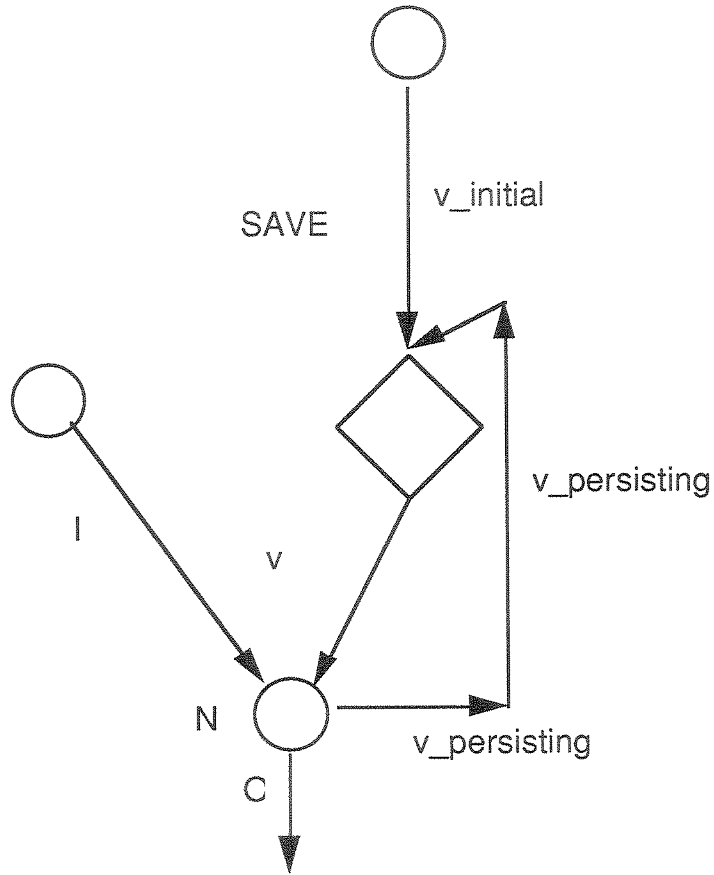


Figure 3
Using Self Loops to Retain State

III. Data Dependencies

This chapter assumes that the reader has read the introduction and has worked through Tutorial Example 1 and Tutorial Example 2.

A. Introduction

Data dependencies are a key element in CODE graphs. They represent data produced and consumed by the SUC nodes. Ultimately, they represent data communication required for the execution of the CODE program. As such, a key runtime issue for CODE graphs is the buffering capacity implied by these communication pathways.

Data dependencies are quite different from exclusion dependencies, at least in their uses, but because of the many similarities in entering the two types of dependencies, a single form is used for both.

Entering dependencies in lists to be associated with arcs is an area which causes some problems for users.

B. Entering Data Dependencies

Single data dependencies are entered and named as described in Tutorial Example 1.

1. Lists of dependencies (this discussion applies also to exclusion dependencies)

To reduce screen clutter, more than one dependency may be associated with an arc. This is referred to as a list of dependencies. The arc for such a dependency is fatter than an arc which carries a single dependency. The name of the arc is taken to be that of the first dependency on the list. This naming convention sometimes leads to trouble, both because the arc (a graphical element) does not have a name of its own and because the names of dependencies other than the first cannot be seen without opening the dependency.

2. Creating and modifying lists of dependencies

To associate multiple dependencies with a single arc, create a single dependency, open its form and use the actions at the bottom of the form

Form Actions: first, previous, next, last, add, delete, quit.

All except **quit** (which has its usual meaning of **exit (or close) form**)are used to create and modify a list of either data or exclusion dependencies. The actions have exactly the meanings you would expect from their names. **First, previous, next,** and **last** are used to move backward and forward through the list. As you move through the list the form changes to reflect the current dependency being viewed. **Add** and **delete** are used to modify the list.

a. Adding new dependencies to the list

Add will create a new dependency in the list immediately behind the current dependency. A new form is presented to the user to be completed. **Unfortunately**, for a variety of practical reasons having to do with the interface, the process of completing this form is filled with opportunities for making errors. These errors are essentially typographical in nature, but frustrating anyway. **The key point is that the cursor must be within the field being completed and that the entry must be terminated with a 'return'.** Startling effects can be achieved if these rules are not followed, since entries typed **will** go into the field in which the cursor happens to lie, and even though data may be typed into the correct field, it **will not** be captured unless the 'return' is entered. If an error is made in entering a dependency (e.g. wrong names is entered), the safe way to correct it is by deleting and re-adding.

Once **add** has been chosen, **a dependency will be added to the list** unless an error is detected by the system. You will not be allowed to quit until the dependency's name has been supplied.

b. Deleting a dependency

Delete will delete the current dependency being displayed, **except** that the first dependency (which is also the arc name) cannot be deleted except by deleting the entire arc! This is done by using the Delete icon from the main menu bar.

3. Replicated switch nodes

See the discussion in Section VII.

C. Execution time

The key questions about data dependencies at execution time are the sizes and queuing disciplines of the buffers associated with them. In CODE 1.2 the buffers are FIFO and "infinite"; that is, they are allocated from a heap and may become as large as the available memory.

If the data dependency is an input to a SUC, then at the start of execution, one data item is removed from a dependency's buffer. If it is an output from a SUC, then at end of execution a data item is added to the buffer. These buffers are organized as FIFO queues.

D. The Data Dependency Form

A typical data dependency form is shown in Figure 1; this form is for data dependency V1 from Tutorial Example 1. An important point is that a single form is used for several different purposes, namely to describe data dependencies, exclusion dependencies, and routing for switch nodes. Consequently, some fields are used only for certain of these cases and otherwise left blank. A list of the fields and their meanings is found in Figure 2.

Dependency: V1

Bounds:

Kind : ☒ data ☐ exclusion ☐ replication

Input SUC/Switch : INIT

Output SUC/Switch : ADD1

Full Start : INIT

Full End : ADD1

Data Type : ☐ int ☐ real ☐ bool ☐ char ☒ int array ☐ real array ☐ bool array ☐ char array

Array Size: 50

Default:

Exclusion SUCs:

Exclusion Constraint:

[first] [previous] [next] [last] [add] [delete] [quit]

Form for Dependency V1
Figure 1

Dependency	;data dependency name
Bounds	;number of identical data dependencies represented by this symbol
Kind:	:data
Input SUC/Switch/Subgraph	;name of source
Output SUC/Switch/Subgraph	;name of sink
Full Start	;actual source SUC in the case source is a subgraph
Full End	;actual sink SUC in the case sink is a subgraph
DataType	;one of int, real, bool, char, or arrays of any of these
Array Size	;length of the array
Default	;exclusion dependencies only
Exclusion SUCs	;exclusion dependencies only
Exclusion Constraint	;exclusion dependencies only
Passing Conditions and Receivers	;for dependencies whose sink is a switch
Form Actions	;All but quit are used to create a list of dependencies to be associated ;with a single physical arc in the CODE graph
first	;display form for the first data item in the list
previous	;display form for the previous data item in the list
next	;display form for the next data item in the list
last	;display form for the last data item in the list
add	;add new data item to the list
delete	;delete the data item from the list whose form is currently displayed
quit:	;exit (close) the form

Fields of the Dependency Form
Figure 2

The fields of the data dependency form (Figure 2) are generally self explanatory, but there are some potential confusions:

1. **Full Start**

This will generally be the same as the entry for the Input SUC/Switch/Subgraph field of the form. However, when the source is a subgraph, this field records the name of the actual node within the subgraph to which the dependency is connected. See dependency LOOP_CONTROL_OUT in Tutorial Example 2 for an example.

2. **Full End**

This will generally be the same as the entry for the Output SUC/Switch/Subgraph field of the form. However, when the sink is a subgraph, this field records the name of the actual node within the subgraph to which the dependency is connected. See dependency LOOP_CONTROL_IN in Tutorial 2 for an example.

3. **Passing Conditions and Receivers**

Used for dependencies whose sink is a switch; see Section IV, Switch Nodes.

IV. Switch Nodes

This chapter assumes that the reader has read the introduction and has worked through Tutorial Example 1 and 2.

A. Introduction

Switch nodes were introduced in CODE for a variety of reasons. These include

1. selectively distributing data to different nodes depending on conditions satisfied by the data (Figure 1)
2. implementing looping constructs (really a special case of 1) (Figure 2)
3. non-deterministic merging of data streams (Figure 3)

Each of these uses is based on the key fact that a switch is enabled for execution whenever data is present on **at least one** of its input data dependencies (as opposed to **all** for SUCs) and that a switch may place output on a **subset** of its output data dependencies (as opposed to **all** for SUCs)

It is important to note that switches do no computations other than testing of data conditions on the input data and routing of the data to output dependencies. Even more narrowly, a combined data condition and routing specification may involve only data from a single input dependency. This is a serious restriction on the capabilities of switches.

B. Entering Switch Nodes

A switch node is placed in the graph, named and has its form filled in as in the Tutorials.

1. Data and routing conditions

The data and routing conditions for a switch are actually associated with the input data dependencies. These conditions are recorded in the **Passing Conditions and Receivers** field of the dependency form. The grammar for these is the following:

```
P_C_R                -> Cond_Routing_Pair ',' P_C_R
                     -> Cond_Routing_Pair
Cond_Routing_Pair    -> Data_Condition '|' Routing
Data_Condition       -> true or any boolean valued C expression with relative
                          operators (<, >, ==, etc) among C expressions.  These
                          expressions may only involve data from the associated
                          data dependency and constants.
Routing              -> list of output data dependency names separated by spaces
```

Notes: 1. No built in functions, e.g. sin (x). are allowed in a Data_Condition
2. The empty list is not allowed as a routing

A Very Important Warning: The data and routing conditions are not parsed by the CODE front end. This means the user may enter illegal expressions and not discover the errors until much later when the declarations file is translated.

Examples :

a. Distributing data (Figure 1)

Suppose there is a input dependency N of type integer for a switch node. If the goal is to suppress all N <= 0 and to send copies of all N > 0 to nodes A, B, C, and D, then the data and routing condition would be N > 0 | NA NB NC ND

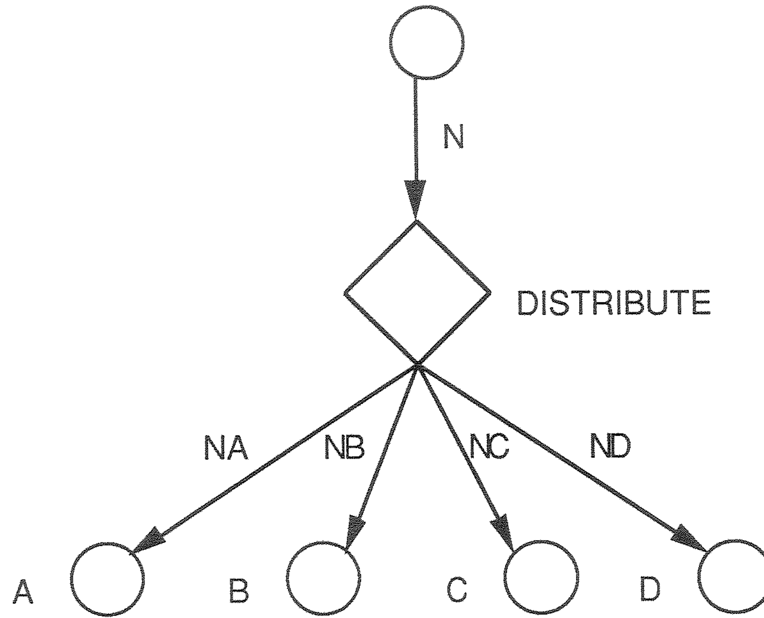


Figure 1
Switch Used to Replicate and Distribute Data. NA, NB, NC, and ND are copies of N

b. Looping (Figure 2. Taken from Tutorial Example 2)
The data and routing conditions for the input data dependency N of switch node Loop
N > 0 | Loop_Control_In, N <= 0 | Final_N
For input data dependency Loop_Control_Out the condition and routing is
Loop_Control_Out > 0 | Loop_Control_In, Loop_Control_Out <= 0 | Final_N

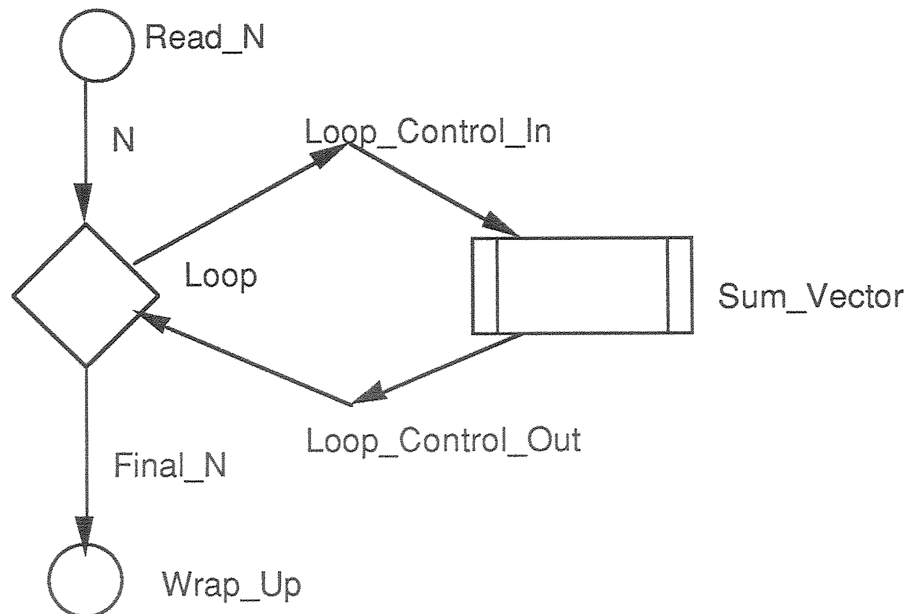


Figure 2
Looping Using a Switch

c. Merging
The data and routing conditions for each input dependency of a merge switch (which simply combines two or more data streams as in Figure 3) is

True | Destination (e.g., True | N in Figure 3.)

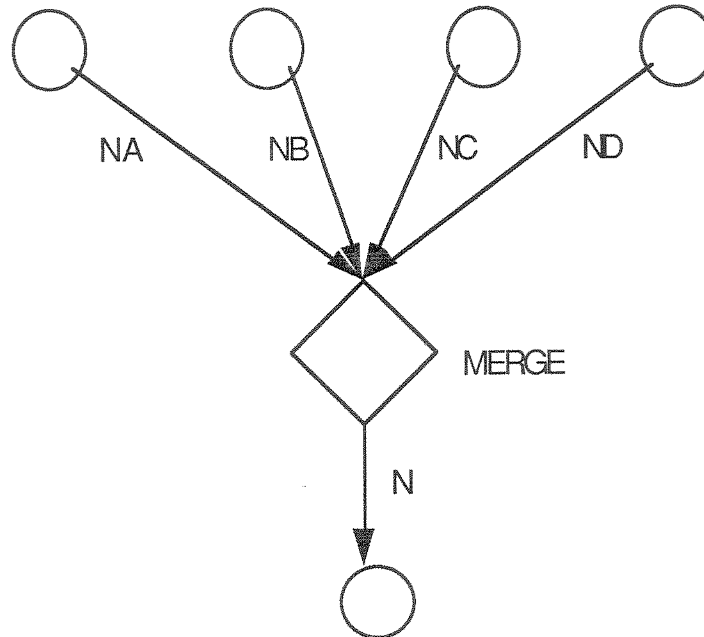


Figure 3
Merge Node. N will be one of NA, NB, NC, ND

2. Replicated switch nodes

See the discussion in Section VII.

C. Execution time

A switch is enabled for execution when data is present on one of its input data dependencies. Execution occurs as follows:

From those input data dependencies which have data present, one is chosen non-deterministically (that is, the user may not program as though they know which arc will be chosen). Data is removed from this dependency, and the data condition of the dependency is tested. If the data condition is satisfied, then the data is distributed to the output dependencies of the switch according to the routing specification.

D. The Switch Node Form

Figure 1 is the form for the switch node LOOP of Tutorial Example 2. The general list of fields in the switch form is in Figure 2. They all have their obvious meanings.

