Durra: A Task-level Description Language

Preliminary Reference Manual

V 0.1 -- November 15, 1986

M.R. Barbacci and J.M. Wing
Carnegie Mellon University

Abstract

Durra is a language designed to support the development of large-grained parallel programming applications. This document is a preliminary reference manual for the syntax and semantics of the language. Comments, suggestions, criticisms, etc., are appreciated. Address them to:

Dr. Mario R. Barbacci
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-7704
Barbacci@sei.cmu.edu.arpa

Professor Jeannette M. Wing
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-3068
Wing@c.cs.cmu.edu.arpa

This research is carried out jointly by the Software Engineering Institute, a Federally Funded Research and Development Center, sponsored by the Department of Defense, and by the Department of Computer Science, sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J.M. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie-Mellon University, the National Science Foundation, the Department of Defense or the US Government.

@edicom[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.'']

1. Introduction

Many computation-intensive, real-time applications require efficient concurrent execution of multiple *tasks* 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, our first pass is influenced by both a specific architecture, HET0 [4], and by a specific application, the Autonomous Land Vehicle (ALV), and more specifically, the perception components of the ALV [5]. We assume there is a cross-bar switch, intelligent buffers on the switch sockets, and a scheduler that can communicate with all processors, buffers, and I/O devices.

1.1. Scenario

Here 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:

1. the creation of a library of tasks,
2. the creation of an application description, and

3. 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 code implementing the tasks. For a given task, there may be possibly many implementations, differing in programming language (e.g., one written in C or one written in W2), processor type (e.g., Motorola 68020 or IBM 1401), 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 enters them into the *library*. This is where Durra first enters the picture. Durra is used to write specifications of each task's performance and functionality, the types of data it produces or consumes, and the ports it uses to communicate with other tasks.

Description creation activities

These happen when the user decides to put together an application (say, autonomous land vehicle) using as building blocks tasks in the 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 compiler retrieves task descriptions matching the *task selections* specified by the user from the library and generates a set of resource allocation and scheduling commands to be interpreted by the *scheduler*.

3. The user links the output of the compiler with run-time support facilities, obtaining a *scheduler program*.

Application execution activities

1. The scheduler downloads the task implementations, i.e., code, to the processors and interprets the scheduling commands and initialization code for the machine.

2. The heterogeneous machine runs the processes on processors as dictated by the scheduler 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). Figures 1 and 2, respectively, illustrate the physical and logical components of the system.

buffers (P)
: computers acting as input or output devices, interfacing processors with the switch. As an optimization, buffers execute predefined tasks such as merge, deal, broadcast, and data transformations.

implementation (A)
: code written in some programming language for a specific processor, and satisfying

Figure 1: Physical Components

Scheduler

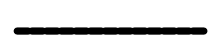
Processors

Buffers
(processor+
memory)

Switch

Buffers
(processor+
memory)

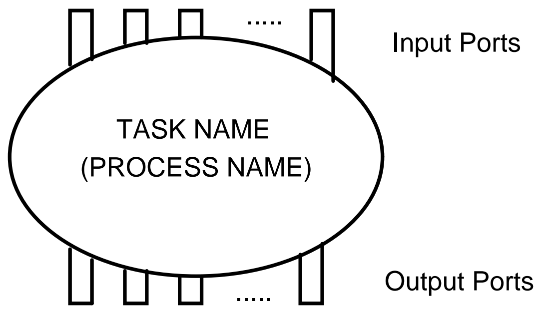
Data Paths

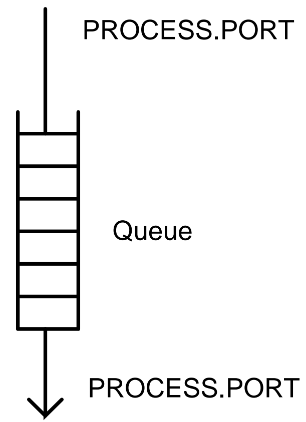
 Switched

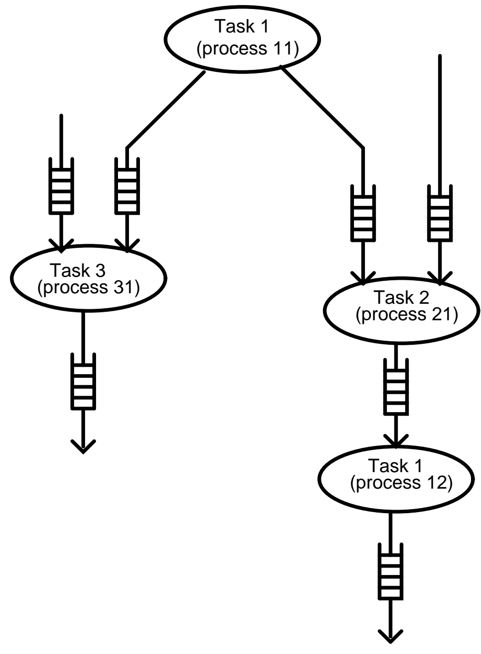
Permanent

Control Paths

Figure 2: Logical Components


.....
Input Ports
TASK NAME
(PROCESS NAME)
Output Ports
.....


PROCESS.PORT
Queue
PROCESS.PORT


Task 1
(process 11)
Task 3
(process 31)
Task 2
(process 21)
Task 1
(process 12)

the performance, functional, and other requirements specified in a task description.

ports (L) processes' logical input or output devices. Input ports remove data from queues; output ports deposit data in queues.

process (L) a uniquely identifiable instance of a task, running on a processo 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 **scheduler** 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.

scheduler (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 3.

