Technical Report

CMU/SEI-89-TR-034
ESD-TR-89-045

Durra: A Task-Level Description Language Reference Manual (Version 2)

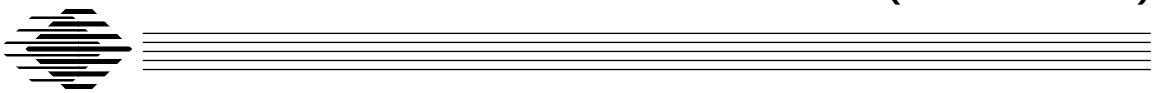
Mario R. Barbacci
Jeannette M. Wing

September 1989

Technical Report

CMU/SEI-89-TR-034
ESD-TR-89-045
September 1989

Durra: A Task-Level Description Language Reference Manual (Version 2)



Mario R. Barbacci
Jeannette M. Wing

Software for Heterogeneous Machines Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

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.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

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

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX: (304) 284-9001 / World Wide Web: http://www.asset.com/sei.html / e-mail: webmaster@www.asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Table of Contents

List of Figures

List of Tables

Durra: A Task-Level Description Language Reference Manual

Abstract: Durra is a language designed to support the development of large-grained parallel programming applications. These applications are often computation-intensive, or have real-time requirements that require efficient concurrent execution of multiple tasks, devoted to specific pieces of the application. During execution time the application tasks run on possibly separate processors, and communicate with each other by sending messages of different types across various communication links. The application developer is responsible for prescribing a way to manage all of these resources. We call this prescription a *task-level application description*. It describes the tasks to be executed, the possible assignments of processes to processors, the data paths between the processors, and the intermediate queues required to store the data as they move from source to destination processes. Durra is a *task-level description language,* a notation in which to write these application descriptions.

This document is a revised version of the original reference manual [2]. It describes the syntax and semantics of the language and incorporates all the language changes introduced as a result of our experiences writing task and application descriptions in Durra. A companion document, *Durra: A Task-Level Description Language User's Manual* [7], describes how to use the compiler, runtime environment, and support tools.

1. Introduction

Durra, also called ''Indian millet'' and ''Guinea corn,'' is a type of grain sorghum with slender stalks, widely grown in warm dry regions. Durra sounds like ''durable'' which isn't a bad connotation. Carnegie Institute personnel indicated that corn is by far the largest in size of all grains. We respectfully declined their suggestion for a name denoting ''largest grain.''

Many computation-intensive, real-time applications require efficient concurrent execution of multiple tasks (independent programs) devoted to specific pieces of the application. Typical tasks include sensor data collection, obstacle recognition, and global path planning in robotics and vehicular control applications. Since the speed and throughput required of each task may vary, these applications can best exploit a computing environment consisting of multiple special and general purpose processors that are logically, though not necessarily physically, loosely connected. We call this environment a *heterogeneous machine*.

During execution time, processes, which are instances of tasks, run on possibly separate processors, and communicate with each other by sending messages of different types. Since the patterns of communication can vary over time, and the speed of the individual processors can vary over a wide range, additional hardware resources, in the form of switching networks and data buffers, are required in the heterogeneous machine.

The application developer is responsible for prescribing a way to manage all of these resources. We call this prescription a *task-level application description*. It describes the tasks to be executed, the possible assignments of processes to processors, the data paths between the processors, and the intermediate queues required to store the data as they move from source to destination processes. A *task-level description language* is a notation in which to write these application descriptions. The problem we are addressing is the design of a task-level description language.

We are using the term *description language* rather than *programming language* to emphasize that a task-level application description is not translated into object code of some kind of executable machine language. Rather, it is to be understood as a description of the structure and behavior of a logical machine, that will be synthesized into resource allocation and scheduling directives. These directives are to be interpreted by a combination of software, firmware, and hardware in a heterogeneous machine.

Although our ultimate goal is to design and implement a task-level description language that can be used for different machines and for varying applications, this implementation was influenced by both a specific architecture, Nectar [1], and by a specific application, the autonomous land vehicle (ALV), and more specifically, by the perception components of the ALV [12]. We assumed a logical machine with a cross-bar switch, intelligent buffers on the switch sockets, and a executive that can communicate with all processors, buffers, and I/O devices. Actual experience with the implementation of the language and runtime environment and building demonstration applications [4, 5, 6, 8, 13, 9] have resulted in a number of changes to the language and these changes are reflected in the revised version of this language reference manual.

1.1. Scenario

The following is a scenario from the user's viewpoint of how the task-level language is used to help develop an application to run on some target, heterogeneous machine. We see three distinct phases in the process: (a) the creation of a library of tasks, (b) the creation of an application description, and (c) the execution of the application.

Library creation activities

These happen early in the life of an application, when the primitive tasks are defined.

1. The developer breaks the application into specific tasks. Typical tasks are sensor processing, feature recognition, map database management, and route planning. Other tasks might be of a more general nature, such as sorting, array operations, etc.

2. The developer writes *task implementations* (i.e., programs). For a given task, there may be possibly many implementations, differing in programming language (e.g., C, Lisp, Ada), processor type (e.g., Motorola 68020, DEC VAX), performance characteristics, or other attributes. The writing of a task implementation is more or less independent of Durra and involves the coding, debugging, and testing of programs in various languages executing on various machines.
3. The developer writes *task descriptions* and compiles them using the Durra compiler. This is where Durra first enters the picture. Durra is used to write specifications of each task implementation's performance and functionality, the types of data it produces or consumes, and the ports it uses to communicate with other tasks. If a compilation is successful, the compiler enters the task description into the Durra library.

Description creation activities

These happen when the user decides to put together an application (say, an autonomous land vehicle) using as building blocks tasks stored in the Durra library.

1. The user writes a *task-level application description*. Syntactically, a task-level application description is a single task description and could be stored in the library as a new task. This allows writing hierarchical task-level application descriptions.
2. The user compiles the description. During compilation, the Durra compiler retrieves from the library task descriptions matching the *task selections* specified by the user and generates a set of resource allocation and scheduling commands to be interpreted by the Durra excutive. These commands constitute a *Durra executive program*.

Application execution activities

1. The executive interprets the resource allocation and scheduling commands. See [6] for details about the runtime environment of Durra.
2. The heterogeneous system runs the processes on processors as dictated by the executive program.

1.2. Terminology

Durra is used for describing process interaction at a logical, not physical, level, and thus it can be used independently of any physical configuration of an actual heterogeneous machine. We will use different terms to distinguish between the physical network (P) of processors, memories, and switches implementing the heterogeneous machine, and the logical network (L) of processes and data queues implementing the application (A). The following are terms used in this manual.

buffer (P) — A computer acting as input or output device, interfacing a processor with the switch. As an optimization, buffers execute predefined tasks and data transformations.

implementation (A) Code written in some programming language for a specific processor that satisfies the performance, functional, and other requirements specified in a task description.

port (L) A logical input or output device of a process. Input ports remove data from queues; output ports deposit data in queues.

process (L) A uniquely identifiable instance of a task, running on a processor of the heterogeneous system. The same task may be instantiated any number of times to obtain multiple processes executing the same code.

processor (P) A computer in the heterogeneous system, not to be confused with the executive processor or the buffers. Each processor in the heterogeneous system has one or two buffers that act as interfaces between the processor and the switch. Processors send data to and receive data from buffers as their means of communication with other processors.

queue (L) A uniquely identifiable logical link between two processes, following a FIFO discipline. Queues serve as intermediaries between input and output ports.

executive (P, L) A computer serving as resource allocator and dispatcher in the heterogeneous system. It controls the switch, all processors, and all buffers.

switch (P) An interconnection network used to tie together all processors in the heterogeneous system. The switch routes data between the buffers attached to the processors.

task (L, A) An abstraction of a set of implementations, each written for a class of processors, implementing part of an application. Tasks are stored in libraries.

The processes of the system are implemented by downloading and executing task implementations, i.e., programs, onto processors of the right kind. The queues of the system are implemented by allocating space in the corresponding buffers' memories. This is illustrated in Figure 1.

1.3. Notes on Syntax

To describe the syntax of Durra, we use standard Backus-Naur-Form (BNF), with the following conventions.

1. Vertical bars (`|`) separate alternative productions. Braces (`{}`) indicate optional components of a production.
2. Terminal symbols are enclosed in quotation marks (`''''`), but the quotation marks do not belong to the terminal.
3. No distinction is made between upper and lower case letters in terminals and non-terminals.
4. A non-terminal of the form xyz_List$_{comma}$ stands for a list of one or more xyz's separated by commas, i.e., the character (`,`), not the string `comma`.

