Technical Report

CMU/SEI-88-TR-20
ESD-TR-88-21

MasterTask:
The Durra Task Emulator

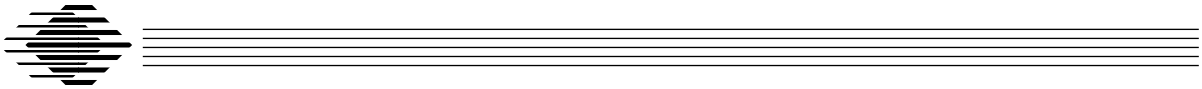
Mario R. Barbacci

July 1988

Technical Report

CMU/SEI-88-TR-20
ESD-TR-88-21
July 1988

MasterTask:
The Durra Task Emulator



Mario R. Barbacci

Software for Heterogeneous Machines Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

MasterTask: The Durra Task Emulator

Abstract. Durra is a language designed to support the construction of distributed applications using concurrent, coarse-grain tasks running on networks of heterogeneous processors. An application written in Durra describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

The tasks and types available to an application developer are described by a collection of Durra *task descriptions* and *type declarations* stored in a library. One of the components of a task description is a specification of the external timing behavior of the task. It describes the sequence of input and output port operations and the amount of processing time spent between port operations.

This report describes MasterTask, a program that can emulate any task in an application by interpreting the timing expression describing the behavior of the task, performing the input and output port operations in the proper sequence and at the proper time.

MasterTask is useful to both application developers and task developers. Application developers can build early prototypes of an application by using MasterTask as a substitute for task implementations that have yet to be written. Task developers can experiment with and evaluate proposed changes in task behavior or performance by rewriting and reinterpreting the corresponding timing expression.

1. Introduction to Durra

Durra [1, 2] is a language designed to support the construction of distributed applications using concurrent, coarse-grain tasks running on networks of heterogeneous processors. An application written in Durra selects and reuses *task descriptions* and *type declarations* stored in a library. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term ‘‘description language’’ rather than ‘‘programming language’’ to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) ‘‘machine language.’’ Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation and scheduling directives, which are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine. This is the translation process depicted in Figure 1-1.

We see three distinct phases in the process of developing an application using Durra: the

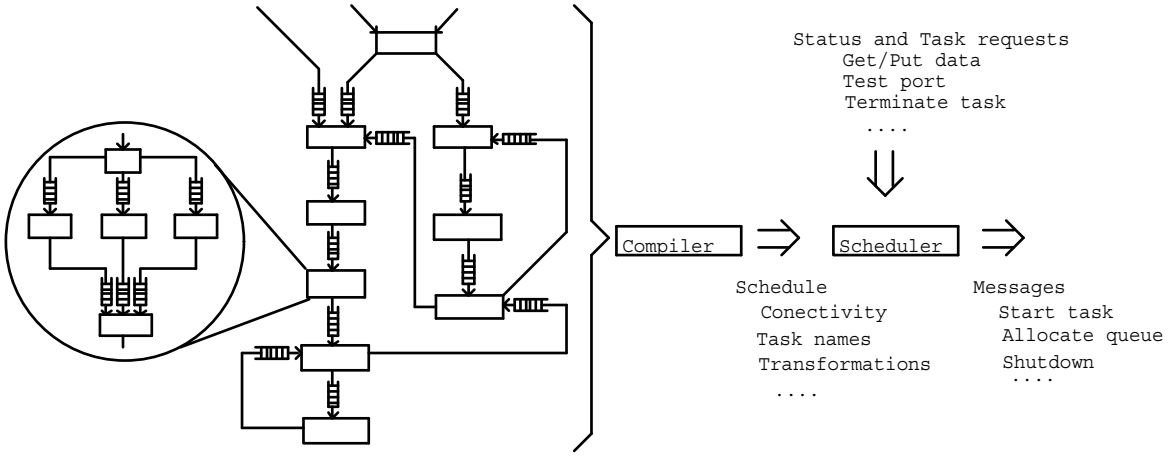


Figure 1-1: Compilation of an Application Description

creation of a library of tasks, the creation of an application using library tasks, and the execution of the application.

During the first phase, the developer writes descriptions of the components tasks. These task descriptions specify the properties of a task implementation (a program). For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. For each implementation of a task, a description must be written in Durra, compiled, and entered in the library.

A task description includes specifications of a task implementation timing behavior and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other miscellaneous attributes of the implementation.

During the second phase, the user writes an *application description*. Syntactically, an application description is a single task description and could be stored in the library as a new task. This allows writing of hierarchical application descriptions. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands or instructions to be interpreted by the scheduler.

During the last phase, the scheduler loads the task implementations (i.e., programs corresponding to the component tasks) to the processors and issues the appropriate commands to execute the programs.

Task descriptions are the building blocks for applications. Task descriptions include the following information (see Figure 1-2): (1) its interface to other tasks (**ports**) and to the

scheduler (**signals**); (2) its **attributes**; (3) its functional and timing **behavior**; and (4) its internal **structure**, thereby allowing for hierarchical task descriptions. For the purposes of this report, the relevant components of a task description are the behavioral specifications.

```
task task-name
   ports
                    -- Used for communication between a process and a queue
      port-declarations

   signals
            -- Used for communication between a process and the scheduler
      signal-declarations

   attributes
                    -- Used to specify miscellaneous properties of the task
      attribute-value-pairs

   behavior
       -- A description of the functional and timing behavior of the task
      requires predicate
      ensures predicate
      timing timing expression
   structure
                  -- A graph describing the internal structure of the task

      process-declarations
                           -- Declaration of instances of internal subtasks

      bind-declarations
                -- Mapping of internal process ports to this task's ports

      queue-declarations
                     -- Means of communication between internal processes

      reconfiguration-statements
                                    -- Dynamic modifications to the structure
end task-name
```

Figure 1-2: A Template for Task Descriptions

2. Durra Behavioral Specifications

The behavioral information part of a task description specifies functional and timing properties of the task. It consists of a functional information part and a timing expression. The functional information part consists of a pre-condition predicate on what is required to be true of the data coming through the input ports and a post-condition predicate on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports.