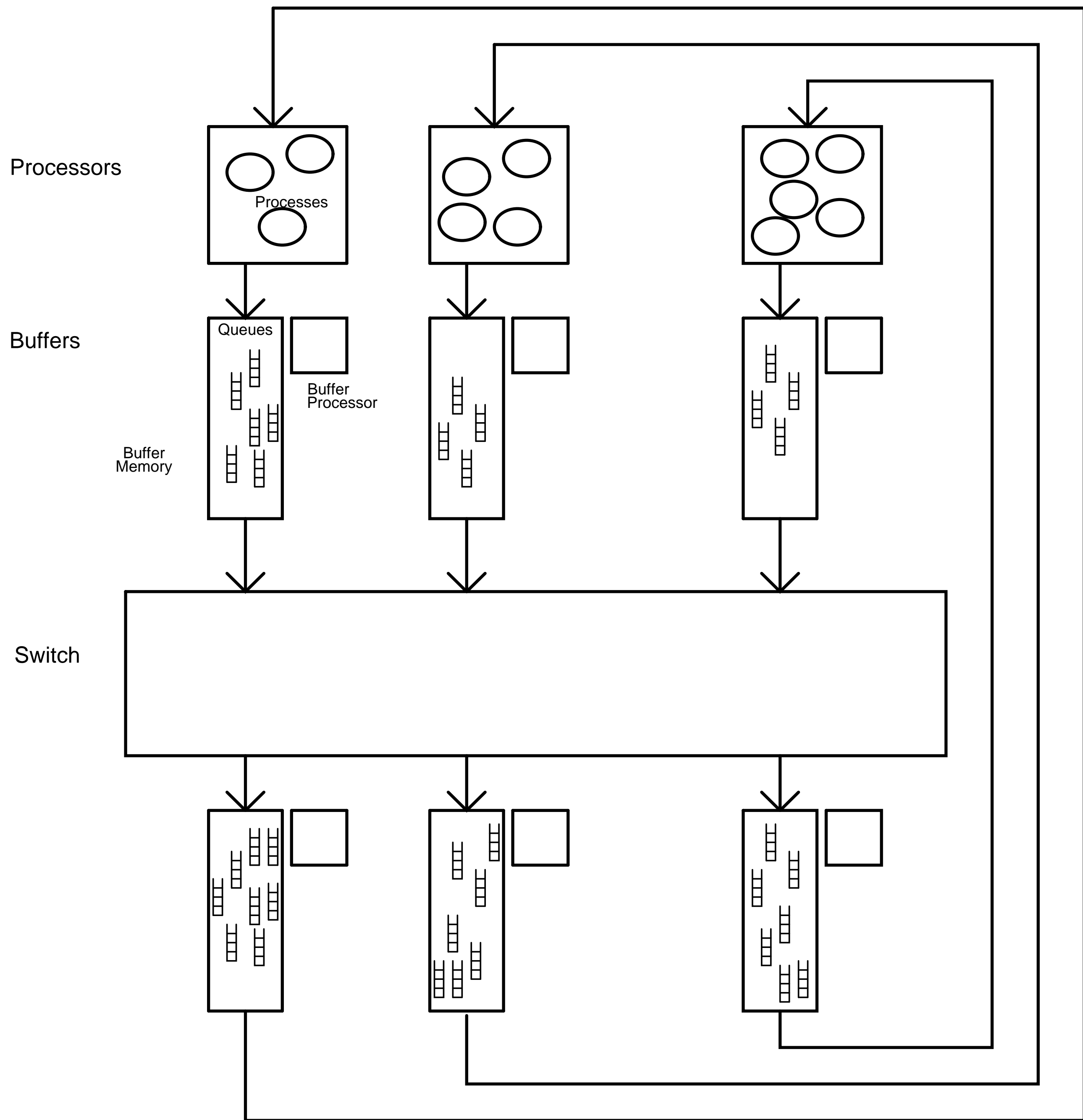
1.3. Notes on Syntax

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

1. Commas separate alternatives. Braces (‘‘{’’ and ‘‘}’’) indicate optionality.
2. Terminal symbols are enclosed in quotes (‘‘ and ’’), but the quotes 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 @qi(,), not the string ‘‘comma.’’
5. Comments start with the characters @qi(--). Any characters between @qi(--) and the end of the line are ignored.
6. Identifiers are, in the usual fashion, sequences of letters, digits, and @qi(_) (underscore), beginning with a letter.
7. Strings are arbitrary sequences of Ascii printable characters, enclosed in double quotes ("). A double quote inside a string must be written as two consecutive double quotes:

   "A string with a double quote, "", inside"
8. Integer and real numbers are always decimal, i.e., base 10. A real number can terminate with a period @qi(.) without a fractional part.

Figure 3: Mapping of Logical and Physical Components



Processors
Processes
Buffers
Queues
Buffer
Processor
Buffer
Memory
Switch

1.4. Keywords and Predefined Identifiers

Keywords and predefined identifiers are highlighted in normal text by writing them in **bold face**, or in ''quotes'', respectively. The following words are keywords in the language: **after**, **and**, **array**, **ast**, **attributes**, **before**, **behavior**, **bind**, **cst**, **date**, **days**, **during**, **end**, **ensures**, **est**, **gmt**, **hours**, **identity**, **if**, **index**, **in**, **is**, **local**, **loop**, **minutes**, **months**, **mst**, **not**, **of**, **or**, **out**, **ports**, **process**, **pst**, **queue**, **reconfiguration**, **remove**, **repeat**, **requires**, **reshape**, **reverse**, **rotate**, **seconds**, **select**, **signals**, **size**, **structure**, **task**, **then**, **timing**, **to**, **transpose**, **type**, **union**, **when**, **years**.

The following words are predefined identifiers in the language: @qi(broadcast), @qi(current_size), @qi(current_time), @qi(deal), @qi(delay), @qi(get), @qi(implementation), @qi(merge), @qi(minus_time), @qi(mode), @qi(plus_time), @qi(processor), @qi(put),

1.5. 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 8) whose values are literals of the appropriate kind, or (c) calls to one of the predefined functions in the language (Section 10.1) returning values of the appropriate kind:

```
IntegerValue        ::=  IntegerLiteral  ,
                         GlobalAttrName  ,
                         FunctionCall

RealValue           ::=  RealLiteral  ,
                         GlobalAttrName  ,
                         FunctionCall

StringValue         ::=  StringLiteral  ,
                         GlobalAttrName  ,
                         FunctionCall

TimeValue           ::=  TimeLiteral  ,
                         GlobalAttrName  ,
                         FunctionCall
```

1.6. How To Read This Manual

This manual is written top-down, so the reader should be aware that there are many forward references. One can read this manual from beginning to end to get an overview of the language, and then read individual sections to understand the details of each language feature.

2. Compilation Units

Syntax:

Compilation	`::=`	**CompilationUnit**`_List`$_{semicolon}$ `@qi(;)`
CompilationUnit	`::=`	**TypeDeclaration** `,`
		TaskDescription

Meaning:

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

Any number of compilation units can be submitted to the **compiler** as a group, in a single text file. Each unit is compiled in order, and if no errors are detected, the unit is entered into the **library**. It can then be used by units compiled later, including units submitted later in the same compilation.

3. Type Declarations

Syntax:

```
TypeDeclaration    ::=  @qi(TYPE) TypeName @qi(IS) TypeStructure ,
                        @qi(TYPE) TypeName @qi(IS) UnionStructure

TypeName           ::=  Identifier

TypeStructure      ::=  @qi(SIZE) ElementSize ,
                        @qi(ARRAY) ArrayDimension @qi(OF) TypeName

ArrayDimension     ::=  '(' IntegerValue_List_space ')'      -- Positive integer

ElementSize        ::=  IntegerValue ,                   -- Positive number of bits
                        IntegerValue @qi(TO) IntegerValue
                                                         -- Non-negative size range

UnionStructure     ::=  @qi(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
```

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 two kinds of type declarations. First, a type declaration can specify the structure of the data moving through a process port. The basic data type is a sequence of bits of fixed or variable (but bound) length. More complex types are declared as multi-dimensional arrays of simpler types. Second, a type 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.

4. Task Descriptions

Syntax:

```
TaskDescription     ::=  @qi(TASK) TaskName
                             InterfacePart
                                 { BehaviorPart }
                                     { AttrDescriptionPart }
                                         { StructurePart }
                                             @qi(END) TaskName
```

Meaning:

Task descriptions are compilation units 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 4 shows a template for a task description, where the **ports** and **signals** clauses constitute the interface information.

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

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

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

  attributes                                                     -- OPTIONAL
    attribute-value-pairs
    -- Additional properties of the task

  structure                                                      -- OPTIONAL
    process-declarations
    queue-declarations
    binding-declarations
    reconfiguration-statements
    -- A process-queue graph describing the internal structure of a task
end task-name;
```

Figure 4: A Template for Task Descriptions

5. Task Selections

Syntax:

```
TaskSelection        ::=  @qi(TASK) TaskName
                            { PortDeclarationPart }
                               { SignalDeclarationPart }
                                  { BehaviorPart }
                                     { AttrSelectionPart }
                                        { @qi(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 9.1. 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 6) requires the declarations of the ports, whereas in a task selection, the declaration of the ports is optional. Figure 5 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
  ports                                                             -- OPTIONAL
    port-declarations
    -- A signature that must match port directions and types of
    -- that of a task description in the library.

  signals                                                           -- OPTIONAL
    signal-declarations
    -- A signature that must match signal directions and names of
    -- that of a task description in the library.

