Chapter 2. Overview of the CODE Language

This chapter will informally define the CODE model of parallel computation and provide an overview of how users prepare programs using the CODE environment. The model will be more thoroughly defined in Chapter 3.

CODE is a graphical parallel programming language. The fundamental idea is that users can create parallel programs with it by drawing and annotating graphs. A graph is a type of diagram that consists of nodes (represented by icons in CODE) and arcs that interconnect the nodes. Such a diagram shows a computation's communication structure. Nodes represent computations (or shared variables) and arcs represent the flow of data from one computation to another. Parallel programs are created by composing nodes into graphs by direct construction. Users draw the nodes and then interconnect them with arcs.

There are several steps in creating a program with CODE. First, the user draws the graph that shows the parallel program's communication structure. Then he or she annotates this graph by filling in a set of forms that describe properties of the nodes, arcs, etc. in the graph. For example, the user must specify which sequential computation a node performs. Finally, the user selects a target machine from a list of machine types for which translators have been prepared, and CODE translates the annotated graph into a program that can be compiled and run on the chosen target machine.

2.1 A Simple Example Program

When a user first runs CODE, an empty window appears. He or she then draws a graph in this window using a mouse. CODE graphs can contain several different node types which have different meanings and uses. Each different node type is represented by a different icon. The most important type of node is the Unit of Computation (UC) node. These represent sequential computations that can be composed into parallel programs by drawing arcs that show dataflow among them. Figure 2-1 shows a complete CODE program that has been created using only UC nodes and dataflow arcs.

This program integrates a function over an interval by dividing the interval in half and integrating over each piece separately. The results are summed to form the final answer. The program consists of four UC nodes. The arcs represent data that is created by one UC and is needed before another UC can begin its sequential computation. When the program runs, it first executes (or fires) the UC called “Split Interval” which divides the interval in half and sends the endpoints of the sub-intervals to both of the “Integ Half” nodes.

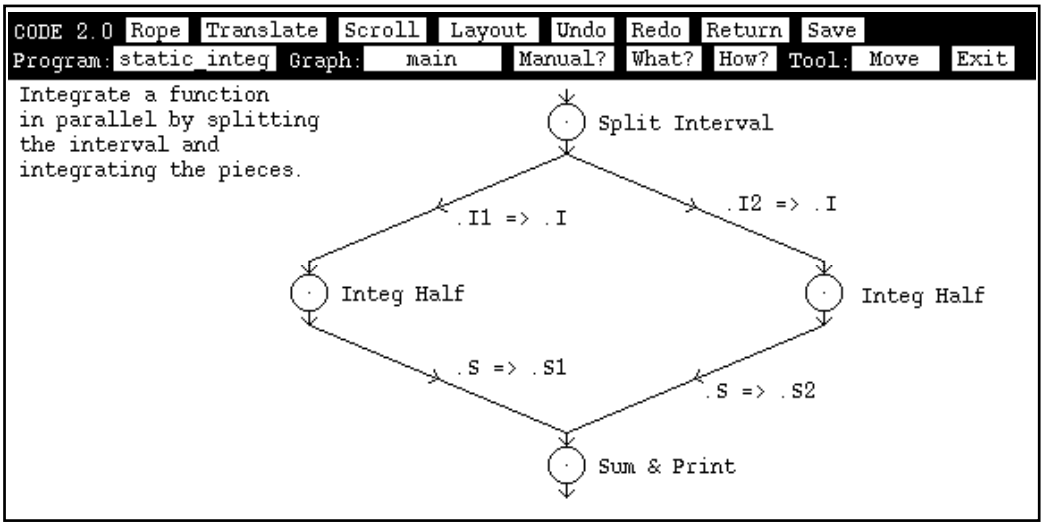

Figure 2-1. A CODE Program.

These nodes integrate, using a sequential procedure, a fixed function over the interval sent to them. They can run in parallel with each other since there is no arc from one to the other. Node “Sum & Print” waits for results from the two integrations and then adds them together and prints this sum. The arcs represent unbounded FIFO queues of data that are sent from one UC node to another. Nodes send and receive data on ports that are defined within them. Arcs bind ports together. These bindings are shown next to the arcs in Figure 2-1. For example, node “Split Interval” sends data out on port “I1” that is received by the “Integ Half” node on the left on port “I.”