Timing expressions are the critical piece of information used by MasterTask. The syntax and semantics of timing expressions are described below. For additional information about the syntax and semantics of the functional information part, see the Durra reference manual [1].

2.1. Timing Expressions

This section is reproduced from [1].

Processes remove data from their input queues and store data in their output queues following a task-specific pattern provided by a timing expression. A timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports; this is the behavior of the task seen from the outside.

2.1.1. Time Literals

Syntax:

```
TimeLiteral       ::= Seconds { TimeBase } ,
                      IndeterminateTime

Seconds           ::= IntegerValue ,
                      RealValue

TimeBase          ::= ``DTIME'' ,
                                                  -- Time since start of day
                      ``ATIME'' ,
                                   -- Time since the start of the application
                      ``PTIME'' ,
                                        -- Time since the start of the process

IndeterminateTime ::= ``*''
```

Examples:

```
    3615.5 atime    -- An application relative time: 1 hour and 15.5 seconds
                                        -- after the start of the application.

    2.25       -- Two and a quarter seconds relative to some previous event

    *                                      -- An indeterminate point in time.
```

Meaning:

Time values are used to specify points in time. These can be either (1) absolute, in which case they must be followed by the name of a time base (the start of the day, the application, or the process); or (2) relative to some prior event in a timing expression, in which case a time base is not allowed. All time values are measured in seconds.

2.1.2. Event Expressions and Time Windows

Syntax:

```
Event              ::= GlobalPortName { ``.'' QueueOperation } ,
                       ``DELAY'' TimeWindow

TimeWindow         ::= ``['' TimeValue ``,'' TimeValue ``]''

QueueOperation     ::= ``ENQUEUE'' ,
                       ``DEQUEUE''
```

Examples:

```
in1     -- An operation (Dequeue, by default) on the queue feeding in1.
in1.dequeue                                -- The same operation as above.
delay[10, 15]    -- A delay interval lasting between 10 and 15 seconds.
delay[*, 10]              -- A delay interval taking at most 10 seconds.
delay[10, *]             -- A delay interval taking at least 10 seconds.
```

Meaning:

Queue operations performed by the processes constitute the basic events of an application description. An event expression represents a queue operation on a queue attached to a specific port, taking a variable amount of time to complete. A pseudo-operation, ''delay,'' is used to represent the time consumed by the process between (real) queue operations.

The name of the queue operation is optional. If the name is not given, a default queue operation is assumed: ''dequeue'' for input ports, ''enqueue'' for output ports.

Intervals of time between queue operations are denoted by a ''delay'' operation whose time window describes the minimum and maximum time consumed by the process in between queue operations.

2.1.3. Timing Expressions

Syntax:

```
TimingExpression   ::= { ``LOOP'' } SequentialEvent

SequentialEvent    ::= ParallelEvent_List(spaces)

ParallelEvent      ::= BasicEvent_List(double_vertical_bar)

BasicEvent         ::= Event ,
                       { Guard ``=>'' } qi[(] SequentialEvent ``)'')
```

```
Guard             ::= ''REPEAT'' IntegerValue ,
                      ''BEFORE'' TimeValue ,                -- Absolute time
                      ''AFTER'' TimeValue ,                 -- Absolute time
                      ''DURING'' TimeWindow ,       -- T_min is Absolute time
                      ''WHEN'' Expression           -- A Boolean expression
```

Examples:

```
in1 || in2   -- Two parallel input operations, starting simultaneously.

in1 delay[10,15] out1                    -- Three sequential operations.

repeat 5 => (in1 delay[10,15] out1)
                    -- Same as above but as a cycle repeated five times.

before 64800 DTIME => ( .  .  .  )
                           -- A sequence constrained to start before 6 pm.
                 (18 hours or 18*3600 seconds after the start of the day)

after 64800 DTIME => ( .  .  .  )
                            -- A sequence constrained to start after 6 pm.

during [64800 DTIME, 7200] => ( .  .  .  )
                -- A sequence constrained to start between 6 pm and 8 pm
              T_max is 2 hours counted from the start of the time window.

when (Current_Size(in1) > 0) and (Current_Size(in2) > 0) =>
                                                    ((in1 || in2) out1);
           -- A sequence that starts after both input queues have data.

loop when (Current_Size(in1) > 0) and (Current_Size(in2) > 0) =>
                                                    ((in1 || in2) out1);
                -- The same sequence as above but repeated indefinitely.
```

Meaning:

A timing expression is a regular expression describing the patterns of execution of operations on the input and output ports of a task. The keyword **loop** can be used to indicate that the pattern of operations is repeated indefinitely.

A timing expression is a sequence of parallel event expressions. Each parallel event expression consists of one or more event expressions separated by the symbol || to indicate that their executions overlap. Since the expressions might take different amounts of time to complete, nothing can be said about their completion, other than a parallel event expression terminates when the last event terminates.