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

  attributes                                                        -- OPTIONAL
    attribute-value-pairs
    -- Named (actual) attributes used to match (formal) attributes of
    -- those of a task description in the library.
end task-name                    -- optional if only the task name is specified
```

Figure 5: A Template for Task Selections

---

6. Interface Information

Syntax:

InterfacePart ::= **PortDeclarationPart** { **SignalDeclarationPart** }

Meaning:

The interface portion of a task description or a task selection provides information about the ports of the processes instantiated from the task and the signals used by the processes instantiated from the task to communicate with the **scheduler**.

6.1. Port Declarations

Syntax:

PortDeclarationPart ::= `@qi(PORTS)` **PortDeclaration**_$\texttt{List}_{\texttt{semicolon}}$ `@qi(;)`

PortDeclaration ::= **PortName**_$\texttt{List}_{\texttt{comma}}$ `@qi(:)` `@qi(IN)` **TypeName**
PortName_$\texttt{List}_{\texttt{comma}}$ `@qi(:)` `@qi(OUT)` **TypeName**

PortName ::= **Identifier**

GlobalPortName ::= { **ProcessName** `@qi(.)` } **PortName**

Examples:

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

Meaning:

A port declaration specifies the direction of the data movement and the type of data moving through the port.

Port names must be unique within a task. Outside the task, ports are identified by their global name, obtained by prefixing the name of a process (instance of a task) to the name of the port, e.g., p1.out2.

6.2. Signal Declarations

Syntax:

SignalDeclarationPart ::= `@qi(SIGNALS)`
SignalDeclaration_$\texttt{List}_{\texttt{semicolon}}$ `@qi(;)`

SignalDeclaration ::= **SignalName**_$\texttt{List}_{\texttt{comma}}$ `@qi(:)` `@qi(IN)` ,
SignalName_$\texttt{List}_{\texttt{comma}}$ `@qi(:)` `@qi(OUT)` ,
SignalName_$\texttt{List}_{\texttt{comma}}$ `@qi(:)` `@qi(IN)` `@qi(OUT)`

SignalName ::= **Identifier**

GlobalSignalName ::= { **ProcessName** `@qi(.)` } **SignalName**

Examples:

signals
```
  Stop, Start, Resume: in;
  RangeError, FormatError: out;
  Read: in out;
```

Meaning:

Signals are special messages exchanged between a process and the **scheduler**. A signal declaration specifies the direction of the signal. An **in** signal is a message that a process can receive from the scheduler; an **out** signal is a message that a process can send to the scheduler; an **in out** signal is used for both directions of communication.

All signal names must be unique within a task. Outside the task, a signal is identified by compounding the name of a process (instance of a task) with the name of the signal, e.g., p1.Restart.

6.3. Rules for Matching Selections with Descriptions

If a task selection provides a port declaration clause, the port names provided in the task selection override the port names provided in the task declaration. The port declaration lists must otherwise be identical, i.e., the number, the order, the directions, and the types must be identical.

If a task selection provides a signal declaration clause, the clause must be identical to that provided in the task description, i.e., the names, number, and directions must be identical.

7. Behavioral Information

Syntax:

```
BehaviorPart      ::=  @qi(BEHAVIOR) FunctionPart TimingPart

FunctionPart      ::=  { @qi(REQUIRES) `"' predicate `"' @qi(;) }
                         { @qi(ENSURES) `"' predicate `"' @qi(;) }

TimingPart        ::=  { @qi(TIMING) TimingExpression @qi(;) }

predicate         ::   @ii(Larch Predicate)[1]
```

Meaning:

The behavioral information part specifies functional and timing information about the task.

The functional information part of a task description consists of a pre-condition (**requires**) on what is required to be true of the data coming through the input ports, and a post-condition (**ensures**) on what is guaranteed to be true of the data going out on the output ports.

The timing information part of a task description consists of a timing expression following the keyword **timing**. The timing expression describes 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 meaning of the individual parts and their combination. See [1] for the formal meaning.

7.1. 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** is used as the assertion language in the predicates of these clauses. We restrict this section to a very brief outline of Larch's approach.

Larch [2, 3] uses a two-tiered approach to specifying program modules: a **trait** defines state-independent properties, and an **interface specification** defines state-dependent properties of a program. A trait is written in the Larch Shared Language (LSL), and it provides the assertion language used to express and define the meaning of the predicates of an interface specification.

For a program module such as a procedure, a Larch interface specification is written in a **Larch Interface Language** and contains predicates about the states before and after the execution of the procedure. The Larch Interface Language (LIL) to be used is specific to the programming language in which the procedure is written (e.g., C, CommonLisp, or Ada.)

---

[1] Essentially, a first-order assertion, [2].

7.1.1. Larch Traits and Specifications

Figure 6 depicts a Larch (two-tiered) specification of queues with @qi(put) and @qi(get) operations. The top part of the specification (Figure 6.a) is a trait written in LSL used to describe values of queues. A set of operators and their signatures following **introduces** defines a vocabulary of terms to denote values of a type. For example, Empty and Insert(Empty, 5) denote two different queue values. The set of equations following the **constrains** clause defines a meaning for the terms; more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from the above trait, one could prove that First(Rest(Insert(Insert(Empty, 5), 6))) = 6.

The bottom part of the specification (Figure 6.b) contains two interfaces written in a ''generic'' Larch interface language. They describe the functional behavior of two queue operations, @qi(put) and @qi (get) (queue operation names are used to write timing expressions, which are described in Section 7.2.3.) A **requires** is a pre-condition on the state of an operation's input data that must be true upon operation invocation; an **ensures** is a post-condition on the state of an operation's input and output data that is guaranteed to be true upon operation termination. An omitted predicate is taken to be **true**. The specification for @qi(get) states that @qi(get) must be called with a non-empty queue and that it modifies the original queue by removing its first element and returning it.

```
QVals: trait
  introduces
    Empty: @RightArrow Q
    Insert: Q, E @RightArrow Q
    First: Q @RightArrow E
    Rest: Q @RightArrow Q
    isEmpty: Q @RightArrow Bool
    isIn: Q, E @RightArrow Bool
 constrains Q so that
    Q generated by [ Empty, Insert ]
    for all q: Q, e, e1: E
        First(Insert(Empty), e)) = e
        First(Insert(q, e)) = if isEmpty(q) then e else First(q)
        Rest(Insert(q, e)) = if isEmpty(q) then Empty else Insert(Rest(q), e)
        isEmpty(Empty) = true
        isEmpty(Insert(q, e)) = false
        isIn(Empty, e) = false
        isIn(Insert(q, e), e1) = (e = e1) | isIn(q, e1)
```

a. A Trait for Queue Values

```
Put = operation (q: queue, e: element)
    ensures q_post = Insert(q, e)

Get = operation (q: queue) returns (e: element)
    requires ~isEmpty(q)
    ensures q_post = Rest(q) & e = First(q)
```

b. Interfaces for Queue Operations

Figure 6: A Larch Two-Tiered Specification for Queues

7.1.2. Functional Specification of a Task

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. A **requires** clause states what is required to be true of the data coming through the input ports; an **ensures** clause states what is guaranteed to be true of the data going out through the output ports.

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.

For example, consider the matrix multiplication task in Figure 7. The task takes input matrices from two queues and outputs the result matrix on an output queue. The **requires** clause states that the task implementor may assume that the number of rows of the matrix entering through the port in1 equals the number of columns of the matrix entering through in2. The **ensures** clause states that the result of multiplying the two input matrices is output through the output port.