Switch:	LOOP
Bounds:	

Input Dependencies: N LOOP_CONTROL_OUT

Output Dependencies: FINAL_N LOOP_CONTROL_IN

quit

Figure 1
Form for Switch Node LOOP of Tutorial Example 2

Switch:	;name of the switch
Bounds:	;number of replications of this switch
Input Dependencies	;list of input data dependencies for this switch
Output Dependencies	;list of output data dependencies for this switch
quit	;Form Action button to exit (close) the form

Figure 2
Fields of a Switch Node Form

E. Remarks

1. Forwarding data based on the state of several dependencies

Suppose you wish to organize a computation Test_and_Forward as follows:

a. There are two inputs, Trigger (produced by computation T) and Data (produced by computation D). T and D may be the same computation. The computation Test_and_Forward should test Trigger. If it is greater than 0 then Data should be forwarded to node Process. Otherwise, Data should not be forwarded. As CODE is currently designed, this computation cannot be organized as a single Switch node with inputs Trigger and Data and output New_Data to Process. This is because a test on Trigger cannot be used to route data from another dependency (Data). However, a single SUC cannot be used either since the execution of such a SUC would promise to output New_Data at the end of each execution. There are three ways to handle this:

i. Use a switch and pass Trigger and Data to the switch node as part of a single array . This may require changes to T and D. Note that records are not supported as a data type for data dependencies.

ii. Use a SUC and modify Process to accept some value of New_Data as an order not to process the data. The SUC can then output that value when Trigger is <= 0.

iii. Note that i. and ii. may both require modification of the existing computations T and D. To avoid this you may use a graph like that of Figure 3. In Figure 3, New_SUC does the test and either sets New_Data to Bad_Value or to Data, depending on the value of Trigger. The data condition and routing on dependency New_Data is

New_Data != Bad_Value | Data'

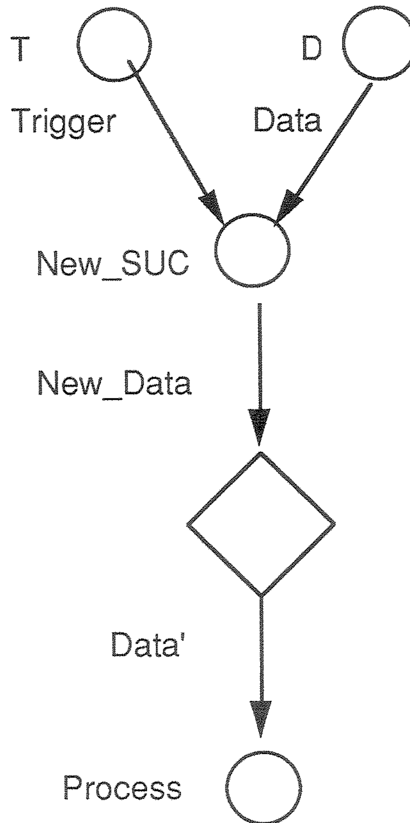


Figure 3

In some sense this is an implementation of ii. above, but it has the merit of not modifying Process.

V. Exclusion Dependencies

This chapter assumes that the reader has read the introduction and has worked through Tutorial Example 1 and Tutorial Example 2.

A. Introduction

Exclusion dependencies are used to share data among SUC nodes. This is a powerful paradigm for parallel programming. The actual implementation of this data sharing is hidden from the programmer and is handled by the CODE backends. This eliminates a common source of errors {data races} in parallel programs. In addition, the user is able to provide an optional default initial value for the data items.

CODE 1.2 programs are limited in the sense that a given SUC may not participate in more than one exclusion dependency. Also, it is assumed that termination of execution of a node can never result in an exclusion dependency error. This excludes conditions of the form: Node A may only execute and access data item X, if Node B is also executing and able to access data item X.

B. Entering Exclusion Dependencies

1. Dependencies

Exclusion dependencies are entered and named as described in Tutorial 1.

2. Lists of dependencies

The process of creating a list of data items associated with the exclusion dependency is identical to that used for data dependencies. See the discussion in Section III.

3. Constraints

Constraints are read only in the Exclusion Constraint field of the Exclusion Dependency Form; they are not entered there. Instead the user specifies in the Constraint Type? field of the participating SUCs whether the SUC participates as a Share node or a Mutex node. The constraint may have many nodes of each type. The constraint is satisfied if one of the following is satisfied:

- i. no node participating in the dependency is executing
- ii. one Mutex node is executing and no other node is executing
- iii. any subset of the Share nodes is executing and no Mutex nodes is executing

Appendix D is a formal listing of the constraint syntax and its interpretation.

The constraint syntax used to describe constraints in the Exclusion Constraint field of the Exclusion Dependency Form is the following:

a. Constraint syntax

```
Exclusion_Constraint  -> Mutex _Constraint  mutex  Share_Constraint
Mutex _Constraint     -> Mutex _Constraint  mutex  SUC_Node_Id
                      -> SUC_Node_Id
                      -> epsilon
Share_Constraint      -> Share_Constraint   or  SUC_Node_Id
                      -> SUC_Node_Id
                      -> epsilon
```

b. Constraint examples

i. Simple mutual exclusion
Figure 1 is the simple exclusion constraint of Tutorial Example 1. If more nodes are involved, say Add1, Add2, Add3, Add4, the constraint is

Add1 mutex Add2 mutex Add3 mutex Add4.

Dependency: SUM
Bounds:
Kind : ☐ data ☒ exclusion ☐ replication
Input SUC/Switch :
Output SUC/Switch :
Full Start :
Full End :
Data Type : ☒ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array
Array Size:
Default:
Exclusion SUCs: ADD2, INIT, ADD1, PRINTSUM
Exclusion Constraint: ADD2 mutex INIT mutex ADD1 mutex PRINTSUM
[first] [previous] [next] [last] [add] [delete] [quit]

Form for the Exclusion Dependency SUM
Figure 1

ii. Readers/Writers
If W1 and W2 are writers and R1, R2, R3 are readers, then the constraint is
W1 mutex W2 mutex (R1 or R2 or R3)

iii. Producer/consumer
If P is the producer and C is the consumer, then the constraint is P mutex C

C. Execution time

At execution time exclusion constraints are regarded as predicates which must be satisfied by the runtime system at all times. Since the termination of the execution of a node can never falsify these predicates because of their restricted form, this translates to a condition for allowing a node to begin execution. The condition is the following:

> From those SUCs which are enabled for execution, the runtime system checks to see which may be started without falsifying the exclusion constraint in which they participate (if any). One such node is allowed to begin execution.

A constraint is satisfied if one of the conditions i, ii, or iii on the previous page is satisfied. Appendix D is a formal listing of the constraint syntax and its interpretation.

D. The Exclusion Dependency Form

The exclusion dependencies and data dependencies share the same form since they have many fields in common. For the exclusion dependencies we must set an exclusion constraint. In addition, the user is able to provide an optional default initial value for the data items.

The fields of the exclusion dependency form are given in Figure 2. They are generally self explanatory except for:

1. Input SUC ;This reflects internal information and is not user settable. It will be deleted soon from the exclusion dependency form.
2. Output SUC ;This reflects internal information and is not user settable. It will be deleted soon from the exclusion dependency form.
3. Default ;A list of values used to initialize the data. The only syntax is a list of values separated by commas. This works well for a few simple data items, but more complex data, for example a large array, should be initialized by invoking a SUC.

Dependency	;exclusion dependency name
Bounds	;number of identical dependencies represented by this symbol
Kind:	:exclusion
Input SUC	;not used for exclusion dependencies
Output SUC	;not used for exclusion dependencies
Full Start	;not used for exclusion dependencies
Full End	;not used for exclusion dependencies
DataType	;one of int, real, bool, char, or arrays of any of these
Array Size	;length of the array
Default	;initial values for dependency data
Exclusion SUCs	;list of SUCs involved in this exclusion dependency
Exclusion constraint	;condition that must be satisfied at runtime for use of the data by a SUC
Form Actions	;Many of these are used to create a list of data items to be associated ;with a single exclusion dependency in the CODE graph
first	;display form for the first data item in the list
previous	;display form for the previous data item in the list
next	;display form for the next data item in the list
last	;display form for the last data item in the list
add	;add new data item to the list
delete	;delete the data item from the list whose form is currently displayed
quit:	;exit the form

Fields of the Dependency Form
Figure 2

E. Remarks

1. A SUC node may not participate in more than one exclusion dependency. That one dependency may involve a list of data items. The restriction is that a SUC node may be grouped with at most one set of other nodes to form an exclusion dependency.
2. Switch nodes may not participate in an exclusion dependency.
3. It is assumed that termination of execution of a node can never result in an exclusion dependency error. Consequently, this excludes synchronizations of the form: Node A may only execute if Node B is also executing.

VI. Subgraphs

This chapter assumes that the reader has read the introduction and has worked through Example 1 and Example 2.

A. Introduction

Subgraphs are used as a hierarchical structuring mechanism in CODE and are consequently very important in the actual engineering of CODE programs. They are also a valuable mechanism for conserving screen real estate.

The graph in which the subgraph node appears is called the **parent** graph of the subgraph. Each subgraph has a dummy start node called the **From_Parent** and a dummy stop node called the **To_Parent**. Any subgraph may flattened into its parent graph, and any collection of nodes in a graph may be designated as a subgraph.

B. Entering Subgraphs

Method 1. Draw a graph and then designate a portion of it as a subgraph. This is one of the operations controlled by the menu bar of the main CODE window. Simply select that icon and then draw a box around the portion of the graph to be reduced.

Method 2. Create an empty subgraph node in an existing graph, and then open the subgraph and edit it. Afterward, connect the subgraph to the rest of the graph.

1. Connecting a subgraph to the rest of the graph

Given a subgraph, it might be connected to the outside world in several ways: input dependencies, output dependencies, and exclusion dependencies. The key point is to realize that the computation has not been completely specified until all dependencies actually connect SUCs/Switches to SUCs/Switches. That is, a dependency can only connect a SUC/Switch node in the subgraph to some other SUC/Switch. The user has the option of leaving these dependencies partially unspecified while creating the CODE graph, but in the end, before the executable code can be generated, all dependencies must be connected to specific nodes.

a. Input dependency

An input dependency to a subgraph is created by drawing a data dependency from a SUC/switch/subgraph to the subgraph. At that point the user is asked

"Do you wish to end or to continue?" and there is a confirmation message

"END here with left button; CONTINUE in other graph with RIGHT or MIDDLE button"

"To end" means to defer for the moment the connection of the dependency to a specific node of the subgraph. **Warning:** there is no explicit notation showing that the connection has been deferred. If the user forgets to connect it later, then the first error message will come from the declarations file generator.

"To continue" means to enter the subgraph and then continue the dependency to the node (SUC or switch) in the subgraph to which it should be connected. Note this dependency passes through From_Parent.

b. Output dependency

An output dependency is drawn from a subgraph node to the To_Parent The user is then asked

"Do you wish to end or continue" and there is a confirmation message

"END here with left button; CONTINUE in other graph with RIGHT or MIDDLE button"

"To end" means to defer for the moment the connection of the dependency to a specific node of the parent graph. **Warning:** there is no explicit notation showing that the connection has been deferred. If the user forgets to connect it later, then the first error message will come from the declarations file generator.

"To continue" means to exit the subgraph and then continue the dependency to the SUC in the parent graph. Note this dependency passes through the node To_Parent.

c. Exclusion dependency

The method is just like that for a data dependency. An exclusion dependency to be extended from inside a subgraph to the parent graph is handled like a output dependency from a node in a subgraph. An exclusion dependency to be extended into a subgraph from the parent is handled like an input dependency.

2. Replicated subgraphs

See the discussion in Section VII.

D. The Subgraph Form

Figure 1 shows the form for the subgraph of Tutorial Example 2.The fields of the subgraph form are described in Figure 2.

```
Subgraph: [SUM_VECTOR                    ]

Parent Graph: code_graph

SUCs & Switches:INIT, ADD1, ADD2, PRINTSUM

Subgraphs    : ** none **

Dependencies: LOOP_CONTROL_IN, LOOP_CONTROL, LOOP_CONTROL_OUT, SUM, SUM2, SUM1, V2, V1

[read files] [display] [quit]
```

Figure 1
Form for Subgraph SUM_VECTOR of Tutorial Example 2

Subgraph	;subgraph name
Parent Graph	;parent graph name
SUCs & Switches	;list of SUCs and Switches found in this subgraph
Subgraphs	;list of subgraphs found in this subgraph
Dependencies	;list of dependencies found in this subgraph
Form Actions	
read files	;capability associated with reuse subsystem. See ROPE manual
display	;opens a window and displays the subgraph. This window cannot ;be edited
quit	;exits (closes) the form

Figure 2
Fields of the subgraph form

VII. Replicated Dependencies and Nodes

A. Introduction

A shorthand is used to indicate static replications of SUCs, switches and dependencies. This technique allows the user to express regular structures in a compact, legible fashion. There are some complications in describing the connection patterns implied by the replication notation. These are discussed below in detail.

At this time, CODE does not formally support replicated subgraphs; this will be changed in the next version of the system. However, there is a programming technique, discussed in Section IX, which allows many of the benefits of replicated subgraphs to be obtained.

B. Entering Replicated Objects

1. Replicated nodes and switches.

If G is a SUC or switch node and n is an integer, then we indicate n copies of G by doing the following:

a. Draw a normal SUC or switch named G
b. In the form for G enter n in the Bounds field.

This will cause the circle or diamond for G to be outlined and denoted by G[n]. This indicates that the node is replicated. The replication factor is n.

2. Replicated data dependencies.

If D is a data dependency and n is an integer, then we indicate n copies of D by doing the following:

a. Draw a normal data dependency named D
b. In the form for D enter n in the Bounds field.

This will cause the dependency to be denoted by D[n]. This indicates that the dependency is replicated. The replication factor is n.

An Important Note: The actual value of n must be used. This is a static facility. There is a small dynamic capability which is discussed in Section IX.

Consider the examples in Figure 1.

i. a single value, A, labels the arc from S to T[10]. The meaning of this is that
   a. the Ti, i = 1, 10, are identical
   b. a copy of A is sent to each Ti.

ii. a vector of values, A[10], labels the arc from S to T[10]. The meaning is that
   a. the Ti are identical
   b. length A[10] equals length T[10] and
   c. ai is input to Ti.

iii. a vector of values, A[r], labels the arc from S[q] to T[p].

let n = number of input dependencies of T

let m = number of output dependencies of S

let d = number of dependencies carried on the arc A

Then the following conditions are implied

a. the Ti are identical and the Si are identical

b. Either p is a multiple of q or q is a multiple of p

c. r is a multiple of Max(p,q)

d. $n*p = m*q = r*d$

e. input dependency cj of Tf takes its data from output dependency dk of Sg where this data is associated with the u-th dependency in the list A[v] and $(f-1)*n+j = (g-1)*m + k = (v-1)*d+u$.

Note that conditions b and c insure that if Tf receives any input from Sg then either Tf receives all its input from Sg or Tf consumes all the input of Sg.