A basic event expression is either a queue operation (including ''delay'') or a timing expression enclosed in parentheses. The latter form also allows for the specification of a guard, an expression specifying the conditions under which a sequence of operations is allowed to start or repeat its execution.

Guard	Description
repeat	This guard indicates repetitions of a timing expression. The number of repetitions is a non-negative integer value.
before	This guard is followed by an absolute time value representing the latest start time allowed. The task is terminated if a deadline has elapsed.
after	This guard is followed by an absolute time value representing the earliest start time allowed. If necessary, the sequence is blocked until the deadline.
during	This guard is followed by a time window during which the sequence is allowed to start. The first value is the earliest start time allowed and must be an absolute time value; the second value is the latest start time allowed and can be an absolute time value or a time value relative to the former.
when	This guard describes what is required to be true of the state of the system (i.e., time and queues; see Section 2.2) before the sequence is allowed to start. It is a pre-condition for starting the sequence.

2.1.4. Restrictions on Time Values and Time Windows

Although the syntax allows both absolute and relative time values to appear in either of the two boundaries in a time window, not all of the possible combinations make sense:

1. In the time window attached to ‘‘delay’’ operation, the time values must be relative (i.e., no time basis allowed) and are interpreted relative to the start of the operation.

2. In the time window of a **during** guard, the first time value (T_{min}) must be absolute. The second time value (T_{max}) can be absolute or relative. In the latter case, the time value is relative to T_{min}.

2.2. Predefined Functions

This section is reproduced from [1].

A small number of functions are predefined in the language. These functions are used to obtain the current time in the various time bases, to perform computations with time values, and to obtain the number of elements stored in a queue. These functions, together with time and numeric literals, constitute the terms used to build expressions in timing guards and reconfiguration conditions.

Syntax:

```
FunctionCall      ::= FunctionName { FunctionParameters }
```

```
FunctionName       ::= ''CURRENT_DTIME'' ,
                       ''CURRENT_ATIME'' ,
                       ''CURRENT_PTIME'' ,
                       ''MINUS_TIME'' ,
                       ''PLUS_TIME'' ,
                       ''CURRENT_SIZE''
FunctionParameters     ::= ''('' DurraValue_List_comma '')'')
                 -- The type and number of parameters is function dependent.
```

Examples:

```
Plus_Time(Current_DTime, 9000)
                                  -- 2.5 hours from the current time of day
Current_Size(Master_Process.Data_Port)
                                     -- the size of a queue feeding a port
```

Meaning:

The following functions are predefined in the language: ''current_dtime,'' ''current_atime,'' ''current_ptime,'' ''minus_time,'' ''plus_time,'' and ''current_size.''

The functions with names like ''Current_?Time'' return the current time as a number of seconds relative to the appropriate time base.

The function call ''Minus_Time(TimeValue$_1$,TimeValue$_2$)'' returns the time value obtained by subtracting TimeValue$_2$ from TimeValue$_1$. The following cases are allowed:

1. If both parameters are absolute times, the result is a relative time, i.e., a duration (in seconds).

2. If TimeValue$_1$ is an absolute time and TimeValue$_2$ is a relative time, the result is an absolute time with the same time base as TimeValue$_1$.

3. If both parameters are relative times, the result is a relative time.

The function call ''Plus_Time(TimeValue$_1$,TimeValue$_2$)'' returns the time value obtained by adding TimeValue$_2$ to TimeValue$_1$. The following cases are allowed:

1. If one parameter is an absolute time and the other parameter is a relative time, the result is an absolute time in the same time base.

2. If both parameters are relative times, the result is a relative time, i.e., a duration.

The function call ''Current_Size(GlobalPortName)'' returns the current number of elements stored in the queue associated with a given port.

Calls to these functions can appear anywhere a value of the same kind as the return value can appear. That is, a call to a function returning an integer, a real, a string, or a time value can appear instead of an integer, a real, a string, or a time value, respectively.

3. Durra Runtime Environment

To understand the operation of MasterTask, it is necessary to understand the components of the Durra runtime environment. MasterTask operates like any other task implementation. It does not receive special treatment by the runtime environment and, in fact, the scheduler and servers have no built-in knowledge of MasterTask. This section provides the summary of the runtime environment necessary to understand how task implementations are invoked and how they can receive information contained in the task description. How MasterTask uses this information is the subject of Section 4.4. For further details about the runtime environment, see [3].