```
task multiply
  ports
    in1, in2: in matrix;
    out1: out matrix;
  behavior
      requires "rows(First(in1)) = cols(First(in2))";
      ensures  "Insert(out1, First(in1) * First(in2))";
end multiply;
```

Figure 7: A Matrix Multiplication Task

7.2. Timing Part

Processes remove data from their input queues and store data into 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.

7.2.1. Time Literals

Syntax:

```
TimeLiteral        ::=  { Date @qi(@) } TimeOfDay { TimeZone }
                        IndeterminateTime

Date               ::=  years @qi(/) months @qi(/) days

years              ::=  IntegerValue

months             ::=  IntegerValue                 -- range is 1..12

days               ::=  IntegerValue                 -- range is 1..31

TimeOfDay          ::=  { { hours @qi(:) } minutes @qi(:) } seconds ,
                        RealValue TimeUnit ,
                        IntegerValue TimeUnit ,

hours              ::=  IntegerValue                 -- range is 0..23

minutes            ::=  IntegerValue                 -- range is 0..59

seconds            ::=  IntegerValue ,
                        RealValue

TimeUnit           ::=  @qi(YEARS) ,
                        @qi(MONTHS) ,
                        @qi(DAYS) ,
                        @qi(HOURS) ,
                        @qi(MINUTES) ,
                        @qi(SECONDS)

TimeZone           ::=  @qi(EST) ,               -- Eastern Standard Time
                        @qi(CST) ,               -- Central Standard Time
                        @qi(MST) ,               -- Mountain Standard Time
                        @qi(PST) ,               -- Pacific Standard Time
                        @qi(GMT) ,               -- Greenwich Meridian Time
                        @qi(LOCAL) ,             -- Local Time
                        @qi(AST)                 -- Application Start Time

IndeterminateTime  ::=  @qi(*)
```

Examples:

```
5:15:00 est     -- An absolute time: 5 hours 15 minutes Eastern Standard Time.

15.5 hours ast        -- An application relative time: 15 hours and 30 minutes
                                     -- after the start of the application.

2:10                         -- An event relative time: 2 minutes 10 seconds
                                                   -- after some base event.

2.1667 minutes        -- Approximately the same event relative time as above
                                      -- 10 seconds is 1/6th of a minute.

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

Meaning:

Time values are used to specify points in time. These can be either (1) absolute, i.e., independent of the application, in which case they must be followed by the name of a time zone; (2) relative to the application start time, in which case they must be followed by the fictitious time zone @qi(ast); or (3) relative to some prior event in the application, in which case neither a date nor a time zone is allowed.

The notation allows for alternative ways of denoting time of day or time elapsed between events. Time can be expressed in the familiar formats "HH:MM:SS", "MM:SS", or just "SS". Thus, a plain number represents a number of seconds. Time can also be expressed as a multiple of other time units by writing a number followed by a unit name such as **seconds**, **minutes**, **hours**, **days**, **months**, or **years**. The use of **seconds** as a time unit is redundant, but allowed for completeness' sake. The format adopted by a user might depend on the nature of the application, on any standard conventions in the application domain, on the magnitude of the time scale, on the precision required, or simply on aesthetic, personal preferences.

7.2.2. Event Expressions and Time Windows

Syntax:

```
EventExpression     ::=  GlobalPortName
                              { @qi(.) QueueOperation }
                                  { TimeWindow }
                         @qi(DELAY) TimeWindow

TimeWindow          ::=  @qi([) TimeValue @qi(,) TimeValue @qi(])

QueueOperation      ::=  Identifier                  -- Configuration dependent
```

Examples:

```
in1            -- An operation (get, by default) on the queue feeding port in1.

in1.get               -- An operation taking a system default time to complete.

in1.get[5, 15]    -- An operation taking between 5 and 15 seconds to complete.

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, @qi(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: @qi(get) for input ports, @qi(put) for output ports. The complete list of queue operations is configuration dependent, as described in Section 10.4.

Time windows are used to describe the duration of a queue operation or the delay between two operations. Time windows are denoted by a pair of time values [T_{min},T_{max}] defining the boundaries of the interval.

The time window associated with a queue operation describes the minimum and maximum time needed to perform the operation. This time window is optional, and if it is missing, a configuration dependent, default window is assumed, as described in Section 10.4. Intervals of time between queue operations are denoted by a @qi(delay) operation whose time window describes the minimum and maximum time consumed by the process in between queue operations.

7.2.3. Timing Expressions

Syntax:

```
TimingExpression         ::=  { @qi(LOOP) } CyclicTimingExpression

CyclicTimingExpression   ::=  ParallelEventExpression_List_spaces

ParallelEventExpression  ::=  BasicEventExpression_List_double_vertical_bar

BasicEventExpression     ::=  EventExpression ,
                              { Guard @qi(=>) } '(' CyclicTimingExpression ')'

Guard                    ::=  @qi(REPEAT) IntegerValue ,
                              @qi(BEFORE) TimeValue ,              -- Absolute time
                              @qi(AFTER) TimeValue ,               -- Absolute time
                              @qi(DURING) TimeWindow ,       -- T_min is Absolute time
                              @qi(WHEN) '"' predicate '"'

predicate                ::   @ii(Larch Predicate)[2]
```

Examples:

```
in1 || in2[10,15]   -- Two parallel input operations, starting simultaneously.

in1[0,5] delay[10,15] out1- Two sequential inputs operations with an interven

repeat 5 => (in1[0,5] delay[10,15] out1)- Same as above but as a cycle repeat

before 18:00:00 local => ( . . . )-- A sequence constrained to start before 6 pm

after 18:00:00 local => ( . . . )  -- A sequence constrained to start after 6 pm.

during [18:00:00 local, 12 hours] => ( . . . )- A sequence constrained to start

when ~empty(in1) and ~empty(in2) => ((in1.get || in2.get) out1.put);
          -- A sequence constrained to start after both input queues have data.