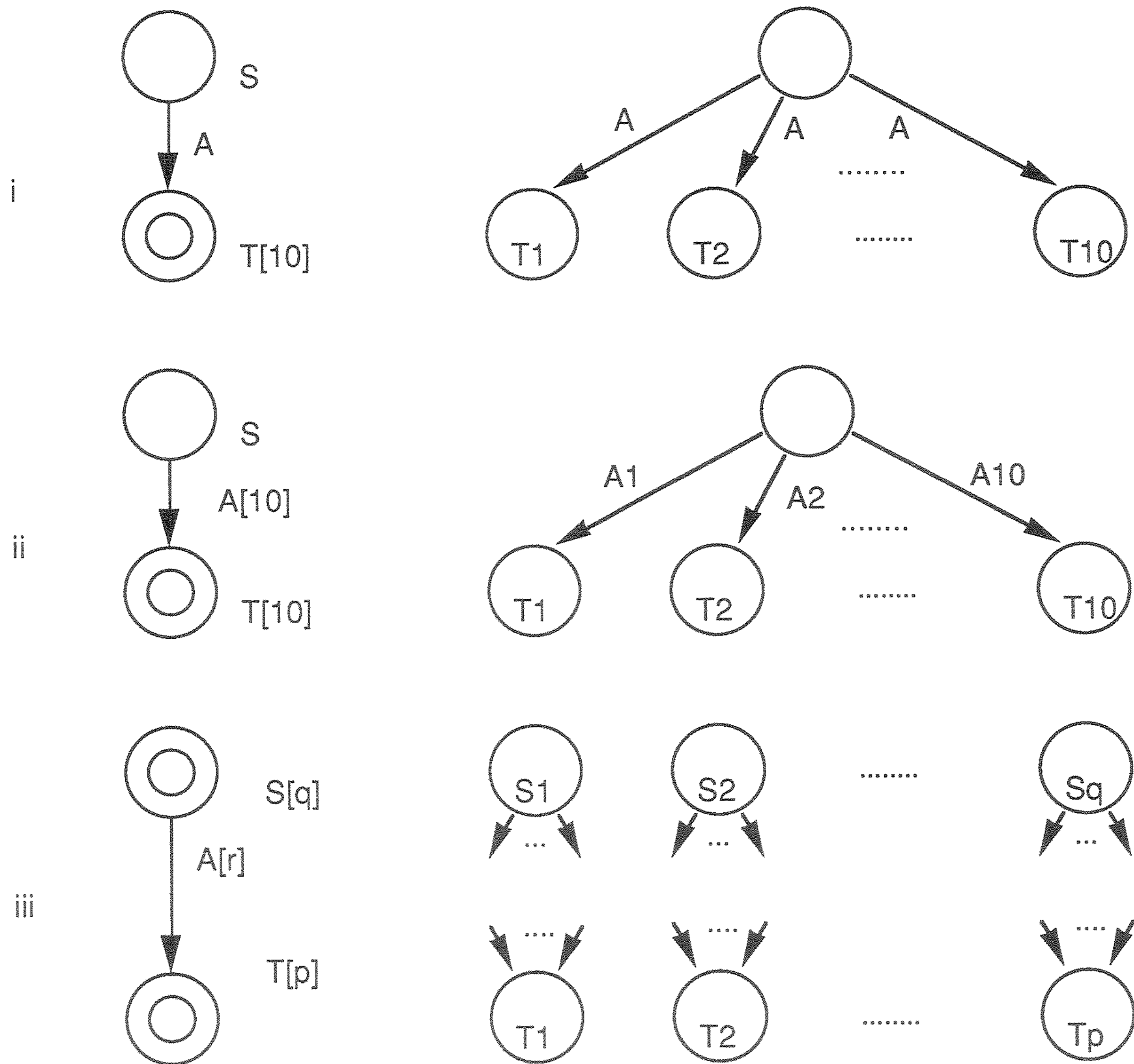


Figure 1
Replication Nodes and Arcs

Consider the even more complex examples of Figure 2.

i. Si supplies p copies of Ai, one to each copy of T

ii. The index ri must be a multiple of p for each i. Tj then receives Ai[(j-1)*p+1 ... j*p] as input dependencies from Si.

iii. Each triple Si(qi), Ai(ri), T(p) must satisfy the conditions of Example iii in Figure 1 and the dependencies are distributed as in that example.

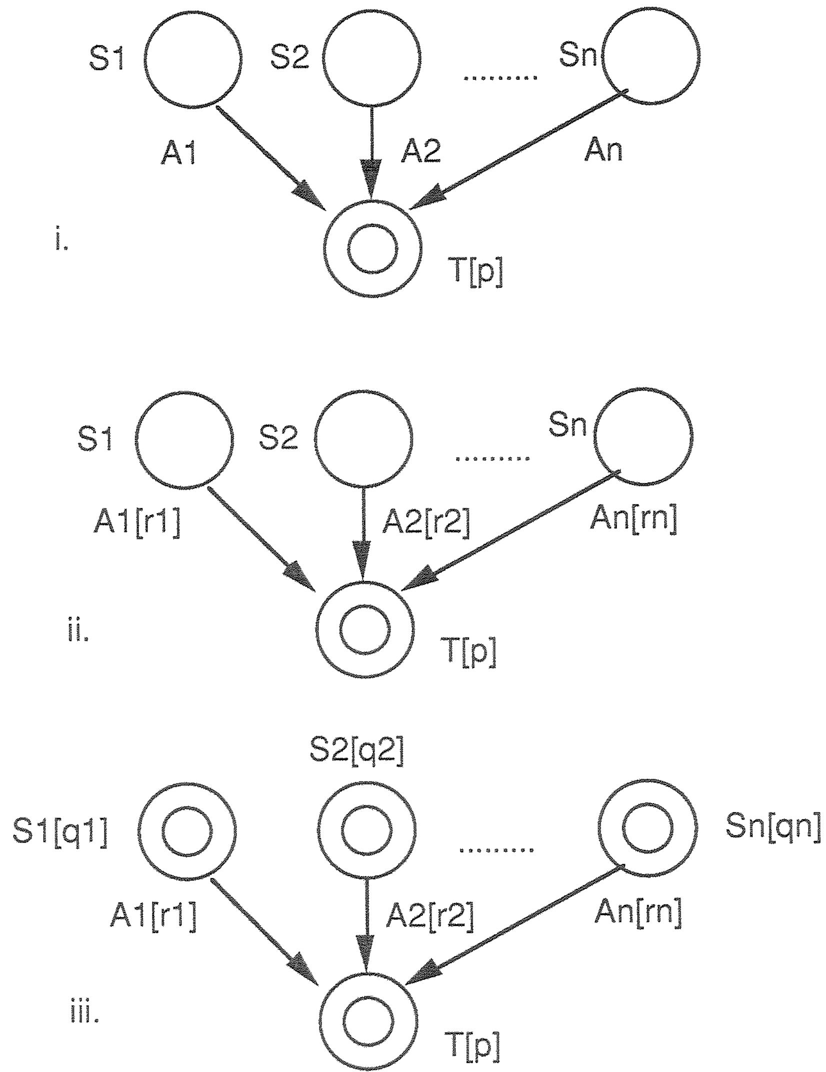


Figure 2
Complex Cases of Replication for Nodes and Arcs

3. Exclusion Dependencies.

If E is a exclusion dependency and n is an integer, then we indicate n copies of E by doing the following:

a. Draw a normal exclusion dependency named E

b. In the form for E enter n in the Bounds field.

This will cause the dependency to be denoted by E[n]. This indicates that the dependency is replicated. The replication factor is n.

Important Notes:

i. The actual value of n must be used. This is a static facility.

ii. As discussed below, exclusion dependencies act differently from data dependencies when involved with replicated nodes.

A simple exclusion dependency involving a replicated node is assumed to extend to all nodes rather than be replicated. For example, in Figure 3.i, we see that the exclusion dependence E simply extends to all of the nodes. Each node participates in the exclusion constraint in the same way as its base node. If S is share then so are S[1] and S[2]. If T is mutex then T[1], T[2], and T[3] are mutex.

In Figure 3.ii, the dependence E[5] is one dependence of dimension 5 in which all of the S's and T's participate.

C. Execution time

These replication techniques are for structuring of programs at program-construction time. The user may imagine that at compile time each such construct is expanded in macro fashion to its corresponding graph as described above and then compiled. These constructs do not effect execution time activities

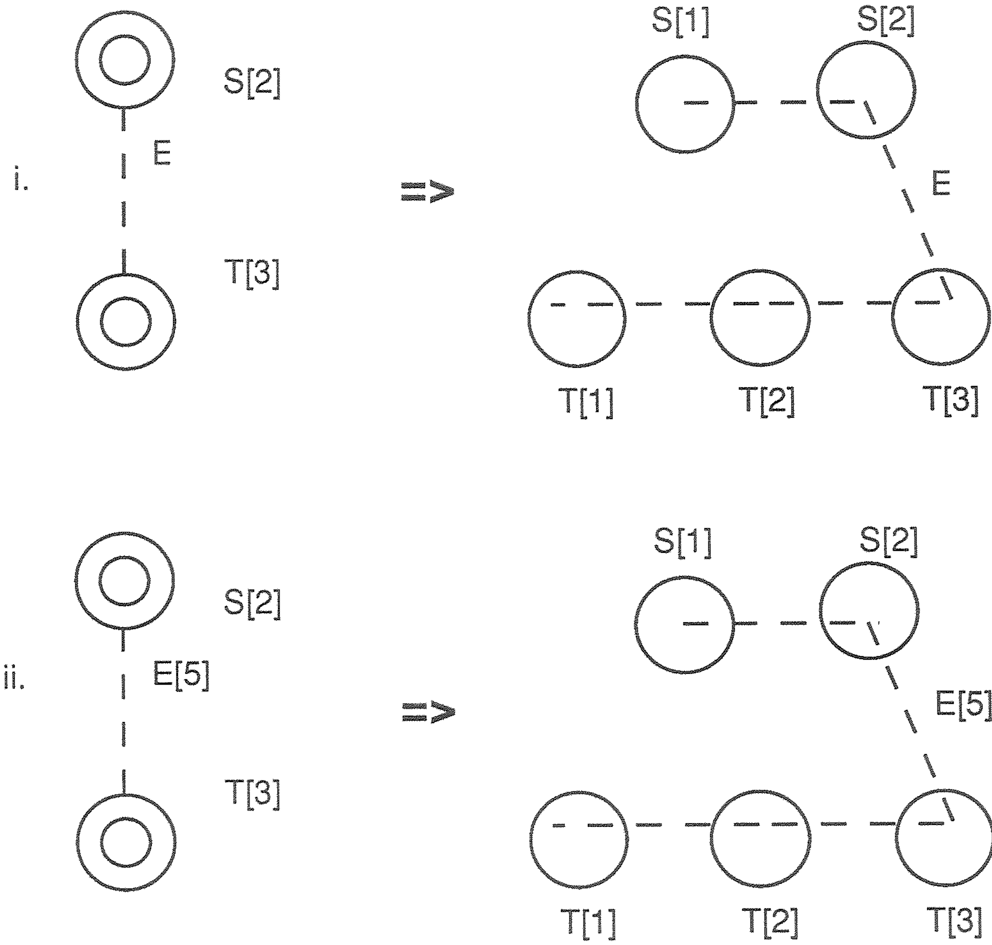


Figure 3
An Exclusion Dependence Among Replicated Nodes

VIII. Executing the CODE Program

This chapter assumes that the reader has read the introduction and has worked through Tutorial Example 1 and Tutorial Example 2.

A. Introduction

Execution of a CODE program requires three steps. First the graph is translated to a so-called declaration file. Then the declaration file is translated to a program for the target language and architecture. This is accomplished by a Translator of A Declaration (TOAD) which has been specialized for a given language and architecture. Finally, the translated program is compiled and executed on the target system.

B. Making the Declaration File

This step is performed in CODE and is independent of the target machine. The user chooses Make Declaration File from the Command Menu. This menu is obtained by clicking the rightmost mouse button with the cursor in the drawing region.

The message bar displays 'File to contain declarations". This is a request for a file name for the declarations file. Enter some standard name like 'decl' followed by a Return. The message bar now reads 'Declarations saved in file decl. The file decl may be viewed with the usual Unix utilities if you are curious.

Of course, this declaration file should not be generated until the program is complete. Some checks for completeness and correctness are done at this point. For example, if a dependency ends at a subgraph and has not been continued to a specific SUC in that subgraph, an error message will be given.

C. Translating the Declaration File

The TOADs (Translators Of A Declaration) convert the declaration file to a program for the target language and architecture. There are different TOADs for different architecture/language combinations. See Figure 1 for a list..

Machine	Language	Toad Name
Cray	Fortran	toad_cft+77

Figure 1
Table of Existing Toads

For example, a Cray Fortran translation is produced by the following command:

toad_cft+77 < decl ('cft' stands for 'cray fortran')

This produces a Fortran source program crayf77.f , which is ready to be compiled and executed.

D. Compiling and Executing on the Target System

Each of these is unique and entirely dependent on the target

1. Executing Example 1 on the Cray.

A. Move the file crayf77.f to the Cray

B. Compile and execute with the following commands:

```
cft+77 -a stack crayf77.f     ;compile to produce object crayf77.o
segldr crayf77.o              ;link to create executable a.out
a.out                         ;execute program
```

IX. Miscellaneous

This chapter assumes that the reader has read the introduction and has worked through Tutorial Example 1 and Tutorial Example 2.

A. Dynamic Graphs

There is a small dynamic capability in CODE. It is enough to allow the run-time specification of the number of copies desired of a replicated SUC. Figure 1 shows the allowable configuration for a dynamic graph. N1 and N2 are equal to N at runtime and inform B and C respectively of the value of N.

An important point is that neither B[N] nor C may have inputs from outside the set {A, B, C}. Similarly, the outputs of A and B are restricted to the set {A, B, C}.

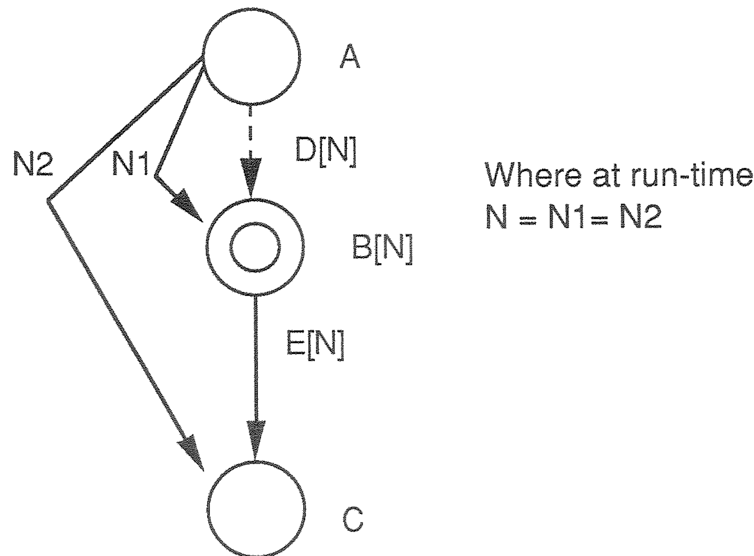


Figure 1
Dynamic Graph

C. Menu Bar Graph Manipulations

1. Moving objects (icon 6 from the top of the menu bar)

This allows the user to rearrange the graph so as to achieve more efficient use of screen real estate or to improve the understandability or esthetics of the graph.

a. Choose menu bar item 6

b. The message "Draw a box around objects to be moved using left button" appears in the message bar.

c. In the usual mouse/graphics style, click the left mouse button and draw a box around the part of the graph to be moved.

d. Click the left mouse button again, an empty box appears representing the objects to be moved. Move this box to the new location and click the left mouse button again. The objects then appear at the new location.

2. Redrawing dependencies (icon 7 from the top of the menu bar)

This allows the user to reroute a dependency. This is useful if the wrong sink was chosen when the dependency was drawn.

a. Choose menu bar item 7

b. The message "Choose a dependency and redraw it using left button" appears in the message bar.

c. Place the cursor on the dependency to be redrawn and click the left button. Then proceed just as in drawing a dependency. There is no need to enter a name, since the old name is kept.

3. Moving names (icon 13 from the top of the menu bar)

This allows the user to move names so as to achieve more efficient use of screen real estate or to improve the understandability or esthetics of the graph.

a. Choose menu bar item 13

b. The message "Point at the object & click left button to move its name" appears in the message bar.

c. Click the left mouse button with the cursor on the object (not on the name). The name then disappears.

d. With the cursor on the new desired location, click the left mouse button again; the name then appears at the new location.

4. Browsing operations

This is covered in the separate manual discussing the reuse capabilities of CODE

5. Rope management

This is covered in the separate manual discussing the reuse capabilities of CODE

D. Graph Manipulation Commands

These commands are found on the menu brought up by the right mouse button.

1. Redisplay

Redisplay the graph.

2. Zoom In

Enlarges the graph, giving the user the impression of zooming in with a camera. The user is asked to choose the object to be displayed in the center of the drawing region after the zoom.

3. New Center

Shifts the graph in the drawing region to place the user designated object in the center of the drawing region.