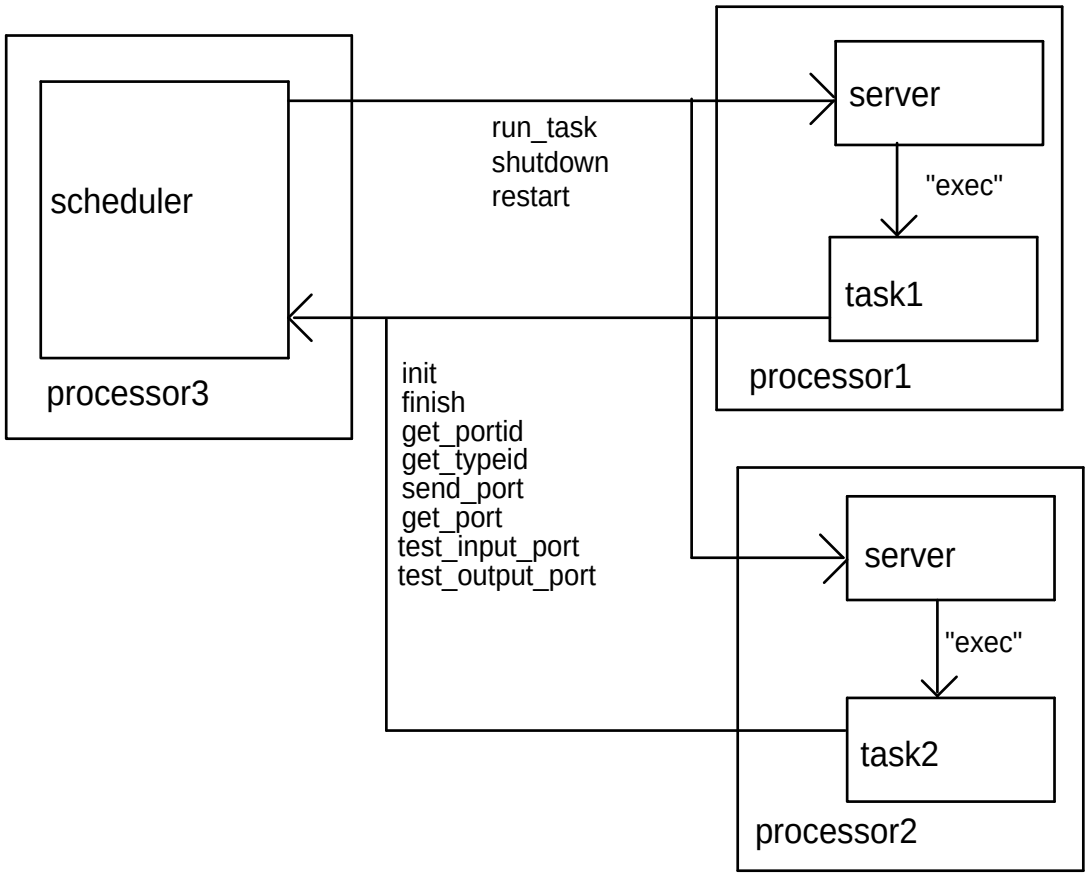


Figure 3-1: The Durra Runtime Environment

There are three active components in the Durra runtime environment: the application tasks, the Durra server, and the Durra scheduler. Figure 3-1 shows the relationship between these components.

After compiling the type declarations, the component task descriptions, and the application description, as described previously, the application can be executed by performing the following operations:

1. The component task implementations must be loaded in a special directory in the appropriate processors. The directory name is known to the Durra servers and scheduler. Since MasterTask is playing the role of a task implementation, it too must be stored in this special directory.

2. An instance of the Durra server must be started in each processor.

3. The scheduler must be started in one of the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

The scheduler is the part of the Durra runtime system responsible for starting the tasks, establishing communication links, and monitoring the execution of the application. In addition, the scheduler implements the predefined tasks (**broadcast**, **merge**, and **deal**) described in [1]. The scheduler operates by reading a file containing the instructions generated by the Durra compiler. For the purposes of this reports, the two relevant instructions are **task_load** and **source**.

The **task_load** instruction takes three parameters: 1) the task name, 2) a processor to run it on (either a specific processor or a class of processors), and 3) the name of the executable file.

```
(task_load MAIN.PC VAX  "sink_task2")
```

The **source** instruction takes two parameters: 1) a task name, and 2) the value of the ‘‘source’’ attribute specified in the task description, if any.

```
(source MAIN.DISPLAY "display.durra.TREE")
```

If the source attribute is not specified in the task description, the parameter is the name of the syntax tree produced by the Durra compiler (see [3] for file-naming conventions). This information might be used by the task implementation to examine its own Durra description. This information will be passed to the indicated application task at startup. The significance of the ‘‘implementation,’’ ‘‘source,’’ and ‘‘processor’’ attributes will become clear in Section 4.4.

4. MasterTask Implementation Overview

As described in Section 2.1, a Durra timing expression can contain concurrent events as well as loops and guards that block execution until some condition is met (e.g., some amount of time has elapsed since the start of the application, an input queue has a given number of data elements). In this section we describe the use of Ada tasking features to implement concurrent events. In this description, wWhenever there is the possibility of confusion, we will use the term ‘‘Ada task’’ to distinguish it from a ‘‘Durra task.’’ Otherwise the context should make it clear which type of task is implied.

When MasterTask starts, it reads the Durra syntax tree of the task it wants to emulate, locates the timing expression, and builds an internal data structure, isomorphic to the original timing expression syntax tree, but with different information stored in the nodes. In particular, each node in the internal tree contains a reference to a task object. This task object is responsible for performing one or more node-dependent operations: 1) execute a queue operation (including ‘‘delay’’); 2) evaluate a guard expression (including ‘‘repeat’’); 3) direct the execution of the tasks responsible for the subtrees rooted at this node.

Generally, MasterTask exhibits the same behavior as a regular Durra task implementation, issuing the same calls to the scheduler (see [3] for a description of the operations) in the following order:

1. Calls the **init** function to establish communication with the scheduler. This operation is invoked by MasterTask’s main program during the initialization phase.

2. Calls **get_portid** for each of the task ports (these are the ports named in the timing expression). These operations are invoked while building the internal structures.

3. Calls **send_port** and **get_port** as necessary to send and receive data. In between port operations, it might wait for varying amounts of time, as indicated in the timing expression. These operations are invoked by the Ada task objects, operating concurrently as they interpret their assigned nodes.