loop when ~empty(in1) and ~empty(in2) => ((in1.get || in2.get) out1.put);
                         -- The same sequence as above but repeated indefinetely.
```

[2]Essentially, a first-order assertion, [2].

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.

Parallel events start simultaneously but are not necessarily completed at the same time. In the expression ‘‘(in1 || in2[10,15])’’, the duration of the input operation on port in1 defaults to some configuration-dependent value (See Section 10.4) and might be shorter or longer than the explicit duration of the input operation on port in2, i.e., between 10 and 15 seconds.

A basic event expression is either a queue operation (including @qi(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 deadline does not include a date, i.e., it is just a time of day, and the deadline has passed, then the sequence is blocked at most until midnight of the current date and will unblock at ‘‘00:00:00’’ of the following day. The task is terminated if a dated deadline has passed.
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. If the deadline does not include a date, i.e., it is just a time of day, then the sequence is blocked at most 24 hours. For example, if it is ‘‘00:00:00.000’’ and the deadline is ‘‘23:59:59.999’’ the sequence will unblock at the end of the day.
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 10.1) before the sequence is allowed to start. It is a pre-condition for starting the sequence.

7.2.4. 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. A date in a time value that uses the @qi(ast) time zone is meaningless.
2. In the time window attached to a queue operation, including @qi(delay), the time values must be relative (i.e., no dates or time zones allowed) and are interpreted relative to the start of the operation.
3. 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}.

7.3. Rules for Matching Selections with Descriptions

The meaning of the behavioral information is a predicate, @Mf(R, T) => @Mf(E, T), where R is the **requires** predicate, E is the **ensures** predicate, T is the **timing** expression, and @Mf is the meaning function mapping a predicate and timing expression into a boolean [1].

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.

Currently there are no facilities to check these implications and timing expressions, so for the time being the behavioral information part of a task description is treated as commentary information. However, timing expressions are used to simulate the behavior of a task and are therefore required by the simulator [6].

8. Attributes

Syntax:

AttrDescriptionPart ::= `@qi(ATTRIBUTES)` **AttrDescription**_$\texttt{List}_{\texttt{semicolon}}$ `@qi(;)`

AttrDescription ::= **AttrName** `@qi(=)` **AttrValue**

AttrSelectionPart ::= `@qi(ATTRIBUTES)` **AttrSelection**_$\texttt{List}_{\texttt{semicolon}}$ `@qi(;)`

AttrSelection ::= **AttrName** `@qi(=)` **AttrDisjunction**

AttrName ::= **Identifier**

GlobalAttrName ::= { **ProcessName** `@qi(.)` } **AttrName**

AttrDisjunction ::= **AttrConjunction** ,
AttrDisjunction `@qi(OR)` **AttrConjunction**

AttrConjunction ::= **AttrPrimary**,
AttrConjunction `@qi(AND)` **AttrPrimary**

AttrPrimary ::= **AttrTerm** ,
`@qi(NOT)` **AttrTerm**

AttrTerm ::= **AttrValue** ,
'(' **AttrDisjunction** ')'

AttrValue ::= **OtherAttrValue** ,
'(' **OtherAttrValue**_$\texttt{List}_{\texttt{comma}}$ ')' ,
ModeAttrValue ,
ImplementationAttrValue ,
ProcessorAttrValue ,

OtherAttrValue ::= **IntegerValue** ,
RealValue ,
StringValue ,
TimeValue

Examples:

```
attributes                               -- Attributes in a task declaration
  author = "jmw";
  color = ("red", "white", "blue");
  implementation = "/usr/jmw/alv/cowcatcher.o";
  Queue_Size = 25 ;

attributes                                 -- Attributes in a task selection
  author = "jmw" or "mrb";
  color = "red" and "blue" and not ("green" or "yellow");
  processor = Warp1;
  mode = grouped_by_4;
```

Meaning:

Attributes specify miscellaneous properties of a task. They are a means of indicating pragmas or hints to the compiler and/or scheduler. 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 lists the desired values of a 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. Attributes defined in other tasks can be accessed by prefixing the name of the attribute with the name of a process instantiated from that task, e.g., p1.author.

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" as in the examples then "Queue_Size" can appear anywhere an integer value is expected. This permits the user to name say, a queue size and use the name to declare queues with identical size in a number of task descriptions. Another use is to instantiate "families" of tasks, i.e., tasks that share the same value for some attribute, as shown in Figure 8.

```
process
  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 value as Master_Process
        ... other attributes, maybe ...
      end foo;

  p2: task bar
        attributes
        Key_Name = Master_Process.Key_Name;  -- Same value as Master_Process
        ... other attributes, maybe ...
      end bar;
```

Figure 8: Use of Global Attribute Names

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 values, or strings, as the case may be, and compatibility is based on value equality. If the attribute is predefined in the language, the syntax for the legal values is given in Section 10.2, and compatibility is attribute dependent.

The following attributes are predefined in the language: @qi(mode) (specifies the mode of operation for a deal or merge predefined task); @qi(implementation) (specifies the location of the task implementation); and @qi(processor) (specifies the processor type on which the implementation can run). These are described in Section 10.2.

8.1. Rules for Matching Selections with Descriptions

If a task selection specifies an attribute not present in a task description, no match occurs, i.e., the compiler skips this description and continues searching for a candidate. If a task description provides an attribute not specified in a task selection, the attribute is ignored.

If a task selection provides a predicate (a disjunction) for an attribute, a matching task description must provide values that satisfy the predicate, i.e., the disjunction yields **true** when evaluated in the context of the values declared for the attribute. If a task description provides a single value for an attribute, a matching task selection must provide exactly that value.

9. Structural Information

Syntax:

```
StructurePart          ::=  @qi(STRUCTURE)
                                StructureClause_List_space
                                    { ReconfigurationClause-List_space }

StructureClause        ::=  @qi(PROCESS) ProcessDeclaration_List_semicolon @qi(;),
                            @qi(QUEUE) QueueDeclaration_List_semicolon @qi(;) ,
                            @qi(BIND) PortBinding_List_semicolon @qi(;)

ReconfigurationClause  ::=  @qi(RECONFIGURATION)
                                 Reconfiguration_List_semicolon @qi(;)
```

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.

9.1. Process Declarations

Syntax:

```
ProcessDeclaration     ::=  ProcessName_List_comma @qi(:) TaskSelection
```

Examples:

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

9.2. Queue Declarations

Syntax:

```
QueueDeclaration       ::=  QueueName { QueueSize } @qi(:) QueueDefinition

QueueDefinition        ::=  GlobalPortName
                                @qi(>) ProcessName @qi(>)
                                    GlobalPortName
                            GlobalPortName
                                 @qi(>) TransformExpression @qi(>)
                                    GlobalPortName

QueueName              ::=  Identifier
```

QueueSize ::= `@qi([)` **IntegerValue** `@qi(])`

GlobalQueueName ::= { **ProcessName** `@qi(.)` } **QueueName**

Examples:

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

q1: p1 > (2 1) transpose > p2 ;-- Two ports connected through an unbounded queu
                              -- The data arrays are transposed in the queue.

q1[100]: p1 > xyz > p2 ;-- Two ports connected through a bounded (size = 100)
                    -- Data are transformed in the queue by a process ``xyz''.
```

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). The queue name must be unique within the task description defining the process-queue graph. The (optional) queue bound declares the maximum number of elements that will be stored in the queue at any one time. If a queue is full when a @qi(put) 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 Section 10.4.

When establishing a logical connection, 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.

9.3. 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.

Complicated transformations can be written as separate tasks, in which case an appropriate task must be selected and instantiated as a process, and the process name must be specified in the queue declaration. Simple transformations can be specified directly in the queue declaration.