4. Zoom Out

Shrinks the graph, giving the user the impression of zooming out with a camera. The user is asked to choose the object to be displayed in the center of the drawing region after the zoom.

5. Clear

Completely erases the graph in the drawing region. The saved graph (if any) is not effected.

Appendix A

CODE project publications

M. Azam, C. Lin, "Programming with CODE: A Computation Oriented Display Environment", Department of Computer Sciences, The University of Texas at Austin, Oct. 1988.

J. C. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach", In Proceedings of IEEE International Conference of Parallel Programming, 1985.

J.C. Browne, M. Azam, S. Sobek, "CODE: A Unified Approach to Parallel Programming", IEEE Software, July 1989.

J.C. Browne, T.J. Lee and C. Lin, "ROPE User's Manual: A Reusability-Oriented Parallel-programming Environment", Department of Computer Sciences, The University of Texas at Austin, Oct. 1988.

J. C. Browne, T.J. Lee and J. Werth, "Experimental Evaluation of a Reusability Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, Vol 16, No. 2, 1990.

J.C. Browne, J. Werth, and T.J. Lee, "Intersection of Parallel Structuring and Reuse of Software Components:A Calculus of Composition of Components for Parallel Programs", International Conference on Parallel Processing, 1989.

J.C. Browne and J. Werth, "Software Engineering of Parallel Programs in the Computation-Oriented Display Environment", 1989 Minnowbrook Conference on Software Engineering of Parallel Programs.

J.C. Browne and J. Werth, "Software Engineering of Large Grain Parallel Programs", 1989 Workshop on Large Grain Parallelism, Carnegie-Mellon.

T.J. Lee, "Software Reuse in Parallel Programming Environments", Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, 1989.

P. Newton, " Translation from a Declarative Model Of Parallel Computation to Multiple Procedural Models", Dissertation in progress, Department of Computer Sciences, The University of Texas at Austin, 1990.

S. Sobek, "A Constructive Unified Model of Parallel Computation", Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, 1990.

S. Sobek, M. Azam, and J.C. Browne, "Architecture and Language Independent Parallel Programming: A Feasibility Demonstration", Proceedings of IEEE International Conference of Parallel Programming, 1988.

Appendix B
The CODE Environment

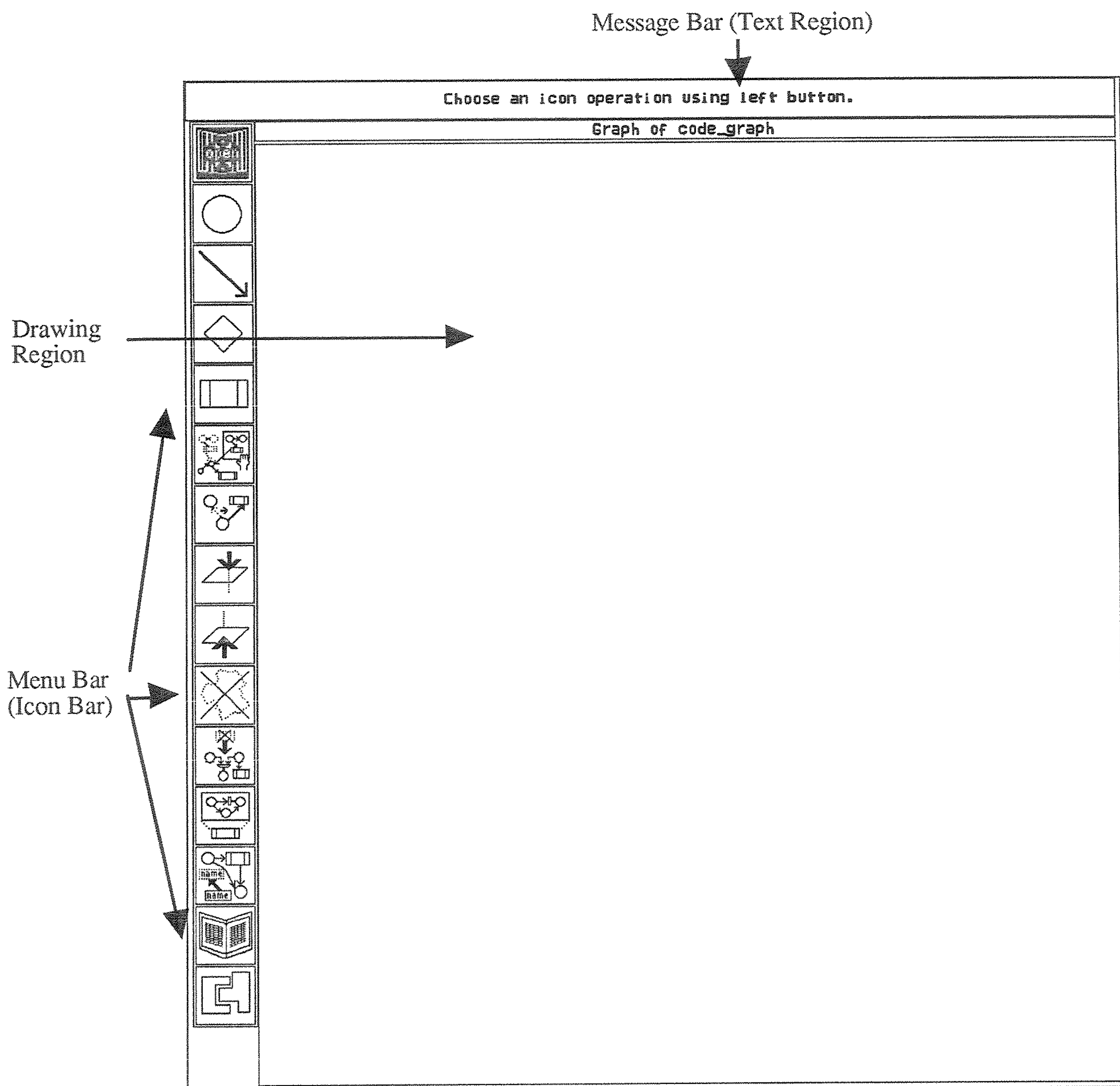


Figure 1
Parts of the CODE Window

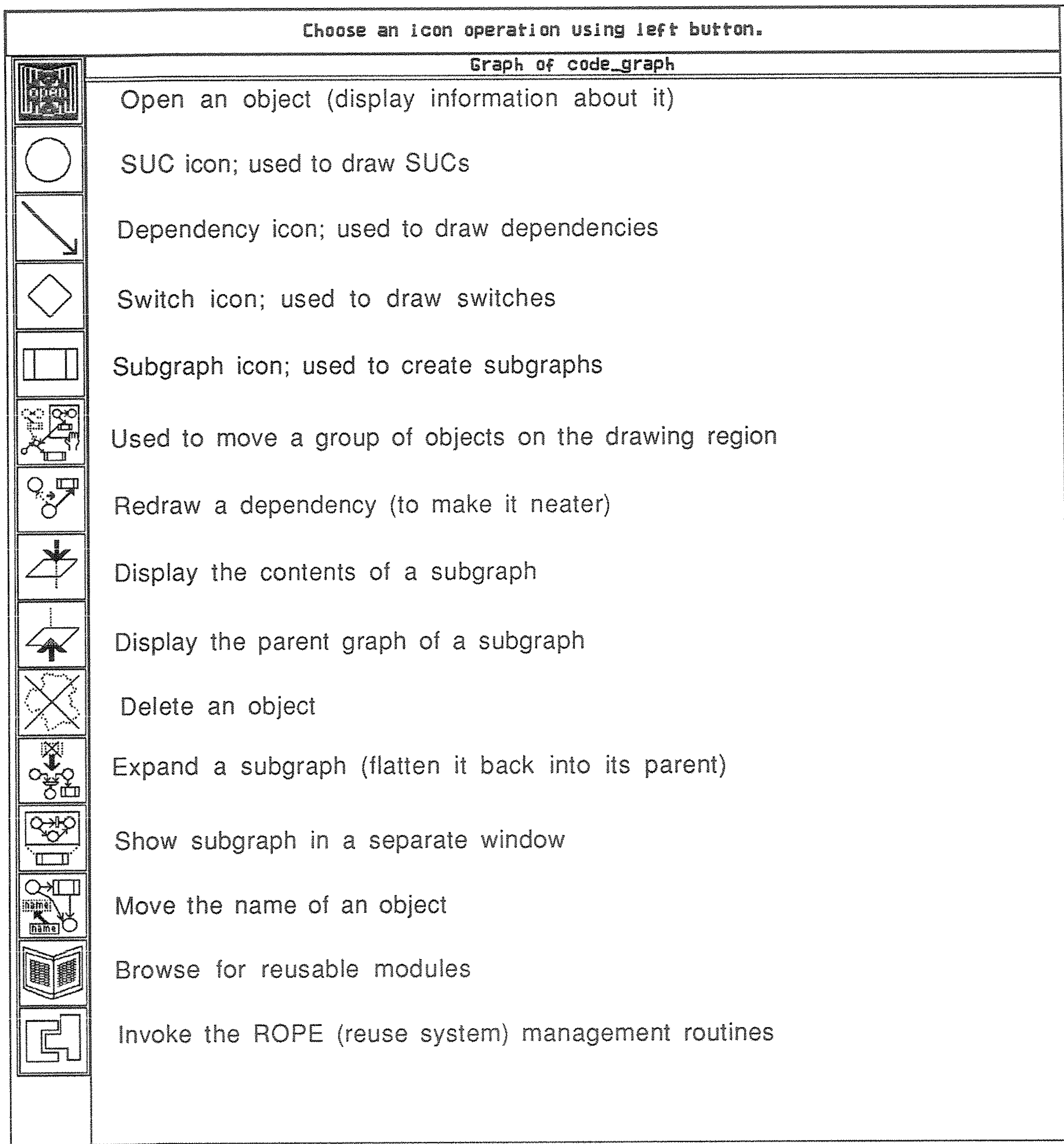


Figure 2
Annotated CODE Menu Bar

Appendix C

Command Menu (Right Mouse Button Menu)

1. Save Graph File

Save graph in a file. The user supplies the file name in response to a query.

2. Load Graph File

Load a graph from a file. The user supplies the file name in response to a query.

3. Change Rope Search Path

This is covered in the separate manual discussing the reuse capabilities of CODE

4. Make Declaration File

Translate CODE graph to an intermediate form.

5. Redisplay

Redisplay the graph.

6. Zoom In

Enlarges the graph, giving the user the impression of zooming in with a camera. The user is asked to choose the object to be displayed in the center of the drawing region after the zoom.

7. New Center

Shifts the graph in the drawing region to place the user designated object in the center of the drawing region.

8. Zoom Out

Shrinks the graph, giving the user the impression of zooming out with a camera. The user is asked to choose the object to be displayed in the center of the drawing region after the zoom.

9. Clear

Completely erases the graph in the drawing region. The saved graph (if any) is not effected.

10. Quit

Leave CODE.

Appendix D
Exclusion Constraints

1. Constraint syntax

Exclusion_Constraint	-> Mutex _Constraint mutex Share_Constraint
Mutex _Constraint	-> Mutex _Constraint mutex SUC_Node_Id
	-> SUC_Node_Id
	-> epsilon
Share_Constraint	-> Share_Constraint or SUC_Node_Id
	-> SUC_Node_Id
	-> epsilon

2. Constraint semantics

At execution time exclusion constraints are regarded as predicates whose truth must be preserved by the runtime system at all times. Since the termination of the execution of a node can never falsify these predicates because of their restricted form, this translates to a condition for allowing a node to begin execution. The condition is the following:

> From those SUCs which are enabled for execution, the runtime system checks to see which may be started without falsifying the exclusion constraint in which they participate (if any). One such node is allowed to begin execution.

The truth or falsity of a predicate is determined by the actions of an interpreter for the attribute grammar of Figure 1. Intuitively, a share constraint (or-constraint) is never false, regardless of the number of SUCs executing. It may be idle {no SUC from the share constraint executing} or behaving {some non-empty subset of SUCs from the share constraint executing}. A mutex constraint among a set of SUC nodes may be idle {no SUCs from the set executing} or behaving {at most one of the SUCs from the set executing}. The exclusion constraint formed from a mutex constraint and a share constraint is true only if at most one of the share constraint and the mutex constraint is behaving.

Associated with each constraint is a list of data items (possibly just one) and a set of SUC nodes participating in the exclusion dependency. Let S be that set of SUCs. Assume that Executing(SUC_Node_Id) is a boolean semantic function returning true if SUC_Node_Id is executing and false otherwise.

```
Exclusion_Constraint -> Mutex_Constraint  mutex  Share_Constraint
        {Exclusion_Constraint.Truth_Value :=
                If Mutex _Constraint = idle then
                        If Share_Constraint = racing then false else true
                Elseif Mutex _Constraint = behaving then
                        If Share_Constraint = idle then true else false
                Else false}

Mutex_Constraint1            -> Mutex_Constraint2 mutex  SUC_Node_Id
        {Mutex_Constraint1.Value :=
        If (Mutex_Constraint2.Value = idle) then
                If (Executing(SUC_Node_Id))) then behaving else idle
        Elseif  (Mutex_Constraint2.Value = behaving) then
                If (Executing(SUC_Node_Id))) then racing else behaving
        Else racing}

                        -> SUC_Node_Id
        {Mutex_Constraint1.Value := If Executing(SUC_Node_Id) then behaving else idle}

                        -> epsilon
        {Mutex_Constraint1.Value := idle}

Share_Constraint1       -> Share_Constraint2 or SUC_Node_Id
        {Share_Constraint1.Value := If (Share_Constraint2.Value =.behaving) or
                                    Executing(SUC_Node_Id) then behaving else idle}

                        -> SUC_Node_Id
        {Share_Constraint1.Value := If Executing(SUC_Node_Id) then behaving else idle.}

                        -> epsilon
        (Share_Constraint1.Value := idle}
```

Figure 1
Attribute Grammar Defining Truth Value of an Exclusion Constraint

Appendix E.
Developing CODE Programs for the IBM 3090

To run the prograns on the IBM 3090 the following changes need to be made:

A. The names of the variables must be limited to six or fewer characters.

B. Variable names should not contain underscores.

C. Read * should be replaced by Read n, where n is a system defined I/O unit.

Tutorial: Example 1
SUCs, Data Dependencies, and Exclusion Dependencies

The following assumes that the user has read the Introduction (Section I 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.

Contents of the tutorial

- I. Preliminaries
- II. General remarks on the interface
  - A. Windows
  - B. Input
- III. Entering Example 1
  - Step 1. Draw and name all the nodes, switches, arcs, and hyperarcs.
  - Step 2. Use forms to enter all information about the nodes and arcs.
  - Step 3. Use edit windows to enter the code located at the nodes.
- IV. Executing Example 1 on the Cray

I. Preliminaries

```
% xstart              ;Start the X-Windows System

% mkdir example1      ;Create a directory to hold the files associated with the program.

% cd example1         ;Change directory to the directory created in Step 1

                      ;Create the following README file in the directory to serve as top
                      ;level documentation for the program.

% more README         ;the following is a listing of the README file for Example 1
```

Specification:
   This program adds a vector of 100 elements and prints the sum.

Algorithm:
   First initialize a vector of 100 real numbers. Then divide the vector into two 50 element vectors and add them separately. The resulting partial sums are added to produce the sum of the elements of the original vector. Finally, print the partial sums and the final sum.