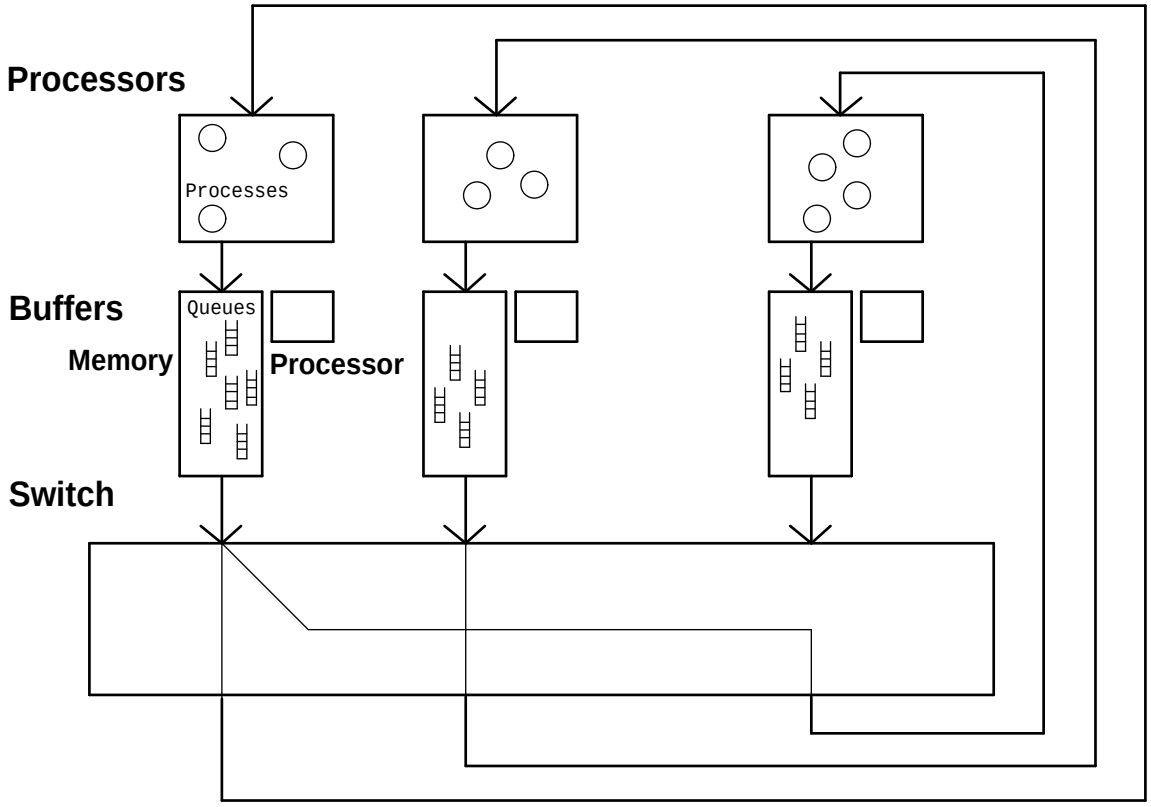


Figure 1: Mapping of Logical and Physical Components

5. Comments start with a double hyphen (`--`). Any text between the double hyphen and the end of the line is ignored.
6. Identifiers are sequences of letters, digits, and underscores (_). An identifier must begin with a letter and end with a letter or a digit. Consecutive underscores in the middle of an identifier are not allowed.
7. Strings are sequences of ASCII printable characters, enclosed in double quotation marks ("). A double quotation mark inside a string must be written as two consecutive double quotation marks:

```
"A string with a double quotation mark, "", inside"
```

8. Integer and real numbers are always decimal, i.e., base 10. A real number can terminate with a period (`.`) without a fractional part.

1.4. Keywords and Predefined Identifiers

Keywords and predefined identifiers are highlighted in normal text by writing them in **bold face**, or inside quotation marks () respectively. The following words are keywords in the language: **after**, **and**, **array**, **atime**, **attribute**, **attributes**, **before**, **behavior**, **bind**, **binds**, **dtime**, **during**, **end**, **ensures**, **exit**, **if**, **in**, **is**, **loop**, **not**, **of**, **or**, **out**, **port**, **ports**, **process**, **processes**, **ptime**, **queue**, **queues**, **reconfiguration**, **record**, **reconnect**, **reconnects**, **remove**, **repeat**, **requires**, **size**, **structure**, **task**, **then**, **timing**, **to**, **type**, **union**, **when**.

Several keywords exist in both singular and plural form and have the same meaning in either form (process/processes, port/ports, queue/queues, bind/binds, and attribute/attribute).

The following words are predefined identifiers in the language: ‘‘broadcast’’, ‘‘current_atime’’, ‘‘current_dtime’’, ‘‘current_ptime’’, ‘‘current_size’’, ‘‘deal’’, ‘‘debug’’, ‘‘delay’’, ‘‘dequeue’’, ‘‘enqueue’’, ‘‘implementation’’, ‘‘merge’’, ‘‘minus_time’’, ‘‘mode’’, ‘‘null’’, ‘‘plus_time’’, ‘‘processor’’, ‘‘signal’’, ‘‘source’’, ‘‘xdisplay’’, ‘‘xwindow’’.

1.5. Local and Global Names

In a task description the writer declares the names of task components such as ports, atributes, queues, etc. These names are local and unique within a task. Outside a task, its components are identified by a global name obtained by prefixing the name of a process (instance of the task) to the name of the component. For example, ‘‘p1.out2’’ could be the name of some output port declared inside process ‘‘p1.’’ If necessary, multiple process names as prefixes can be used. For example, ‘‘p1.p2.name’’ is the global name for some component ‘‘name’’ declared inside the process ‘‘p2’’ declared inside the process ‘‘p1.’’

```
TypeName          ::= Identifier
FieldName         ::= Identifier
TaskName          ::= Identifier
PortName          ::= Identifier
GlobalPortName    ::= {ProcessName_List(period)‘‘.’’} PortName |
                      ‘‘NULL’’
AttrName          ::= Identifier
GlobalAttrName    ::= {ProcessName_List(period)‘‘.’’} AttrName
QueueName         ::= Identifier
GlobalQueueName   ::= {ProcessName_List(period)‘‘.’’} QueueName
ProcessName       ::= Identifier
GlobalProcessName ::= ProcessName_List(period)
ProcessQueueName  ::= GlobalProcessName |
                      GlobalQueueName
```

1.6. Literal Values

Each of the non-terminals IntegerValue, RealValue, StringValue, and TimeValue stands for (a) literals (constants) of the appropriate kind, or (b) names of attributes (Section 7) whose values are literals of the appropriate kind, or (c) calls to predefined functions returning values of the appropriate kind. The predefined functions are described in Appendix B.a.

```
IntegerValue      ::= IntegerLiteral |
                      GlobalAttrName |
                      FunctionCall

RealValue         ::= RealLiteral |
                      GlobalAttrName |
                      FunctionCall

StringValue       ::= StringLiteral |
                      GlobalAttrName |
                      FunctionCall

TimeValue         ::= TimeLiteral |
                      GlobalAttrName |
                      FunctionCall

Value             ::= IntegerValue |
                      RealValue |
                      StringValue |
                      TimeValue

FunctionCall      ::= FunctionName { FunctionParameters }

FunctionName      ::= Identifier

FunctionParameters    ::= ‘‘(’’ Value_List_comma ‘‘)’’
             -- The type and number of parameters is function dependent.
```

1.7. Expressions

Boolean expressions are used in the language to denote two kinds of conditions or predicates: attribute values and reconfiguration conditions. Expressions used to specify attribute values in a task selection (Sections 7 and 8.1) are evaluated at compile time because their values are used by the Durra compiler to identify tasks from the library. Expressions used to specify reconfiguration conditions in a task description (Section 8.5) are evaluated at execution time.

```
Expression        ::= Disjunction

Disjunction       ::= Conjunction |
                      Disjunction ‘‘OR’’ Conjunction

Conjunction       ::= Primary |
                      Conjunction ‘‘AND’’ Primary

Primary           ::= {‘‘NOT’’} Term

Term              ::= Relation |
                      ‘‘(’’ Disjunction ‘‘)’’

Relation          ::= Value ‘‘=’’ Value |                      -- Equal
                      Value ‘‘/=’’ Value |                  -- Not equal
                      Value ‘‘>’’ Value |                     -- Greater
                      Value ‘‘>=’’ Value |           -- Greater or equal
                      Value ‘‘<’’ Value |                        -- Less
                      Value ‘‘<=’’ Value                -- Less or equal
```

1.8. Compilation Units

There are two kinds of compilation units (i.e., separately compilable units): type declarations and task descriptions.

```
CompilationUnit   ::= TypeDeclaration |
                      TaskDescription
```

Each compilation unit must be submitted to the Durra compiler as a single text file. If no errors are detected, the unit is entered into the library.