Drawing the graph is only the first part of creating a CODE program. Users also have to annotate it to complete its specification. These annotations define many aspects of the program-- what sequential computation a UC node will perform, under what conditions it can fire, what data types are defined for the program, etc. Annotations are performed by filling out attribute forms associated with the entity being defined. Figure 2-2 shows the attribute form associated with a UC node.

Notice the fields for “Terminate node?” and “Start node?”. The programmer must designate exactly one UC node in the program to be the Start Node and exactly one node to be the Terminate node. When a CODE program is run, the system first executes the Start Node to get the computation started. The firing of the Terminate Node signals the end of the computation to the CODE system. The system will not end the computation until the Terminate Node fires.

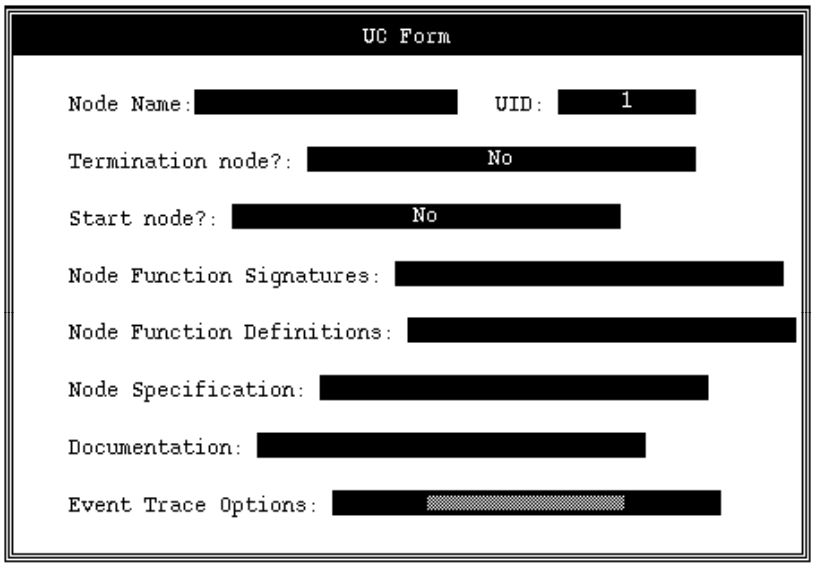

Figure 2-2. UC Node Attribute Form.

Recall that a UC node represents a sequential computation. These computations are specified by the user in the “Node Specification” field of the UC

attribute form. This specification also includes the mapping between the data on the incoming and outgoing arcs and the local data used by the UC's computation, and it also defines the conditions under which the UC node is allowed to fire.

The node specification is text-based and mostly declarative. It is divided into units called "stanzas" each of which has a different function. Stanzas consist of a name followed by some text enclosed in curly braces { }. The specification for the UC node named "Split Interval" is shown below.

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

The first stanza is the "output_ports" stanza. Output ports are names that are bound to arcs that leave the node, and input ports are names that are bound to arcs that enter nodes. This binding is specified as an arc attribute and will be discussed later. For now, it is enough to realize that two arcs that it refers to by names I1 and I2 leave Split Interval. The type name of the data that will be placed on these arcs is IntegInfo, a structure that contains the endpoints of an interval and the number of points to use in the integration.

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

The “comp” stanza defines the sequential computation the node performs when it fires. Split Interval defines two intervals which it stores in structure variables I1 and I2.

The language used in the comp stanza resembles a subset of C, but is not C. It is in fact, rather limited. It is expected that all substantial sequential computations will be encapsulated in ordinary sequential functions (usually written in C) that are defined outside of CODE (although CODE has facilities to help the user in managing, placing, and compiling them). Notice the calls to ReadInt and ReadReal. These are C functions, written separately.

UC nodes are the fundamental building blocks in CODE. UC nodes run in parallel with UC nodes. Hence, CODE programs express parallelism at a coarse level of granularity, roughly at the level of a subroutine call, since UC nodes are expected to perform fairly significant computations, often involving calls to external C functions.

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

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

Guards are Boolean expressions and Bindings are “Arc Outputs” that use a “<-” operator to place a value onto an arc. Other forms of bindings are allowed and will be discussed elsewhere. All bindings are performed from left to right on all routing rules whose guards all evaluate to TRUE. Node Split Interval places struct i1 onto arc I1 and struct i2 onto arc I2. These define the intervals that the Integ Half nodes will integrate.

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

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

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

```
Guard, Guard, ..., Guard =>
```

Guards represent "Arc Inputs" that extract information from arcs using the "->" operator.