4. Calls **finish** to break communication with the scheduler when it completes its job. This operation is invoked by MasterTask’s main program after the Ada task object assigned to the root node has terminated (and by implication, all of the task objects are terminated). If the timing expression contains a never-ending loop or a guard that is not satisfied, at least one of the task objects will not terminate and the **finish** operation will not be invoked. In this case MasterTask must be terminated by the scheduler.

4.1. The Ada Task Type Specification

All of the task objects assigned to the nodes are instances of the same task type:

```
task type Master_Task_Template is
  entry Start(e: in FSA_Node);
  entry Stop;
end Master_Task_Template;
```

The task type specification indicates that the task objects will have two entry points, ‘‘Start’’ and ‘‘Stop.’’ All coordination of activities will be performed by calling these entries.

4.2. The Ada Task Type Body

```
task body Master_Task_Template is
  e1, e2: FSA_Node;
begin
  loop

    accept start(e: in FSA_Node) do
      e1 := e;        --| remember the node we are responsible for
    end start;

    case e1.kind is   --| perform a node-dependent action
      when FSA_Event =>
        ..............
      when FSA_Repeat =>
        ..............
      when FSA_Guard =>
        ..............
      when FSA_List =>
        ..............
    end case;

    accept stop;

  end loop;
end Master_Task_Template;
```

The body of the task type implementation consists of a loop containing three steps:

1. Perform an **accept** ‘‘Start’’ statement. When the rendezvous occurs, the parent task (the caller) is released immediately after a simple assignment statement is executed by the entry proper.

2. Perform some node-dependent action (e.g., delay, invoke a queue operation, start several children processes in parallel).

3. Perform an **accept** ‘‘Stop’’ statement. The entry call to ‘‘Stop’’ serves as a lock to the parent task. The parent cannot continue until the child task arrives here, i.e., until it has completed performing the node-dependent action. When the rendezvous occurs, the parent task (the caller) is released and the child task loops to its **accept** ‘‘Start’’ statement.

The node-dependent actions are shown in an abridged fashion. We eliminated all details except those that illustrate the synchronization between parent and children tasks:

```
when FSA_Event =>
  do_operation(e1);
```

The simplest node-dependent action performs a delay or a queue input or output operation. A delay operation is associated with a time window that specifies a minimum and maximum delay. MasterTask draws a random number from a uniform distribution between these two boundaries and uses it as the length of time it delays its operation (this is implemented with an Ada **delay** statement).

Normally, queue input and output operations are performed by issuing the **send_port** and **get_port** schedulers call, as described in [3]. MasterTask can also be used as a freestanding program, running without the help of the Durra runtime system (see Section 4.5). In freestanding mode, queue operations are emulated by using a default delay duration (0.1 second).

```
when FSA_Repeat =>
  for i in 1..e1.count
  loop
    e1.rexpression.node_task.start(e1.rexpression);
    e1.rexpression.node_task.stop;
  end loop;
```

A node that implements a ‘‘repeat’’ guard invokes its child task a number of times ‘‘count.’’ The parent task calls the ‘‘Start’’ entry and blocks until the child task gets there. Once they are both synchronized, the parent is released and the child goes on to execute the loop body. The parent task immediately calls the ‘‘Stop’’ entry and blocks until the child task gets there (i.e., until the child task has completed one iteration). At that point both tasks are released. The child task goes back to the head of the outer loop to the **accept** ‘‘Start’’ statement and waits there until the next time it is called upon to evaluate its subtree. The parent task either loops again, or terminates the loop and goes to its **accept** ‘‘Stop’’ statement.

```
when FSA_Guard =>
  while not Do_Guard(e1)
  loop
    delay Delay_Between_Retries;
  end loop;
  e1.expression.node_task.start(e1.expression);
  e1.expression.node_task.stop;
```

A node that implements a guard (other than a ‘‘repeat’’ guard) loops evaluating the guard expression until it is satisfied. When that occurs, the node invokes its child task to perform the guarded timing expression. The parent first calls the ‘‘Start’’ entry, thus triggering the child, and then calls the ‘‘Stop’’ entry and blocks until the child task gets there. To avoid wasting resources and to let other task objects run, MasterTask delays for a period of 0.1 second before reevaluating a guard.

```
when FSA_List =>
  if e1.is_parallel then
    e2 := e1;                    --| Start the branches together
    while e2 /= NULL
    loop
      e2.this_event.node_task.start(e2.this_event);
      e2 := e2.next_event;
    end loop;

    e2 := e1;                    --| Wait for all of them to finish.
    while e2 /= NULL
    loop
      e2.this_event.node_task.stop;
      e2 := e2.next_event;
    end loop;

  else                           --| Start the branches in sequence
    e2 := e1;
    while e2 /= NULL
    loop
      e2.this_event.node_task.start(e2.this_event);
      e2.this_event.node_task.stop;
      e2 := e2.next_event;
    end loop;
  end if;
```

Finally, we have the cases of nodes implementing concurrent and sequential actions. The structure of these nodes is the same except for a flag that distinguishes between these two cases.

If the node implements a list of concurrent actions, it first loops issuing calls to the ‘‘Start’’ entry of all of its children. Since only the ‘‘Start’’ entry is called, the parent does not block for long and loops back immediately to start the next child. After starting all of its children, the parent executes a similar loop, but this time calling the ‘‘Stop’’ entry of its children. Notice that it is irrelevant in which order the children reach the **accept** ‘‘Stop’’ statement and rendezvous with the parent, since the parent must eventually rendezvous with all its children.