2. Type Declarations

Syntax:

```
TypeDeclaration   ::= ‘‘TYPE’’ TypeName ‘‘IS’’ TypeStructure |
                      ‘‘TYPE’’ TypeName ‘‘IS’’ UnionStructure

TypeStructure     ::= ‘‘SIZE’’ ElementSize |
                      ‘‘ARRAY’’ ‘‘OF’’ TypeName |
                      ‘‘ARRAY’’ ArrayDimension ‘‘OF’’ TypeName |
                      ‘‘RECORD’’ ‘‘(’’ Field_List_comma ‘‘)’’

ArrayDimension    ::= ‘‘(’’ IntegerValue_List_space ‘‘)’’
                                              -- Positive integers

ElementSize       ::= IntegerValue |
                                          -- Positive number of bits
                      IntegerValue ‘‘TO’’ IntegerValue
                                          -- Non-negative size range

Field             ::= FieldName : TypeName

UnionStructure    ::= ‘‘UNION’’ ‘‘(’’ TypeName_List_comma ‘‘)’’
```

Examples:

```
type packet is size 128 to 1024;
                                  -- Packets are of variable length
type tails is array (5 10) of packet;
                              -- Tails are 5 by 10 arrays of packets
type mix is union (heads, tails);
                                -- Mix data could be heads or tails
type rec is record (header: integer, size: integer, data: packet);
            -- Recs have two integers and one packet, in that order.
```

Meaning:

Type declarations are compilation units that define the structure of the data produced or consumed by the tasks. A type declaration introduces a global name for a data type, or a set of previously declared types, which can then be used in port declarations.

There are several kinds of type declarations. The simplest data type is a sequence of bits of fixed or variable (but bound) length. More complex types are declared as multi-dimensional arrays and records of simpler types. An array type declaration that omits the dimensions represents an array of unknown dimensionality. A record type declaration describes the structure of data objects by listing the field names and their types. Finally, a type declaration can specify the union of a number of previously declared, i.e., named, types where data items moving through a process port could be one of any of the member types.

3. Task Descriptions

Syntax:

```
TaskDescription   ::= ‘‘TASK’’ TaskName
                          InterfacePart
                              {BehaviorPart}
                                  {AttributeDescPart}
                                      {StructurePart}
                                          ‘‘END’’ TaskName
```

Meaning:

Task descriptions are compilation units that define the properties of task implementations (i.e., user programs). Task descriptions are used as building blocks for task-level application descriptions.

A task description is divided into four components: (1) interface information, (2) behavioral information, (3) attributes, and (4) structural information. All these components will be described in later sections. Figure 2 shows a template for a task description, where the **port** clause constitutes the interface information.

---

task *task-name*
  port -- REQUIRED
    port-declarations
    -- A description of the input/output interface of the task

  behavior -- OPTIONAL
    function-predicates
    timing-expressions
    -- A description of the behavior of the task

  attribute -- OPTIONAL
    attribute-value-pairs
    -- A description of additional properties of the task

  structure -- OPTIONAL
    process-declarations
    queue-declarations
    binding-declarations
    reconfiguration-statements
    -- A description of the internal structure of the task
end *task-name*;

Figure 2: A Template for Task Descriptions

---

4. Task Selections

Syntax:

```
TaskSelection     ::= ‘‘TASK’’ TaskName
                          {InterfacePart}
                              {BehaviorPart}
                                  {AttributeSelPart}
                                      {‘‘END’’ TaskName}
```

Meaning:

Task selections are templates used to identify and retrieve task descriptions from the library.

A given task, e.g., convolution, might have a number of different implementations that differ along dimensions such as algorithm used, code version, performance, or processor type. In order to select among a number of alternative implementations, the user provides a task selection as part of a process declaration, as described in Section 8.1. This task selection lists the desirable features of a suitable implementation.

Syntactically, a task selection looks somewhat like a task description without the **structure** part, and all other components except for the task name are optional. For example, notice that in the syntax of a task declaration, the interface part (Section 5) requires the declarations of the ports, whereas in a task selection, the declaration of the ports is optional. Figure 3 shows a template for a task selection. For brevity, if only the task name is given, the terminating ‘‘**end** *task-name*’’ is optional.

---

task *task-name* -- REQUIRED
  port -- OPTIONAL
    port-declarations
    -- A signature that must match port directions and types of
    -- that of a task description in the library.

  behavior -- OPTIONAL
    function-predicates
    timing-expressions
    -- A specification of the desired functional and timing
    -- behavior of a task description in the library.

  attribute -- OPTIONAL
    attribute-expression
    -- An expression of named, actual attribute values used to match
    -- formal attribute values of a task description in the library.
end *task-name* -- optional if only the task name is specified

Figure 3: A Template for Task Selections

---

5. Interface Information

5.1. Port Declarations

Syntax:

```
InterfacePart     ::= ''PORT'' PortDeclaration_List(semicolon)'';''

PortDeclaration   ::= PortName_List(comma) '':'' ''IN'' TypeName |
                      PortName_List(comma) '':'' ''OUT'' TypeName
```

Examples:

```
ports
  in1: in heads;
  out1, out2: out tails;
```

Meaning:

The interface portion of a task description or a task selection provides information about the ports of the processes instantiated from the task. A port declaration specifies the direction of the data movement and the type of data moving through the port.

5.2. Rules for Matching Selections with Descriptions

If a task selection provides a port declaration clause, the port declarations must be identical, i.e., names, directions, and types of the ports must be identical.

6. Behavioral Information

Syntax:

```
BehaviorPart      ::= ‘‘BEHAVIOR’’
                        { ‘‘REQUIRES’’ ‘‘"’’predicate‘‘"’’ ‘‘;’’ }
                            { ‘‘ENSURES’’ ‘‘"’’predicate‘‘"’’ ‘‘;’’ }
                                { ‘‘TIMING’’ TimingExpression ‘‘;’’ }

predicate         ::= Larch Predicate
```

Meaning:

The behavior part of a task description specifies functional and timing information about the task.

The functional information consists of a pre-condition (**requires**) on what is required to be true of the data coming from the input ports, and a post-condition (**ensures**) on what is guaranteed to be true of the data going to the output ports. The timing information part consists of a timing expression (**timing**) describing the behavior of the task in terms of the operations it performs on its input and output ports.

The formal meaning of the behavioral information is essentially based on first-order logic. In what follows, we give only an informal introduction to the individual parts and their combination. Appendix A provides the formal details.

6.1. Function Part

The functional information of a task description describes the behavior of the task in terms of predicates about the data in the queues, before and after each execution cycle of the task. The Larch Shared Language [10] is used as the assertion language in the predicates of these clauses.

We use a similar approach as Larch’s for the specification of the functional behavior of a task. That is, we view the task as a procedure whose input and output ‘‘parameters’’ are defined by the ports of the task. If one were to view each cycle of a task as one execution of a procedure, the **requires** and **ensures** are exactly the pre- and post-conditions on the functionality of that cycle. An omitted predicate is taken to be **true**.

These are not assertions about the queues connected to the ports. For instance, an assertion could be made that a datum of some type was sent to an output port. It cannot be asserted that the datum is in the associated output queue at the end of the task execution, because it could have been removed by then.

It is up to the implementor of a task to verify that the functionality of the task satisfies the **requires** and **ensures** predicates. A task description writer and user may assume that the task implementor performed such verification either formally or informally.

6.2. Timing Part

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.

6.2.1. Time Literals

Syntax:

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

Seconds           ::= IntegerValue |
                      RealValue

TimeBase          ::= ‘‘DTIME’’ |             -- Time since start of day
                      ‘‘ATIME’’ |     -- Time since start of application
                      ‘‘PTIME’’ |         -- Time since start of 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.

An absolute time value using ‘‘DTIME’’ cannot exceed 86400, the number of seconds in a day.

6.2.2. 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 a timing expression. An event 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.