9.3.1. Off-Line Data Transformations

Complex data transformations can be specified as regular tasks by writing a procedure in some programming language suitable for either the **buffers** or one of the heterogeneous processors and entering an appropriate task description in the **library**. These data transformation tasks must declare exactly one input port and one output port.

```
task corner_turning
  ports
    in1: in landmark_row_major;
    out1: out landmark_column_major;
  attributes
    implementation = "/usr/mrb/screetch.o";
    processor = buffer_processor;
end corner_turning;
```

9.3.2. In-Line Data Transformations

Syntax:

```
TransformExpression   ::=  TransformOp_List_space

TransformOp           ::=  ReshapeOp ,
                           SelectOp ,
                           TransposeOp,
                           RotateOp,
                           ReverseOp,
                           DataOp

ReshapeOp             ::=  VectorArgument @qi(RESHAPE)

SelectOp              ::=  ArrayArgument @qi(SELECT)

TransposeOp           ::=  VectorArgument @qi(TRANSPOSE)

RotateOp              ::=  ArrayArgument @qi(ROTATE)

ReverseOp             ::=  IntegerValue @qi(REVERSE)

DataOp                ::=  Identifier

VectorArgument        ::=  '(' IntegerValue_List_space ')' ,
                           '(' IntegerValue @qi(IDENTITY) ')' ,
                           '(' IntegerValue @qi(INDEX) ')' ,
                           '(' @qi(*) ')'                      -- Empty vector

ArrayArgument         ::=  VectorArgument ,
                           '(' ArrayArgument_List_space ')'
```

Examples:

If the input is a 2x2x3 3-dimensional array:

```
(3 4) reshape       -- reshapes the input array into a 3x4 2-dimensional array.

(12) reshape                                           -- unravels the array.
```

If the input is a 2-dimensional array:

```
((5 2 3) (*)) select-- generates an array consisting of rows 5 2 and 3, in that

((*) (5 2 3)) select- generates an array consisting of columns 5 2 and 3, in t

(2 1) transpose                   -- Transposes the array in the normal manner.
```

```
(1 -2) rotate          -- Rotates each row left 1 position and then rotates
                          -- each column of the result down 2 positions.
```

Additional examples:

```
(5 identity)                          -- Generates the vector (1 1 1 1 1).

(5 index)                             -- Generates the vector (1 2 3 4 5).

2 reverse  -- Reverses the elements along the 2nd coordinate of an input array.
```

Meaning:

The most common cases of data transformations are expected to be n-dimensional array manipulations. For these operations, the language provides a short-cut: it is not necessary to write task implementations, i.e., program code, and task descriptions and to enter them in the **library**. It suffices to specify the transformations as part of the queue declaration.

In-line data transformations are specified in post-fix notation, interpreted left-to-right, with arguments preceding the operators, and with the input port providing the initial argument. In general, the arguments are multi-dimensional arrays (nested vectors) of scalar data values.

Operator	Description
integer **identity**	generates the vector (1 1 ... 1 1).
integer **index**	generates the vector (1 2 ... N).
vector **reshape**	unravels an array (i.e., linearizes it) and then reshapes into an array with the dimensionality of the argument vector. The input array is linearized in row order, i.e., by scanning all of the positions varying the highest dimension first. This operation must be specified if the input and output array do not have the same shape but the array elements are in the right order when the arrays are unraveled.
array **select**	extracts (slices) pieces of a data array. If the input is a vector, (5) **select** represents the 5th element, and (5 2 3) **select** is a new vector consisting of the 5th, 2nd, and 3rd elements in that order. A vector of the form "(*)" selects all components along one dimension.
vector **transpose**	permutes the dimensions of a data array according to the argument vector (V). The ith coordinate of the input array becomes coordinate V[i] of the result.

scalar_or_vector **rotate**

specifies rotations of n-dimensional data arrays. The operator is preceded by an argument which must be either a scalar (signed) integer value or a parenthesized array of (signed) integer values. The magnitude of the values specify the number of positions to rotate the input data, and the sign of the values specify the direction of the rotation: a positive amount indicates rotation towards lower indices.

A scalar argument specifies how to rotate an input vector. An n-length vector of scalars specifies how to rotate an n-dimensional input array along each dimension (one element per dimension). An n-length vector of vectors argument specifies how to rotate an n-dimensional input array along each dimension (one top level vector per dimension) and within each dimension, how to rotate each "row" (one element of a second level vector per row.)

For example, consider the transformation "((1 2 0) (-3 -4)) **rotate**" applied to a 2-dimensional 3x2 input array. The vector (1 2 0) specifies how to rotate the rows; the vector (-3 -4) specifies how to rotate the columns. The first row is rotated left 1 position, the second row is rotated left 2 positions, the third row is left unchanged. Then the first column is rotated down 3 positions, and finally, the second column is

	rotated down 4 positions.
integer **reverse**	reverses the order of the elements of an array along an arbitrary coordinate specified by the integer argument. If the input is a vector, the argument must be "1". In the transformation "2 **reverse**", if the input is a 2-dimensional array, this operation shuffles columns; if the input is a 3-dimensional array, this operation shuffles planes.
Data Operations	scalar operations applied to each element of an input array. The set of operations is configuration dependent. The initial set will include operations to round, truncate, or otherwise convert between various integer and floating-point formats, as described in the **configuration file**, Section 10.4.

This is a first attempt at defining the set of the operations a user is likely to perform on n-dimensional arrays. The guiding principle is to keep the notation simple; more complex transformations should probably be specified as off-line transformations.

A data transformation operation is more than just a way to achieve type compatibility between ports. It also serves to specify operations that would be inappropriate or inefficient if written as part of one of the 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>(2 1) **transpose**>p2". Arrays produced by p1 are transposed while in the queue, before they are delivered to p2.

9.4. Binding Port Names

Syntax:

PortBinding	`::=`	**ExtPortName** `@qi(=)` **IntPortName**	
ExtPortName	`::=`	**PortName**	`-- External port`
IntPortName	`::=`	**GlobalPortName**	`-- Internal port`

Example:

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

Meaning:

A port binding maps a port of the process-queue graph defining the internal structure of a task to a port defining the external interface of a task.

9.5. Process-Queue Graph Reconfiguration

Syntax:

Reconfiguration `::=` `@qi(IF)` **RecPredicate** `@qi(THEN)`
{ **ProcessTermination**-`List`$_{space}$ }
Structure`_List`$_{space}$
`@qi(END) @qi(IF)`

```
ProcessTermination   ::=  @qi(REMOVE) GlobalProcessName_List_comma @qi(;)

RecPredicate         ::=  RecDisjunction ,
                          RecPredicate @qi(OR) RecDisjunction

RecDisjunction       ::=  RecConjunction ,
                          RecDisjunction @qi(AND) RecConjunction

RecConjunction       ::=  RecRelation ,
                          @qi(NOT) '(' RecPredicate ')'

RecRelation          ::=  RecTerm @qi(=) RecTerm ,                   -- Equal
                          RecTerm @qi(/=) RecTerm ,              -- Not equal
                          RecTerm @qi(>) RecTerm ,                 -- Greater
                          RecTerm @qi(>=) RecTerm ,  -- Greater than or equal
                          RecTerm @qi(<) RecTerm ,                    -- Less
                          RecTerm @qi(<=) RecTerm ,     -- Less than or equal

RecTerm              ::=  IntegerValue ,
                          RealValue ,
                          StringValue ,
                          TimeValue
```

Examples:

```
if Current_Time >= 6:00:00 local and Current_Time < 18:00:00 local
then
    process
        p_vision: task vision attributes processor = warp2;
    queue
        q_vision_road: p_deal.out3   >  > p_vision.in1;
        q_obstacles:   p_vision.out1 >  > p_merge.in3;
    end if;
```

Meaning:

A reconfiguration statement is a directive to the **scheduler**. It is used to specify changes in the current structure, i.e., 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 predicate is a boolean expression involving time values, queue sizes, and other information available to the **scheduler** at run time.