If the node implements a list of sequential actions, the behavior is very similar to the implementation of the ‘‘repeat’’ guard that we saw before, except that instead of looping over a counter, the looping takes place over the list of children tasks.

4.3. The Main Program

The main program invokes the different components of MasterTask, as described below:

```
procedure master is
  Root: Tree_Node;               --| Syntax tree structure
  Header_String: String_Type;
  fnode: FSA_Node;               --| Internal tree structure
begin
  Init;          --| Call the Durra initialization procedure

  Read_Tree(User_Source_Parameter, Root, Header_String);
                 --| Read the Durra syntax tree
```

```
  fnode := Parse_Timing(Extract_Timing(Root));
               --| Generate the internal data structure

  fnode.node_task.start(fnode);
               --| Start the root task

  fnode.node_task.stop;
               --| Wait until the root task is done.

  Finish;
               --| Call the Durra termination procedure
end Master;
```

4.4. Using MasterTask in a Task Description

To use MasterTask to emulate an implementation, the writer of the task description must write (or modify) the real task description to look like this:

```
task task_name                                              -- required
ports port declarations                                     -- required
signals signal declarations                                 -- optional
behavior                                                    -- required
    requires Larch predicate                                -- optional
    ensures Larch predicate                                 -- optional
    timing timing expression                                -- required
Attributes                                                  -- required
    implementation = "master";                              -- required
    processor = VAX;                                        -- required
    any other attributes EXCEPT "source"                    -- optional
Structure structure declarations                            -- optional
end task_name
```

Except for the fixed ''implementation'' and ''processor'' attributes and the absence of the ''source'' attribute, any other components of the task description should be written as if the real task implementation were to be used.

The ''implementation'' attribute indicates that MasterTask is implemented by a program, ''master,'' stored in a special directory as indicated in Chapter 3.

The ''processor'' attribute indicates that this program can only execute on a VAX processor. However, since MasterTask is written in Ada, it should be easily portable to other machines, in which case the appropriate version of the attribute should be used.

Finally, the absence of a ''source'' attribute forces the Durra compiler to use as a replacement value the name of the syntax tree produced during the compilation of the task description. This is how MasterTask knows what to do. It gets the file name, reads the syntax tree, looks for the timing expression, builds the appropriate internal structures, and interprets them.

It should be fairly obvious but nevertheless worth mentioning that MasterTask has no way of generating ''meaningful'' output data or of making use of its input data. As output data, it generates blocks of random bytes of appropriate length. All input data are dropped.

4.5. Using MasterTask In Freestanding Mode

Although MasterTask was designed to help the developers of an application emulate missing task implementations, it is also convenient to be able to ‘‘test’’ a task’s timing expression without having to build an ‘‘application’’ around it. MasterTask can be invoked directly via a UNIX command:

dmaster *task_file_name*

The parameter, *task_file_name*, is the name of the file containing the syntax tree produced by the Durra compiler. The file name has the form ‘‘*name*.durra.TREE,’’ but the user only needs to specify *name* (or *name*.durra) since the **dmaster** command can supply the missing extensions. In freestanding mode, queue operations are not performed, and instead, their duration is emulated by using a fixed default delay (0.1 seconds).

5. A Durra Example

This section contains a complete example of a Durra application. As show in Figure 5-1, it consists of two type declarations, three component task descriptions (and their implementations), and the application description.

The example illustrates the use of a predefined task, **broadcast**, which is implemented directly by the scheduler. In this application, one task (‘‘taska’’) sends out strings of data to the **broadcast** task which, in turn, sends them on to two other tasks (‘‘taskb’’ and ‘‘taskc’’). The application description (‘‘main’’) cements all of the component tasks together in the configuration shown in Figure 5-2.

5.1. Type Declarations and Type Descriptions

‘‘Byte’’ is the basic type (a scalar type 8 bits long). ‘‘String’’ is an unbounded sequence of bytes.

‘‘Taska’’ has a single output port, ‘‘out1,’’ which produces strings. ‘‘Taskb’’ and ‘‘taskc’’ both have a single input port, ‘‘in1,’’ which consume strings. Not surprisingly, all three tasks are implemented by the program ‘‘master.’’ However, each task has a different behavior, specified by its timing expression.

‘‘Taska’’ executes a simple loop three times. In each pass, it first *computes* for some random amount of time between 5 and 10 seconds, and then performs an output operation on its output port. This output operation produces the string to be broadcast to the other two user tasks.

‘‘Taskb’’ waits until it is 30 seconds after process start-time before executing a loop three times, each time performing an input operation on its input port. This input operation consumes one of the strings that were broadcast.

‘‘Taskc’’ starts by trying to perform an input operation on its input port, following which it *computes* for some random amount of time between 10 and 13 seconds. Notice that the input operation is likely to take some time to complete because the producer (Taska) will only generate its first string after some delay. The third step in Taskc is a guarded time expression. The task waits until there are two elements in its input queue before proceeding to remove in parallel. The removal operations take place in two concurrent paths, each of which waits for some random amount of time before performing the actual operation.