Implementation
   A SUC INIT is used to initialize a vector, V, of 100 real numbers, and then divide V into two 50 element vectors, V1 and V2. Each of V1 and V2 is then passed to its own SUC, ADD1 and ADD2 respectively, and added to produce partial sums SUM1 and SUM2. The partial sums, SUM1 and SUM2, are used by the ADD routines to update a shared variable, SUM, which at termination should contain the sum of the original vector. SUM1 and SUM2 are then passed to a final SUC, PRINTSUM, which prints them and the final sum, SUM.

```
% /projects/code/bin/xcode_hp        ;Start up the CODE system (on HPs). Consult your own
                                     ;system documentation for the proper name and directory
```

II. General remarks on the interface:

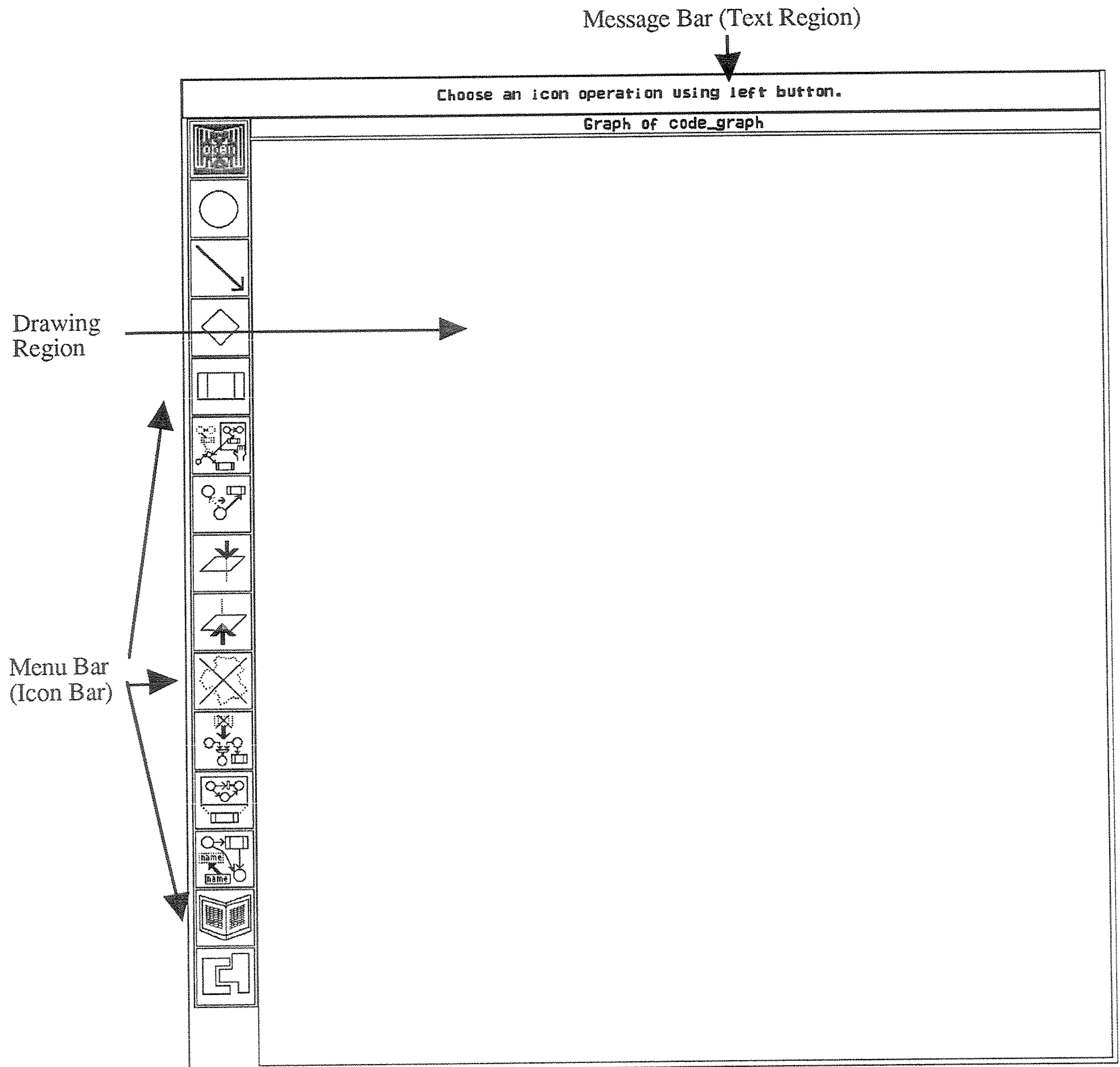
A. Windows

There are three types of windows in the system:

1. The **CODE window** used to create the top level graph
2. **Forms** used to enter information about objects in the system.
3. **Edit windows** used to enter subprograms

The CODE window (Figure 1) is divided into three regions:

i. The message (text) region at the top for messages and some text input
ii. The menu bar on the left where operations are selected with the mouse
iii. The drawing region where the CODE graph appears



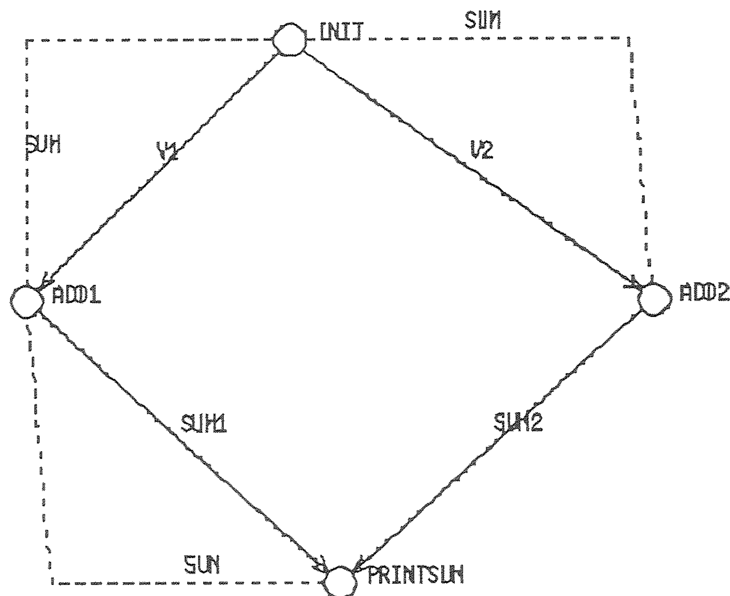
CODE Window
Figure 1

B. **Input**

1. The left mouse button is used to select objects and menu icons, to confirm actions, and to draw various graphical elements.
2. The right mouse button is used to bring up a menu of top level commands and to cancel selected actions before they are taken.
3. The keyboard is used to enter alphanumeric information in the text region, in forms and in edit windows.

III. Entering Example 1

Our goal is to draw the following graph:



Example 1
Figure 2

Users generally encounter the fewest obstacles if they enter their program in the following order:

Step 1. **Draw and name all the SUCs, switches, arcs, and hyperarcs.**
Step 2. **Use forms to enter all information about the nodes and arcs.**
Step 3. **Use edit windows to enter the code located at the nodes.**

Before we start

Saving your work as you go:

With cursor in the drawing window, the rightmost mouse button brings up a menu of high level commands. The first of these is Save Graph File. By invoking this regularly you can save your work as you go.

Correcting mistakes

a. Finish entering the node, dependency or whatever. (There are generally no provisions for aborting an operation in the middle.)

b. Select the delete icon (the dotted outline with the crossed lines) (Notice the message bar at the top of the CODE window now says "Select an object to be deleted using left button.")

c. Place the cursor on the object to be deleted and click the left button (Notice the message bar at the top of the CODE window now says "Please confirm deletion of graph object with left button". In addition, a banner through the middle of the drawing area says "CONFIRM delete with LEFT button, CANCEL with RIGHT or MIDDLE button")

d. Click left button (assuming you really wish to delete)

Step 1. Draw and name all the SUCs, switches, arcs, and hyperarcs.

A. Draw and name all SUCs and switches:

1. Draw the SUC INIT by

a. Select the SUC icon (the circle) with the mouse from the menu bar (Notice the message bar at the top of the CODE window now says "Create a SUC using left button")

b. Click the left button in the drawing region at the desired location (Notice the message bar at the top of the CODE window now says "Enter name for the new SUC:".)

c. Enter the SUC name, "INIT", followed by a Return (Warning: the name is only captured if the cursor is in the drawing region, (where the cursor is shaped like a pencil).)

2.. Draw the SUCs ADD1, ADD2, and PRINTSUM

At this point the graph looks like Figure 3.

B. Draw and name all data dependencies:

1. Draw the data dependency V1 by

a. Select the dependency icon (the arrow)with the mouse (Notice the message bar at the top of the CODE window now says "Choose the dependency starting and ending object using left button.")

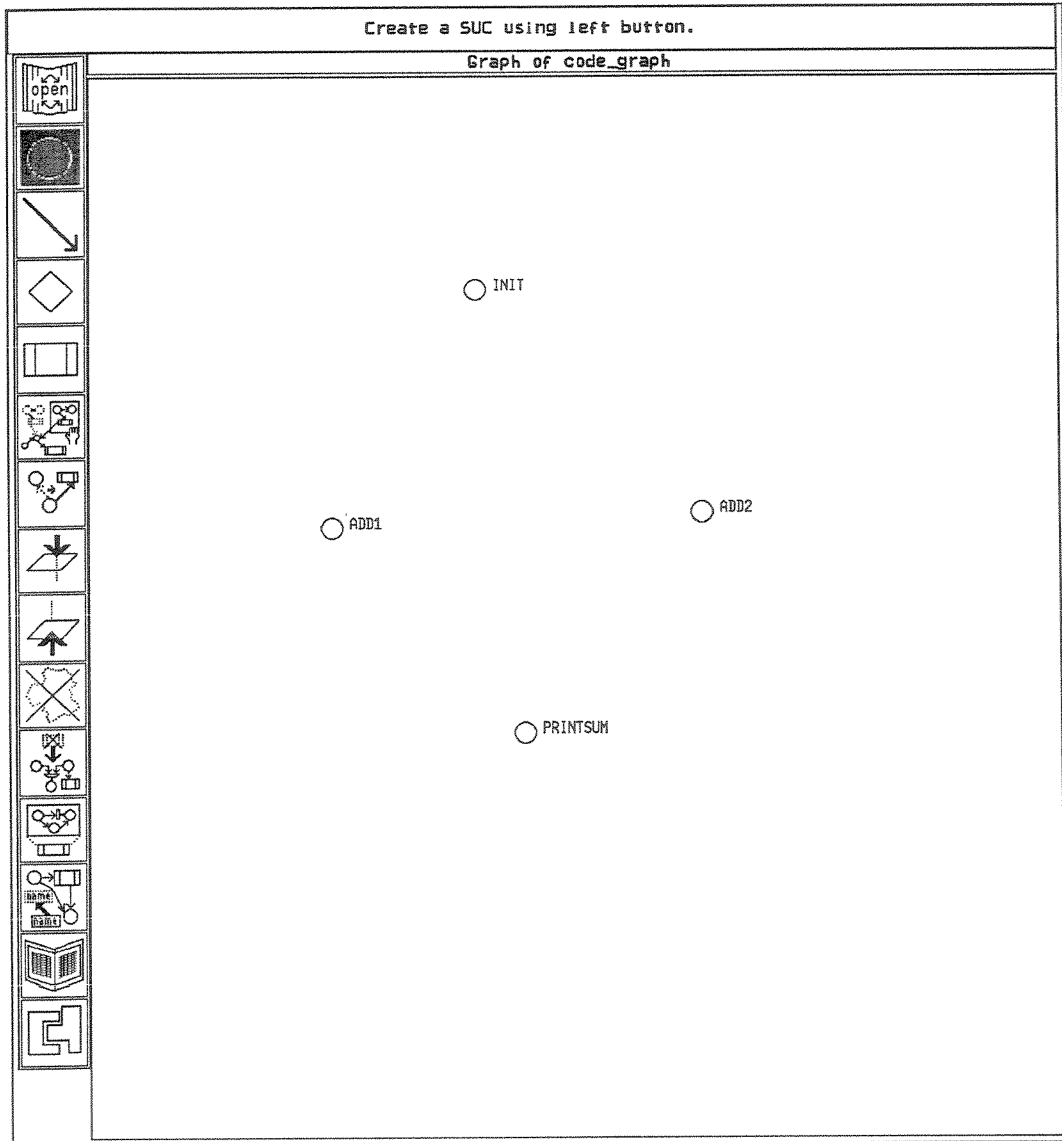
b. Place the cursor on the start node (source node) and click the left button (Notice a line is now anchored at the start node and follows the cursor)

c. Place the cursor on the end node (sink node) and click the left button (Notice the message bar at the top of the CODE window now says "Enter name for the new dependency:")

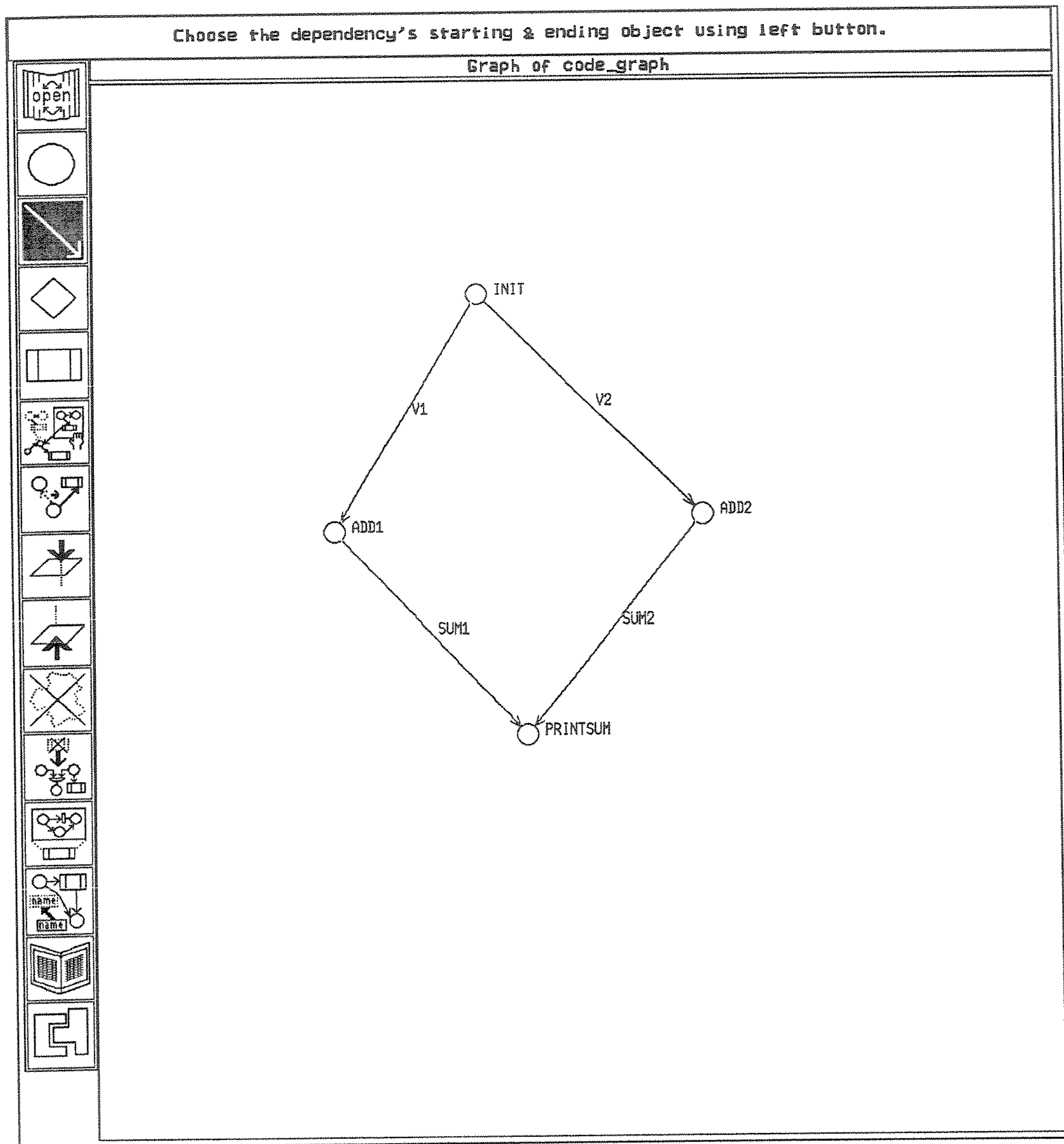
d. Entering the dependency name, "V1", and entering a Return (Warning: the name is only captured if the cursor is in the drawing region, (where the cursor is shaped like a pencil).)

2. Draw the data dependencies V2, SUM1, SUM2

At this point the graph looks like Figure 4.