6.2.3. Timing Expressions

Syntax:

```
TimingExpression  ::= {‘‘LOOP’’} SequentialEvent

SequentialEvent   ::= ParallelEvent_List(spaces)

ParallelEvent     ::= BasicEvent_List(double_vertical_bar)

BasicEvent        ::= Event |
                      {Guard ‘‘=>’’} ‘‘(’’ 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
          Tmax 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 optional 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 a double vertical bar || 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. If the time base is ‘‘PTIME’’ or ‘‘ATIME’’ and the deadline has elapsed, the task is terminated. If the time base is ‘‘DTIME’’ and the deadline has elapsed, the task waits until the end of the day (when the ‘‘time-of-day’’ resets to 0)
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 times allowed and can be an absolute time value or a time value relative to the former. If the time base of the latest start time is ''PTIME'' or ''ATIME'' and the deadline has elapsed, the task is terminated.

when This guard describes what is required to be true of the state of the system (e.g., time values and queues sizes) before the sequence is allowed to start. It is a pre-condition for starting the sequence.

6.2.4. 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}.

6.3. Rules for Matching Selections with Descriptions

The meaning of the behavioral information is a predicate, $M_f(R, T) => M_f(E, T)$, where R is the **requires** predicate, E is the **ensures** predicate, T is the **timing** expression, and M_f is the meaning function mapping a predicate and timing expression into a Boolean [3].

A task description matches a task selection if the predicate associated with the behavioral information of the task description implies that of the task selection. If no timing expression appears, the predicate simplifies to R => E, and that of a task description must imply that of the task selection.

7. Attributes

Syntax:

```
AttributeDescPart ::= ‘‘ATTRIBUTE’’ Attribute_List_semicolon ‘‘;’’

Attribute         ::= AttrName ‘‘=’’ AttrValue |
                      AttrName ‘‘=’’ ‘‘(’’ AttrValue_List_comma ‘‘)’’

AttributeSelPart  ::= ‘‘ATTRIBUTE’’ Expression‘‘;’’
```

Examples:

```
                                        -- Attributes in a task declaration
    attributes
      author = "jmw";
      color = ("red", "white", "blue");
      implementation = "cowcatcher";
      queue_size = 25 ;

                                             --Attributes in task selections
    attributes
      processor = Sun;

    attributes
      author = "jmw" or author = "mrb";

    attributes
      (color = "red" or color = "blue") and not (author = Mr_Ed);
```

Meaning:

Attributes specify miscellaneous properties of a task. They are a means of indicating pragmas or hints to the Durra compiler and the Durra executive. In a task description, the developer of the task lists the possible values of a property; in a task specification, the user of a task writes an expression listing the desired name/values of the task’s property. All attribute values used in matching task selections with task descriptions must be constants, computable before execution time, i.e., tasks and their implementations are static properties of an application.

Example attributes include: author, version number, programming language, file name, and processor type. There may be as many attributes as desired.

The name of an attribute can appear in any context in which its value can appear. For instance, if the user defines an attribute `queue_size` with an integer value, as in the examples above, then `queue_size` can appear anywhere an integer value is expected. This permits the user to define, for example, the size of message queues and use the name to specify ‘‘families’’ of tasks, i.e., tasks that share the same value for some attribute, as shown in Figure 4.

The syntax and semantics of the attribute values are attribute dependent. If the attribute is not predefined in the language, the values are treated as uninterpreted numbers, time

```
processes
  Master_Process: task Master_Task
                                                    -- A task selection
        attributes
          Key_Name = some value;
          ... other attributes, maybe ...
        end Master_Task;

  p1: task foo
        attributes
        Key_Name = Master_Process.Key_Name;
                      -- Same attribute value as in Master_Process
      end foo;

  p2: task bar
        attributes
        Key_Name = Master_Process.Key_Name;
                      -- Same attribute value as in Master_Process
      end bar;
```

Figure 4: Use of Global Attribute Names

values, or strings, as the case may be, and compatibility is based on value equality. If the attribute is predefined in the language, compatibility is attribute dependent. See Appendix B.b for details about the predefined attributes.

7.1. Rules for Matching Selections with Descriptions

If a task selection includes an attribute expression (a predicate), a matching task description must provide values that satisfy the predicate, i.e., the expression yields **true** when evaluated in the context of the values declared for the attribute. If a task description provides a list of values for an attribute, a matching task selection is satisfied if any of these values satisfies the expression. For example, if a task description includes an attribute ‘‘size’’ with values (1, 3, 5) then a task selection with an attribute expression of the form ‘‘size > 4’’ is satisfied by the task description because the value ‘‘5’’ makes the attribute expression **true**.

If a task selection includes an attribute expression (a predicate) that uses an attribute not present in a task description or if a task description provides attributes not used in the attribute expression of a task selection, these attributes are ignored for the purposes of matching a library task.

8. Structural Information

Syntax:

```
StructurePart ::= ‘‘STRUCTURE’’ StructureClause_List(space)

StructureClause   ::= ‘‘PROCESS’’ ProcessDeclaration_List(semicolon) ‘‘;’’ |
                      ‘‘QUEUE’’ QueueDeclaration_List(semicolon) ‘‘;’’ |
                      ‘‘RECONNECT’’ QueueReconnection_List(semicolon) ‘‘;’’|
                      ‘‘BIND’’ PortBinding_List(semicolon) ‘‘;’’ |
                      ‘‘RECONFIGURATION’’ Reconfiguration_List(semicolon) ‘‘;’’
```

Meaning:

Process and queue declarations appear under the keyword **structure** in a task description. These declarations define a graph in which processes are the nodes, and queues are the links. These graphs depict the internal structure of a compound task. The **structure** part of a task description provides the means for developing hierarchical task descriptions.

8.1. Process Declarations

Syntax:

```
ProcessDeclaration    ::= ProcessName_List(comma) ‘‘:’’ TaskSelection
```

Examples:

```
p1: task finder;
p2: task finder ports foo: in, bar: out end finder;
p3, p4: task finder attributes author="mrb" end finder;
```

Meaning:

An instance of a task is bound to each process’s name. The name of a task is the minimal part of a task selection. Local, actual names (e.g., ports `foo` and `bar` in the example) can be introduced by providing a port declaration, provided that the types of ports specified in the task declaration are identical to those provided in the task selection. If they are left out, the formal names used in the task description are used instead.

8.2. Queue Declarations

Syntax:

```
QueueDeclaration  ::= QueueName {QueueSize} ‘‘:’’ QueueDefinition

QueueReconnection ::= QueueName ‘‘:’’ QueueDefinition

QueueDefinition   ::= GlobalPortName
                          ‘‘>’’ Transformation ‘‘>’’
                              GlobalPortName

QueueSize         ::= ‘‘[’’ IntegerValue ‘‘]’’
```

Examples:

```
    queue q1: p1 > > p2 ;
                  -- Two ports connected through a new, unbounded queue.
                                  -- The two ports must have the same type.

    reconnect q1: p1 > > p2 ;
             -- Two ports connected through a previously declared queue.
                                  -- The two ports must have the same type.

    queue q2[100]: p1 > xyz > p2 ;
            -- Two ports connected through a bounded (size = 100) queue.
                  -- Data are transformed in the queue by a process xyz.

    queue q3: p1 > > NULL ;
                       -- A queue with a ‘‘grounded’’ destination port.
```

Meaning:

A queue definition establishes a logical link between two ports that communicate by passing data from the first port (source) to the second port (destination). A queue is not permanently attached to the ports specified in the queue’s declaration and can be reconnected (i.e., attached to other ports via the reconnect statement) in the structure clause of a reconfiguration statement (Section 8.5).

The (optional) queue bound in a queue declaration specifies the maximum number of elements that will be stored in the queue at any one time. If a queue is reconnected, the original, declared queue bound is retained. If a queue is full when a ‘‘enqueue’’ operation is attempted, the process trying to store the data waits until the queue has space for the new item. If the queue bound is not provided, a configuration-dependent, default queue length is assumed, as described in [6, 7].

The predefined port name ‘‘NULL’’ can be used to declare or reconnect a queue to an idle source or destination port. That is, to a ‘‘port’’ that will not produce or consume any data. This feature is useful to plug unused task ports by connecting them to NULL ports.

When declaring or reconnecting a queue, the ports are checked for type compatibility. Non-union types are compatible if they have the same name. Union types are compatible if the source set is a subset of the destination set. A non-union source type is compatible with a union destination type if the source type name is a member of the destination set.

If the types are not compatible, the user must provide a data transformation operation that will convert objects of one type into the other, as described below.

A reconnect statement can not specify a queue size or a transformation. These are inherited from the original queue declaration.

8.3. Data Transformations

Data transformations are operations applied to data coming from a source port in order to make them acceptable to a destination port. A data transformation is required if the input and output port types are not compatible. Such transformations are needed if, for instance, the types have the same structure but the data are in the wrong format, e.g., turning a square array on its side or converting between floating-point formats.

Transformations must be written as separate tasks and instantiated as processes. The processes can then be used to build transformation expressions in the queue declaration. Furthermore, since the data can be of a structured type, there can be transformations that apply to a whole datum (e.g., corner-turning an array) or to the subcomponents (e.g., rounding and converting to integers the elements of an array of floating point numbers).

Syntax:

```
Transformation    ::= TransformOp_List(space)

TransformOp       ::= ArrayTransform |
                      RecordTransform |
                      ScalarTransform

ArrayTransform    ::= TransformName {''(''''Transformation'''')''}

RecordTransform   ::= TransformName {''(''FieldTransform_List(space)'')''}

FieldTransform    ::= FieldName'':'' ''(''''Transformation'''')''