Actually, firing rules can be somewhat more complicated than this. Assignment statements can appear on the right of the "=>" and guards can also be Boolean expressions. Firing rules in the general case have a very rich syntax and can express quite elaborate firing conditions and bindings. This will be discussed elsewhere since such elaborate firing rules are needed only occasionally.

Arc inputs represent both a condition and a binding. The notation "I -> i" represents the condition or guard "there is a value on arc I" and the binding "remove a value from arc I and place it into variable i." A UC node may execute whenever all of the guards on any of its firing rules are TRUE. Such a rule is said to be "satisfied". When the node fires, the bindings associated with the satisfied rule are performed. Hence, Integ Half may fire whenever there is a

value on arc I. It extracts a value from I and places it into local variable i for use by the node's computation.

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

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

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

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

The last issue in the Integration program is the annotation of arcs. Each arc's annotation is shown in Figure 2-1. (Such display is not automatic. The annotations have been written as comments on the graph.)

Arc annotations are called "arc topology specifications". They serve to bind names between nodes. As mentioned above, nodes use ports as local names for arcs. Arc topology specifications bind port names together. For example, the specification

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

indicates that output port I1 (declared in UC node Split Interval) is to be bound to input port I (declared in node Integ Half on the left). When you

draw an arc between UC nodes, you must specify what pair of ports the arc binds.

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

2.2 CODE Elements Are Actually Templates

It is time to confess that the description of CODE so far, although completely correct, has been oversimplified. The notations as they have been described are inadequate as the basis for a programming system because they are static. As described so far, the graph drawn represents a completely fixed computation and communication structure. Many real-world algorithms do not fit into this view. Aspects of their structure depend on runtime information. As a simple case, one may wish to prepare a program that will utilize as many processors as happen to be available at a particular moment to perform a certain task. Perhaps the desired structure is as shown in Figure 2-3 where there are N replicated nodes in the center, where N is a runtime determined value. Structures that are dictated by runtime parameters are called "dynamic".

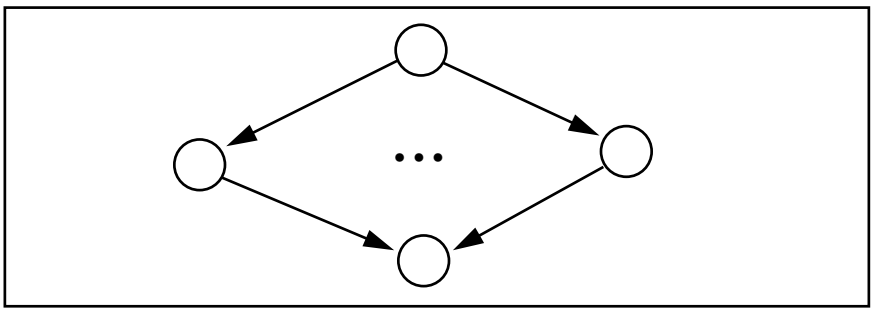

Figure 2-3. A Dynamic Computation Structure.

CODE directly supports the specification of dynamic structures. Rather than being static, every node or arc users draw in CODE can be instantiated

any number of times at runtime. The instantiations are named by integer valued indices. Hence, it is more accurate to say that you are creating a *template* for a node rather than a node itself when you draw and annotate a UC node.

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

The Integration program is limited to two-way parallelism. There are only two Integ Half nodes to run in parallel. An N-way parallel program can easily be envisioned that would use N "Integ" nodes each of which integrates a fraction of the interval of size (b - a)/N. A CODE program that implements this scheme is shown in Figure 2-4. The node Integ is instantiated N times using the same communication structure shown in Figure 2-3.