Notice that nothing is being said about the internal representation of time values. They are definitely not like integer or real values -- time values cannot be mixed with regular numeric values in an expression. In addition, currently the language does not provide any arithmetic operators for time values. However, a few predefined system functions provide for the computation of past or future time values, as described in Section 10.1.

10. Predefined Language Facilities

10.1. Functions

Syntax:

```
FunctionCall         ::=  FunctionName { FunctionParameters }

FunctionName         ::=  @qi(CURRENT_TIME) ,
                          @qi(MINUS_TIME) ,
                          @qi(PLUS_TIME) ,
                          @qi(CURRENT_SIZE)

FunctionParameters   ::=  '(' Parameter_List_comma ')'      -- Function dependent

Parameter            ::=  IntegerValue ,
                          RealValue ,
                          StringValue ,
                          TimeValue
```

Examples:

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

Meaning:

The following functions are predefined in the language: @qi(current_time), @qi(minus_time), @qi (plus_time), and @qi(current_size).

The function call @qi(Current_Time) returns the current time as an absolute date in the local time zone.

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. $TimeValue_1$ must be later than $TimeValue_2$.
2. If $TimeValue_1$ is an absolute time and $TimeValue_2$ is a relative time, the result is an absolute time in the same time zone as $TimeValue_1$.
3. If both parameters are relative times, the result is a relative time. $TimeValue_1$ must be larger than $TimeValue_2$.

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 zone.
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.

10.2. Attributes

The following attributes are predefined in the language: @qi(mode), @qi(implementation), and @qi (processor).

10.2.1. Mode Attribute

Syntax:

ModeAttr `::=` `@qi(MODE) @qi(=)` **ModeAttrValue**

ModeAttrValue `::=` **Identifier**

Meaning:

The values of the @qi(mode) attribute are identifiers denoting the operation performed by one of the predefined tasks: @qi(broadcast), @qi(merge), and @qi(deal), as described in Section 10.3.

The formal specification of the operation is given by the behavioral part of the task description. The identifiers used as values for the @qi(mode) attribute are just a convenient shorthand to select what are expected to be frequently used tasks. Users are more likely to select predefined tasks by specifying a mode value (i.e., an identifier) than by specifying a timing expression or a function predicate.

The following identifiers are representative of typical values for the @qi(mode) attribute: @qi(random), @qi(fifo), @qi(round_robin), @qi(by_type), @qi(balanced), @qi(grouped_by_2). The actual values are implementation dependent.

10.2.2. Implementation Attribute

Syntax:

ImplementationAttr `::=` `@qi(IMPLEMENTATION) @qi(=)` **ImplementationAttrValue**

ImplementationAttrValue `::=` **StringValue**

Examples:

```
implementation = "/usr/cbw/het0/demo.o";
```

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.

10.2.3. Processor Attribute

Syntax:

ProcessorAttr `::=` `@qi(PROCESSOR) @qi(=)` **ProcessorAttrValue**

ProcessorAttrValue `::=` **Identifier** `,`
Identifier `'('` **Identifier**_List$_{comma}$ `')'`

Examples:

```
processor = m68000(m68020, m68032);
processor = m68020(p1, p2, p3);
processor = m68032(p4, p5);
processor = ibm1401;
processor = warp(warp1, warp2);
processor = buffer_processor;
```

Meaning:

The configuration of the heterogeneous machine specifies the different values for the @qi(processor) attribute, including names of classes of processors as well as names of individual processors, as illustrated above. See Section 10.4 for details about specifying the configuration of the heterogeneous machine.

The value of the @qi(processor) attribute can vary in specificity by using a processor class name or an individual processor name. For example, WARP means any Warp processor; WARP1 means that Warp processor.

If the user specifies the name of a class of processors as the value of the @qi(processor) attribute, any one of the members of the class can be used to execute the task. If the user specifies a class name and a set of members (in parentheses), any one of the members of this set can be used to execute the task. The members of the set must be a subset of the class as defined by the configuration.

10.3. Tasks

The following tasks are predefined in the language: @qi(broadcast), @qi(merge), and @qi(deal).

10.3.1. Broadcast

@qi(broadcast) is a task with one input port and as many output ports as needed. Input data are replicated and sent to all the output ports. Port names are *in1* for the input port and *out1, out2,..., outN* for the output ports.

10.3.2. Merge

@qi(merge) is a task with 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. Port names are *in1, in2,..., inN* for N input ports and *out1* for the output port.

A merge discipline must be provided as a value to the @qi(mode) attribute, as described in Section 10.2.1. Possible values include @qi(random) (unordered), @qi(fifo) (ordered by time of arrival to the merge process), and @qi(round_robin) (one from each 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.

10.3.3. Deal

@qi(deal) is a task with 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. Port names are *in1* for the input port and *out1, out2,..., outN* for the output ports.

A deal discipline must be provided as a value to the @qi(mode) attribute, as described in Section 10.2.1. Possible values include @qi(random) (unordered), @qi(round_robin) (one to each output port and repeating), @qi(by_type), @qi(grouped_by_2), and @qi(balanced). If dealing by type, the output port must be uniquely identifiable (i.e., there is exactly one output port of the right type for each possible type accepted by the input port.) This is the only kind of @qi(deal) process in which multiple output types make sense. Other kinds of @qi(deal) processes require compatible output types.

10.3.4. Illustrative Task Descriptions

Figure 9 illustrates typical task descriptions for the predefined tasks. The task description in Figure 9.a depicts a 2-output broadcast task that handles items of some type "packet" in parallel. The task description in Figure 9.b depicts a 2-input merge task that handles items of type packet in round robin fashion. Finally, the task description in Figure 9.c depicts a 2-output deal task that handles items of type packet in round robin fashion.

These descriptions do not really exist in the **library**. The **compiler** generates them on demand to satisfy process declarations of the form:

pb: task broadcast attributes mode = *identifier*; **end broadcast;**
pm: task merge attributes mode = *identifier* **end merge;**
pd: task deal attributes mode = *identifier* **end deal;**

where *identifier* is "parallel", "sequential_round_robin", etc., as defined by the implementation.

10.4. Configuration File

Information about the configuration of the heterogeneous machine, the location of system files and libraries, and other random information required by the **compiler**, **library**, and **scheduler** appears in a **configuration file**.

The configuration file in Figure 10 illustrates the definition of the hardware configuration (values for the @qi(processor) attribute), the location of the system task implementations, and various pieces of information about queues and queue operations.

In the @qi(processor) attribute, the meaning of a class name is understood by the **scheduler** as standing for any of the specific values in the class (i.e., a run-time choice of processors). Notice that this choice can be restricted by the user in a task description by specifying a subset of the class, and restricted even further in a task selection by specifying an even smaller subset of allowable processors.

The example configuration file also specifies the location of system files, in particular, the implementations of system tasks. Additional information in the file would describe default queue operations, data transformations, etc.