ScalarTransform   ::= TransformName

TransformName     ::= GlobalProcessName |
                      NULL
```

Examples:

Let's assume we have implemented tasks that perform a few useful transformations such as transposing an array and converting floating point numbers into integers. In order to use these tasks, we need to instantiate them in the usual manner (assume that no other information, e.g., attributes, is needed to select the tasks):

```
process transpose_process = task transpose;

process fix_process = task fix;
```

Assuming that some source port produces arrays of some shape, with elements of type float, and that a destination port consumes transposed versions of the same arrays. We could apply the first transformation process as follows:

```
queue
    q1: source_port_name > transpose_process > destination_port_name
```

The process `transpose_process` will be applied to each array after it is placed in the queue and before it is delivered to the destination.

Now assume that the destination port consumes arrays whose shape is that of the transposed version of the same input array but whose elements are of type `fix`. We could apply both transformation processes as follows:

```
queue
    q1: sport > transpose_process (fix_process) > dport
```

The process `transpose_process` will be applied to each array after it is placed in the queue, and then process `fix_process` will be applied to each element of the transformed array, before the result is delivered to the destination.

Sometimes it is necessary to apply a transformation to the elements of an array but without changing the array itself. This can be achieved by specifying the null transformation process NULL as the array transformation, followed by the desired element transformation process:

```
queue q1: sport > NULL (fix_process) > dport
```

If we need to apply more than one transformation to the elements of an array, this can be specifed by listing a sequence of transformations:

```
queue q1: sport > t1 (t11 t12) > dport
```

After applying `t1,` each element is transformed by applying `t11` and `t12,` in that order.

We can apply rather complicated sequences of transformations, as the following example suggests:

```
queue q1: sport > t1 (t11 t12) t2 (t21 (t211 t212) ) > dport
```

Transformation `t1` produces objects of some array type, whose elements will be transformed by `t11` and `t12.` The result at this point is an array with the same structure as that produced by `t1` but whose elements are of whatever type `t12` produces. Transformations can be nested as shown in the example. Transformation `t2` produces some array type whose elements (scalars or arrays) are transformed by `t21` which produces some array type whose elements are transformed by the sequence `t211` and `t212.`

Record transformations follow a similar pattern, except that since not all ‘‘components’’ (i.e., fields) are of the same type, we need to specify, in addition to whole-record transformations (if any), specific field transformations:

```
queue q1: sport > t1 (head: (t11 t12) tail: (t2) ) > dport
```

The example illustrates a queue connecting a source port to a destination port. The source port generates data of some type suitable as input to a transformation process `t1`. The output of `t1` is some record type with fields `head` and `tail` which are transformed independently of each other. The `head` field is transformed by the sequence `t11` and `t12`, while the `tail` field is transformed by `t2`. The records produced by `t1` could have additional fields and these are not disturbed. The null transformation ‘‘NULL’’ could be used when it is necessary to transform one or more fields of a record but without applying any transformation to the record as a whole:

```
queue q1: sport > NULL ( head: (t1) tail: (t2) ) > dport
```

Meaning:

A sequence of data transformation expressions is convenient short-hand for a series of anonymous, transformationless queue declarations connecting the transformation processes.

A data transformation is a way to achieve compatibility between the data produced by some source port and the data expected by some destination port. The definition of compatibility between types depends on the types:

1. Non-union types are compatible if they have the same name.
2. Union types are compatible if the set of alternative type names in the source type is a subset of the set of alternative type names in the destination type.
3. A non-union source type is compatible with a union destination type if the source type name is a member of the set of alternative type names in the destination type.
4. Array types produced by a chain of transformations are compatible with the destination port type if the two array structures are identical: they have the same shape, size, and element type.
5. Record types produced by a chain of transformations are compatible with the destination port type if the two record structures are identical: they have the same number of fields, in the same order, and of the same type. It is not required that the field names be identical.

A data transformation operation also serves to specify operations that would be inappropriate or inefficient if written as part of one of the application’s tasks. For example, consider an application that requires scanning an array in different directions (e.g., first by rows, then by columns) and performing some operation on each element (e.g., computing the average of the neighbors). Rather than writing several versions of the task, one for each traversal pattern, one could simply write one version of the task, and then instantiate it as many times as necessary. Each process so instantiated could then take its input arrays from queues that perform the appropriate transposition, as in ‘‘q1: p1 > transpose > p2’’. Arrays produced by p1 are transposed while in the queue, before they are delivered to p2.

There are a two restrictions on the tasks that could be applied as data transformations. First, each task must have exactly one input port and one output port. Second, the port types must match all along the chain of transformations, such that data is ‘‘pipelined’’ through the sequence.

The same transformation process can be named in several transformation expressions. The user should think of them as if each use is a different instantiation. Whether or not there exists only one or more than one instantiation of the process is an implementation-dependent optimization.

8.4. Port Bindings

Syntax:

```
PortBinding       ::= ExternalPortName ‘‘=’’ InternalPortName

ExternalPortName  ::= PortName

InternalPortName  ::= GlobalPortName
```

Example:

```
binds
  in1 = p_deal.in1;
  out1 = p_merge.out1;
```

Meaning:

A port binding maps a port defined by an inner task to a port defined by an outer task.

8.5. Reconfigurations

Syntax:

```
Reconfiguration   ::= ReconfigurationLabel ‘‘:’’
                          ‘‘IF’’ Expression ‘‘THEN’’
                              ProcessQueueTermination
                                  StructureClause_List(space)
                                      {ReconfigurationExit}
                                          ‘‘END’’ ‘‘IF’’

ReconfigurationLabel  ::= Identifier

ProcessQueueTermination   ::= ‘‘REMOVE’’ ProcessQueueName_List(comma)‘‘;’’

ReconfigurationExit   ::= ‘‘EXIT’’ { ReconfigurationLabel }
                          ‘‘WHEN’’ Expression‘‘;’’
```

Examples:

```
reconfiguration
    L_1: if condition_to_enter_this_configuration then
         remove process_and_queue_names;
         process ...;                                  -- declare new processes
         queue ...;                                    -- declare new queues
         reconects ...;                                -- reconnect old queues
         exit L_1 when condition_to_exit_this_configuration;
         end if;
```

Meaning:

A reconfiguration statement is a directive to the Durra executive. It is used to specify changes in the current structure i.e., the process-queue graph of the application and the conditions under which these changes take effect. Typically, a number of existing processes and queues are substituted by new processes and queues, which are then connected to the remainder of the original graph. The reconfiguration condition is a Boolean expression involving time values, queue sizes, signal values, and other information available to the executive at run time.

More than one reconfiguration statement can be specified in a structure clause. The reconfiguration conditions are tested by the executive concurrently. If more than one of these conditions becomes true at the same time, the executive selects one of the reconfiguration statements and changes the structure of the application accordingly. This choice of alternative configuration is non-deterministic.

It is useful to think of the possible configurations of an application as the nodes of a tree. The root node corresponds to the initial structure of the application description. Each alternative reconfiguration statement in the structure part of the application description corresponds to the root of a subtree or alternative structure. Each of these, in turn, can have one or more reconfiguration statements of its own, and so on. The tree is traversed down when a reconfiguration condition is satisfied and traversed up when an exit condition is satisfied.

Multiple (i.e., nested) reconfiguration levels can be exited at once by writing the label of the outermost reconfiguration statement to be exited. The structure of the application changes reverts to the form it had when that reconfiguration took place. If no label follows the **exit** keyword, the immediately enclosing label is assumed.

The condition following the **when** keyword acts as an assertion attached to the current structure of the application. It can be used to terminate reconfigurations derived from it, as shown in Figure 5. If *condition_2* is ever satisfied, the structure of the application changes back to the form it had when reconfiguration L_1 took place, regardless of what reconfigurations (e.g., L_2a, L_2b, L_3) might have taken place later. In addition to these implicit terminations, the inner reconfiguration statements can also have termination statements of their own.

