CODE 2.0 Language Reference

This document defines the CODE 2.0 graphical parallel programming language. It is intended to supplement the CODE 2.0 User Manual with a concise definition of all CODE graphical objects and their attributes.

If you know nothing about CODE, this is not the place to begin reading. The CODE 2.0 User Manual contains introductory material. Readers of this reference are assumed to be familiar with CODE's fundamentals and with general programming language concepts and terminology commonly used in describing programming languages.

1.0 Overview

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

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

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

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

Formal parameters are defined using special nodes in the called graph, Interface nodes and Creation Parameter nodes.

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

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

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

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

(O) - Optional attribute. It is possible to create a correct CODE program without supplying a value.

(R) - Required attribute. You must supply a value.

2.0 Programs and their Attributes

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

-- Program Attributes --

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

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

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

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

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

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

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

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

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

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

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

3.0 Graphs and their Attributes

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

banner button.

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

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

-- Graph Attributes --

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

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

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

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

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

4.0 Interface Nodes and their Attributes

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

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

-- Interface Node Attributes --

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

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

5.0 Creation Parameter Nodes and their Attributes

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

-- Creation Parameter Node Attributes --

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

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

6.0 Call Nodes and their Attributes

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

-- Call Node Attributes --

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

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

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

7.0 Unit of Computation (UC) Nodes and their Attributes

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

-- UC Node Attributes --

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

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

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

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

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

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

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

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

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

8.0 Name Sharing Relation Nodes and their Attributes

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

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

-- Name Sharing Node Attributes --

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

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

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

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

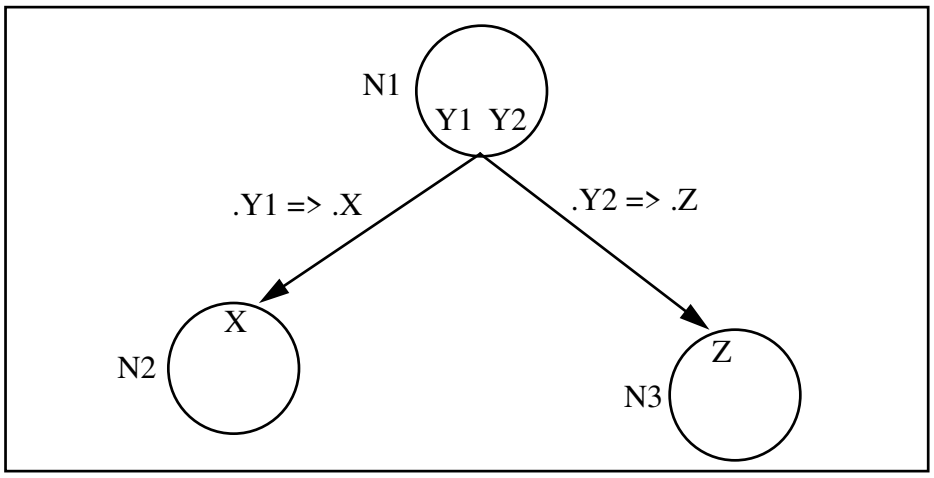
```
shared_vars { // These variables are shared.
      Matrix1 m[10];
      int MyInt;
}
vars { // Local vars for use by init_comp.
      int i;
}
init_comp { // Run when node is created.
      i = 0;
      while (i < 10) {
             m[i] = 2.5;
             i = i + 1;
      }
}
```

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

9.0 Arcs and their Attributes

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

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



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

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

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

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

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

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

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

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

The following are legal, although unusual.

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

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

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

- **UC node to UC node**

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

- **UC node to Call node**

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

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

- **Call node to UC node**

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

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

- **Call node to Call node**

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

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

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

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

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

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

• Input Interface node to UC node

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

• UC node to Output Interface node

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

• Input interface node to Call node

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

• Call node to output interface node

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

• Input interface node to Name Sharing Relation node

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

• Input Interface node to Output Interface node

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

10.0 Identifiers and Constants

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

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

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

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

11.0 Predeclared Identifiers

There are several identifiers that are predeclared in CODE.

TRUE The integer value 1.

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

12.0 Comments

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

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

13.0 Scope in CODE

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

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

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

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

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

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

14.0 Types in CODE

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

Predefined Types

int
real
char

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

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

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

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

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

<array_type_spec> := array of OLDTYPE ;

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

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

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

Examples of Legal Type Definitions

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

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

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

type PolyLine is array of Point;

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

CODE is a statically typed language and is type-safe with the exception of ellipses (...) in function signatures. Despite the fact that all types must be named before that can be used, CODE uses type

structure (rather than name) to determine when types are equivalent. For example, if we define the following types:

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

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

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

Examples of Illegal Type Definitions

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

15.0 Expressions and Operators

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

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

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

char => int => real

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

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

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

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

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

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

16.0 Statements

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

The syntax of CODE statements is as follows.