Nodes for Example 1
Figure 3



Nodes and Data Dependencies
Figure 4

C. Draw and name all exclusion dependencies
(Part of this is like drawing a data dependency)

1. Draw the exclusion dependency SUM by

a. drawing a data dependency between two of the nodes participating in the exclusion dependency, say INIT and ADD1. This requires a trick to keep the picture looking nice; bring the line straight out to the left side till it is above ADD1, click the left button (this anchors the line) and then draw down to the node ADD1. This gives us a neat elbow and keeps the picture separate from the one for dependency V1.

b. Now designate this as an exclusion dependency

i. Select the Open icon (the open book at the top)
(Notice the message bar at the top of the CODE window now says "Choose an object to open using left button.")

ii. place the cursor on the dependency SUM and clicking left

iii. Place the resulting X window, called the form of dependency SUM, at some convenient location by clicking the left button. These forms are discussed in greater detail later.

iv. Use the mouse to choose the box labeled "exclusion" on the third line of the form (labeled "Kind :")
(Notice that the solid dependency line converts to dotted in the graph . The other fields of the form will be discussed later)

v. Close the form by selecting "quit" from the bottom line

Dependency: SUM

Bounds:

Kind : ☐ data ☒ exclusion ☐ replication

Input SUC/Switch :

Output SUC/Switch :

Full Start :

Full End :

Data Type : ☒ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array

Array Size:

Default:

Exclusion SUCs: ADD2, INIT, ADD1, PRINTSUM

Exclusion Constraint: ADD2 mutex INIT mutex ADD1 mutex PRINTSUM

[first] [previous] [next] [last] [add] [delete] [quit]

Form for Dependency SUM
Figure 5

c. Extend the exclusion dependency SUM to the node PRINTSUM

i. Draw a data dependency from ADD1 to PRINTSUM and name it SUM
(Notice the message bar at the top of the CODE window now says "Do you want to extend an existing exclusion dependency")
In addition, a banner through the middle of the drawing area says "CONFIRM delete with LEFT button, CANCEL with RIGHT or MIDDLE button")
ii. click left and the previously solid line becomes dotted

d. Extend the exclusion dependency SUM to the node ADD2 as in c.

At this point we have finished drawing the graph, and it looks like Figure 1.

Step 2. Use forms to enter all information about the nodes and arcs

A. Enter information about the nodes

1. Enter information about INIT

a. select the Open icon (the book)
(Notice the message bar at the top of the CODE window now says "Choose an object to open using left button.")

b. select INIT using the left button. An X window will be created; place it in some convenient place like the left corner of the screen. This is the form for INIT. It looks like Figure 6.

```
Suc:                 [INIT                                   ]

Bounds:              [                                       ]

Termination Node?:   [■] No   [ ] Yes

Input  Dependencies: ** none **

Output  Dependencies: V1 V2

Exclusion Dependencies: SUM

Code File: [INIT                                   ]

Language: [ ] Ada  [ ] C  [■] Fortran

Constraint Type?: [■] Mutex  [ ] Shared

[edit code]  [update code]  [quit]
```

Form for SUC INIT
Figure 6

Suc:	;SUC name
Bounds:	;number of identical SUCs represented by this symbol
Termination Node:	;yes or no, does execution of this node imply termination of ;the program?
Input Dependencies:	;list of input dependencies for this SUC. Information ;supplied by system.
Output Dependencies:	;list of output dependencies for this SUC. Information ;supplied by system.
Exclusion Dependencies:	;list of exclusion dependencies in which the SUC participates ;(at most 1 in CODE 1.2). Supplied by the system.
Code File	;name of file holding actual code of SUC
Language:	;choice of Ada, C, Fortran
Constraint Type?	;type of participation in an exclusion dependence if there is ;one, meaningless otherwise. Default is mutex
Form Actions	
edit code:	;enter user code into the SUC
update code:	;rewrite template. This may be invoked repeatedly as the ;graph is changed
quit:	;exit the form

Fields of a SUC Form
Figure 7

These fields (Figure 7) of a SUC form are filled with default values. The meanings of most of them are obvious. The edit code and update code Form Actions will be discussed in step 3. Information about the exclusion dependency SUM is actually indicated on this form by setting the field Constraint Type?; however in the interest of keeping concerns separated, we will set this field below. At this time we just set the Language and quit.

c. select Fortran as the language
d. close the window by selecting quit on the bottom line

2. Enter information about ADD1, ADD2 and PRINTSUM as in 1.

The one difference is that PRINTSUM must be designated as a Termination Node.

B. Enter information about the data dependencies

1. Enter information about V1 by

a. selecting the Open icon (the book)
(Notice the message bar at the top of the CODE window now says "Choose an object to open using left button")

b. selecting V1 using the left button. An X window will be created; place it in some convenient place like the left corner of the screen. This is the form for V1 (Figure 8).

These fields (Figure 9) are filled with default values. The meanings of some are obvious. Others are discussed elsewhere in the manual. We set the data type.

c. selecting int array as the Data Type
(Notice that this causes the Array Size to be set to the default value 1)

d. entering 50 as the Array Size followed by a Return. A very important point: the cursor must be in the field in order for you to type. At this point the window looks like Figure 6.

e. selecting quit to close the window

C. Enter information about the exclusion dependencies

1. Enter information about the exclusion dependency SUM

The constraint for an exclusion dependency is entered by indicating for each SUC in the dependency the nature of its participation in the dependency (see Section V for more discussion of this point). In our simple exclusion dependency the constraint is

INIT mutex ADD1 mutex ADD2 mutex PRINTSUM

which means that at most one of INIT,ADD1, ADD2 and PRINTSUM access SUM at a time. Since mutex is the default participation style of SUCs (in the interest of safety), we do not need to take any action.

The system automatically displays the exclusion constraint as in Figure 5.

Dependency: V1

Bounds:

Kind : [x] data [ ] exclusion [ ] replication

Input SUC/Switch : INIT

Output SUC/Switch : ADD1

Full Start : INIT

Full End : ADD1

Data Type : [ ] int [ ] real [ ] bool [ ] char [x] int array [ ] real array [ ] bool array [ ] char array

Array Size: 50

Default:

Exclusion SUCs:

Exclusion Constraint:

first previous next last add delete quit

Form for dependency V1
Figure 8

Dependency	;data dependency name
Bounds	;number of identical data dependencies represented by this symbol
Kind:	:data
Input SUC/Switch/Subgraph	;name of source
Output SUC/Switch/Subgraph	;name of sink
Full Start	;actual source SUC in the case source is a subgraph
Full End	;actual sink SUC in the case sink is a subgraph
DataType	;one of int, real, bool, char, or arrays of any of these
Array Size	;length of the array
Default	;exclusion dependencies only
Exclusion SUCs	;exclusion dependencies only
Exclusion Constraint	;exclusion dependencies only
Passing Conditions and Receivers	;for dependencies whose sink is a switch
Form Actions	;All but quit are used to create a list of dependencies to be associated ;with a single physical arc in the CODE graph
first	;display form for the first data item in the list
previous	;display form for the previous data item in the list
next	;display form for the next data item in the list
last	;display form for the last data item in the list
add	;add new data item to the list
delete	;delete the data item from the list whose form is currently displayed
quit:	;exit (close) the form

Fields of the Dependency Form
Figure 9

Step 3. Use edit windows to enter the code located at the nodes

A. Components of the SUC Program

The key point to understand in entering the program for a SUC is that such programs are made up of two parts:

i. a **template** generated by the system and
ii. **user written code.**

1. The Template

The template for a node generated by the system is a subprogram with parameters corresponding to the data and exclusion dependencies of the node. This subprogram is of the following form (exact syntax depends on the language):

The template is of the following form (exact syntax depends on the language):

```
Subprogram Header (with parameter list)
Declarations of parameters and shared variables
Comment Section describing parameters
Comment Section describing shared variables

        (*User Written code*)

Return and/or End Statements
```

For instance, the Fortran template for ADD1 of our example is the following:

```
      subroutine ADD1(V1,SUM1)
      integer   V1(50)
      integer   SUM1
      COMMON /SUM/SUM
c*  V1 IS INPUT ONLY
c*  SUM1 IS OUTPUT ONLY
c*  SUM IS SHARED AS COMMON BLOCK
c*   begin user-written code
c
c     user written code
c
      return
      end
```

2. The User Written Code

User written code goes between the begin statement (begin comment in the case of Fortran). An extremely important point is that the update operation will completely rewrite the area belonging to the template. Only the user written code between the Begin and the Return/End will be carried over into the new template. **Any other user written code will be deleted. So keep your code in the user area!!**

User written code for ADD1 would be something like the following:

```
          SUM1 = 0
          Do 100  I = 1, 50
                SUM1 = SUM1 + V1(I)
100       Continue
          SUM = SUM + SUM1
```

3. The Complete Node Program

The complete node program is the combination of the template and the user written code. For node ADD1 the complete program is as follows:

```
          subroutine ADD1(V1,SUM1)
          integer   V1(50)
          integer   SUM1
          COMMON /SUM/SUM
c*  V1 IS INPUT ONLY
c*  SUM1 IS OUTPUT ONLY
c*  SUM IS SHARED AS COMMON BLOCK
c*   begin user-written code
c
c   Add the elements of V1 forming the partial sum SUM1
          SUM1 = 0
          do 100  I = 1, 50
                SUM1 = SUM1 + V1(I)
100       continue
c
c   Update the shared variable SUM
          SUM = SUM + SUM1
c
          return
          end
```

B. Creating the Components

Each of these components of the complete node program has a Form Action corresponding to it:

i. **update code** - (re)generates the template to match the graph
ii. **edit code** - allows user to enter code

1. update code

This Form Action causes the template to be (re)written to match the graph. The system **does not** automatically maintain consistency between the graph and the node program. Whenever the dependencies involving a node are changed, this Action must be invoked. The key point is that any user written code which appears outside the allocated user area will be deleted when this Action is invoked.

2. edit code

This Form Action invokes the user's default editor on the complete node program. Note that this means the user can edit the template portion of the program, **a very dangerous action!**

C. Creating the node programs for Example 1

1. Creating the node program for ADD1

    a. open the node ADD1

    b. select **update code** from the Form Actions. This will generate the template.

    c. select **edit code** from the Form Actions. This will create an edit window containing the template. Enter the user code from the previous page. At this point the complete node program should look like the one on the previous page.

    d. close the edit window by issuing the editor quit command

    e. close the ADD1 form by selecting quit

2.. Create the node programs for INIT, ADD2, and PRINTSUM in the same way. Their listings follow:

```
      subroutine  INIT(V1,V2)
      integer   V1(50)
      integer   V2(50)
      COMMON /SUM/SUM
C*  V1 IS OUTPUT ONLY
C*  V2 IS OUTPUT ONLY
C*  SUM IS SHARED AS COMMON BLOCK
C*   begin user-written code
C
      integer   V(100)
C
C   Initialize V(100) to the first 100 integers
      do 100 I = 1, 100
            V(I) = I
100   continue
C
C  Partition V into V1 and V2
      do 200 I = 1, 50
            V1(I) = V(I)
            V2(I) = V(50 +I)
200   continue

C  Initialize the shared variable SUM
      Sum = 0
C
      return
      end
```

```
      subroutine  ADD2(V2,SUM2)
      integer   V2(50)
      integer   SUM2
      COMMON /SUM/SUM
C*  V2 IS INPUT ONLY
C*  SUM2 IS OUTPUT ONLY
C*  SUM IS SHARED AS COMMON BLOCK
C*   begin user-written code
C
C   Add the elements of V2 forming the partial sum SUM2
      SUM2 = 0
      do 100  I = 1, 50
             SUM2 = SUM2 + V2(I)
100   continue
C
C   Update the shared variable SUM
      SUM = SUM + SUM2
C
      return
      end


      subroutine  PRINTSUM(SUM1,SUM2)
      integer   SUM1
      integer   SUM2
      COMMON /SUM/SUM
C*  SUM1 IS INPUT ONLY
C*  SUM2 IS INPUT ONLY
C*  SUM IS SHARED AS COMMON BLOCK
C*   begin user-written code
C
C  Print the values
      Print *, Sum1, Sum2, Sum
C
      return
      end
```

IV. Executing Example 1 on the Cray

We now take the first step in the compiling and executing of the CODE graph by creating an intermediate text form of the program. This intermediate form is called the declarations file.

A. Creating the Declarations File

1. Press the right mouse button to get the COMMANDS menu. Choose 'Make Declaration File'. The message bar displays 'File to contain declarations". This is a request for a file name for the declarations file. Enter some standard name like 'decl' followed by a Return. The message bar now reads 'Declarations saved in file decl'.

2. The file decl may be viewed with the usual Unix utilities if you are curious.

B. Creating the Source File

In a separate Unix window (or after exiting CODE) execute the command

toad_cft+77 < decl ('cft' stands for 'cray fortran')

This produces a Fortran source program crayf77.f, which is ready to be compiled and executed.

C. Executing Example 1 on the Cray (using UNICOS)

A. Move the file crayf77.f to the Cray

B. Compile and execute with the following commands:

```
cft+77 -a stack crayf77.f    ;compile to produce object crayf77.o
segldr crayf77.o             ;link to create executable a.out
a.out                        ;execute program
```

Tutorial: Example 2
Subgraphs

The following assumes a basic familiarity with graphical interfaces which employ mice and menus to communicate with users, with X windows, and with Unix commands and editors. It also assumes that the user has finished Tutorial Example 1.

Contents of the tutorial

I. Preliminaries
II. Entering Example 2
  Step 1. Draw and name all the nodes, switches, arcs, and hyperarcs.
  Step 2. Use forms to enter all information about the nodes and arcs
  Step 3. Use edit windows to enter the code located at the nodes.
III. Executing Example 2 on the Cray

I. Preliminaries

%mkdir example2 ;Create a directory to hold the files associated with the program.

%cd example2 ;Change directory to the directory created in Step 1

;Create the following README file in the directory to serve as top
;level documentation for the program.

%more README ;the following is a listing of the README file

Specification:
Read the integer N, then N times do the following:
initialize a vector, add it, and print its sum.

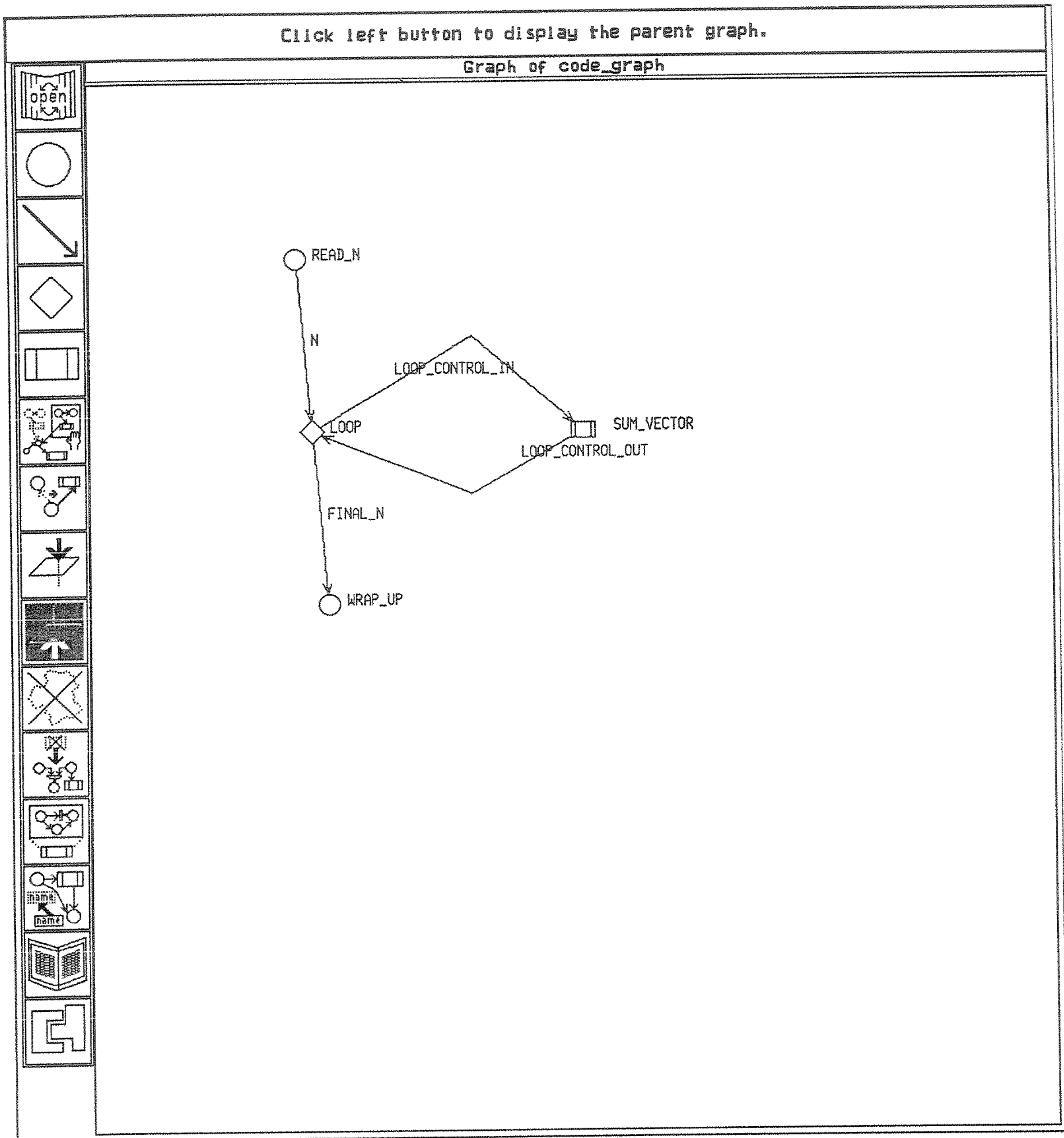
Algorithm:
Read the variable N. If it is positive, then loop repeatedly through the algorithm of Example 1 until all vectors have been initialized, added and values printed. If N is not positive, then quit.