```
reconfiguration
   L_1: if condition_1 then
        remove .....;
        process .....;
        queue .....;
        reconfiguration
           L_2a: if ..... then
                 .....;
                 end if;
           L_2b: if ..... then
                 .....;
                 reconfiguration
                    L_3: if ..... then
                              .....;
                         end if;
                 end if;
        exit when condition_2;        -- Terminates L_2a, L_2b, L_3, etc.
        end if;
```

Figure 5: Nested and Alternative Reconfiguration Statements

The exit statement condition, if present, might never be satisfied, in which case only a parent configuration's exit statement can terminate the current reconfiguration.

References

[1] E.A. Arnould, F.J. Bitz, E.C. Cooper, H.T. Kung, R.D. Samson, and P.A. Steenkiste.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April, 1989.

[2] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3 (DTIC: ADA178975), Software Engineering Institute, Carnegie Mellon University, December, 1986.

[3] M.R. Barbacci and J.M. Wing.
Specifying Functional and Timing Behavior for Real-time Applications.
Lecture Notes in Computer Science. Volume 259, Part 2.*Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE).*
Springer-Verlag, 1987, pages 124-140.

[4] 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.

[5] M.R. Barbacci.
MasterTask: The Durra Task Emulator.
Technical Report CMU/SEI-88-TR-20 (DTIC: ADA199429), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[6] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18 (DTIC: ADA199480), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[7] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
Durra: A Task-Level Description Language User's Manual.
Technical Report CMU/SEI-89-TR-33, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[8] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, S.L. Baur, D.C. Bixler, M.T. Heins.
Command, Control, Communications, and Intelligence Node: A Durra Application Example.
Technical Report CMU/SEI-89-TR-9 (DTIC: ADA206575), Software Engineering Institute, Carnegie Mellon University, February, 1989.

[9] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, and J.M. Wing.
Developing Applications for Heterogeneous Machine Networks: The Durra Environment.
Computing Systems 2(1), March, 1989.

[10] J.V. Guttag, J.J. Horning, and J.M. Wing.
Larch in Five Easy Pieces.
Technical Report 5, DEC Systems Research Center, July, 1985.

[11] F. Jahanian and A.K. Mok.
Safety Analysis of Timing Properties in Real-Time Systems.
Transactions on Software Engineering 12(9):890-904, September, 1986.

[12] S.A. Shafer, A. Stenz, C.E. Thorpe.
An Architecture for Sensor Fusion in a Mobile Robot.
In *Proceedings of the International Conference on Robotics and Automation*, pages 2002-2011. IEEE Computer Society Press, San Francisco, California, April, 1986.

[13] C.B. Weinstock.
Performance and Reliability Enhancement of the Durra Runtime Environment.
Technical Report CMU/SEI-89-TR-8 (DTIC: ADA207445), Software Engineering Institute, Carnegie Mellon University, February, 1989.

Appendix A: Formal Meaning of Timing Expressions

We use Jahanian and Mok's Real-Time Logic (RTL) [11] to give meaning to our timing expressions. Furthermore, we use their logic to give meaning to the combination of our functional and timing specifications. We use four of their notational conventions:

Syntax	***Meaning***
↑A	The start of an operation (''action'' in RTL's terminology).
↓A	The end of an operation.
@(E, i)	The time of the i^{th} occurrence of event E, where events in our context are the start of an operation or the end of an operation. @ is an occurrence function that captures the notion of real-time.
P(t1, t2)	The interval of time during which the predicate P holds. P holds before or at t1, from t1 to t2, and at or after t2. If t1 and t2 are identical, then P holds at an interval around t1. For brevity, we will use P(t) when t1=t2 (i.e., ''P holds around time t'').

A.a. Assigning Meaning to Timing Specifications

In this section we describe the meaning of our timing specifications in terms of RTL logic. In the following discussion, we assume E, E1, and E2 are arbitrary timing expressions; A, A1, and A2 are operations; t1 and t2 are times (absolute or relative); a1 and a2 are absolute times; r1 and r2 are relative times; and W is a predicate of a **when** guard.

To simplify the exposition, we introduce a simple rewrite rule: Any timing expression of the form ''**repeat** *n* => E'' can be rewritten as a sequence of *n* occurrences of the unguarded expression E (''E E E ... E''). Thus, the only guards we need to consider are **before**, **after**, **during**, and **when**. Table A-1 gives the axioms that describe the start and end times of operations and composition of operations.

We assign a meaning to timing expressions by introducing a function, M_{tb} (Table A-2.a), which maps timing expressions to Boolean values:

$$M_{tb} : \text{Timing Expression} \rightarrow \text{Boolean}.$$

In the definition of M_{tb} we use an auxiliary function, M_{to} (Table A-2.b), which maps timing expressions to operations:

$$M_{to}: \text{Timing Expression} \rightarrow \text{Operation}.$$

M_{to} is needed because ''start time'' and ''end time'' are meaningful only for queue operations.

As an example of how to interpret the formalism intuitively, consider the entries for the **during** guard in Table A-2.a. This guard specifies a time window during which the operation is allowed to start. The first time value of the window is the earliest start time

1. For any queue operation A, and for some implementation defined time window [T1,T2], the following axiom expresses the (default) duration of the operation:

$$\forall\, i\ [\ T1 \le @(\downarrow A,i) - @(\uparrow A,i) \le T2\ ]$$

2. For any queue operation A[t1,t2], with a duration defined by the time window [t1,t2], the following axiom expresses the duration of the operation:

$$\forall\, i\ [\ t1 \le @(\downarrow A,i) - @(\uparrow A,i) \le t2\ ]$$

3. For any sequence of queue operations, A = A1 ... An, the following axiom relates the start and end times of the composition to the start and end times of the individual operations:

$$\forall\, i\ [@(\uparrow A, i) = @(\uparrow A1, i) \wedge @(\downarrow A, i) = @(\downarrow An, i)]$$

4. For any parallel queue operations, A = A1 || ...|| An, the following axiom relates the start and end times of the composition to the start and end times of the individual operations:

$$\forall\, i\ [@(\uparrow A, i) = \min(@(\uparrow A1, i), ..., @(\uparrow An, i)) \wedge @(\downarrow A, i) = \max(@(\downarrow A1, i), ..., @(\downarrow An, i))]$$

5. Cycles in a repeating task do not overlap. The following two axioms express that an input operation cannot finish after the last output operation finishes, and that an output operation cannot start before the earliest input operation starts:

$$\forall\, i\ [\max(@(\downarrow out_1,i),@(\downarrow out_2,i),...,@(\downarrow out_J,i)) > \max(@(\downarrow in_1,i),@(\downarrow in_2,i),...,@(\downarrow in_K,i))\ ]$$
$$\forall\, i\ [\min(@(\uparrow out_1,i),@(\uparrow out_2,i),...,@(\uparrow out_J,i)) > \min(@(\uparrow in_1,i),@(\uparrow in_2,i),...,@(\uparrow in_K,i))\ ]$$

where J and K are the number of output and input queues, respectively.

Table A-1: Axioms About Operation Start and End Times

allowed and must be an absolute time value. The second time value is the latest start time allowed and can be an absolute time value or a time value relative to the former. The meaning of the guarded expression is the conjunction of the meaning of the expression proper and a predicate stating the restriction on starting times.

A.b. Assigning Meaning to the Combined Specifications

Given a task description of the form:

Timing Expression	M_{tb}***(Timing Expression)***
E =	$M_{tb}(E)$ =
(E1)	$M_{tb}(E1)$
E1 ... En	$M_{tb}((E1\ E2)\ ...En)$
E1 ||...|| En	$\wedge\ M_{tb}(Ei \parallel Ej)$ for all $i \neq j$
E1 E2	$M_{tb}(E1) \wedge M_{tb}(E2) \wedge$ $\forall\ i\ [\ @(\downarrow M_{to}(E1), i) \leq @(\uparrow M_{to}(E2), i)\ ]$
E1 || E2	$M_{tb}(E1) \wedge M_{tb}(E2) \wedge$ $\forall\ i\ [@(\uparrow M_{to}(E1), i) < @(\downarrow M_{to}(E2), i) \wedge$ $@(\uparrow M_{to}(E2), i) < @(\downarrow M_{to}(E1), i)]$
when W => E1	$M_{tb}(E1) \wedge \forall\ i\ [\ W(@(\uparrow M_{to}(E1), i))\ ]$
before a1 => E1	$M_{tb}(E1) \wedge \forall\ i\ [\ @(\uparrow M_{to}(E1), i) \leq a1\ ]$
after a1 => E1	$M_{tb}(E1) \wedge \forall\ i\ [\ @(\uparrow M_{to}(E1), i) \geq a1\ ]$
during [a1, a2] => E1	$M_{tb}(E1) \wedge \forall\ i\ [\ a1 \leq @(\uparrow M_{to}(E1), i) \leq a2\ ]$
during [a1, r2] => E1	$M_{tb}(E1) \wedge \forall\ i\ [\ a1 \leq @(\uparrow M_{to}(E1), i) \leq a1 + r2\ ]$
A[r1, r2]	$\forall\ i\ [\ @(\uparrow A, i) + r1 \leq @(\downarrow A, i) \leq @(\uparrow A, i) + r2\ ]$
A[*, r1]	$\forall\ i\ [\ @(\downarrow A, i) \leq @(\uparrow A, i) + r1\ ]$
A[r1, *]	$\forall\ i\ [\ @(\uparrow A, i) + r1 \leq @(\downarrow A, i)\ ]$
A	true

a. M_{tb} -- Mapping from Timing Expressions to Booleans

Timing Expression	M_{to}***(Timing Expression)***
E =	$M_{to}(E)$ =
loop E1	$M_{to}(E1)$
E1 ... En	$M_{to}(E1)$ ... $M_{to}(En)$
E1 ||... || En	$M_{to}(E1) \parallel ... \parallel M_{to}(En)$
guard => E1	$M_{to}(E1)$ for all guards: **when**, **before**, **during**, **after**
A [t1, t2]	A
A	A

b. M_{to} -- Mapping From Timing Expressions to Operations

Table A-2: Assigning Meaning to Timing Specifications