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

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

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

<null_stmt> := ;
```

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

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

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

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

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

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

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

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

Here are some example CODE statements.

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

17.0 Functions and Function Calling

Sequential functions (and procedures but we will often just use the word function) are very much a part of CODE even though you cannot define them within it. Rather, they are written in a regular

sequential language, most often C. It is expected that most significant UC computations will be specified using such external sequential functions. CODE supports their use in several ways.

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

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

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

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

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

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

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

Formal arguments have the following syntax.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In CODE, a call to foo would take the form

```
foo(k)
```

rather than

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

as would be required in C.

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

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

is an illegal call.

18.0 UC Node Specifications

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

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

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

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

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

<vars> -- Local variables definitions.

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

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

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

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

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

```
<replication> :=

       { <thing_to_be_replicated> : <repl_list> } |

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

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

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

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

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

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

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

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

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

18.1 Input Ports Stanza

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

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

Example:

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

18.2 Output Ports Stanza

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

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

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

18.3 Shared Variables Stanza

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

<use> := reader | writer
```

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

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

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

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

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

This is legal. But the following arc topology specifications are not legal since they bind two

shared variable name to two different instances of the same variable.

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

18.4 Variable Declaration Stanza

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

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

<size_spec> := <array_sizes> | <member_sizes> |
               <array_sizes> <member_sizes>

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

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

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

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

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

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

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

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

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

18.5 Initial Computation Stanza

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

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

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

18.6 Firing Rules Stanza

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

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

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

<fbind> := <assign_stmt> (See "Statements")
```

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

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

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

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

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

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

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

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

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

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

Here are some example firing rule stanzas.

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

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

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

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

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

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

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

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

18.7 Computation Stanza

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

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

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

18.8 Routing Rules Stanza

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

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

<rguard> := <expr>

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

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

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

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

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

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

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

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

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

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

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

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

Binding sections for all rules with TRUE guards are performed.

Here are some example routing rules stanzas.

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

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

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

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

19.0 Name Sharing Relation Specifications

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

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

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

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

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

19.1 Shared Variables Stanza

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

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

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

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

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

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

19.2 Variable Declaration Stanza

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

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

19.3 Initial Computation Stanza

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

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

20.0 Graphs and Graph Calling

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

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

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

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

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

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

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

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

Here are some examples.

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

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

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

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

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

Here are some examples.

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

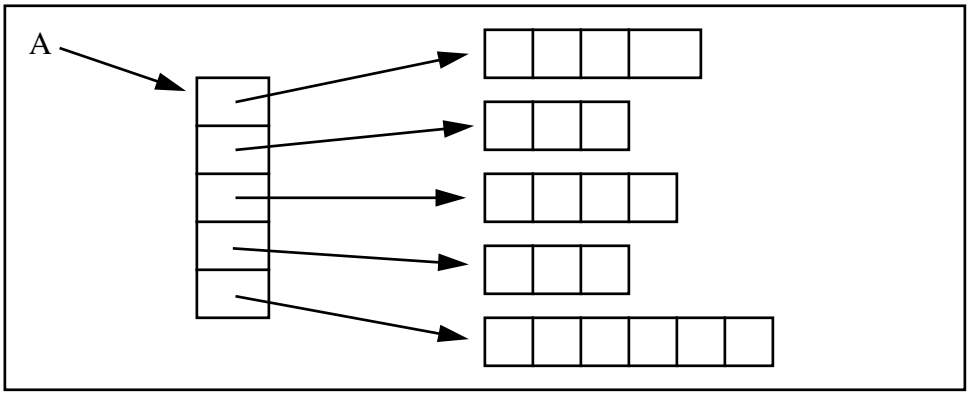
Just as with arcs going from UC to UC, only index variable are allowed on the left hand side, and the variables must be unique. They are place holders containing the indices of the entity that sent the data on the arc. The call index variables refer to the index of the Call node (and hence graph) the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

21.0 CODE Arrays

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

returns an array of size 10 x 3.

Function copy returns a copy of the array that is its argument. Also array assignment copys arrays. For example, the following assignment sets A to be a copy of B;

```
A = B;
```

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

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

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

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

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

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

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

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

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

Assuming, of course that array is regularly sized.

```
int SumArray(M, i_size, j_size, k_size)
        int ***M;
        int i_size;
        int j_size;
        int k_size;
{
        int i, j, k;
```

```
        int **t1, *t2;
        int sum=0;

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

        return sum;
}
```

The use of temporary variable t1 and t2 can substantially decrease the execution time of such functions, but use this trick only within sequential routines-- not directly from CODE. Array assignment in CODE implied copying the array.