Implementation
Use a switch node and variables Loop_Control_In and Loop_Control_Out to handle the looping. Follow the implementation of Example 1.

%/projects/code/bin/xcode_hp ;Start up the CODE system (on HPs)
;Place the resulting X window (Figure 1) to the right
;side of the screen.

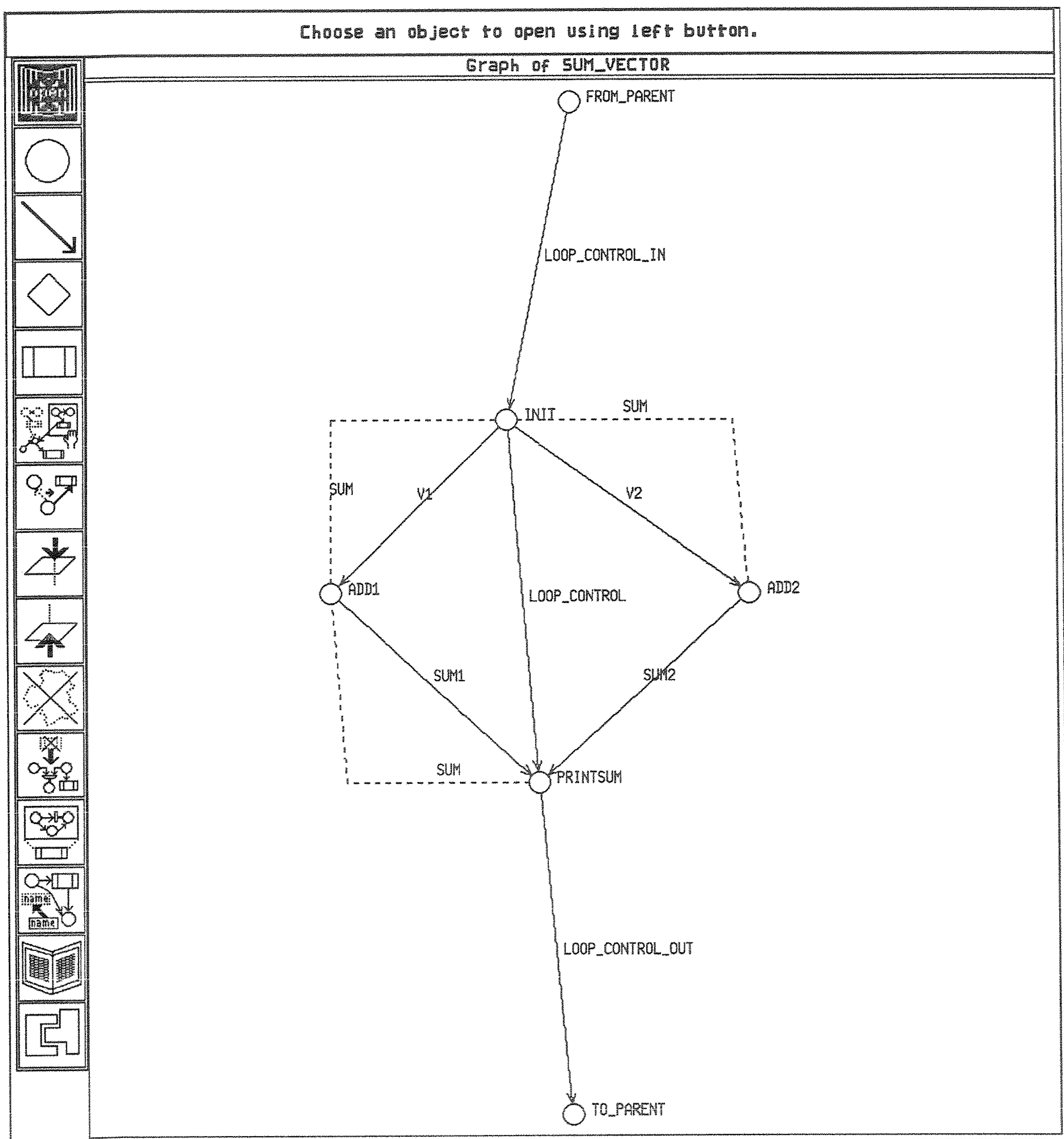
III. Entering Example 2

Our goal is to draw the graph of Figure 1.



Example 2 Top Level Graph
Figure 1

The subgraph SUM_VECTOR (Figure 2) is essentially the graph of Example 1.



Example 2 Subgraph SUM_VECTOR
Figure 2

Step 1. Draw and name all the nodes, switches, arcs, hyperarcs, and subgraphs.

A. Entering the subgraph

There are two ways to do this: either load the graph for Example 1 and edit it or enter the entire graph from scratch. We will use the first method since it gives us a chance to try some different commands.

1. Copy the following files from the directory for Example 1 to the directory for Example 2 using the normal Unix cp command.

gph	graph file,
ADD1	node program for node ADD1
ADD2	node program for node ADD2
INIT	node program for node INIT
PRINTSUM	node program for node PRINTSUM

2. Load the graph for Example 1

With the cursor in the drawing region, press the rightmost mouse button. From the resulting menu choose Load Graph File. This causes the message "File to be read:" to be displayed. Enter "gph" as the file to be read. Remember that the cursor must be in the drawing region, and that you must follow the name with a return.

3. Add the data dependency LOOP_CONTROL to the graph just as in Example 1. Its source is INIT, and its sink is PRINTSUM. This is an integer variable. The graph now looks like Figure 2 *EXCEPT* the nodes TO_PARENT and FROM_PARENT and the dependencies LOOP_CONTROL_IN and LOOP_CONTROL_OUT are missing. These will be inserted later when the subgraph is created and connected to the top-level graph.

B. Forming the subgraph

1. Choose the subgraph icon from the menu bar. The message area reads "Draw a box around objects to be included in a subgraph using left button"

2. In the usual window interface style, hold down the left mouse button and draw a box around the graph. When you have the entire graph surrounded, click the left button once and the graph will disappear to be replaced by a subgraph symbol. The message bar reads "Enter name for the new object".

One unpleasant thing that can happen is that you may start at a point that makes it impossible to include the entire graph. Unfortunately once you begin the process, a subgraph will be formed even if it winds up containing the wrong part of the graph (or is even empty). If this happens, then the subgraph may be re-expanded using the fifth icon from the bottom in the menu bar. In that way you can try again. Another alternative is to clear the drawing region using the rightmost mouse button menu and then loading the graph again.

3. Enter the name SUM_VECTOR for the subgraph

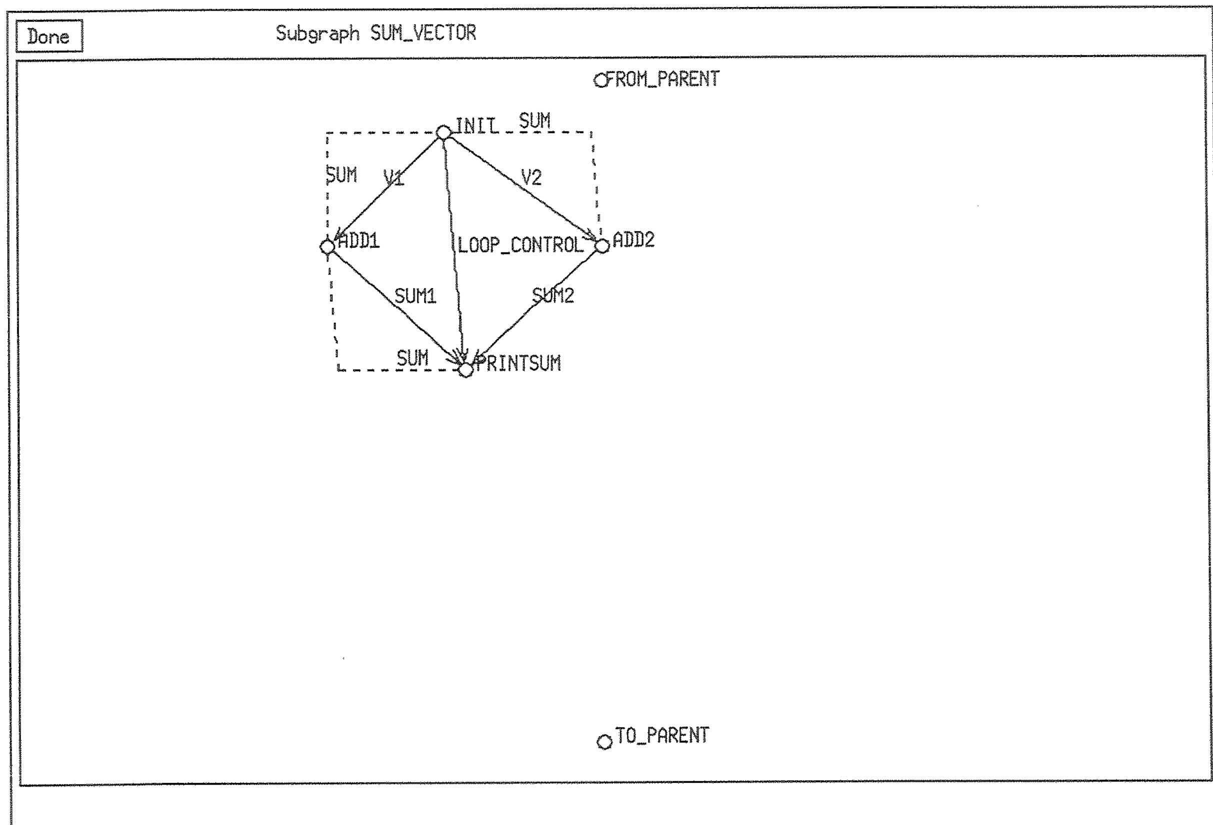


Figure 3
Display Window for Subgraph SUM_VECTOR

C. Displaying the subgraph

There are three ways to display a subgraph.

1. Display the subgraph in a separate window (Figure 3) . This window is read only and is invoked by the fourth icon from the bottom. You may continue to edit the top-level graph while viewing the subgraph. Close the window for the subgraph by selecting "done" in the upper left corner of the subgraph window.

2. Display the subgraph and edit it. This is invoked by icon number 8 (counting from the top). Return to the top level graph by invoking icon 9, and then clicking the left mouse button in the drawing region.

3. Open the subgraph. This is invoked by icon 1. The form for the subgraph is then displayed as in Figure 4. One may then choose the Form Action "display graph" to open the subgraph for editing. Return to the top level graph by invoking icon 9, and then clicking the left mouse button in the drawing region.

a. Try each of the above methods for displaying the subgraph, each time returning to the top level. Notice that the system has now automatically inserted the nodes TO_PARENT and FROM_PARENT in the subgraph. The dependencies LOOP_CONTROL_IN and LOOP_CONTROL_OUT will be inserted later when we connect the subgraph to the main graph.

```
Subgraph: SUM_VECTOR

Parent Graph: code_graph

SUCs & Switches:INIT, ADD1, ADD2, PRINTSUM

Subgraphs    : ** none **

Dependencies: LOOP_CONTROL_IN, LOOP_CONTROL, LOOP_CONTROL_OUT, SUM, SUM2, SUM1, V2, V1

[read files]  [display]  [quit]
```

Figure 4
Form for Subgraph SUM_VECTOR

D. Complete the top level graph

We use our usual methodology of first drawing and naming everything, then inserting information in the forms, and finally writing the code for the nodes.

1. Draw and name the SUCs READ_N and WRAP_UP

2. Draw and name the switch LOOP. This is just like drawing a SUC but you start with the switch icon

3. Draw the dependencies N and FINAL_N

4. Connect the subgraph

   a. Draw the dependency LOOP_CONTROL_IN.

When a dependency ends at a subgraph, the message "Do you wish to end or to continue?" appears in the message bar, and the message "END with LEFT button, CONTINUE in other graph with RIGHT (or MIDDLE) button" appears across the drawing pane. We choose to Continue by pressing the right button. This brings up the subgraph in the window, and the dependency arc continues from the node FROM_PARENT. Terminate the dependency at INIT. The drawing region reverts back to the parent graph and we name the dependency as usual.

b. Draw the dependency LOOP_CONTROL_OUT.

a. Display the subgraph for editing.

b. Draw a dependency from PRINTSUM to the node TO_PARENT. When a dependency in a subgraph ends at the node TO_PARENT, the message "Do you wish to end or to continue?" appears in the message bar and the message "END with LEFT button, CONTINUE in other graph with RIGHT (or MIDDLE) button" appears across the drawing region.. We choose to Continue by pressing the right button. This brings up the parent graph (the top-level graph in this case), and the dependency arc continues from the subgraph node, SUM_VECTOR. Complete the dependency by using LOOP as its sink. The drawing region reverts back to the subgraph, and we name the dependency as usual.

Step 2. Use forms to enter all information about the nodes and arcs

Most information about the subgraph is already correct; there are just a few things to change.

A. Enter information about READ_N, WRAP_UP, and PRINTSUM

All are Fortran routines and WRAP_UP (but not READ_N) is a Termination Node. The node PRINTSUM in the subgraph is no longer a Termination Node. All these changes require opening the subgraph for editing, if it is not already open.

B. Enter information about the Switch node LOOP

Open the form (Figure 5) for LOOP. There are no changes that need to be made.

```
Switch:          LOOP

Bounds:

Input  Dependencies: N LOOP_CONTROL_OUT

Output  Dependencies: FINAL_N LOOP_CONTROL_IN

[quit]
```

Figure 5
Form for Switch Node LOOP

C. Enter data type information about the dependencies N, FINAL_N, LOOP_CONTROL, LOOP_CONTROL_IN and LOOP_CONTROL_OUT.

These are all integer variables (the default, so no changes need to be made).