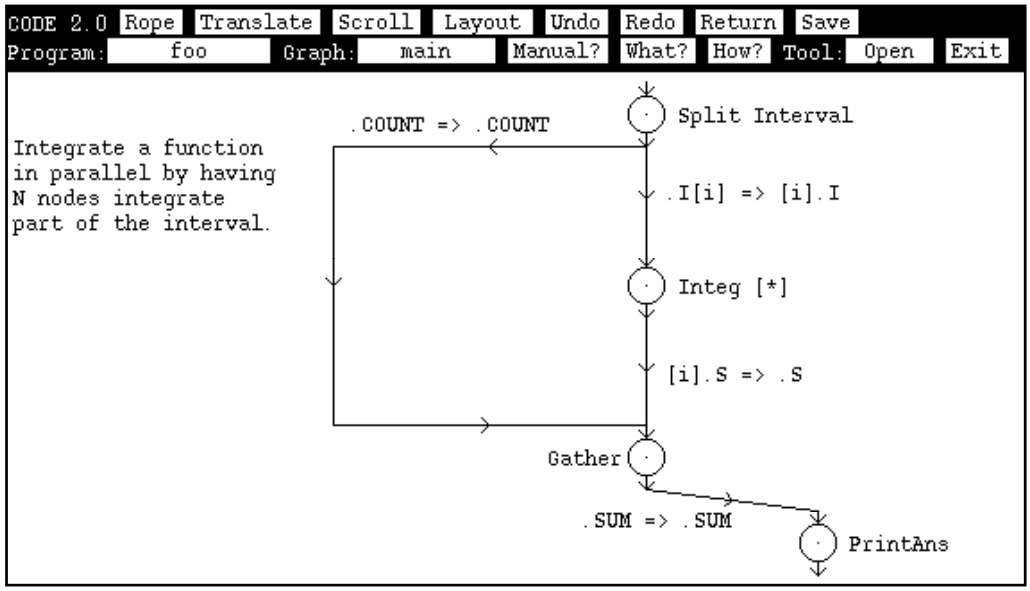

Figure 2-4. Another Integration Program.

The "[*]" by the name is a comment reminding the viewer that the node is instantiated multiple times. In fact, UC node names serve only as comments and, hence, are optional attributes. Actually, all of the text that appears in Fig-

ure 2-4 is either an optional name or a comment drawn on the graph. Such text increases the readability of the graph.

The multiple instantiation is specified by annotations that will be described in later chapters. The important idea to remember now is that all CODE elements (nodes,arcs, etc.) you draw are in reality templates that may be instantiated many times. The graph depicts static aspects of the program. Annotations capture dynamic aspects.

2.3 Other CODE Nodes

There are other node types than UC nodes in CODE. Many of these serve to hierarchically structure programs, exactly as subprograms and Call statements do in conventional languages such as C or Fortran. Figure 2-5 shows all of CODE's node types.

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

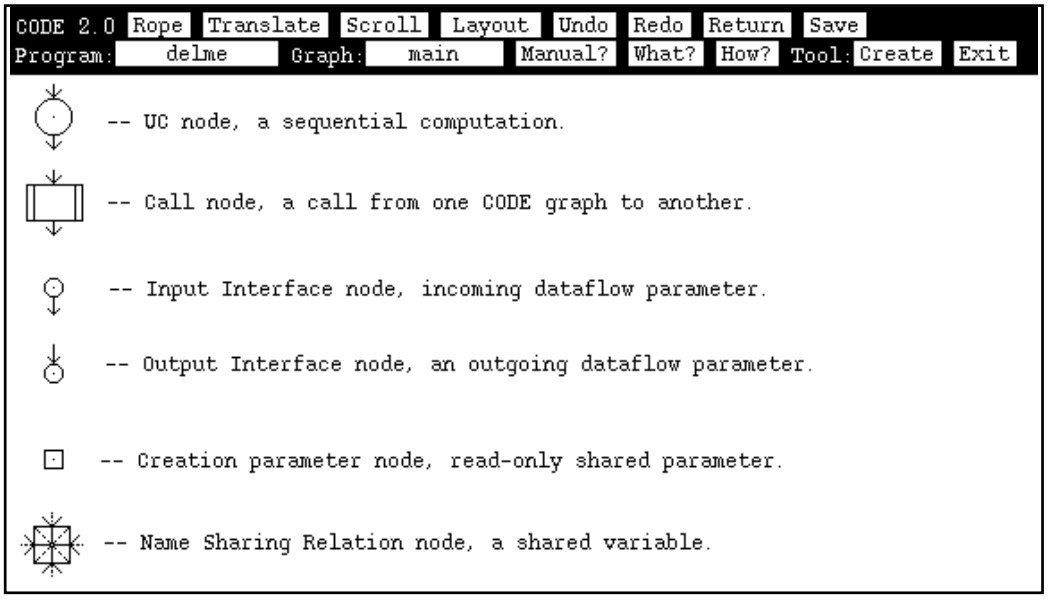

Figure 2-5. All CODE Node Types.

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

Name Sharing Relation nodes introduce shared variables. UC nodes that access them must declare themselves to be either readers or writers of the variable and an arc must be drawn from the UC node to the Name Sharing Relation node that defines the shared variable. This also will be described later.

2.4 Translating and Running CODE Programs

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

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