Task ‘‘main’’ is the application description. It specifies the three tasks that make up the application, plus an instance of the predefined task **broadcast**. The structure part specifies the interconnection of those four tasks.

```
type byte is size 8;
type string is array of byte;
```

a -- Type Declarations

```
task taska
    ports
        out1: out string;
    behavior
        timing (repeat 3 => (delay[5, 10] out1.enqueue) );
    attributes
        processor = vax;
        implementation = "master";
end taska;
task taskb
    ports
        in1: in string;
    behavior
        timing (after 30 ptime => (repeat 3 => (in1)));
    attributes
        processor = vax;
        implementation = "master";
end taskb;
task taskc
    ports
        in1: in string;
    behavior
        timing (in1.dequeue delay [10,13]
               (when current_size(in1) = 2 =>
                   ( (delay [3,5] in1.dequeue) ||
                     (delay [1,10] in1.dequeue) ) ) );
    attributes
        processor = vax;
        implementation = "master";
end taskc;
```

b -- Task Descriptions

```
task main
    structure
        process p1: task taska;
                p2: task taskb;
                p3: task taskc;
                pb: task broadcast
                        ports   in1: in string;
                                out1, out2: out string;
                        end broadcast;
        queues  q1b[10]: p1.out1 >> pb.in1;
                qb2[10]: pb.out1 >> p2.in1;
                qb3[10]: pb.out2 >> p3.in1;
end main;
```

c -- Application Description

Figure 5-1: Durra Type Declarations and Task Descriptions

After all of the above files are compiled, the Durra compiler generates a file with instructions to the scheduler. See [3] for further information about the scheduler instructions and a description of the UNIX commands that invoke the compiler and scheduler-instruction generator. For this example application, these details are not important.

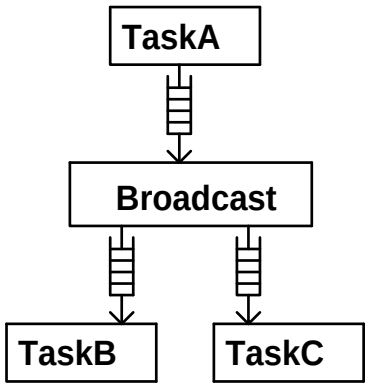


Figure 5-2: Application Structure

5.2. Execution Traces

Each instance of MasterTask produces a separate trace file in which each line is prefixed with a string of the form ‘‘PID=n,’’ where ‘‘n’’ is the process id number assigned by the scheduler to the task (i.e., instance of MasterTask). A trace of the execution of this example application is shown in Figures 5-3, 5-4, and 5-5.

The first line of each trace file displays the various time bases assumed by the task: the application start time (AT), the process start time (PT), and the day time (DT).

Following the time bases, each task displays the internal data structure generated during the initialization phase. It is a *prettyprinted* tree in which each node corresponds to a timing expression construct. The structure is displayed for later identification of events in the trace.

The bulk of the trace files display each timing expression event and the time of ocurrence of the event. Events include queue operations ‘‘Enqueue’’ and ‘‘Dequeue,’’ delays (treated as a pseudo queue operation), and guard evaluations.

Following the time stamp, each event prints some information about its operation. ‘‘Repeat’’ guards have the string ‘‘I=’’ followed by the current iteration number and the total number of iterations. Other guards are identified by the guard type (e.g., ‘‘OP_AFTER’’). ‘‘Enqueue,’’ ‘‘Dequeue,’’ and ‘‘Delay’’ operations generate two lines of trace. The first line (marked with the string ‘‘->’’) indicates the time at which the operation started. The second line (marked

with the string ‘‘<-’’) indicates the time at which the operation completed. Depending on the event type, some lines will make reference to a node in the timing expression syntax tree. These are marked with the string ‘‘N=’’ followed by an integer, where the integer is the node number, as labeled in the prettyprinted tree shown at the beginning of the trace file.

```
PID=1 1988/04/25 75548.929 ATIME
PID=1 1988/04/25 75548.960 PTIME
PID=1 1988/04/25 75548.979 DTIME
PID=1 N= 1      FSA_GUARD(OP_NOGUARD)
PID=1 N= 2        FSA_REPEAT Count: 3
PID=1 N= 3          FSA_LIST Parallel: FALSE
PID=1 N= 4            FSA_EVENT Op= FSA_DELAY P=OUT1 ( 0)
PID=1 N= 5              FSA_TIMEWINDOW
PID=1 N= 6                5.000
PID=1 N= 7                10.000
PID=1 N= 8            FSA_LIST Parallel: FALSE
PID=1 N= 9              FSA_EVENT Op= FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75549.359 ATIME N= 2 I= 1 ( 3)
PID=1 T=1988/04/25 75549.380 ATIME N= 4 -> FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75559.280 ATIME N= 4 <- FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75559.299 ATIME N= 9 -> FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75559.349 ATIME N= 9 <- FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75559.400 ATIME N= 2 I= 2 ( 3)
PID=1 T=1988/04/25 75559.419 ATIME N= 4 -> FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75564.780 ATIME N= 4 <- FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75564.830 ATIME N= 9 -> FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75564.880 ATIME N= 9 <- FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75564.900 ATIME N= 2 I= 3 ( 3)
PID=1 T=1988/04/25 75564.940 ATIME N= 4 -> FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75572.359 ATIME N= 4 <- FSA_DELAY P=OUT1 ( 0)
PID=1 T=1988/04/25 75572.380 ATIME N= 9 -> FSA_ENQUEUE P=OUT1 ( 0)
PID=1 T=1988/04/25 75572.419 ATIME N= 9 <- FSA_ENQUEUE P=OUT1 ( 0)
```