```
task taskname
   . . . . .
   behavior
      requires Req ;
      ensures  Ens ;
      timing   E ;
   . . . . .
end taskname ;
```

we give meaning to the predicates of the functional specification as related to time (i.e.,

at what times are these predicates to hold?) via a function M_f (Table A-3) which maps from behavioral specifications to Boolean values:

$$M_f : \text{Predicate} \times \text{Timing Expression} \rightarrow \text{Boolean}$$

Pred.	***Expr.***	M_f***(Predicate, Timing Expression)***
Req	E	$M_f(\text{Req}, E) = \forall\, i\, [\text{Req}(@(\uparrow M_{to}(E), i)) \wedge M_{tb}(E)]$
Ens	E	$M_f(\text{Ens}, E) = \forall\, i\, [\text{Ens}(@(\downarrow M_{to}(E), i)) \wedge M_{tb}(E) \wedge$ **Consistent**$(\text{Ens}, E)]$

Table A-3: Assigning Meaning to Combined Specifications

The predicate **Consistent**(Ens, E) used in the definition of M_f checks to see if the **ensures** Ens predicate is meaningful with respect to the timing expression E. **Consistent** is defined by using two auxiliary predicates, **Uses** and **Depends**.

We define **Uses**:

Uses: element $\times$ input queue $\times$ output queue $\times$ Predicate $\rightarrow$ Boolean

such that for all input queues q_{in}, output queues q_{out}, elements in the output queues x:

Uses$(x, q_{in}, q_{out}, \text{Ens}) =$
true, if q_{in} *appearsIn* $x \wedge \text{Ens} \Rightarrow \text{isIn}(q_{out}, x)$;
false, otherwise.

UsesSet$(x, q_{out}, \text{Ens}) =$
$\{q_{in} \mid$ **Uses**$(x, q_{in}, q_{out}, \text{Ens})\}$ for all x such that $\text{isIn}(q_{out}, x)$

where ''a *appearsIn* b'' is a syntactic relation that checks if the text a occurs in the text b. Intuitively, **Uses** checks to see if the computation of x, the element enqueued on q_{out}, can be proven from the Ens to use any of the elements from q_{in}.

We define **Depends**:

Depends: element $\times$ input queue $\times$ output queue $\times$ Timing Expression $\rightarrow$ Boolean

such that for all input queues q_{in}, output queues q_{out}, elements in the output queues x, and for all $1 \leq i \leq \text{length}(q_{out})$ where $i^{th}(q_{out})$ is $\text{first}(\text{rest}^{i-1}(q_{out}))$:

Depends$(i^{th}(q_{out}), q_{in}, q_{out}, E) =$
true, if $E=E1\ q_{out}\ E2$ or $E=E1\ q_{out} \parallel E2$ or $E=E1 \parallel q_{out}\ E2$ and
q_{in} *appearsIn* E1 and q_{out} *appearsIn* E1 i-1 times;
false, otherwise.

DependsSet$(x, q_{out}, E) =$
$\{q_{in} \mid$ **Depends**$(x, q_{in}, q_{out}, E)\}$ for all x such that $\text{isIn}(q_{out}, x)$

Intuitively, **Depends** says that output elements can depend on only elements that were previously, or concurrently input.

We now define **Consistent**:

Consistent: Predicate × Timing Expression → Boolean

as follows:

Consistent(Ens, E)=
$\forall$ x, $\forall$ q_{out} [isIn(q_{out}, x) => (**UsesSet**(x, q_{out}, Ens) $\subseteq$ **DependsSet**(x, q_{out}, E))]

Intuitively, we check that each element x in each output queue depends on only elements that have been dequeued from input queues strictly before or concurrently with the enqueueing of x.

A.c. Examples

In the absence of a timing expression, we can perform standard first-order reasoning on a functional specification. For example, if the multiply task's **ensures** predicate had the additional conjunct, first($out1_{post}$) = first(in1), then by equational reasoning (substitution of equals by equals), we see that the **ensures** predicate is satisfiable only if first(in1) * first(in2) = first(in1).

In the absence of a functional specification, we can use the axioms and rules of RTL plus our extensions listed in Section A.a to determine inconsistent timing expressions. For example, if the expression is in1 out1 in2, we can apply axiom 5 of Section A.a to show that, for each task cycle, the end of the last input operation (in2) cannot follow the end of the last output operation (out1), thus invalidating the timing expression.

More interestingly, however, is to show how a combined specification can be proven inconsistent, where in fact, each separately is consistent and meaningful. For example, consider a task that merges data coming from two input into one output queue, as shown in Figure 6.

```
task merge
    ports
        in1, in2: in item;
        out1: out item;
    behavior
        ensures out1post=insert(insert(out1,first(in1)),first(in2));
        timing loop (in2 out1 in1 out1);
end merge;
```

Figure 6: Merge Task

The **ensures** clause specifies that the output queue's items be ordered such that the item from in1 is before that from in2, but the timing expression specifies that if the item from in1 is output on the queue out1, it must be the second, not first, item in the queue (here we assume that the output queue is initially empty.) This inconsistency can be formally proven:

UsesSet(first(out1), out1, Ens) = {in1}
DependsSet(first(out1), out1, E) = {in2}

Since the **UsesSet** is not a subset of the **DependsSet** for first(out1), **Consistent**(Ens, E) is false.

The example in Figure 7 illustrates why subsetting and not equality is used in the definition of **Consistent**. It also shows the use of the Ensures predicate and the need for equational reasoning about elements in a queue (see the second conjunct in the **Uses** predicate).

```
task divide
    ports
        a, b: in real;
        q, r: out real;
    behavior
        ensures first(q_post) * first(a) + first(r_post) = first(b)
        timing loop (a q b r);
end merge;
```

Figure 7: Divide Task

The **ensures** clause in the Divide task specifies that the quotient of b divided by a is in q and the remainder in r; however, the timing expression says that the computation of the quotient need depend on only what is in a, and not what is in b. This inconsistency can be formally proven since:

UsesSet(first(q), q, Ens) = {a, b}
DependsSet(first(q), q, E) = {a}

More specifically, to show **UsesSet**(first(q), q, Ens) = {a, b} we first note that:

Uses(*quotient*(first(a), first(b)), a, q, Ens) = true
Uses(*quotient*(first(a), first(b)), b, q, Ens) = true

since a and b both ‘‘appear in’’ the first argument (assume *quotient* is a trait operator for real numbers.)

Using equational reasoning on the Ens, we can show

first(q) = *quotient*(first(a), first(b))

By substitution, we get

Uses(first(q), a, q, Ens) $\wedge$ **Uses**(first(q), b, q, Ens)

yielding:

UsesSet(first(q), q, Ens) = {a, b}

Appendix B: Predefined Language Facilities

In this appendix we define the functions, attributes, and tasks that are built-in into the language and are implemented by the Durra compiler or the Durra executive.

B.a. Predefined Functions

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

Some functions compute the time elapsed from the beginning of the day, the start of the application, or the start of a process. Other functions perform computations with time values. Finally, there is one function that returns 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’’|
                      ‘‘SIGNAL’’

FunctionParameters    ::= ‘‘(’’ Value_List_comma ‘‘)’’
             -- The type and number of parameters is function dependent.
```

Examples:

```
    Plus_Time(Current_DTime DTIME, 9000)
          -- 2.5 hours later (i.e., 9,000 seconds from the current time)
    Current_Size(Master_Process.Data_Port)
                                  -- the size of a queue feeding a port
```

Meaning:

The functions with names like ‘‘current_?time’’ return the number of seconds (a real value) from the start of the day, the start of the application, or the start of a process, as suggested by their names.

The function call ‘‘current_ptime(*task_or_process_name*)’’ returns the elapsed time (in seconds) since the start of the current task (i.e. the task in whose description this function is used), or the start of a process instantiated inside the current task.

The function call ‘‘minus_time(*some_time_value*, *seconds*)’’ returns the time value ob-

tained by subtracting a number of seconds (a real or integer value) from some time value.

The function call ‘‘plus_time(*some_time_value*, *seconds*)’’ returns the time value obtained by adding a number of seconds (a real or integer value) to TimeValue.