Keep in mind that the **configuration file** is not written in the task description language. The example shown is just an illustration of the kinds of information that are likely to be contained in the file — form and content of the file are implementation dependent.

```
task broadcast
  ports
    in1: in packet;
    out1, out2: out packet;
  behavior
    ensures "insert(out1, first(in1)) & insert(out2, first(in1))";
    timing  loop (in1 (out1 || out2));
  attributes
    mode = parallel;
end broadcast;
```

a. Parallel Broadcast

```
task merge
  ports
    in1, in2: in packet;
    out1: out packet;
  behavior
    ensures "insert(insert(insert(out1,first(in1)),first(in2)),first(in3))";
    timing  loop ((in1 in2 in3) (repeat 3 => out1));
  attributes
    mode = sequential_round_robin;
end merge;
```

b. Round-Robin Merge

```
task deal
  ports
    in1: in packet;
    out1, out2: out packet;
  behavior
    ensures "insert(out1, first(in1)) & insert(out2, second(in1))";
    timing  loop (in1 out1 in1 out2);
  attributes
    mode = sequential_round_robin;
end deal;
```

c. Round-Robin Deal

Figure 9: Predefined Task Descriptions

```
processor = warp(warp_1, warp2);
processor = sun(sun_1, sun_2, sun_3);
implementation = "/usr/cbw/hetlib/";
default_input_operation = ("get", 0.01 seconds, 0.02 seconds);
default_output_operation = ("put", 0.05 seconds, 0.10 seconds);
default_queue_length = 100;
data_operation = ("fix", "fix.o");
data_operation = ("float", "float.o");
data_operation = ("round_float", "round.o");
data_operation = ("truncate_float", "trunc.o");
```

Figure 10: Configuration File

11. Appendix -- An Extended Example

This appendix illustrates a task-level description of a fictional application. A process-queue graph of the application appears in Figure 11.

11.1. Data Transformation Tasks

```
task corner_turning
  ports
    in1: in landmark_row_major;
    out1: out landmark_column_major;
  attributes
    implementation = "/usr/mrb/screetch.o";
    processor = buffer_processor;
    ...   other attributes uniquely identifying an implementation ...
end corner_turning;
```

11.2. Type Declarations

```
type map_database         is .....  ;
type destination          is .....  ;
type local_path           is .....  ;
type recognized_road      is .....  ;
type road_selection       is .....  ;
type vehicle_position     is .....  ;
type vehicle_motion       is .....  ;
type wheel_motion         is .....  ;
type landmark             is .....  ;
type landmark_list        is .....  ;
type landmark_row_major   is .....  ;
type landmark_column_major is .....  ;
type vision_road          is .....  ;
type sonar_road           is .....  ;
type laser_road           is .....  ;
type road                 is .....  ;
type obstacles            is .....  ;
```

11.3. Task Descriptions

```
task navigator
  ports
    in1: in map_database;
    in2: in destination;
    out1: out road_selection;
    out2: out landmark_list;
  attributes
    author = "jmw";
    version = "1.0";
    processor = "m68020";
end navigator;

task road_predictor
  ports
    in1: in map_database;
```

Figure 11: Example Process-Queue Graph

```
    in2: in road_selection;
    in3: in vehicle_position;
    out1: out road;
end road_predictor;

task landmark_predictor
  ports
    in1: in landmark_list;
    in2: in vehicle_position;
    out1: out landmark_row_major;
end landmark_predictor;

task road_finder
  ports
    in1: in road;
    out1: out recognized_road;
end road_finder;

task landmark_recognizer
  ports
    in1: in landmark_column_major;
    out1: out landmark_column_major;
end landmark_recognizer;

task vision
  ports
    in1: in vision_road;
    out1: out obstacles;
  attributes
    processor = warp;
end vision;

task sonar
  ports
    in1: in sonar_road;
    out1: out obstacles;
  attributes
    processor = warp;
end sonar;

task laser
  ports
    in1: in laser_road;
    out1: out obstacles;
  attributes
    processor = warp;
end laser;

task position_computation
  ports
    in1: in landmark_column_major;
    in2: in vehicle_motion;
    out1, out2: out vehicle_position;
end position_computation;

task local_path_planner
  ports
```

```
    in1: in wheel_motion;
    in2: in obstacles;
    out1: out local_path;
    out2: out vehicle_motion;
end local_path_planner;

task vehicle_control
  ports
    in1: in local_path;
    out1: out wheel_motion;
end vehicle_control;

task obstacle_finder
  ports
    in1: in recognized_road;
    out1: out obstacles;
  behavior
    timing loop (in1[10, 15] out1[3, 4]);
  structure
    process
      p_deal:  task deal attributes mode = by_type end deal;
      p_merge: task merge attributes mode = fifo end merge;
      p_sonar: task sonar;
      p_laser: task laser attributes processor = warp1 end laser;
    bind
      in1 = p_deal.in1;
      out1 = p_merge.out1;
    queue
      q1: p_sonar.out1 > > p_merge.in1;
      q2: p_laser.out1 > > p_merge.in2;
      q3: p_deal.out1  > > p_sonar.in1;
      q4: p_deal.out1  > > p_laser.in1;

    --for dynamic reconfiguration

    if Current_Time >= 6:00:00 local and Current_Time < 18:00:00 local
    then
        process
            p_vision: task vision attributes processor = warp2; end vision;
        queue
            q5: p_deal.out3   > > p_vision.in1;
            q6: p_vision.out1 > > p_merge.in3;
        end if;
end obstacle_finder;
```

11.4. Application Description

```
task ALV
  attributes
    version = "Fall 1986";
    processor = HET0;
    speed  = fast;
  structure
    process
      navigator: task navigator attributes author = "jmw" end navigator;
```

```
      road_predictor:      task road_predictor;
      landmark_predictor:  task landmark_predictor;
      road_finder:         task road_finder;
      landmark_recognizer: task landmark_recognizer;
      obstacle_finder:     task obstacle_finder;
      position_computation:task position_computation;
      local_path_planner:  task local_path_planner;
      vehicle_control:     task vehicle_control;
      ct_process:          task corner_turning;
    queue
      q1: navigator.out1           > > road_predictor.in2;
      q2: navigator.out2           > > landmark_predictor.in1;
      q3: road_predictor.out1      > > road_finder.in1;
      q4: road_finder.out1         > > obstacle_finder.in1;
      q5: obstacle_finder.out1     > > local_path_planner.in2;
      q6: local_path_planner.out1  > > vehicle_control.in1;
      q7: local_path_planner.out2  > > position_computation.in2;
      q8: vehicle_control.out1     > > local_path_planner.in1;
      q9: landmark_predictor.out1  > ct_process > landmark_recognizer.in1;
  -- requires data transformation between row_major and column_major landmarks
      q10:landmark_recognizer.out1 > > position_computation.in1;
      q11:position_computation.out1> > road_predictor.in2;
      q12:position_computation.out2> > landmark_predictor.in2;
end ALV;
```

References

[1] M.R. Barbacci and J.M. Wing.
Specifying Functional and Timing Behavior for Real-time Applications.
Technical Report (in process), Software Engineering Institute, Carnegie Mellon University, 1986.

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

[3] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[4] H.T. Kung.
Private communication.

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

[6] R.G. Stockton.
The Heterogeneous Machine Simulator.
Technical Report (in process), Software Engineering Institute, Carnegie Mellon University, 1986.

Index

" 5, 14

’’ 5

(9, 19, 22, 27, 29, 31, 32

) 9, 19, 22, 27, 29, 31, 32

Comment 5
Configuration file 18, 19, 20, 26, 33
Current_Size 31

DataOp 29

Identifier 5
Integer 5

Larch Interface Specification 14
Larch Trait 14

Minus_Time 31

Plus_Time 31
Processor 26

Real 5

String 5

“ 5
{ 5
|| 20
} 5