D. Enter **Passing Conditions & Receivers** for the dependency N (Figure 6)

1. Open the form
2. With the cursor positioned on the line for **Passing Conditions & Receivers** enter the following condition:

   N > 0 | LOOP_CONTROL_IN, N<= 0 | FINAL_N

E. Enter **Passing Conditions & Receivers** for the dependency LOOP_CONTROL_OUT (Figure 7)

1. Open the form
2. With the cursor positioned on the line for **Passing Conditions & Receivers** enter the following condition:

   LOOP_CONTROL_OUT = 0 | FINAL_N,
   LOOP_CONTROL_OUT > 0 | LOOP_CONTROL_IN

Dependency: LOOP_CONTROL_OUT

Bounds:

Kind : ☒ data ☐ exclusion ☐ replication

Input subgraph : SUM_VECTOR

Output SUC/Switch : LOOP

Full Start : SUM_VECTOR/PRINTSUM

Full End : LOOP

Data Type : ☒ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array

Array Size:

Default:

Exclusion SUCs:

Exclusion Constraint:

Passing conditions & Receivers: LOOP_CONTROL_OUT = 0 | FINAL_N, LOOP_CONTROL_OUT > 0 | LOOP_CONTROL_IN

[first] [previous] [next] [last] [add] [delete] [quit]

Figure 6
Form for Data Dependency N

Dependency: LOOP_CONTROL_OUT

Bounds:

Kind : ■ data ☐ exclusion ☐ replication

Input subgraph : SUM_VECTOR

Output SUC/Switch : LOOP

Full Start : SUM_VECTOR/PRINTSUM

Full End : LOOP

Data Type : ■ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array

Array Size:

Default:

Exclusion SUCs:

Exclusion Constraint:

Passing conditions & Receivers: LOOP_CONTROL_OUT = 0 | FINAL_N, LOOP_CONTROL_OUT > 0 | LOOP_CONTROL_IN

first previous next last add delete quit

Figure 7
Form for Data Dependency LOOP_CONTROL_OUT

Step 3. Use edit windows to enter the code located at the nodes

We need to create SUC node programs for the nodes of the subgraph (INIT, ADD1, ADD2, PRINTSUM) and for the nodes of the top level graph (READ_N, WRAPUP). We do this just as in Tutorial Example 1. Remember that the code is made up of two parts, the template and the user written code. As we discussed in Tutorial Example 1, there are two different Form Actions corresponding to these two different parts of the node program.

i. **update code** - (re)generates the template to match the graph
ii. **edit code** - allows user to enter code

1. Update Code

This Form Action causes the template to be (re)written to match the graph. The system **does not** automatically maintain consistency between the graph and the node program. Whenever the dependencies are changed at a node, this Action must be invoked. The key point is that any user written code which appears outside the allocated user area will be deleted when this action is invoked.

2. Edit Code

This Form Action invokes the unix editor vi on the complete node program. Note that this means the user can edit the template portion of the program, **a very dangerous action!**

A. Subgraph nodes (INIT, ADD1, ADD2, PRINTSUM)

The programs for the subgraph nodes are similar to those of the graph of Example 1 but not identical. The node names are the same, and all of the old dependencies are present and unchanged. However, there are new dependencies (LOOP_CONTROL, LOOP_CONTROL_IN and LOOP_CONTROL_OUT), and some code changes at the nodes INIT and PRINTSUM. All of the changes needed are made in INIT and PRINTSUM.

1. INIT
   a. Open the form for INIT
   b. Choose **update code** from the Form Actions. This will generate a new template to account for the new dependencies LOOP_CONTROL, and LOOP_CONTROL_IN incident on INIT
   c. Choose **edit code** from the Form Actions. Edit the code for INIT to read as follows:

```
      subroutine INIT(LOOP_CONTROL_IN, V1,V2,
+     LOOP_CONTROL)
      integer LOOP_CONTROL_IN
      integer  V1(50)
      integer  V2(50)
      integer LOOP_CONTROL
      COMMON /SUM/SUM
C*  LOOP_CONTROL_IN is input only
C*  V1 IS OUTPUT ONLY
C*  V2 IS OUTPUT ONLY
C*  LOOP_CONTROL is output only
C*  SUM IS SHARED AS COMMON BLOCK
C*  begin user-written code
C
      integer  V(100)
C  Initialize V
      do 100 I = 1, 100
            V(I) = I + LOOP_CONTROL_IN
100   continue
C
C  Partition V into V1 and V2
      do 200 I = 1, 50
            V1(I) = V(I)
            V2(I) = V(50 +I)
200   continue
C
C  Initialize the shared variable SUM
      Sum = 0
C
C     Set Loop_Control to Loop_Control_In
      LOOP_CONTROL = LOOP_CONTROL_IN
      return
      end
```

2. PRINTSUM

a. Open the form for INIT

b. Choose **update code** from the Form Actions. This will generate a new template to account for the new dependencies LOOP_CONTROL and LOOP_CONTROL_OUT incident on PRINTSUM

c. Choose **edit code** from the Form Actions. Edit the code for PRINTSUM to read as follows:

```
      subroutine  PRINTSUM(SUM1,SUM2,  LOOP_CONTROL,
+      LOOP_CONTROL_OUT)

        integer   SUM1
        integer   SUM2
        integer   LOOP_CONTROL
        integer   LOOP_CONTROL_OUT
        COMMON /SUM/SUM
C*  SUM1 IS INPUT ONLY
C*  SUM2 IS INPUT ONLY
C*   LOOP_CONTROL is input only
C*  LOOP_CONTROL_OUT is output only
C*  SUM IS SHARED AS COMMON BLOCK
C*   begin user-written code
C
C  Print the values
        Print *, LOOP_CONTROL, SUM1, SUM2, SUM
C
C  Decrement LOOP_CONTROL
        LOOP_CONTROL_OUT = LOOP_CONTROL - 1
C
        return
        end
```

B. Top Level Graph Nodes (READ_N, WRAP_UP)

These SUC node programs are entered as in Tutorial Example 1 and the subgraph SUM_VECTOR above.

1. READ_N

```
      subroutine READ_N(N)
        integer   N
c*  N IS OUTPUT ONLY
c*   begin user-written code

C  Read N, the number of vectors to be read
        read *, N
        return
        end
```

2. WRAP_UP

```
subroutine WRAP_UP(FINAL_N)
        integer   FINAL_N
c*  FINAL_N IS INPUT ONLY
c*  begin user-written code
C
C   Print closing message
C
        If (FINAL_N .EQ. 0) then
                Print *, "Computation successful"
        Else
                Print *, "Bad value of N"
        Endif
        return
        end
```

IV. Executing Example 1 on the Cray

This is just like Example 1. We take the first step in the compiling and executing of the CODE graph by creating an intermediate text form of the program. This intermediate form is called the declarations file.

A. Creating the Declarations File

1. Press the right mouse button to get the COMMANDS menu. Choose 'Make Declaration File'. The message bar displays 'File to contain declarations". This is a request for a file name for the declarations file. Enter some standard name like 'decl' followed by a Return. The message bar now reads 'Declarations saved in file decl'.

2. The file decl may be viewed with the usual Unix utilities if you are curious.

B. Creating the Source File

In a separate Unix window (or after exiting CODE) execute the command

```
toad_cft+77 < decl      ('cft' stands for 'cray fortran')
```

This produces a Fortran source program crayf77.f, which is ready to be compiled and executed.

C. Executing Example 1 on the Cray.

A. Move the file crayf77.f to the Cray

B. Compile and execute with the following commands:

```
cft+77 -a stack crayf77.f      ;compile to produce object crayf77.o

segldr crayf77.o               ;link to create executable a.out

a.out                          ;execute program
```

Tutorial: Examples 3
Statically Replicated Objects

The following assumes that the user has finished Tutorial Examples 1 and 2 and has at least skimmed the entire manual.. On the assumption that the user has mastered the basics of using CODE, only those parts of the examples representing new ideas are discussed in detail.

This example is an altered version of Example 1. The shared variable SUM is eliminated. There are now ten adders instead of two and each adder prints its number and a message when it completes. Replication is used to produce the 10 adders. The requirement that an adder "know its number" forces us to associate more than one data dependency with a single arc.

I. Preliminaries

These are the same in nature as those for Examples 1 and 2.

II. Entering Example 2

Our goal is to draw the graph of Figure 1. We do this as follows:

A. Draw and name the nodes INIT, ADD, and PRINTSUM. In the form for ADD set the Bounds field to 10. See Figure 2. Remember that PRINTSUM is a Termination Node.

B. Draw and name data dependencies V_IN and SUM_OUT. Note that the system sets the Bounds field of SUM_OUT to 10 automatically, but the Bounds field of V_IN must be set to 10 by you. See Figures 4 and 5.

C. The arc labeled V_IN[10] is heavier than normal because it carries more than one dependency. To enter the second dependency {ADDER_NUMBER which is the identifying number of the copy of ADD which receives it} open the form for V_IN and use the add form action to add another dependency to the arc as described in Section III of the manual. See Figure 6.

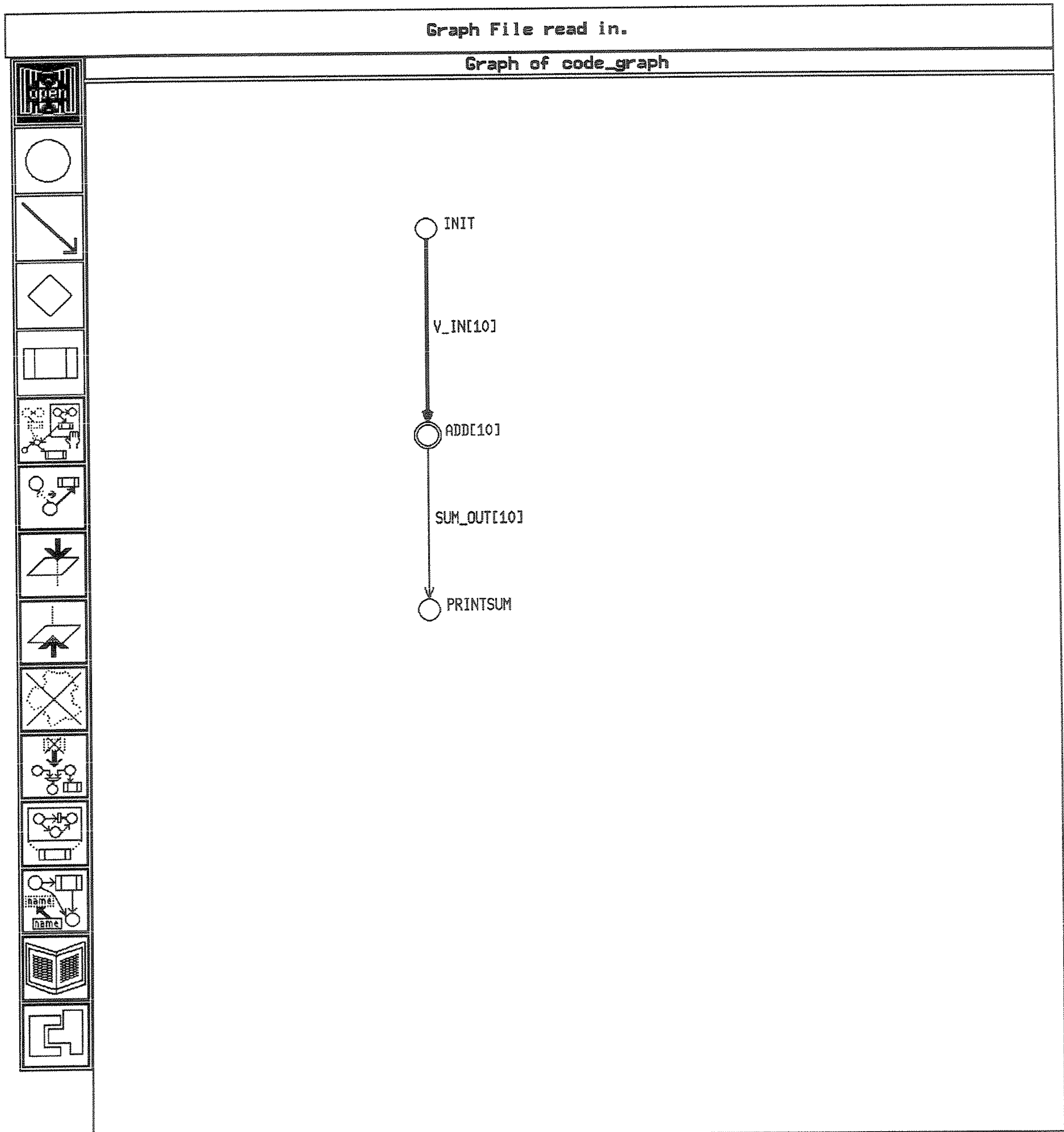


Figure 1
Graph of Example 3.1

Multiple Suc: ADD

Bounds: 10

Termination Node?: [x] No [ ] Yes

Input Dependencies: V_IN ADDER_NUMBER

Output Dependencies: SUM_OUT

Exclusion Dependencies: ** none **

Code File: ADD

Language: [ ] Ada [ ] C [x] Fortran

Constraint Type?: [x] Mutex [ ] Shared

[edit code] [update code] [quit]

Figure 3
Form for the Replicated SUC ADD

Multiple Dependency: V_IN

Bounds: 10

Kind : [x] data [ ] exclusion [ ] replication

Input SUC/Switch : INIT

Output SUC/Switch : ADD[10]

Full Start : INIT

Full End : ADD

Data Type : [ ] int [ ] real [ ] bool [ ] char [x] int array [ ] real array [ ] bool array [ ] char array

Array Size: 10

Default:

Exclusion SUCs:

Exclusion Constraint:

[first] [previous] [next] [last] [add] [delete] [quit]

Figure 4
Form for Data Dependency V_IN

Multiple Dependency: SUM_OUT

Bounds: 10

Kind : ☒ data ☐ exclusion ☐ replication

Input SUC/Switch : ADD[10]

Output SUC/Switch : PRINTSUM

Full Start : ADD

Full End : PRINTSUM

Data Type : ☒ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array

Array Size:

Default:

Exclusion SUCs:

Exclusion Constraint:

[first] [previous] [next] [last] [add] [delete] [quit]

Figure 5
Form for Data Dependency SUM_OUT

Multiple Dependency: ADDER_NUMBER

Bounds: 10

Kind : ☒ data ☐ exclusion ☐ replication

Input SUC/Switch : INIT

Output SUC/Switch : ADD[10]

Full Start : INIT

Full End : ADD

Data Type : ☒ int ☐ real ☐ bool ☐ char ☐ int array ☐ real array ☐ bool array ☐ char array

Array Size:

Default:

Exclusion SUCs:

Exclusion Constraint:

[first] [previous] [next] [last] [add] [delete] [quit]

Figure 6
Form for Data Dependency ADDER_NUMBER

III. SUC Code

A. INIT

```
        subroutine INIT(V_IN,ADDER_NUMBER)
        integer   V_IN(10, 10)
        integer   ADDER_NUMBER(10)
c* V_IN IS OUTPUT ONLY
c* ADDER_NUMBER IS OUTPUT ONLY
c* begin user-written code
C
        integer V(100)
C
C  Initialize V
        do 100 I = 1, 100
                V(I) = I
100     continue
C
C  Partition V into V_IN(1,1..10) to V_IN(10, 1..10)
        do 300 I = 1, 10
        do 200 J = 1, 10
                V_IN(I,J) = V(10*(I-1) + J)
200     continue
300     continue
C
C  Initialize ADDER_NUMBER
        do 400 I = 1, 10
                ADDER_NUMBER(I) = I
400     continue
C
        return
        end
```

B. ADD

```
        subroutine ADD(V_IN,ADDER_NUMBER,SUM_OUT)
        integer   V_IN(10)
        integer   ADDER_NUMBER
        integer   SUM_OUT
c* V_IN IS INPUT ONLY
c* ADDER_NUMBER IS INPUT ONLY
c* SUM_OUT IS OUTPUT ONLY
c*  begin user-written code
C
C  Add the elements of V_IN forming the partial sum SUM_OUT
        SUM_OUT = 0
        do 100 I = 1,10
                SUM_OUT = SUM_OUT + V_IN(I)
100     continue
C
C  Print a message
        Print *, ADDER_NUMBER, "finishing"
C
        return
        end
```

C. PRINTSUM

```
      subroutine PRINTSUM(SUM_OUT)
      integer   SUM_OUT(10)
c*  SUM_OUT IS INPUT ONLY
c*  begin user-written code
      integer SUM
C
C  Initialize the value of SUM
      SUM = 0
C  Print the values for each partial sum and accumulate
      do 100 I = 1, 10
            Print*, I, SUM_OUT(I)
            SUM=SUM+SUM_OUT(I)
100   continue
C
C  Print the final sum
      Print *, SUM
C
      return
      end
```