It should be obvious that ‘‘current_ptime(*any_name*)’’ can never return a value larger than ‘‘current_atime’’. That is,

```
Current_PTime(any_name) <= Current_ATime
```

is always true, and the result of

```
Minus_Time(Current_ATime ATIME, Current_PTime(any_name))
```

is constant and known (by the Durra executive) at the time *any_name* starts. It is the delay between the start of the application and the start of the process.

The function call ‘‘current_size(*port_name*)’’ returns the current number of elements stored in the queue associated with a given port.

The function call ‘‘signal(*process_name*, *integer_value*)’’ returns **true** if the last signal raised by a process had a given value and returns **false** if no signals have been raised so far by the process, or if the last signal raised had a different value. The ‘‘signal’’ function can only appear as part of a conditional expression in a reconfiguration statement.

Signal values are integer numbers that have no prespecified meaning in the language. It is up to the application and task developers to adopt the appropriate conventions. See [6] for details about signal raising and other means of communication between the application processes and the executive.

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.

B.b. Predefined Attributes

The following attributes are predefined in the language: ‘‘mode’’, ‘‘implementation’’, ‘‘processor’’, ‘‘source’’, ‘‘xwindow’’, ‘‘xdisplay’’, and ‘‘debug’’.

B.b.1. Mode Attribute

Syntax:

```
ModeAttr          ::= ‘‘MODE’’ ‘‘=’’ Identifier
```

Examples:

```
    mode = Round_Robin;
```

Meaning:

The values of the ‘‘mode’’ attribute are identifiers denoting the operation performed by one of the predefined tasks: ‘‘broadcast’’, ‘‘merge’’, and ‘‘deal’’. Possible values are described in Section B.c.

The identifiers used as values for the ‘‘mode’’ attribute are just a convenient shorthand to select what are expected to be frequently used tasks.

B.b.2. Implementation Attribute

Syntax:

```
ImplementationAttr    ::= ‘‘IMPLEMENTATION’’ ‘‘=’’ StringValue
```

Examples:

```
    implementation = "demo";
```

Meaning:

The value of the implementation attribute is the name of the file containing the actual object code. The format of a file name may vary with the host operating system.

B.b.3. Processor Attribute

Syntax:

```
ProcessorAttr     ::= ‘‘PROCESSOR’’ ‘‘=’’ Identifier
```

Examples:

```
    processor = ibm1401;
    processor = sun;
    processor = sun1;
```

Meaning:

The configuration of the heterogeneous machine specifies the different values for the ‘‘processor’’ attribute, including names of classes of processors as well as names of individual processors, as illustrated above. This information is maintained in a ‘‘configuration file’’ accessed by the runtime environment. See the *Durra User’s Manual* [7] for additional details.

The value of the ‘‘processor’’ attribute can vary in specificity by using a processor class name or an individual processor name. For example, SUN means any SUN processor; SUN1 means a specific SUN processor. If the user specifies the name of a class of processors as the value of the ‘‘processor’’ attribute, any member of the class can be used to execute the task.

B.b.4. Source Attribute

Syntax:

```
SourceAttr        ::= ‘‘SOURCE’’ ‘‘=’’ StringValue
```

Examples:

```
    source = "this is a string";
```

Meaning:

The value of the ‘‘source’’ attribute specifies an initial parameter to be passed to the task implementation when it is started by the Durra executive. The actual means of passing this parameter is implementation dependent (e.g., command line, environment variable, predetermined file location). The parameter can be used by the task implementation for any purpose (or can be ignored).

B.b.5. Window Attribute

Syntax:

```
WindowAttr        ::= ‘‘XWINDOW’’ ‘‘=’’ StringValue
```

Examples:

```
    xwindow = "-geometry 80x24+200+200";
```

Meaning:

The value of the ‘‘xwindow’’ attribute specifies the properties of a terminal emulator under the X Window screen management system. The values of the attributes are used by the Durra executive to start a terminal emulator under which the task implementation will execute. These attributes are useful when the task implementation is an interactive program. Otherwise, the input and output streams of all the task implementations running on a given processor might be scrambled together.

B.b.6. Display Attribute

Syntax:

```
DisplayAttr       ::= ‘‘XDISPLAY’’ ‘‘=’’ Identifier
```

Examples:

```
xdisplay = SUN_1
```

Meaning:

The configuration of the heterogeneous machine specifies the different values for the ‘‘xdisplay’’ attribute, including names of classes of processors as well as names of individual processors, as illustrated above. This information is maintained in a ‘‘configuration file’’ accessed by the runtime environment. See the *Durra User’s Manual* [7] for additional details.

The value of the ‘‘xdisplay’’ attribute specifies the name of a display for a terminal emulator under the X Window screen management system. The value of the attribute is used by the Durra executive to start a terminal emulator under which the task implementation will execute. This attribute is useful when, for instance, one would like all interactive tasks to display their output at a given console, regardless of which processor the task is running on. The default is the display associated with the processor on which the executive is running.

B.b.7. Debug Attribute

Syntax:

```
DebugAttr         ::= ‘‘DEBUG’’ ‘‘=’’ StringValue
```

Examples:

```
debug = "a.db -p /usr/projects/hetsim/tasklib/current/.vax";
```

Meaning:

The value of the debug attribute is the command line that would be used to execute a task under control of the appropriate source-level debugger. The format of the command line depends on the language, debugger, and host operating system.

For example, in the Unix implementation, if the debug attribute is present, the Durra executive starts the debugger by ‘‘executing’’ the command obtained by concatenating the debug attribute value with the implementation attribute value, as in:

```
a.db -p /usr/projects/hetsim/tasklib/current/.vax demo
```

Since a separate terminal interface is required for each debugger activated, this feature is only available when the environment supports the X Window Manager. The effect is that, instead of starting the task directly, the runtime environment starts an ‘‘xterm’’ terminal emulator and runs the specified debugger in it, giving it the task name as an argument.

B.c. Predefined Tasks

The following tasks are predefined in the language: ‘‘broadcast’’, ‘‘merge’’, and ‘‘deal’’.

B.c.1. Broadcast Task

The task ‘‘broadcast’’ has one input port and as many output ports as needed. Input data are replicated and sent to all the output ports.

```
task broadcast
    ports port-declarations;
end broadcast;
```

B.c.2. Merge Task

The task ‘‘merge’’ has one output port and as many input ports as needed. The type of the output port is the union of all the input types. Input data items are merged and sent to the output port.

```
task merge
    ports port-declarations;
    attributes mode = identifier;
end merge;
```

A merge discipline must be provided as a value to the ‘‘mode’’ attribute in a ‘‘merge’’ task selection, as described in Section B.b.1. Possible values include ‘‘fifo’’ (ordered by time of arrival to the merge process) and ‘‘round_robin’’ (one datum from each consecutive input port and repeating). Because of transmission delays, the order of arrival of the data might differ from the order in which the data were sent out. A FIFO merge process uses time of arrival, not time of creation, to order the data.

B.c.3. Deal Task

The task ‘‘deal’’ has one input port and as many output ports as needed. The type of the input port is the union of all the output types. Input data items are sent to one output port.

```
task deal
    ports port-declarations;
    attributes mode = identifier;
end deal;
```

A deal discipline must be provided as a value to the ‘‘mode’’ attribute in a ‘‘deal’’ task selection, as described in Section B.b.1. Possible values include ‘‘fifo’’ (output to the first port with a waiting consumer, with a random choice if there is more than one waiting consumer), ‘‘round_robin’’ (output to each port in order and repeating), and ‘‘by_type’’ (output to the appropriate port, depending on the type of the data.) In the latter case there must be exactly one output port for each possible type accepted by the input port. This is the only kind of ‘‘deal’’ process in which multiple output types are allowed. Other kinds of ‘‘deal’’ processes require compatible output types in all the output ports.

B.c.4. Examples

Task descriptions for the predefined tasks are not stored in the library. The Durra compiler uses the task selection part of the process declaration as the task description. In other words, predefined tasks are ‘‘created’’ on demand, to satisfy a specific process declaration. The following examples illustrate process declarations that are instances of predefined tasks.

```
process pb: task broadcast
                ports
                    in1: in packet;
                    out1, out2: out packet;
             end broadcast;
```

This process declaration selects a 2-output ‘‘broadcast’’ task.

```
process pm: task merge
                ports
                    in1, in2: in packet;
                    out1: out packet;
                attributes
                   mode = round_robin;
            end merge;
```

This process declaration selects a 2-input ‘‘merge’’ task that handles items of type packet in round-robin fashion.

```
process pd: task deal
                ports
                   in1: in packet;
                   out1, out2: out packet;
                attributes
                   mode = round_robin;
            end deal;
```

This process declaration selects a 2-output ‘‘deal’’ task that handles items of type packet in round-robin fashion.

Index