Figure 5-3: Tracing of Taska

```
PID=2 1988/04/25 75548.539 ATIME
PID=2 1988/04/25 75548.770 PTIME
PID=2 1988/04/25 75548.789 DTIME
PID=2 N= 1      FSA_GUARD(OP_NOGUARD)
PID=2 N= 2        FSA_GUARD(OP_AFTER)
PID=2 N= 5          1988/04/25 75571.780 PTIME
PID=2 N= 3          FSA_REPEAT Count: 3
PID=2 N= 4            FSA_EVENT Op= FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75549.080 ATIME N= 2 G= OP_AFTER FALSE... looping
PID=2 T=1988/04/25 75571.869 ATIME N= 2 G= OP_AFTER TRUE ... continuing
PID=2 T=1988/04/25 75571.880 ATIME N= 3 I= 1 ( 3)
PID=2 T=1988/04/25 75571.909 ATIME N= 4 -> FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75571.950 ATIME N= 4 <- FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75571.969 ATIME N= 3 I= 2 ( 3)
PID=2 T=1988/04/25 75571.989 ATIME N= 4 -> FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75572.039 ATIME N= 4 <- FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75572.059 ATIME N= 3 I= 3 ( 3)
PID=2 T=1988/04/25 75572.070 ATIME N= 4 -> FSA_DEQUEUE P=IN1 ( 0)
PID=2 T=1988/04/25 75572.530 ATIME N= 4 <- FSA_DEQUEUE P=IN1 ( 0)
```

Figure 5-4: Tracing of Taskb

```
PID=3 1988/04/25 75552.809 ATIME
PID=3 1988/04/25 75552.890 PTIME
PID=3 1988/04/25 75552.929 DTIME
PID=3 N= 1      FSA_GUARD(OP_NOGUARD)
PID=3 N= 2        FSA_LIST Parallel: FALSE
PID=3 N= 3          FSA_EVENT Op= FSA_DEQUEUE P=IN1 ( 0)
PID=3 N= 4          FSA_LIST Parallel: FALSE
PID=3 N= 5            FSA_EVENT Op= FSA_DELAY P=IN1 ( 0)
PID=3 N= 6              FSA_TIMEWINDOW
PID=3 N= 7                10.000
PID=3 N= 8                13.000
PID=3 N= 9            FSA_LIST Parallel: FALSE
PID=3 N= 10             FSA_GUARD(OP_NOGUARD)
PID=3 N= 11               FSA_GUARD(OP_WHEN)
PID=3 N= 30                 FSA_OPERATOR(OP_EQL)
PID=3 N= 31                   FSA_OPERATOR(OP_CURRENTSIZE)
PID=3 N= 32                     0
PID=3 N= 33                   2
PID=3 N= 12                 FSA_LIST Parallel: TRUE
PID=3 N= 13                   FSA_GUARD(OP_NOGUARD)
PID=3 N= 14                     FSA_LIST Parallel: FALSE
PID=3 N= 15                       FSA_EVENT Op= FSA_DELAY P=IN1 ( 0)
PID=3 N= 16                         FSA_TIMEWINDOW
PID=3 N= 17                           3.000
PID=3 N= 18                           5.000
PID=3 N= 19                       FSA_LIST Parallel: FALSE
PID=3 N= 20                         FSA_EVENT Op= FSA_DEQUEUE P=IN1 ( 0)
PID=3 N= 21                   FSA_LIST Parallel: TRUE
PID=3 N= 22                     FSA_GUARD(OP_NOGUARD)
PID=3 N= 23                       FSA_LIST Parallel: FALSE
PID=3 N= 24                         FSA_EVENT Op= FSA_DELAY P=IN1 ( 0)
PID=3 N= 25                           FSA_TIMEWINDOW
PID=3 N= 26                             1.000
PID=3 N= 27                             10.000
PID=3 N= 28                         FSA_LIST Parallel: FALSE
PID=3 N= 29                           FSA_EVENT Op= FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75553.320 ATIME N= 3 -> FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75559.690 ATIME N= 3 <- FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75559.710 ATIME N= 5 -> FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75572.669 ATIME N= 5 <- FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75572.770 ATIME N= 11 G= OP_WHEN TRUE
PID=3 T=1988/04/25 75572.799 ATIME N= 15 -> FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75572.849 ATIME N= 24 -> FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75575.969 ATIME N= 15 <- FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75575.989 ATIME N= 20 -> FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75576.039 ATIME N= 20 <- FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75578.229 ATIME N= 24 <- FSA_DELAY P=IN1 ( 0)
PID=3 T=1988/04/25 75578.250 ATIME N= 29 -> FSA_DEQUEUE P=IN1 ( 0)
PID=3 T=1988/04/25 75578.299 ATIME N= 29 <- FSA_DEQUEUE P=IN1 ( 0)
```

Figure 5-5: Tracing of Taskc

References

[1] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3, Software Engineering Institute, Carnegie Mellon University, December, 1986.
Also Technical Report CMU-CS-86-176, Department of Computer Science, Carnegie Mellon University, December 1986, and NTIS Report No. AD-A178 975.

[2] M.R. Barbacci, C.B. Weinstock, and J.M. Wing.
Programming at the Processor-Memory-Switch Level.
In *Proceedings of the 10th International Conference on Software Engineering*. Singapore, April, 1988.

[3] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18, Software Engineering Institute, Carnegie Mellon University, July, 1988.

Table of Contents